5-2018

Lightweight Programming Abstractions for Increased Safety and Performance

Leo Osvald
Purdue University

LIGHTWEIGHT PROGRAMMING ABSTRACTIONS FOR INCREASED SAFETY AND PERFORMANCE

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Leo Osvald

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

May 2018

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF DISSERTATION APPROVAL

Dr. Tiark Rompf, Chair

Department of Computer Science

Dr. Milind Kulkarni

School of Electrical and Computer Engineering

Dr. Benjamin Delaware

Department of Computer Science

Dr. Hubert E. Dunsmore

Department of Computer Science

Approved by:

Dr. Voicu Popescu by Dr. William J. Gorman

Head of the Graduate Program

ACKNOWLEDGMENTS

I would like to thank my Ph.D. advisor, Tiark Rompf, for guiding me ever since he joined the department in Fall 2014 and refining my (earlier) ideas into high-quality publications. This dissertation would not have seen the light of day without his help, encouragement and constant enthusiasm.

Most of the work presented in this dissertation was supported by the National Science Foundation through awards 1047962, 1553471 and 1564207. A special acknowledgment also goes to Jan Vitek, my former academic advisor, who quickly recognized my programming skills and enabled me to jump-start a scientific career, and later warmly welcomed and improved my research proposal on data views. Moreover, the corresponding chapter would not have been included in this dissertation without Benjamin Delaware's suggestions, and the findings not nearly as good if I had not been pointed to the related work by David Gleich. I also thank the rest of my dissertation committee, Buster Dunsmore and Milind Kulkarni, for their timely and helpful feedback, and William J. Gorman for writing tips.

Finally, I express gratitude to multiple members of my family: my mom, Lahorka, for sacrificing herself and raising me in a loving and healthy environment where I would shine, by investing in my education and well-being, as well as my grandma, Ivanka, for all her support and dedication; my brother, Denis, for continuous help and advice; and my smart and loving wife, Baharak, who has been my best friend and stood beside me through the hardest times of my life, and encouraged me to pursue my dreams to the fullest. I love you all and would not have done it without you!

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

CRTP	Curiously Recurring Template Pattern
CPS	Continuation-Passing Style
CPU	Central Processing Unit
DOT	Dependent Object Types
DSL	Domain Specific Language
FIFO	First-In, First-Out
JVM	Java Virtual Machine
LMS	Lightweight Modular Staging (a Scala library)
MSP	Multi-Staged Programming
RAII	Resource Acquisition Is Initialization
RBT	Red-Black Tree
VM	Virtual Machine

NOMENCLATURE

bag	unordered collection of elements (i.e., a multiset)
deque	double-ended queue
list	ordered collection of elements
random-access	with the property that lookups or updates at any position are efficient (practically constant- and asymptotically sublinear-time)

ABSTRACT

Osvald, Leo PhD, Purdue University, May 2018. Lightweight Programming Abstractions for Increased Safety and Performance. Major Professor: Tiark Rompf.

In high-level programming languages, programmers do not need to worry about certain implementation details that compilers or interpreters do behind the scenes. However, this oftentimes results in some loss; in the former case, it is the inability to precisely communicate programmer's intentions to a compiler that compromises safety, and in the latter case, it is the loss of performance because an interpreter needs to do extra work at runtime. Modern languages tend to address this problem differently, albeit rarely without serious limitations. In this dissertation, we develop lightweight programming abstractions whose implementation is practical in multi-paradigm high-level languages such as Scala and C++. The main idea of this work is exploitation of the type system to guide both the code generation (for performance) and type checking (for safety), so that more efficient specialized code is produced or more compiler errors are raised, respectively. This is done by encoding properties of the data as well as data layout, and employing metaprogramming techniques such as staging and template instantiation. We make five main scientific contributions. First, we formalize second-class values with stack-bounded lifetimes as an extension of simply-typed λ calculus, as well as its generalization to polymorphic type systems such as $F_{<:}$, and calculi with path-dependent types described in the Dependent Object Types (DOT) family; we further generalize the binary first- vs second-class distinction to an arbitrary type lattice—or, more generally, a privilege lattice—then show that abstract type members naturally enable privilege parametricity. Second, we propose a model of checked exceptions based on second-class values, which unlike monads, do not suffer from well-established shortcomings of requiring users to rewrite their code

in monadic style throughout. Third, we develop a memory model with data views, which decouple the presentation/interface of a data structure from its layout/storage, and offer not only performance gains through code specialization but also increased safety due to a finer grained control of references to the underlying storage (similar to ownership type systems). Fourth, we design lexically scoped borrowed references with Rust's semantics, including no mutable aliasing, but in a flow-insensitive setting using second-class values. Fifth, we empirically show within a realistic subset of Scala (MiniScala) that performance gains enabled by stack in place of heap allocation, which may be significant according to previous studies, can be guaranteed via second-class values; in fact, the usage of the more expensive heap is reduced to $\mathcal{O}(1)$ in the majority of the benchmarks ported from Scala Native and the Computer Languages Benchmarks Game. Finally, all of these findings are backed by artifacts: an extension of the Scala language with type-checking rules for second-class values and multiple case studies, data views as a library-based framework in C++/Scala along with an evaluation pipeline involving microbenchmarks, an implementation of Rust-like borrowed references as a Scala library, and a modified MiniScala's type-checker and memory allocation scheme, as well as accordingly ported and annotated benchmarks.

1 INTRODUCTION

1.1 Problem statement

In high-level programming languages, programmers do not need to worry much about certain implementation details that compilers or interpreters do behind the scenes. Oftentimes, however, this results in some loss; in the former case, it is the inability to precisely communicate the programmer's intentions to a compiler that *compromises safety*, and in the latter case, it is the *loss of performance* because an interpreter needs to do extra work at runtime. Modern languages tend to address this problem differently, albeit rarely without serious limitations.

For instance, C++ has compile-time template instantiation, but overuse of templates can easily result in code explosion due to excessive code specialization. Also, uninstantiated templates are not type-checked. Nevertheless, templates enable some interesting design patterns such as *compile*-time polymorphism and Curiously Recurring Template Pattern (CRTP), which are fairly specific to C++. C++ also supports a limited form of *scoped capabilities* via unique pointers (its pointees have exactly one owner) and stateful destructors, which is better known as Resource Acquisition Is Initialization (RAII). However, move semantics of unique pointers is insufficient or cumbersome in situations where so-called *borrowing* is desired, such as in function calls; i.e., a uniquely owned pointed-to data or the pointer itself is loaned for a shorter duration than the one of the pointer's lifetime. This negatively impacts the programming style because unique pointer should be moved or have its data released before a function call in order for a function to be able to use the uniquely owned data, then the function must return the unique pointer to the same data so that the caller regains a unique ownership after moving in the returned value. On the other hand,

using shared pointers causes a run-time overhead because a C++ compiler cannot reliably deduce borrowing patterns via pointer analysis.

Another attempt of increasing safety is the `final` keyword in Java. If an anonymous class instantiated in a method body refers to a local variable, that variable needs to be marked `final`. The justification is that such a construct is equivalent to a closure in which the variable is *free* (unbound), so to prevent potentially unexpected behavior in case of its reassignment, Java takes a radical approach; it is *forbidden* to change the free reference to point to another object. (C++ has undefined behavior for such by-reference captures that go out of scope.) Nevertheless, free variables can inadvertently create expensive, first-class closures, since they may extend lifetimes of objects they refer to. Some compilers, such as Go, go to great lengths in order to deduce such lifetimes and thus avoid allocation of closure objects at run-time, but there is little hope for guarantees without exposing this information at the type system level.

The Go programming language has built-in abstractions for views of contiguous storage known as *slices*. (The very same idea has been streamlined for at least a decade via Google's (open-source) C++ libraries, even though slices first appeared in the C++98 Standard Library.) These slices can be written through, and reslicing a slice or an array is a *constant*-time operation—amortized constant-time if the slice grows an underlying backing array. However, these abstractions incur performance overhead in terms of both CPU time and memory, as the Go runtime needs to keep track of slice ranges, and they are also susceptible to *run-time* panics (i.e., out-of-bounds exceptions) as such. Further, the programmer cannot control memory deallocation upon their shrinkage, and they can grow by appending only at the end (not the beginning nor the middle).

Finally, the Rust programming language goes furthest in terms of a memory model. Its type system statically prevents *mutably aliased* memory, use after free, double free, and use of uninitialized memory (including null-pointer dereferencing). While Rust does eliminate many pitfalls at compile-time and without runtime over-

head, it puts a significant burden on the programmers, who may now need to thread variable lifetimes through their code or settle for a different, perhaps inferior, code design. This issue is exacerbated by *flow sensitivity* (i.e., the type of an expression depends on control flow), making Rust programs one of the hardest to debug (which is perhaps why the authors put significant effort into pretty printing each compile error along with its explanation).

1.1.1 Programming language choice

In this dissertation, we propose lightweight programming abstractions whose implementation is practical in multi-paradigm high-level languages. As the two representative languages of implementation, we choose Scala and C++, which are both compiled languages, in order to eliminate interpretation overhead.

Scala is both functional and object-oriented, and runs atop Java Virtual Machine after being compiled into Java bytecode. Therefore, it is as dynamic as Java, but the additional compilation step allows for its much richer and thus safer type system, featuring: *path-dependent* types, *declaration*-site type co/contra/in-variance, higher-kinded polymorphism (via a library), just to name a few. Scala was selected primarily because it is a successful pioneer in bringing the latest programming language research into practice, and because of its similarity to Java. That being said, attempts to model the Resource Acquisition Is Initialization design pattern in Scala like in C++ fail miserably, since Scala has reference semantics (borrowed from Java), i.e., copying cannot be controlled by the user. The automatic garbage collection does not help, either; the core problem is lack of facilities to communicate to the compiler that a variable (i.e., a resource) must not *escape* its declaring scope.

C++ is a more rigidly typed language, with a Turing-complete template system that allows for *duck* typing (unlike Scala). Memory is not managed in C++ (unless user opts in to reference-counted `std::shared_pointer`s), but there is a flexible copy and ownership semantics via copy and move constructors/assignments. How-

ever, due to its backward-compatibility with C, most of the safety guarantees are void as soon as as one uses regular pointers or arrays, as opposed to unique/shared pointers or `std::array`, respectively; but these replacements impose a number of restrictions, and thus put a heavy burden on the programmer. More specifically, using `std::array` requires size to be known at *compile*-time, using move requires careful design of classes and sacrifices ease of extension for performance, unique ownership is not always possible due to complicated objects' lifetimes, yet shared ownership hurts performance.

1.2 Overview

1.2.1 Second-class values

Second-class values as they appeared in ALGOL have the benefit of following a *strict stack discipline* ("downward funargs"), i.e., they cannot escape their defining scope. This makes them cheaper to implement, but more importantly, phasing out second-class entities has eliminated some useful programming patterns and static guarantees. They are naturally used for functions in ALGOL as well as Pascal, since neither language has automatic garbage collection to collect escaping variables that would otherwise be possibly created via first-class closures. Since first-class objects *may* escape their defining scope, they cannot be used to represent static capabilities or access tokens—a task that second-class values are ideally suited to because they have bounded lifetimes and they have to "show up in person".

Unfortunately, most modern languages have abolished these restrictions and admit functions (or objects with methods) as first-class citizens alongside integers and real numbers, leading to an undesirable situation where inexpensive and restricted "second-class" constructs are no longer available. One of our key findings is that their *non-escaping property* can be statically guaranteed by enforcing the following rules:

1. First-class functions may not refer to second-class values through free variables

2. All functions must return first-class values, and only first- class values may be stored in object fields or mutable variables

We propose a type system in which a violation of either of the above rules results in a compilation error. Our system supports objects of any type as second-class values, unlike systems that expose a distinct category of second-class functions, reference cells, or other entities. The imposed rules are similar to those on *borrowed references* [1, 2] in ownership type systems, e.g., as implemented in Rust [3], but there are two key differences:

- We claim that these restrictions have important benefits as a *programming model*, orthogonal to the goals of ownership types (controlling aliasing, ensuring uniqueness, preventing race conditions, etc.).

- In contrast to sophisticated ownership type systems, such a type system is straightforward to formalize and integrate with *existing* languages and other advanced type system features.

To support the latter claim, our system has been implemented as an extension of the Scala language. The Scala type system is backed by System $D_{:<}$, which at its core is a system of first-class type objects and path-dependent types. Recently, the calculus of Dependent Object Types (DOT) has been proved sound by Rompf and Amin [4].

1.2.2 Data views

We present a library-based framework of data views over chunks of memory segments. Such views not only enable a uniform treatment of references and arrays, but they provide a more general abstraction in the sense that parts of arrays, references, or even views, can be combined into hierarchies to form new logical data structures. To provide efficient implementations in widely used industrial languages such as C++ and Scala, we employ static and dynamic multi-staging techniques, respectively. Through staging and code specialization, the overhead of traversal and tracking of such view hierarchies is mostly eliminated. Thus, our data views can be used as building blocks for creating data structures for which programmers need not pick a specific representation but can rely on code generation and specialization to provide the right implementation that meets asymptotic running time and space guarantees.

The simplest type of view we propose is a one-dimensional *array* view, which is basically an ordered collection of chunks of contiguous memory (also called array slices) and/or views themselves. We refer to either of these constituents as view

portions. A so-called *simple* view is the one in which no portion is another view; Figure 1.1 shows one such view.

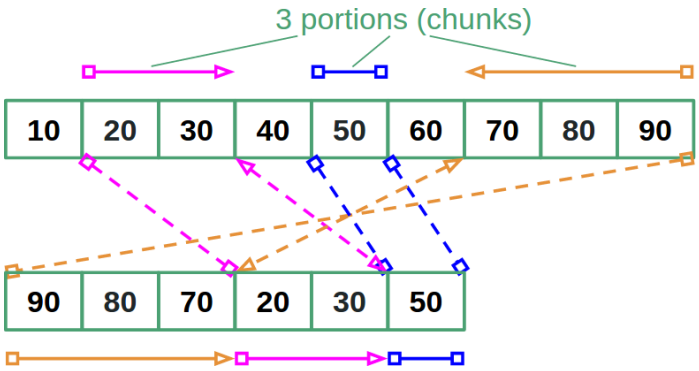


Figure 1.1. A view (at the bottom) composed of three memory chunks (at the top): the last three elements in the reverse order, the middle element, and the second and third element, respectively.

By extension, we define N-dimensional (ND) array views as a generalization that supports *logical* layout as of ND arrays (e.g., a matrix if N=2) but with their physical layout hidden. For example, such an abstraction should provide an efficient indexing by coordinates as well as efficient iteration along any of its dimensions. To illustrate that such a problem is not trivial, consider a well-known representation of a sparse matrix in Compressed Sparse Row (CSR) format, which contiguously stores column coordinates of non-zero elements. However, such a representation sacrifices efficiency of column-wise access for a more efficient row-wise access; traversing along a specific column requires some sort of a binary search in each of the rows, and thus requires more than (amortized) constant-time per element. Other formats have their own trade-offs.

Instead of settling for a specific representation, we provide a general framework for *specializing* representations of data depending on its *structure* and properties. Some examples are:

- a view that sees every k-th element of an array can be stored as a pair (array a, indexed access function $\lambda i.a[k \cdot i]$);
- tridiagonal matrix as a *composite* view of three 1D array views;
- a view of *immutable* (infinite) series of elements can be represented in $\mathcal{O}(1)$ space using an indexed access function;
- record, class or environment as an *unordered* view.

When two instances of an ordered data structure are catenated together to form a bigger instance in a *persistent* way (i.e., that both the instances as well as the merger can be accessed), this necessitates multiple levels of nesting in order to avoid decreased performance after many such operations. A simple scenario that results in such a tree-like hierarchy is when a bigger view is repeatedly created out of two or more smaller views. However, having a deep nesting hierarchy hurts performance due to indirection while reading through such composite views. Therefore, we propose using efficient tree-like data structures that we review in Section 1.5.2 for nested views, depending on their (statically) declared properties.

As hinted by the examples, using both properties and layout of the data allows for a more efficient access or storage. So, one of the key ideas in this work is to encode that information into the types. This can be achieved in two ways: *explicitly*, by requiring usage of special types; and *implicitly* via staging, by compiling the code at run-time and evaluating first-stage values, then inspecting the Abstract Syntax Tree and emitting the specialized code in the second stage. In the former case, C++ templates alleviate the burden of pattern matching on types, since the family of closely related types can be represented via a type template (e.g., `Diag<T, BlkHeight, BlkWidth>`) in order to easily refer to their variations with different parameters via function templates that act as metafunctions or in partial specialization (e.g., `template<typename T, size_t...S> Diag<T, Same(S...)>`). The template instantiation allows the compiler to inline certain computation and specialize the code based on the actual template parameters computed at compile-time. In the latter case, the parameters that are known only at run-time must be staged (i.e., evaluated in a

later stage), but the rest of the code will be executed and hence inlined or specialized through staging. Compilation at run-time is possible due to the virtualized environment (i.e., Java Virtual Machine). Therefore, the end result is the same, although the latter approach has additional advantages (see Section 3.4.2).

Ultimately, using the multi-stage programming framework Lightweight Modular Staging (LMS) [5] in Scala, we support fine-grained specialization of view types at run-time. The compilation overhead is negligible when lots of data is read or written through a view, since we use efficient data structures and view merging algorithms. The whole machinery (staging, code generation and compilation) is hidden from the user by exposing the view framework as a Scala library that relies heavily on lazy evaluation and implicit conversions.

1.2.3 Flow-insensitive Rust-like references

The Rust programming language[1] demonstrates that memory safety is achievable in a practical systems language, based on a sophisticated type system that controls object lifetimes and aliasing through notions of ownership and borrowing. It features a borrow checker that enforces Rust *unique ownership* semantics with borrowing. Each object is stack-allocated by default and owned by a unique variable that is either mutable or immutable. The variable can temporarily hand off the ownership through *borrowed references* to or into the object. Such a borrowed reference must be immutable unless the source (object or its reference) is mutable and there are no other references (that are live [6]). In other words, it is ensured by typing rules that a mutable access is exclusive (i.e., unaliased), while the sole immutable access can be shared. It must be noted that the owner itself is not allowed to mutate the object for the duration of the mutable borrow, nor any of the inactive mutable borrows [3]. Therefore, mutable references to objects have a uniqueness property in a sense that, for a duration of a function call, they can be either exclusively borrowed as

[1] `https://www.rust-lang.org`

mutable or multiply borrowed as immutable. Conversely, immutable references can be copied, and thus freely shared or immutably borrowed. (Of course, the conversion from immutable to mutable is not allowed.) There is an explicit syntax in Rust for describing the *lifetime* bounds associated with a borrowed reference. A function needs to constrain its borrowing input arguments to ensure that its body does not violate the abovementioned aliasing rule that guarantees safety. The lifetimes bounds are inferred automatically in simple cases (e.g., a single borrowed reference in input parameters), but they also mix together with lifetime polymorphism (analogous to type polymorphism) or appear as template arguments to provide more expressiveness.

While Scala has traditionally targeted only managed language runtimes, the Scala Native[2] effort makes Scala a viable low-level language as well. Thus, memory safety becomes an important concern, and the question bears asking what, if anything, Scala can learn from Rust. In addition, Rust's type system can encode forms of protocols, state machines, and session types, which would also be useful for Scala in general. A key challenge is that Rust's typing rules are inherently flow-sensitive, but Scala's type system is not.

Our solution presented in Chapter 4 achieves static guarantees similar to Rust with only mild extensions to Scala's type system. It is based on two components:

- the observation that monadic or continuation-passing style can transform a flow-sensitive checking problem into a type-checking problem based on scopes; and
- on our type system extension with second-class values (presented in Chapter 2), which we use to model scope-based lifetimes.

Despite the former, our approach is still practical because the burden of writing programs in monadic style can be eliminated through macros, like Scala async/await[3], or by using Scala's existing CPS transformation plug-in [7]. The additional benefit is that by modeling Rust's borrowed references in Scala, one can further generalize them

[2] `http://scala-native.org`

[3] `https://github.com/scala/async`

and increase their performance by applying the concepts of data views and dynamic specialization presented in Section 3.4.2. Finally, the pointed-to (referenced) data can be allocated on stack in many situations as Chapter 5 demonstrates (albeit on a subset of Scala), thus further closing the performance gap between Scala and Rust.

1.3 Contributions

The lightweight abstractions we introduce were rigorously tested in the above languages. Their important theoretical foundations have also been proved mechanically using the Coq proof assistant. Our key contributions are as follows:

1. *formal model* of second-class values with stack-bounded lifetimes, a generalization of second-class functions from Pascal/Algol, for *improved safety* guarantees via more precise type checking as well as the more *efficient allocation on stack*;

2. several corroborating case studies, showing the implementation of second-class values alongside first-class values is practical in a modern programming language (Scala) and useful as a *programming model*;

3. introducing the *first practical* model of *checked exceptions* in the Scala standard library by employing second-class values as (implicit) scoped capabilities;

4. design and implementation of view abstractions for *flexible* and independent data *layout* and *storage*, aimed at improving performance via metaprogramming techniques for generating specialized code, both statically and dynamically;

5. experimental results showing that our *data views perform* well in practice, compared against general-purpose and domain-specific representations in state-of-the-art libraries based on a series of common microbenchmarks;

6. prototype implementation of a *memory model* based on view abstractions and the associated results indicating that its application is feasible in concurrent and/or multi-threaded environments as well as hard real-time systems;

7. implementation of *borrowing* including references with ownership and *no-mutable-alias* semantics as defined in Rust (Rust-like) on top of System $D_{:<}$, more precisely Scala, which is *flow-insensitive* unlike Rust, using second-class values;

8. empirical evaluation of *how much heap allocation is avoided* when second-class values are instead allocated on stack, on a series of well-known benchmarks.

The central idea that connects these points is exploitation of the type system to guide both the code generation (for performance) and type checking (for safety), so that more efficient specialized code is emitted or more compiler errors are raised, respectively. This is done by encoding *properties* of the data including lifetimes and mutability, as well as data *layout* using views to better control aliasing, and employing metaprogramming techniques such as *staging* and *template instantiation*.

The second-classiness is one such property that enables the compiler to either safely allocate the value on the stack—cheaper than the heap (where discontiguous portions are used)—or raise the error that the code does not type-check, thereby delivering on the promise to provide static safety guarantees. It enforces the stack-bounded lifetime of a value, as opposed to heap-allocated values with the unbounded lifetimes that may cause *memory thrashing* in environments (VMs or linked language run-times) with garbage collection. We empirically explore how much heap allocation can be instead done on stack in Chapter 5.

Combining these two concepts together into second-class views enables a cheap yet fine-grained notion of borrowed references—in fact, a more powerful one that can alias a specific subset of data, create a new logical layout, or even allocate new data analogously to Go slices (if the view itself is mutable), as we discuss in Chapter 3. We provide a practical Scala solution to borrowing and static prevention of mutable aliasing—a safety issue that Rust statically prevents—by concisely modeling Rust's notion of ownership and borrowing semantics for mutable and immutable references using second-class values (provided via our compiler plug-in) and advanced features of Scala's type system in Chapter 4.

All of the above contributions are supported by artifacts; moreover, the first three corresponding artifacts won the Distinguished Artifact Award at OOPSLA'16. Contribution 1 (formalizing second-class values) is accompanied by mechanized Coq proofs, which were mostly written by Grégory Essertel. In order to achieve Contributions 2–3 (applications of second-class values), a Scala compiler plug-in and additional compiler stage have been developed in Scala. For Contribution 3 (checked exceptions),

Scala's type checker has been modified to support three newly introduced constructs analogous to `try`, `catch`, and `@SuppressWarnings` in Java. To support Contribution 4, a template-based C++ view library (cppviews), as well as the Scalaviews library, has been written. For Contribution 5, a generic pipeline for code generation and benchmarking has been developed, which is capable of comparing the running time of operations on the C++/Scala views versus arbitrary implementations of data structure (via appropriate facades that map to view-defined behavior). Regarding Contribution 6, performance-critical parts have been implemented in Java in order to benchmark the proposed algorithms and data structures. The relevant chunks of code for Contribution 7 are self-contained in this dissertation. Finally, Contribution 8 uses derivatives of copyright-protected code, but we may grant access to certain individuals (not corporations); the ported benchmarks are open-source, however.

1.3.1 Publications

The work in this dissertation has been rigorously peer-reviewed; Chapters 2,3,4 are based on the scholarly articles published in prominent conference and workshop proceedings[4]:

- *Gentrification Gone Too Far? Affordable 2nd-class Values for Fun and (Co-)Effect* by **Leo Osvald**, Grégory Essertel, Xilun Wu, Lilliam I. González Alayón, and Tiark Rompf, in the Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'16), `http://doi.org/10.1145/2983990.2984009`;
- *Flexible Data Views: Design and Implementation* by **Leo Osvald** and Tiark Rompf, in the Proceedings of the 4th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY'17), `https://doi.org/10.1145/3091966.3091970`;
- *Rust-like Borrowing with 2nd-class Values (Short Paper)* by **Leo Osvald** and Tiark Rompf, in the Proceedings of the 8th ACM International Symposium on Scala (Scala'17), `https://doi.org/10.1145/3136000.3136010`; respectively.

Although Chapter 5 is unpublished at the time, it is just a performance study for second-class values (from Chapter 2) that uses the methodology and artifacts developed for a previously peer-reviewed and published article and an abstract (for a talk):

- *Evaluating the Design of the R Language - Objects and Functions for Data Analysis* by Floréal Morandat, Brandon Hill, **Leo Osvald**, and Jan Vitek, in the Proceedings of the 26th European Conference on Object-Oriented Programming (ECOOP'12);
- *TraceR: A framework for understanding R performance* by **Leo Osvald**, Brandon Hill, Floréal Morandat, and Jan Vitek, at the 8th International useR! conference, 2012.

[4]The definitive versions were published by ACM in the listed proceedings.

1.4 Hypothesis

Using a combination of second-class values and fine-grained memory access through data views leads to increased safety and performance, and allows more expressiveness.

1.4.1 Rationale

Second-class values alongside first-class values enable selective enforcement of stack-bounded lifetimes of annotated or type-inferred values at compile-time. They offer a number of safety benefits (capabilities, checked exceptions, etc.) that are unparalleled by previous approaches.

Further, recent advances in the area of data structures (and algorithms), especially purely functional ones, can be applied to create and maintain data views in asymptotically equivalent time complexity compared to ubiquitous data structures used for run-time memory layout such as references, object fields, arrays and slices. However, our data views are more general than references or pointers, since they can represent discontiguous or replicated parts of virtual memory. Using views can, in fact, result in asymptotic space savings of memory in the latter case as well as in cases where a programmer would inadvertently create deeply linked (nested) data structures that serve as more convenient access.

Therefore, data views provide a more flexible and fine-grained memory access than traditional approaches and in many cases, such as when a view is constructed solely for programmers' convenience, their stack-allocation can be guaranteed by marking them as second-class. This shifts the overhead of view creation from run-time to compile-time, thus enabling abstraction without performance loss.

Finally, the restrictions that arise from statically enforcing no mutable aliasing and lexical scoping of borrowed references are lifted by not requiring the granularity at the data structure level (i.e., a reference to a variable or a field), which keeps the static guarantees but provides more expressiveness.

1.5 Related work

1.5.1 Second-class values

Strachey [8] publicized the terminology of first- and second-class objects in 1967:

> In ALGOL a real number may appear in an expression or be assigned to a variable, and either may appear as an actual parameter in a procedure call. A procedure, on the other hand, may only appear in another procedure call either as the operator (the most common case) or as one of the actual parameters. There are no other expressions involving procedures or whose results are procedures. Thus in a sense procedures in ALGOL are second class citizens—they always have to appear in person and can never be represented by a variable or expression (except in the case of a formal parameter).

The issues around stack-implementability of functions in LISP is also known as the funarg problem [9, 10], and conditions for stack implementation of the simply-typed call-by-value lambda calculus have been given by Banerjee and Schmidt [11]. Hannan presented a type-based escape analysis [12], to infer when variables can be allocated on the stack. The type systems in this paper are similar to Hannan's internal formulation. Taha and Nielson have proposed environment classifiers [13] to ensure non-escaping behavior in the context of program generation. Tanter has proposed notions of scope more fine grained than the usual notions of lexical vs dynamic scope [14].

Capabilities Capabilities as a programming model in dynamic languages were made popular by Miller's E language [15]. The capabilities we study take a similar approach to static checking as recent work on co-effects [16]. The idea is to view program behavior such as side effects not as part of the program term, but as part of the context, where an appropriate license or capability must be present. Recent proposals call for their use in more general effect systems [17].

Types, regions and effect systems Early work on memory regions based on RC, a dialect of C proposed by David E. Gay [18] that guarantees temporal safety. Effect and region polymorphism [19], for example in the FX programming language [20]. Talpin and Jouvelot [21, 22] introduce subeffecting and present the first effect and region inference algorithm. Lippmeier [23] extends Haskell with mutable state and call-by-value semantics for effectful parts of programs. Tofte and Talpin [24] show how type, region and effect inference can lead to a stack based implementation for languages with reference allocations and updates, as implemented in MLKit [25]. Siek, Vitousek, and Turner present a type and effect system focused on supporting both stack-allocation and expressive higher-order programming patterns (e.g., currying) [26].

Type-and-effect systems were proposed by Gifford [27]. Particular systems have been designed for exceptions [28], purity [29], and atomicity [30], among others. Work by Marino and Millstein [31] and by Rytz [32, 33] abstracts such individual systems into generic frameworks for larger classes of effect domains. Nielson and Nielson [34] go from flow-insensitive to flow-sensitive effects.

In the presence of global type inference as in Haskell or ML, it is natural to look for similar procedures for global effect inference. This work, however, has a different focus, and seeks to provide *programming abstractions* for describing and checking effects. It aims at languages like Scala that combine object-oriented and functional programming with subtyping, parametric polymorphism, and that in general do not support global type inference [35]. In this setting, small and comprehensible type annotations are of key importance.

Monads Monads [36] are a popular approach to encapsulate side effects in pure functional languages, especially Haskell [37, 38]. Despite their great success, they are not without issues. First, programs that use more than one kind of side effect has to combine multiple monads, which is not straightforward [39]. Monad transformers [40] help, but they often require programmers to explicitly lift operations. Second,

introducing side effects into existing code requires refactoring that code into monadic style, and also any other code that uses it. The fact that monadic and pure code have incompatible types leads to code duplication, as evidenced by functions `map` and `mapM` in Haskell [23]. Monads have been linked to type-and-effect systems [41] and generalized in a variety of ways, e.g., as parametrized monads [42]. Tate formalized the sequential semantics of "producer" effects using indexed monads [43].

Kiselyov and Shan [44, 45] introduced an SIO monad for lightweight monadic regions, based on phantom types and rank-2 polymorphism, that can also manage file handlers safely and efficiently. Their approach ensures that all resources used are deallocated exactly once, and they support improperly nested lifetimes using explicit lifting operations.

Alternative systems for controlling effects Algebraic effects have gained attention recently [46, 47]. Unlike monads, combining effects is straightforward, but most systems do not check effects statically. Potentially, a program might evaluate to an undefined state where an effect operation appears outside a handler. The situation is different in languages with dependent types [39]. Other lines of work worth noting are linear types [48], uniqueness types [49], witnesses for side effects [50]. Koka [51] is a programming language that can express effect-polymorphism and also constructs like exception handlers that mask effects. In the context of Scala, simple type-and-effect systems have been used to implement Delimited continuations, based on a type-directed selective CPS transform [7]. Effects and static checking are particularly important in the context of domain-specific languages [52–56]. Applications such as preventing scope extrusion are important in the context of generative programming using Lightweight Modular Staging [5, 57, 58].

Memory allocation schemes Opportunities and performance gains that are due to stack allocation performed by the compiler in place of heap allocation have been studied theoretically [59] and in the context of JVM-based object-oriented languages [60–65] as well as embedded systems [66]. The issue of memory management, as well

as object lifetimes, has recently been analyzed for Scala in particular and contrasted with Java programs [67, 68].

1.5.2 Data structures and views

The View Template Library [69] is the most closely related work we are aware of; it implements views in C++ as container adaptors which provide access to different representations of data that are generated on the fly. Such a concept view is a generalization of a *smart iterator* [70], which can filter data while iterating over a data structure (i.e., a container), as opposed to providing a transformed access over it. A *live* data view [71] has also been studied in the context of parallel and mobile environments as a programming abstraction of a time window over streaming data.

Persistent data structures A general framework for turning ephemeral pointer-based data structures into persistent ones was provided by Driscoll et al. [72] and improved by Brodal [73].

Arrays The concept of an array for contiguous storage has been introduced by Konrad Zuse [74], and Fortran was the first language that implemented it. *Discontiguous* arrays divided into indexed chunks have been proposed by several researchers [75–77], and have been extensively studied in the scope of virtual machines, where fragmentation caused by large arrays results unpredictable space-and-time performance during garbage collection. To reduce fragmentation, Siebart [77] groups such chunks into a tree, but this requires an expensive tree traversal on every access. Bacon et al. [75], Pizlo at al. [78], and Sartor et al. [79] use a single level of indirection to fixed size *arraylets*. Sartor et al. further reduce the indirection overhead by a constant factor via their first-N optimization, and use other optimization techniques such as zero compression, lazy allocation, and arraylet copy-on-write [79].

Trees Kuszmaul [80] provides a technique for merging balanced binary trees in $\mathcal{O}(1)$ amortized time. *Compressed* trees have been studied by Eltabakh et al. [81] for online search of Run-Length-Encoded data in the context of databases as well as for asymptotically faster algorithms by Larkin [82].

Red-black trees were invented by Guibas and Sedgewick [83], and remain one of the few balanced search trees in which rebalance after every operation requires $\mathcal{O}(1)$ rotations in the worst case (including the deletion).

AVL trees [84] have remained one of the most rigidly balanced trees ever since their introduction in 1962, and require at most two rotations per insertion. That a deletion in an AVL tree can cause $\Theta(\log n)$ rotations, even in the amortized case, has been proved by Amani et al. [85].

Sen, Tarjan and Kim [86, 87] recently described a relaxation of balanced binary search trees in which deletions do not rebalance the tree at all, yet worst-case access time remains logarithmic in the number of insertions, provided that it is periodically rebuilt. In their 2016 paper [87], , they show in particular that *relaxed* AVL and red-black tress perform $\sim$50% less rotations, while the access time is increased by only 5% on average (11% and 33% in worst case, respectively).

For cases when access is localized, faster trees exist. Levcopolous and Overmars [88] invented search tree in which the time to insert or delete a key is $\mathcal{O}(1)$ once the position of the key to be inserted or deleted was known. Dietz and Raman [89] describe an enhanced data structure that additionally supports *fingers* and to additionally allow logarithmic time access around a finger proportional to the distance to it, provided that the RAM with logarithmic word size model is used. Hinze and Paterson invented 2-3 trees known as Finger trees [90], which are purely functional and designed with simplicity of implementation in mind.

Finally, some trees were invented to perform better on *non-uniform* access patterns; their amortized time to access an item v is in $\mathcal{O}(1+\log \frac{m}{c(v)})$, which matches the theoretical optimum [91] as a function of access frequencies $c(v)$. The earliest such is the splay tree by Sleator and Tarjan [92]. In a recent work with Tarjan, Yehuda et

al. [93] devised the CB tree—a practical concurrent alternative that achieves the same asymptotic optimality—in which the number of rotations is *subconstant* amortized if the majority of operations are lookups and/or updates (not insertions).

Lists A Skip list [94] is a simpler and significantly faster alternative to traditional self-balancing search trees, but with the same asymptotic expected time bounds (i.e., $\mathcal{O}(\log n)$) proved by randomized analysis. It supports catenation and splitting, but it is not particularly efficient for order queries [95]. A purely functional *sorted* list that supports join (i.e., order-preserving catenation) in $\mathcal{O}(1)$, while also supporting $\mathcal{O}(\log n)$ insertion, deletion, and lookup, is described by Brodal et al. [96].

Purely functional *arrays* with lookup/update in $\mathcal{O}(\log \log n)$ amortized time were designed by Dietz [97].

Driscoll, Sleator and Tarjan [98] devised purely functional *stacks* with constant-time push/pop and catenation in amortized $\mathcal{O}(\log \log k)$ time and space, where k is the number of stack operations before catenation. Kaplan and Tarjan improved the worst-case running time to $\mathcal{O}(1)$ for the above operations, as well as for the newly supported push/pop at the opposite end, on a simpler data structure [99] using the recursive slow-down technique; they also used this technique to further simplify and improve the efficiency of such catenable *deques* (*d*ouble-*e*nded *que*ues) [100]. If *memoization* is allowed, using Okasaki's implicit recursive slow-down [101] yields even more general but asymptotically equally efficient persistent data structures—albeit no longer purely functional.

A purely functional *random-access* list that supports $\mathcal{O}(\min\{i, \log n\})$ time lookup or update at index i, and stack operations in $\mathcal{O}(1)$ time, was presented by Okasaki [102]. If external immutability suffices, there are simpler fully persistent random access deques that rely on memoization and lazy evaluation to achieve amortized $\mathcal{O}(1)$ deque operations including catenation, in addition to access in $\mathcal{O}(\log i)$ amortized time, as shown by Brodal et al. [103]. The RRB vector [104] is a random-access deque

that supports appending/deleting at either end in amortized $\mathcal{O}(1)$ time, catenation and lookup/update at index in $\mathcal{O}(\log n)$, but exploits spatio-temporal locality.

Metaprogramming C++ templates were first presented in 1988 by Stroustrup [105], the C++ language inventor, who also wrote about early history of C++ [106]. Siek and Taha [107] formalize semantics of C++ templates, which provide a Turing-complete sub-language within C++ through specialization. Cole and Parker [108] develop a method for *dynamic* compilation of C++ templates that delays code generation for instantiated templates until they are actually used at run-time. Multi-staged programming (MSP) was pioneered by Taha [109], mostly through MetaML [110] that allows code generation at run-time. Czarnecki et al. [111] show how to implement Domain Specific Languages (DSLs) using MSP: dynamically in MetaOCaml [112], but also statically in Template Haskell [113] and C++ via template metaprogramming. Lightweight Modular Staging (LMS) [5] is a Scala library for MSP that relies solely on types to distinguish the computational stages, unlike previous approaches—MetaML [110] and MetaOCaml [112]—which rely on quasiquotes. Scala LMS is inspired by Carette et al. [114] and Hofer et al. [115], can generate the code at run-time, and allows for *deeply embedded* DSL implementation through Scala-Virtualized [116].

1.5.3 Ownership and borrowing

Ownership type systems [117–119] were designed to protect against unintentional aliasing and unexpected side effects in object-oriented programs. The notion of borrowing [1,2], denoting a temporary transfer of ownership for the duration of a method call, greatly improves the usability of such systems. Borrowed references are subject to similar constraints as second-class values we define. Our contribution is to show that such second-class constraints are useful as a programming model independent of ownership, aliasing, and even of mutable state and a store abstraction altogether. We are also not aware of any ownership type system that provides facilities like our

privilege lattice and privilege parametricity, leveraging host language features such as abstract type members and path-dependent types.

Temporary aliasing on (borrowed) objects does not require destructive reads, and is thus similar to our approach. Clarke and Wrigstad [120,121] allow it in the form of borrowing, and statically enforce external uniqueness otherwise. In AliasJava [122], this is done via lent references, which cannot be stored to fields, and thus can safely point to any ownership context. Boyland uses technique called alias burying [123], which is based on static analysis that tracks live aliases. Haller and Odersky [124,125] model unique and borrowed references in Scala using capabilities, and also support ownership transfer. In contrast, our work exploits the fact that second-class values cannot be stored in a field or returned from a function, and is entirely type-directed. It is similar to generic universe types [126,127], except that we do not support ownership transfer; however, this is not limiting because our references may be temporarily aliased when they are passed to second-class function parameters during a function call. Overall, our design is similar to LaCasa [128]—their boxes map to our variable wrappers (references), and opening a box is similar to introducing a scope-based dereferenced value in our case—but we also distinguish between immutable and mutable references, like the Pony language [129].

Rust [3] is a recent language by Mozilla that incorporates region-like memory handling based on ownership and borrowing of references. Its semantics is informally explained by its authors [3] and developers [6]. Formalizing Rust's type system has recently been an active area of research; efforts started in 2015 [130], but the first formal (and machine-checked) proof for a realistic subset of Rust was published in 2018 [131]. Cyclone [132] is an earlier approach to build a safe dialect of C based on similar ideas.

2 AFFORDABLE SECOND-CLASS VALUES

Modern programming languages offer much greater expressiveness than their ancestors from the 1960s and '70s. Many of the advancements that directly translate to programmer productivity are the result of removing restrictions on how certain entities can be used, and granting "first-class" status to more and more language constructs. Even conservative languages, like Java and C++, have added closures, albeit with some limitations. First-class functions dramatically increase expressiveness, at the expense of static guarantees. In ALGOL or PASCAL, functions could be passed as arguments but never escape their defining scope. Therefore, function arguments could serve as temporary access tokens or capabilities, enabling callees to perform some action, but only for the duration of the call. In modern languages, such programming patterns are no longer available. (Many languages still distinguish between, e.g., normal functions and closures, but most allow converting second- to first-class values via eta-expansion, which effectively removes the distinction.)

The central thrust of this chapter is to reintroduce second-class values alongside first-class entities in modern languages, and to demonstrate that this combination leads to novel and elegant implementation techniques for desirable static guarantees.

We formalize second-class values with stack-bounded lifetimes as an extension to simply-typed λ calculus, and for richer type systems such as $F_{<:}$ and systems with path-dependent types. We generalize the binary first- vs second-class distinction to arbitrary privilege lattices, with the underlying type lattice as a special case. In this setting, abstract types naturally enable privilege parametricity. We prove type soundness and lifetime properties in Coq.

We implement our system as an extension of Scala, and present several case studies. First, we modify the Scala Collections library and add privilege annotations to all higher-order functions. Privilege parametricity is key to retain the high degree

of code-reuse between sequential and parallel as well as lazy and eager collections. Second, we use scoped capabilities to introduce a model of checked exceptions in the Scala library, with only few changes to the code. Third, we employ second-class capabilities for memory safety in a region-based off-heap memory library. Last, we show that differentiation between relative privileges of second-class values enables enforcement of a security model based on information (data) flow.

2.1 Motivating examples

To demonstrate the versatility and usefulness of our programming model, we discuss a series of motivating examples. These are presented in Scala but would directly map to other modern call-by-value languages.

2.1.1 Scoped capabilities

Many entities come with a life cycle protocol that guards access. For example, when accessing a file or network connection, a program needs to *open* it, and *close* it when it is done. Accessing a file after closing it or forgetting to close a file is an error. A common and extremely useful pattern is to associate the *dynamic lifetime* of the access window with a *lexical scope*. In `C++` this can be realized with constructors and destructors for stack-allocated objects, Python has `with`, Go has `defer`, and in Scala we can define a higher-order function `withFile` that takes care of opening and closing the file, delegating to a handler `fn` for the actual processing:

```
def withFile[U](n: String)(fn: File => U): U = {
  val f = new File(n); try fn(f) finally f.close()
}
```

Client code can use `withFile` as follows:

```
withFile("out.txt") { file => file.print("Hello, World!") }
```

Thus, `file` can be seen as a *capability*: to write data to disk, we need to be given access to a `File` object via `withFile`, and when `withFile` exits, this capability is revoked.

Unfortunately, in Scala, or any other language where `file` is a first-class value, this programming pattern is merely a convention, but nothing *actually* prevents `file` from being accessed outside its lifetime window. This can lead to subtle errors, undesirable exceptions, or potential security vulnerabilities. Here are two easy ways to thwart the pattern, by assigning the file to a mutable variable or by returning it as result from the `withFile` block:

```
var f1: File = null;  withFile(n) { f => f1 = f }
val f1: File =        withFile(n) { f => f }
```

The file may also escape indirectly, through a closure:

```
val print: String=>Unit = withFile(n) { f => (s => f.print(s)) }
```

In addition, a programmer might call other functions from within `withFile` which are unaware of the protocol, and might attempt to store the `File` for later use.

Our solution is a type system extension that lets us define `file` as a *second-class* value, and that ensures that such second-class values will not escape their defining scope. We introduce an annotation `@local` to mark second-class values, and change the signature of `withFile` as follows:

```
def withFile[U](n: String)(@local fn: (@local File) => U): U
```

Now whatever handler is passed as callback `fn` has to be a function that expects a second-class, non-escaping, argument. Note that the callback function `fn` itself is also required to be second-class, so that it can close over other second-class values. This enables, for example, nesting calls to `withFile`.

Since function types like `(@local File) => U` are so common, we provide a shorter notation: `File -> U`:

```
def withFile[U](n: String)(@local fn: File -> U): U
```

Second-class values cannot be stored in mutable variables, they cannot be returned from functions, and they cannot be accessed by first-class (named or anonymous) functions through free variables. Therefore, our earlier problem cases, instead of failing at runtime, now produce compile-time errors:

```
var f1: File = null;  withFile(n) { f => f1 = f }       // error
val f1: File =        withFile(n) { f => f }            // error
val print    =        withFile(n) { f =>
                                    (s => f.print(s)) }  // error
```

2.1.2 Second-class composes

Can we still do anything useful with second-class values? Yes, we can pass them to other functions or methods that expect second-class arguments. For example:

```
val data = new Data { def dump(@local f: File): Unit = ... }
withFile("out.txt") { f => data.dump(f) }
```

Inside the `dump` method, the same second-class restrictions apply to the argument `f` as directly in a `withFile` block: `f` cannot be stored, captured, returned, or otherwise escape its scope.

In addition, functions with second-class *arguments* remain first-class values. This means that we can freely use patterns such as decorators, currying, or η-expansion, on them, as long as we do not capture any second-class arguments. For example, we can capture `data.dump` in a closure, and wrap it in some code that prints additional text:

```
def prettify(wrapped: File -> Unit): (File -> Unit) = { f =>
  f.print("BEGIN ["); wrapped(f); f.print("] END")
}
val pretty = prettify(data.dump)
```

Note that variable `f` will not be allowed to escape. The result of this transformation, `pretty`, is again a first-class function that expects a second-class `File` argument. We can safely store it wherever we like and use it at our convenience:

```
withFile("out") { f => pretty(f) }
```

Thus, by cleverly combining first- and second-class values, we obtain safety without giving up expressiveness.

2.1.3 Higher-order functions and second-class closures

We have seen above how second-class values cannot be captured by first-class closures. Does this rule out the following code, where a closure closing over `file` is passed to `map`?

```
withFile("out.txt") { file =>
  List("Hell", "o, ", "World!") map { x => file.print(x) }
}
```

Not necessarily. We can define `map` in class `List[T]` to take a second-class closure argument as follows:

```
class List[T] {
  def map[U](@local fn: T => U): List[U] =
    if (isEmpty) Nil else fn(head) :: tail.map(fn)
  ...
}
```

The key observation here is that `map` itself treats `fn` in a strictly second-class way. The above snippet type-checks because the closure closing over `file` type-checks as a second-class value, and second-class functions are allowed to refer to other second-class values through their free variables.

One might wonder: would the same work with a *lazy* collection such as `Stream` or `Iterator`?

Suppose we would like to print in a fashion that allows for truncation of long lines and counting printed characters. For that purpose, we define a function that returns an iterator whose `next()` method prints a chunk and return its length:

```
def printingIter(ss: String*)(@local f: File): Iterator[Int] =
  ss.iterator.map(s => { f.print(s); s.length })
```

It seems as though the following code might leak a file:

```
val chunkPrinter = withFile("out.txt") { file =>
  printingIter("Hell", "o, ", "World!")(file)
}
chunkPrinter.next()  // prints to a file (?)
```

Fortunately, this is impossible. Closing over a `File` argument in `printingIter` would require `Iterator`'s `map` parameter to be second-class, i.e.:

```
class Iterator[A] { self =>  // self is alias for this
  def next(): A = ...
  def map[B](@local fn: A => B) = new Iterator[B] {
    def next(): B = fn(self.next()) // error: 1st-class next()
    ...                             //  refers to 2nd-class fn
  }
}
```

Consequently, the `next` method which accesses the mapping function `fn` and in fact the whole `Iterator` object that is returned from `map` would also need to be second-class, which our type system disallows.

We discuss our modifications to the Scala Collections library to deal with second-class values in detail in Section 2.5.

2.1.4 Implicit capabilities as (co-)effects

In the code above, we have already regarded `File` objects as capabilities, guarding access to their associated functionality, including `print`. We can extend this model to other kinds of capabilities. Opening a file and creating a `File` object should perhaps be guarded by a general `CanIO` capability. Likewise, a second-class `throw` function or a `CanThrow` object can embody the capability to throw an exception:

```
def withFile[U](...)(implicit @local c: CanIO): U
def throw(e: Exception)(implicit @local c: CanThrow)
```

Using Scala's `implicit` parameters, such capabilities need not be passed explicitly. For a call like `throw(e)` to type-check, it suffices to have a `CanThrow` capability in scope.

More generally, second-class values as capabilities enable a radical new take on static effect checking: instead of making effects explicit in the *type* of an expression, the capabilities available in scope characterize the effects an operation can have. Thus, it is instructive to compare this approach with other methods of statically checking side effect behavior, such as monads or traditional type-effect systems [27].

Monads and effect systems encode computational properties in the type of an expression, on the right of the turnstile;

$$G \vdash e : \texttt{CanIO}[T] \qquad \texttt{(monad)}$$

$$G \vdash e : T\ \texttt{@canIO} \qquad \texttt{(effect type)},$$

whereas our `@local` annotations are *co-effects* [16, 133], encoded on the left of the turnstile:

$$G, (\texttt{@local}\ c : \mathrm{CanIO}), G' \vdash e : T.$$

This is a subtle but important detail. The major benefits are that the type of an expression remains standard and that it allows for easier encoding of fine-grained information. In particular, different capabilities, such as multiple open files, can be present in the environment without interference, and without picking an ordering:

```
def copyFile(@local src: File, @local dst: File): Unit = {
  dst.print(src.readAll())
}
```

In further comparison, monads offer additional power by abstracting over sequential composition through the *bind* operator. It is well known that monads essentially correspond to delimited continuations, and therefore easily encode patterns like nondeterminism, probabilistic evaluation, and so on. Our second-class values, by contrast, use the normal control flow of the existing language. Thus, continuations need to be provided as an additional language feature to achieve comparable functionality. Monads further encapsulate computation as first-class values. A similar effect can be achieved with second-class capabilities, by η-expanding expressions that require capabilities in the environment. A function `(@local CanIO) => T` can be seen as roughly equivalent to the monadic `CanIO[T]`.

2.1.5 Effect polymorphism

Second-class capabilities also provide an elegant solution to the *effect polymorphism* problem for higher-order functions such as `map`. By taking a second-class function argument, the given definition of `map` in `List[T]` is oblivious to what effect capabilities an actual argument closure uses. The effect (as in: required capabilities) of an expression `map(f => ...)` is exactly the effect of the function `(f => ...)`. By contrast, type-and-effect systems, such as Java's checked exceptions or monads in Haskell, require two implementations of `map`, one for pure and one for impure/-monadic function arguments.

That it could be possible to build general-purpose effect systems based on implicit capabilities has been suggested previously by Odersky [17]. We present the first instantiation of such a system, as a case-study on effect-tracking for checked exceptions in Section 2.6.

2.1.6 Unshareable (local) resources

In distributed programming systems like Apache Spark [134] higher-order functions on `RDD` objects (Resilient Distributed Datasets) are normally evaluated across

a cluster of machines. The first-class functions that are given as arguments to `map`, `reduce` and `foreach` are serialized and shipped across the network to each node that will take part in the computation. Problems occur when a function references non-serializable data. This may well happen indirectly:

```
val conn = connectDatabase("jdbc:mysql://...")
val rdd = loadData(...).map(...).reduce(...)
rdd.foreach { row =>  // store each row to database
  conn.execute("INSERT INTO ...", row)
}
```

Serializing the anonymous function will also need to serialize a closure passed to `foreach`, which includes a reference to `conn`, a database connection. However, an open connection is not something that can reliably be shipped to other machines for distributed computation. (Even if deserialization reopens a connection from each worker, there is usually a tight limit on the number of open connections. More importantly, the worker machine must not fail after the transaction is committed to ensure idempotency in case the operation is rescheduled.) The result will, therefore, be either a runtime exception, or an undefined behavior (if we were able to sensibly ship the connection).

Note that variations of the above scenario may also lead to hard-to-diagnose performance bugs; one such example would be atomically checking for already inserted rows or bailing on duplicates, while another would be replacing `conn` with a piece of shared mutable state or a large memory buffer. Either case is prone to a non-deterministic overhead caused by contention on a single *shared resource*, requiring transactions or locking to avoid race conditions.

How can we fix this? Instead of a runtime exception we would like to get an error at compile time. With this use case in mind, recent work has proposed *Spores* [135], closures that need to list their free variables explicitly and can impose certain type bounds such as serializability on them.

Our solution is to turn `conn` into a second-class value, by adding a `@local` annotation:

```
@local val conn = connectDatabase("jdbc:mysql://...")
```

With this modification, the closure would now need to be (coerced to) second-class (to avoid type error). Consequently, the type checking will fail because `RDD.foreach` expects a first-class function.

2.2 Formal development

We develop our theoretical foundation as an operational semantics for a λ-calculus with first- and second-class bindings and evaluation, along with a sound type system that enforces stack-based lifetimes for second-class bindings. Some key parts of the formalization, as well as mechanized Coq proofs, were developed by Grégory Essertel.

2.2.1 Dynamic semantics

We formalize our model as an extended λ-calculus $\lambda^{1/2}$, where first-class and second-class identifiers use different binding forms x^1 and x^2. These correspond to names without and with `@local` annotations from Section 2.1. The syntax, operational semantics, and type system for this $\lambda^{1/2}$ calculus is shown in Figure 2.1. The semantics is defined in big-step call-by-value style with explicit closures. We can think of evaluation as being split between two judgments $H \vdash t \Downarrow^1 v$ and $H \vdash t \Downarrow^2 v$ for first-class and second-class evaluation, respectively, or as one parameterized judgment $H \vdash t \Downarrow^n v$. An auxiliary definition $H^{[\leq n]}$ restricts H to bindings of names x^m with $m \leq n$. For identifiers, first-class evaluation requires a first-class identifier (Evar). For abstractions, first-class evaluation removes all second-class identifiers from the environment that is to be stored in the closure, rendering them inaccessible (Eabs). For applications, the function itself is evaluated second-class, the function body is always evaluated first-class, and for the argument, it depends on whether the formal parameter is a first-class or second-class symbol (Eapp). These evaluation rules formalize the key ideas stated earlier for combining first-class and second-class values in the same language.

2.2.2 Mechanized implementation

To prove various properties of our system, we have mechanized it in Coq. For this implementation, we had to pick a representation of bindings and environments. We chose a representation based on DeBrujin levels, where names are numeric indexes into the environment, from outermost to innermost. In this setting, we assume that all names x in the program are denoted by x^1 or x^2. This structure is canonical taking the environment bindings as a well-formedness condition. To model the two kinds of bindings for x^1 and x^2, as well as the restriction operator $H^{[\leq n]}$, we found it useful to implement environments as triple $H = (H^1, H^2, k)$, where H^1 holds the x^1 bindings, H^2 holds the x^2 bindings, and k is a lower bound on the accessible bindings in H^2. The last bit deserves some further explanation. We can picture an environment H as

$$H = \{v_1^1, \ldots, v_m^1\}, \{\underbrace{v_1^2, \ldots, v_{k-1}^2}_{\text{inaccessible}} \mid v_k^2, \ldots, v_n^2\}$$

where the vertical bar $\mid$ is at position k in the list of x^2 bindings, denoting that only bindings that are to the right of it, i.e., for names represented by DeBrujin levels $\geq k$ are valid indexes. Restricting H to $H^{[\leq 1]}$ moves the bar k all the way to the right, disabling all existing second-class bindings:

$$H^{[\leq 1]} = \{v_1^1, \ldots, v_m^1\}, \{\underbrace{v_1^2, \ldots, v_{k-1}^2, v_k^2, \ldots, v_n^2}_{\text{inaccessible}} \mid\}$$

However, new second-class bindings can be added to the right. A restriction $H^{[\leq 2]}$ leaves the environment unchanged.

This representation, which preserves the structure of environments, considerably simplifies the proofs, as we do not need to worry about substitution or reasoning about sets of names. A variation would be to use DeBrujin indexes, i.e., to index environments from the right instead of the left. This removes the need for a numeric bound k at this point, at the expense of complicating developments for type systems with abstract types, which require shifting of indexes when moving type variables across contexts.

To prove properties about evaluation, such as type soundness, we follow the technique of Siek [136] and Ernst, Ostermann and Cook [137], which consists in extending a big-step operational semantics $\Downarrow$ to a total evaluation function `eval` by adding a numeric fuel value and explicit `Timeout` and `Error` results:

$$r ::= \text{Timeout} \mid \text{Done}\ (\text{Error} \mid \text{Val}\ v)$$

The fuel value can serve as induction measure.

2.2.3 Lifetime properties

Based on this high-level semantics, which is just an annotated simply-typed λ-calculus, we prove that second-class values exhibit the expected second-class characteristics. In particular, we show that the lifetimes of second-class values follow a stack discipline. To do this, we define a lower-level operational semantics $H, S_1..S_k \vdash t \Downarrow_s^n v$, shown in Figure 2.2, that again splits environments into first-class and second-class parts, but in addition maintains a stack of second-class environments through all function calls. Closures contain a first-class environment but only a stack pointer to represent the second-class part. When invoking a closure, the stack pointer will be used to find the correct caller environment S_i in which to resolve the callee's free second-class variables. This S_i will become the new top stack frame. If the stack pointer is 0, as is the case for first-class functions, the empty environment will be used. Function arguments will be either added to the environment (first-class) or to the top stack frame (second-class).

We define a predicate wf^n to define well-formedness of values v and classify them as first- or second-class value. An environment can be first or second-class, only if all elements are well-formed first- or second-class values, respectively. Well-formed first-class values include exactly the constants c and closures with no second-class references: $wf^1\ c$ and if $wf^1\ H$, then $wf^1\ \langle H, 0, \lambda x^n.t\rangle$. Well-formed second-class values are all well-formed values, since first-class values are also second-class. The abstractions need to have a first-class environment heap reference: $wf^1\ H$, then $wf^2\ \langle H, i, \lambda x^n.t\rangle$.

Lemma 2.2.1 *Evaluation produces only well-formed values:*

$$\frac{wf^{\,1}\ H \qquad wf^{\,2}\ S_1..S_k \qquad H, S_1..S_k \vdash t \Downarrow^n v}{wf^{\,n}\ v}$$

Proof By induction on the derivation. ■

This result establishes that first-class evaluation can only yield values that contain no stack references. The interesting case in the proof is in (Eapp1), when H is extended with a new binding. We know by induction that the new value is well-formed, too. Thus, we can establish the following stronger result.

Theorem 2.2.2 *Evaluation never leaks stack references: If $wf^{\,1}\ H$, then for all H' encountered in a derivation of $H, S_1..S_k \vdash t \Downarrow^n v$, we have $wf^{\,1}\ H'$.*

Proof By induction on the derivation, and Lemma 2.2.1. ■

We now define equivalence relations $\sim$ between values and environments from $\lambda^{1/2}$ and $\lambda_s^{1/2}$, respectively. In order to make the notation clearer, the environment of $\lambda^{1/2}$ will be explicitly (H^1, H^2) and the closures $\langle H^1, H^2, \lambda x^n.t\rangle$. For $\lambda_s^{1/2}$, closures take the shape $\langle H, i, \lambda x^n.t\rangle$. Equivalence between values is with respect to a stack $S_1..S_k$. The key case for closures looks up the correct stack frame given the stack pointer:

$$\frac{S_1..S_i..S_k \vdash (H^1, H^2) \sim (H, S_i)}{S_1..S_i..S_k \vdash \langle H^1, H^2, \lambda x^n.t\rangle \sim \langle H, i, \lambda x^n.t\rangle}$$

With these correspondences at hand, we can show that the total formulations of the high-level semantics $\Downarrow^n$ and low-level semantics $\Downarrow_s^n$, $\texttt{eval}^n$ and $\texttt{eval}_s^n$, are equivalent.

Theorem 2.2.3 *The fully environment-based and (second-class) stack-based semantics are equivalent. For all k,*

$$\frac{S_1..S_k \vdash (H^1, H^2) \sim (H, S_k)}{S_1..S_k \vdash eval^{\,n} k\ (H^1, H^2)\ t \sim eval^{\,n}_{\,s} k\ (H, S_1..S_k)\ t}$$

Proof By induction on the fuel value k. ■

Using $\texttt{eval}^n$ and $\texttt{eval}_s^n$ instead of $\Downarrow^n$ and $\Downarrow_s^n$ in the proofs yields a result that includes equivalent error and divergence behavior. Importantly, the result holds for empty environments, as $(\emptyset, \emptyset) \sim (\emptyset, \emptyset)$.

Corollary 2.2.4 *The lifetimes of second-class bindings in $\lambda^{1/2}$ follow a stack discipline.*

From this result follows that a realistic implementation can use the more efficient stack-based semantics as a basis, and also that second-class values can be used as temporary access tokens.

2.2.4 Type system and static checking

Having defined the correct desired runtime behavior, we would like to be able to rule out erroneous executions statically. To this end, we define a type system for $\lambda^{1/2}$, shown in Figure 2.1, and prove it sound with respect to the given operational semantics. The syntax of types contains a function type $T_1^n \to T_2$ where n distinguishes second-class and first-class parameters, respectively.

Type assignment aims to mirror the operational semantics. Again the rules can be read as two judgments, $G \vdash t :^1 T$ and $G \vdash t :^2 T$ for first-class and second-class type assignment, or as one parameterized judgment $G \vdash t :^n T$. For identifiers, first-class typing requires a first-class identifier (Tvar). For abstractions, first-class typing removes all second-class identifiers from the environment and all function bodies are treated as first-class (Tabs). For applications, the function itself is second-class, and the formal parameter type decides the type assignment of the argument (Tapp).

For the proof of type soundness, we follow the technique of Siek [136]. We need straightforward auxiliary judgments $v :^n T$ that assign types to runtime values and $G \vDash H$ that establishes consistency between type and value environments.

Theorem 2.2.5 *The type system is sound with respect to the operational semantics: for all k, if* `eval` *does not time out, its result is also not stuck, and the result is well typed.*

$$\frac{G \vdash t :^n T \qquad G \vDash H \qquad eval\ ^n k\ H\ t = Done\ r}{r = Val\ v \qquad v :^n T}$$

Proof By induction on the fuel value k, and case analysis on the term t, using helper lemmas to establish soundness of environment lookup. ■

This result implies that "well-typed programs don't go wrong", i.e., that all run-time failures are transformed into compile errors. This includes failures caused by trying to access second-class values that have been removed from an environment via a $H^{[\leq n]}$ operation.

Corollary 2.2.6 *All well-typed programs are guaranteed to respect stack-based lifetimes for second-class values.*

This basic model based on simply-typed λ-calculus captures the essence of combining first- and second-class values in a single language, and it already enables us to write interesting programs with second-class capabilities. The motivating examples from Section 2.1 are almost entirely expressible with just the λ-calculus fragment, except for some simple uses of parametric types, and of course assuming that we access to the filesystem. However, we can gain additional expressiveness by moving to richer type systems, as we motivate and formalize next.

2.3 Extension to richer types

We now move beyond simply-typed λ-calculus as a base calculus. Our motivation is twofold. First, we would like to gain confidence that our model scales to realistic languages, in particular Scala, since this is the testbed for our case studies. Second, we show that specific features, such as subtyping and path-dependent types, enable interesting programming patterns with second-class capabilities.

Parametric polymorphism In a realistic language, we clearly want some form of parametric polymorphism to support generic data structures, and we could base our model on System F instead of λ-calculus without much difficulty. For second-class capabilities, there are also many specific use cases: for example, an exception throwing capability `CanThrow` can be refined to designate specific kinds of exceptions it enables to throw by using `CanThrow[IOException]`, `CanThrow[NullPointerException]`, and so on.

Subtyping Subtyping is specifically useful to create hierarchies of capabilities, some more general than others. For example, instead of a simple `CanIO` capability, we can envision a hierarchy as follows:

```
type CanIO                // unspecific IO
type CanDisk   <: CanIO   // local filesystem
type CanNet    <: CanIO   // network send/receive
type CanHadoop <: CanNet  // remote filesystem
```

Using advanced language features like mixin-composition, reflected as intersection types on the type level, we can create and request capabilities like `CanDisk & CanHadoop` that enable sets of functionality as a whole, and specific capabilities can be masked via up-casts; for example, treating a `CanDisk & CanHadoop` capability as its supertype `CanNet`.

Path-dependent types In Section 2.1, we have used second-class `File` objects directly as capabilities. Sometimes this is undesirable, for example, when only parts of the functionality of `File` objects should be guarded by a capability. For those cases, we can use *path-dependent types* to associate an external capability with a *specific* file object, and require this capability only for some of the operations:

```
class File(val path: String) {
  type Cap
  def read(implicit @local c: Cap): String = ...
}
```

Each `File` object now has an abstract type member `Cap`, and reading the file requires a second-class capability of that type. The `File`'s path, by contrast, can be used

freely without accessing the filesystem, and extracting it hence does not require the file to be opened.

Method `withFile` now introduces both the file, which is first-class, and the implicit capability `c`, which is second-class and has type `file.Cap`, i.e., a path-dependent type referencing a *specific* file object. Here is a possible usage scenario:

```
val usedFiles = new ArrayBuffer[File]()
withFile("out.txt") { file => implicit c =>
  usedFiles += file
  ... file.read() ... // ok, capability available
}
println("this program used the following files:")
for (f <- usedFiles)
  println(f.path)
```

This means that we can freely let the file object escape, knowing that we will not be able to read from it outside of a `withFile` scope without the capability. We make key use of a similar model in our case study on region-based memory (Section 2.7) and for checked exceptions in the presence of parallel collections (Section 2.6).

2.3.1 Formal model

We have shown why we want richer type systems than λ-calculus as our base. We could extend System F for parametric polymorphism alone, or $F_{<:}$ for parametric polymorphism plus subtyping. But in order to cover all the features we want, including path-dependent types, we base our exposition on the DOT (Dependent Object Types) calculus [4, 138, 139], that has been proposed as a foundation for Scala's type system. More precisely, we use a slightly restricted variant of DOT called $D_{<:}$ [139], which encodes $F_{<:}$ in a relatively straightforward way, and which we extend to $D^{1/2}_{<:}$.

System $D_{<:}$ is at its core a system of first-class type objects and path-dependent types. Type objects can be seen as single-field records containing an abstract type member. Type selections, or path-dependent types serve to access these abstract type members.

The syntax and typing rules are shown in Figure 2.3. The type language includes $\bot$ and $\top$, as least and greatest element of the subtyping relation, first-class abstract types (Type $T_1..T_2$), lower-bounded by T_1 and upper bounded by T_2, type selections on a variable x.Type (i.e., path-dependent types), where x is a term variable bound to a type object, and finally dependent function types $(x^n : T) \to T$. The term language includes variables x, creation of type objects (Type T), λ-abstractions $\lambda x^n.t$, and applications $t_1\ t_2$.

The subtyping relation can compare type selections with the bounds of the underlying abstract types, and compare type objects and dependent functions, respectively. Type assignment contains standard cases for dependent abstraction and application.

To relate System $D_{<:}$ to Scala, let us take a step back and consider two ways to define a standard `List` data type:

```
class List[E]              // parametric, functional style
class List { type E }      // modular style, with type member
```

The first one is the standard parametric version. The second one defines the element type `E` as a type member, which can be referenced using a path-dependent type. To see the difference in use, here are the two respective signatures of a standard `map` function:

```
def map[E,T](xs: List[E])(fn: E => T): List[T] = ...
def map[T](xs: List)(fn: xs.E => T): List & { type E = T } = ...
```

Again, the first one is the standard parametric version. The second one uses the path-dependent type `xs.E` to denote the element type of the particular list `xs` passed as argument, and uses a refined type `List & { type E = T }` to define the result of `map`.

It is easy to see how the modular surface syntax directly maps to the formal $D_{<:}$ syntax, if we express fully abstract types `{ type E }` as (Type $\bot..\top$) and concrete type aliases `{ type E=T }` as (Type $T..T$). It is also important to note that the modular style with first-class type objects can directly encode the functional style, which corresponds to bounded parametric polymorphism as in System $F_{<:}$, but with increased expressiveness due to the $\bot$ type and potential lower bounds on type variables.

First-class and second-class values Since the stratification between first- and second-class values happens on the level of identifiers and bindings, not types, parametric polymorphism does not pose major difficulties. Still, moving to a system based on subtyping requires an additional result:

Lemma 2.3.1 *First-class values can be treated as second-class values:*

$$\frac{H \vdash t \Downarrow^n v \qquad n \leq m}{H \vdash t \Downarrow^m v} \qquad\qquad \frac{G \vdash t :^n T \qquad n \leq m}{G \vdash t :^m T}$$

Proof By induction over the respective derivations, showing that the evaluation and type assignment rules for second-class values subsume those for first-class values. ■

This result entails that one can admit coercions from first-class to second-class values, and thus eta-expand t of type $T_1^2 \to T_2$ to $\lambda x^1.t\ x^1$ of type $T_1^1 \to T_2$. Thus, we can define a subtyping relation that justifies $T_1^2 \to T_2 <: T_1^1 \to T_2$.

The operational semantics for $\mathrm{D}_{<:}^{1/2}$ is the same as for $\lambda^{1/2}$, with an additional rule for construction of type values:

$$H \vdash \text{Type } T \Downarrow^n \langle H, \text{Type } T \rangle$$

We can prove type soundness using the same overall technique as for $\lambda^{1/2}$. The proof follows the one given for $\mathrm{D}_{<:}$ by Rompf et al. [139].

Theorem 2.3.2 *Type soundness for $D_{<:}^{1/2}$. If* `eval` *does not time out, it returns a well-typed value:*

$$\frac{\Gamma \vdash t :^n T \qquad \Gamma \vDash H \qquad \mathit{eval}^{\,n} k\ H\ t = \mathit{Done}\ r}{r = \mathit{Val}\ v \qquad H \vdash v :^n T}$$

Proof By induction on the fuel value k. ■

Syntax

$$
\begin{array}{lcll}
n & ::= & 1 \mid 2 & \text{1st/2nd class} \\
t & ::= & c \mid x^n \mid \lambda x^n.t \mid t\ t & \text{Terms} \\
v & ::= & c \mid \langle H, \lambda x^n.t \rangle & \text{Values} \\
T & ::= & B \mid T_1^n \to T_2 & \text{Types} \\
\\
G & ::= & \emptyset \mid G, x^n : T & \text{Type Envs} \\
H & ::= & \emptyset \mid H, x^n : v & \text{Value Envs}
\end{array}
$$

$$G/H^{[\leq n]} = \{x^m : \_ \in G/H \mid m \leq n\}$$

Operational Semantics $\boxed{H \vdash t \Downarrow^n v}$

$$H \vdash c \Downarrow^n c \qquad \text{(Ecst)}$$

$$\frac{x^m : v \in H^{[\leq n]}}{H \vdash x \Downarrow^n v} \qquad \text{(Evar)}$$

$$H \vdash \lambda x^m.t \ \Downarrow^n \ H^{[\leq n]}, \lambda x^m.t \qquad \text{(Eabs)}$$

$$\frac{H \vdash t_1 \Downarrow^2 \langle H', \lambda x^m.t_3 \rangle \qquad H \vdash t_2 \Downarrow^m v_2 \qquad H', x^m : v_2 \vdash t_3 \Downarrow^1 v_3}{H \vdash t_1\ t_2 \Downarrow^n v_3} \qquad \text{(Eapp)}$$

Type System $\boxed{G \vdash t :^n T}$

$$G \vdash c :^n B \qquad \text{(Tcst)}$$

$$\frac{x^m : T \in G^{[\leq n]}}{G \vdash x :^n T} \qquad \text{(Tvar)}$$

$$\frac{G^{[\leq n]}, x^m : T_1 \vdash t :^1 T_2}{G \vdash \lambda x^m.t :^n T_1^m \to T_2} \qquad \text{(Tabs)}$$

$$\frac{G \vdash t_1 :^2 T_1^m \to T_2 \qquad G \vdash t_2 :^m T_1}{G \vdash t_1\ t_2 :^n T_2} \qquad \text{(Tapp)}$$

Figure 2.1. $\lambda^{1/2}$: syntax, operational semantics, and type system.

Syntax

$$
\begin{array}{lcll}
n & ::= & 1 \mid 2 & \text{1st/2nd class} \\
t & ::= & c \mid x^n \mid \lambda x^n.t \mid t\ t & \text{Terms} \\
v & ::= & c \mid \langle H, k, \lambda x^n.t \rangle & \text{Values} \\
H & ::= & \emptyset \mid H, x^1 : v & \text{Value Envs} \\
S & ::= & \emptyset \mid S, x^2 : v & \text{Stack Frames}
\end{array}
$$

Operational Semantics $\boxed{H, S_1..S_k \vdash t \Downarrow_s^n v}$

$$H, S_1..S_k \vdash c \Downarrow_s^n c \qquad \text{(Ecst)}$$

$$\frac{x^1 : v \in H, \emptyset}{H, S_1..S_k \vdash x^1 \Downarrow_s^1 v} \qquad \text{(Evar1)}$$

$$\frac{x^m : v \in H, S_k}{H, S_1..S_k \vdash x^m \Downarrow_s^2 v} \qquad \text{(Evar2)}$$

$$H, S_1..S_k \vdash \lambda x^m.t \ \Downarrow_s^1 \langle H, 0, \lambda x^m.t \rangle \qquad \text{(Eabs1)}$$

$$H, S_1..S_k \vdash \lambda x^m.t \ \Downarrow_s^2 \langle H, k, \lambda x^m.t \rangle \qquad \text{(Eabs2)}$$

$$\frac{\begin{array}{c} H, S_1..S_i..S_k \vdash t_1 \Downarrow_s^2 \ H', i, \lambda x^1.t_3 \\ H, S_1..S_i..S_k \vdash t_2 \Downarrow_s^1 v_2 \\ (H', x^1 : v_2), S_1..S_i..S_k, S_i \vdash t_3 \Downarrow_s^1 v_3 \end{array}}{H, S_1..S_k \vdash t_1\ t_2 \Downarrow_s^n v_3} \qquad \text{(Eapp1)}$$

$$\frac{\begin{array}{c} H, S_1..S_i..S_k \vdash t_1 \Downarrow_s^2 \ H', i, \lambda x^2.t_3 \\ H, S_1..S_i..S_k \vdash t_2 \Downarrow_s^2 v_2 \\ H', S_1..S_i..S_k, (S_i, x^2 : v_2) \vdash t_3 \Downarrow_s^1 v_3 \end{array}}{H, S_1..S_k \vdash t_1\ t_2 \Downarrow_s^n v_3} \qquad \text{(Eapp2)}$$

Figure 2.2. $\lambda_s^{1/2}$: syntax and operational semantics.

Syntax

$$
\begin{array}{lcl}
T & ::= & \bot \mid \top \mid \text{Type } T..T \mid x.\text{Type} \mid (x^n : T) \to T \\
t & ::= & x \mid \text{Type } T \mid \lambda x^n.t \mid t\ t \\
\Gamma & ::= & \emptyset \mid \Gamma, x^n : T \\
v & ::= & \langle H, \lambda x^n : T.t \rangle \mid \langle H, \text{Type } T \rangle
\end{array}
$$

Subtyping $\boxed{\Gamma \vdash S <: U}$

$$\Gamma \vdash \bot <: T \quad (\textsc{Sbot}) \qquad \Gamma \vdash T <: \top \quad (\textsc{Stop})$$

$$\frac{\Gamma \vdash x : \text{Type } T..\top}{\Gamma \vdash T <: x.\text{Type}} \quad (\textsc{Ssel1}) \qquad \frac{\Gamma \vdash x : \text{Type } \bot..T}{\Gamma \vdash x.\text{Type} <: T} \quad (\textsc{Ssel2})$$

$$\Gamma \vdash x.\text{Type} <: x.\text{Type} \quad (\textsc{SselX}) \qquad \frac{\Gamma \vdash S_2 <: S_1\ ,\ U_1 <: U_2}{\Gamma \vdash \text{Type } S_1..U_1 <: \text{Type } S_2..U_2} \quad (\textsc{SselaX})$$

$$\frac{m_2 \leq m_1 \qquad \Gamma \vdash S_2 <: S_1 \qquad \Gamma, x : S_2 \vdash U_1 <: U_2}{\Gamma \vdash (x^{m_1} : S_1) \to U_1 <: (x^{m_2} : S_2) \to U_2} \quad (\textsc{Sall})$$

Type assignment $\boxed{\Gamma \vdash t : T}$

$$\frac{x^m : T \in \Gamma^{[\leq n]}}{\Gamma \vdash x :^n T} \quad (\textsc{Tvar}) \qquad \Gamma \vdash \text{Type } T :^n \text{Type } T..T \quad (\textsc{Ttyp})$$

$$\frac{\Gamma^{[\leq n]}, x^m : T_1 \vdash t_2 :^1 T_2}{\Gamma \vdash \lambda x^m.t_2 :^n (x^m : T_1) \to T_2} \quad (\textsc{Tabs}) \qquad \frac{\Gamma \vdash t :^2 (x^m : T_1) \to T_2\ ,\ y :^m T_1}{\Gamma \vdash t\ y :^n T_2[y/x]} \quad (\textsc{Tdapp})$$

$$\frac{\Gamma \vdash t :^2 (x^m : T_1) \to T_2\ ,\ t_2 :^m T_1}{\Gamma \vdash t\ t_2 :^n T_2} \quad (\textsc{Tapp}) \qquad \frac{\Gamma \vdash t :^n T_1\ ,\ T_1 <: T_2}{\Gamma \vdash t :^n T_2} \quad (\textsc{Tsub})$$

(runtime sybtyping and value type assignment not shown)

Figure 2.3. System $D^{1/2}_{<:}$: a generalization of $F_{<:}$ with value types and path-dependent types.

2.3.2 Arbitrary privilege lattice

The model presented so far enables us to control the lifetimes of capabilities, but in many settings, not all capabilities have the same status. What if we want to have a more control over the relative visibilities of capabilities, while ensuring their non-escaping status as non-first-class values? Suppose we want to prevent race conditions or out-of-order writes when a file is passed to a non-deterministic higher-order function such as a parallel `reduce` operation, yet allow non-deterministic reads, which are far less dangerous:

```
withFile("file.txt") { f =>
  f.readCharAt(0)       // ok
  f.print(...)          // ok: deterministic context
  reduce(data) { (a,b) =>
    f.readCharAt(a)     // ok
    f.print(...)        // error: race condition
    a+b
} }
```

To model such scenarios, we need to treat capabilities for reading and writing differently. We informally introduce a degree of "second classiness", which we achieve by parameterizing `@local` as `@local[P]`, where `P` denotes a *privilege level* and is in *contravariant* position. Implicitly, a `@local` annotation denotes the most restricted privilege level, while its absence denotes no restrictions (first class). In general, annotating a function parameter with `@local[P]` requires each free reference of a passed closure to be annotated with `@local[T]`, for some `T <: P`. In Scala, we can represent privileges directly as types, and their relationships via subtyping: `@local[Nothing]` denotes first-class, equivalent to no annotation, and `@local[Any]` denotes second-class, equivalent to just `@local`, and any other type `P` defines an in-between level.

We now exploit this mechanics to implement the example above. The key is that files themselves will live at a less restricted (i.e. smaller) level than write capabilities:

```
trait R // privilege level >: Nothing (1st) and <: Any (2nd)
class File(val path: String) {
  def print(s: String)(implicit @local w: CanWrite) { ... }
  def readCharAt(i: Int) = { ... }
}
def withFile[U](...)(@local fn: (@local[R] File) => U): U
def reduce[U](...)(@local[R] fn: (U,U) => U)
```

We introduce a privilege level `R` between first- and second-class and implement `withFile` to make file objects available at this new level. In the simplest model, files serve as their own read capabilities, but the `print` method requires an additional *second-class* `CanWrite` capability.

Method `reduce` takes its function argument as `@local[R]`, so files can be accessed from the closure, but truly second-class objects and in particular write capabilities will be precluded. A single global `CanWrite` capability is all that is left to complete the example.

As an alternative, we can model read and write capabilities specific to a given file as path-dependent types, extending the example from the beginning of Section 2.3:

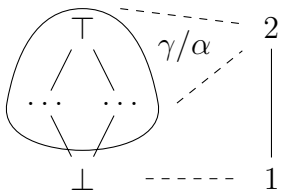
```
class File(val path: String) {            // path-dependent
  type CapW <: CanIO; type CapR <: CanIO  // capabilities
  def print(s: String)(implicit @local w: CapW) { ... }
  def readCharAt(i: Int)(implicit @local[R] r: CapR) = ... }
}
```

In this model, the definition of `withFile` needs to introduce both the `CapR` and the `CapW` objects as separate "fractional" [140] capabilities, with different privilege levels:

```
withFile(path) { f => implicit cr => implicit cw => ... }
```

One could go further and require unequal privilege for sequential reads or random-access writes, thus extending the privilege lattice to more than three levels.

Formal model We generalize the binary first- vs second-class distinction to an arbitrary privilege lattice L. We require a Galois connection γ, α between L and the lattice $\{1, 2\}_{\leq}$, which maps $\top$ to 2 and $\bot$ to 1 via its concretization function γ. All values except $\top$ and $\bot$ can be mapped to either 1 or 2. In the limit, where everything except $\bot$ is mapped to 2, the previous second-class lifetime guarantees extend to all non-first-class bindings:



While picking specific static lattices may be of interest, the key application relies on a much more general insight: in a system with subtyping, we can use the underlying type lattice as privilege lattice.

In the case of $D_{<:}^{1/2}$ and similar systems, we can use the *types* $\bot$ and $\top$ to denote first- and second-class values, respectively. Any desired privilege lattice can be built within a program from phantom types that are in a corresponding subtyping relation. As already discussed, in Scala, we achieve this by parameterizing `@local` as `@local[P]`, where `@local[Nothing]` denotes first-class, equivalent to no annotation, and `@local[Any]` denotes second-class, equivalent to just `@local`. Any other Scala type `P` must be between `Nothing` $= \bot$ and `Any` $= \top$, and gives rise to a more fine-grained lattice structure, subject to existing subtyping relations between `T` and other types.

To make this change explicit in the context of the formal model in Figure 2.3, interpret all m as types and replace all occurrences of $m_1 \leq m_2$ with $m_1 <: m_2$.

Privilege parametricity It is sometimes desirable to abstract over the *level* of privilege in order to prevent code duplication and keep an existing interface unmodified. If a type system includes abstract types, as is the case in $D_{<:}^{1/2}$ and in Scala, abstract types naturally enable such *privilege parametricity*. This means that we can abstract over whether a variable holds first-class or second-class values in a more specific context. The main motivation here is code reuse: we need to write a function or class only once, and we can use it with both first-class and second-class instantiations.

A key use case comes from our handling of the Scala collection library in Section 2.5. We have already mentioned that method `map` should behave differently for eager and lazy collections:

```
@local def println(x: Any): Unit = ...
list.map(x => println(x))   // ok
stream.map(x => println(x)) // error
```

Thus, these collection implementations need to have different signatures for `map`:

```
def map[B](@local fn: A => B) = ... // eager
def map[B](fn: A => B) = ...        // lazy
```

Lazy collections like `Stream[A]` may leak the closure argument to `map`, and therefore it needs to be first-class. Conversely, for eager collections like `List[A]`, we would like a second-class closure argument.

How can we achieve that `List[A]` and `Stream[A]` can be derived from a common superclass? We use `@local[LT]` in the generic `map` signature, where `LT` is an abstract type parameter defined in base class `Iterable[A]`, and refined to `Nothing` or `Any` (first- or second-class) for specific subclasses:

```
// Abstract base class:
trait Iterable[A] {
  type LT
  type plocal = local[LT]
  def map[B](@plocal fn: A => B)
}
// Implementation classes:
class List[A] extends Iterable[A] {
  type LT = Any
  def map[B](@local fn: A => B) = {
    // implement eager version here
} }
class Stream[A] extends MySeq[A] {
  type LT = Nothing
  def map[B](fn: A => B) = {
    // implement lazy version here
} }
```

This design enables the desired usage patterns shown above.

As we can see, abstract base classes can have abstract privileges that are instantiated to second- or first-class in implementation subclasses. In Section 2.5, we will discuss code sharing between collections further and demonstrate that we can indeed share large pieces of the internal implementation in our modified version of the Scala library.

2.3.3 Recursive functions

Our development so far did not consider recursive functions. Adding recursion does not pose particular difficulties. The simplest and most practical implementation of recursive functions extends rule (Eapp) from Figure 2.1 to pass the closure object

itself as argument to the function. The λ syntax is extended to include the self identifier f^k where k denotes first- or second-class binding as usual:

$$\frac{\begin{array}{c} H \vdash t_1 \Downarrow^k v_1 \\ v_1 = \ H', \lambda f^k(x^m).t_3 \\ H \vdash t_2 \Downarrow^m v_2 \\ H', f^k : v_1, x^m : v_2 \vdash t_3 \Downarrow^1 v_3 \end{array}}{H \vdash t_1\ t_2 \Downarrow^n v_3} \quad \text{(EAPP)}$$

Note that this modified (Eapp) rule is no longer deterministic, as the evaluation rule for the function needs to match the class of the closure type. A simple way to make the rule deterministic in the formalism is to extend the syntax of function application to determine if the function is first- or second-class: $t_1^k\ t_2$.

For a realistic implementation, this piece of information can easily be extracted from the type assigned to expression t_1. In this setting, recursive functions are also related to the treatment of objects and `this` pointers, as we will discuss.

2.4 Implementation in Scala

We have implemented a plug-in[1] for the Scala compiler that closely implements the formal system described in Section 2.2 and Section 2.3. Given the nature of the Scala language, and the structure of the Scala compiler, a number of aspects needed additional work. First, Scala is a large language with many constructs in addition to λ-calculus and $D_{<:}$. In particular, objects, classes, traits, and separate compilation posed some challenges. Second, the Scala compiler is structured around a global, hierarchical symbol table as opposed to flat environments, so the formal model of removing certain bindings required different implementation techniques, e.g., traversing scope chains to find common ancestors.

To implement the API introduced in Section 2.1, we define a class `local` as a piece of library code, which the compiler plug-in knows about:

[1] `https://github.com/TiarkRompf/scala-escape`

```
package scala.util
class local[-T] extends StaticAnnotation
```

This class can be used as annotation on declarations:

```
@local val log = new File("out.txt")
```

Since the type parameter `T` is contravariant, writing `@local` is equivalent to `@local[Any]`, which denotes a second-class binding. By contrast `@local[Nothing]` denotes a first-class binding, equivalent to no annotation at all. Any type between `Nothing` and `Any` can be used for finer-grained control, and an abstract type can be used to abstract over the class of binding (Section 2.3.2).

Scala's first-class functions map to anonymous classes that implement a given base trait `Function1`, with the usual `A=>B` notation as type alias:

```
trait Function1[-A,+B] {
  def apply(x:A): B
}
type '=>'[-A,+B] = Function1[A,B]
```

To model functions with second-class arguments, we provide a subtrait `FunctionEsc1`:

```
trait FunctionEsc1[-A,+B,-LA,+LS] extends Function1[A,B] {
  @local[LS] def apply(@local[LA] x:A): B
}
type '->'[-A,+B] = FunctionEsc1[A,B,Any,Nothing]
```

If `A->B` is the expected type for some closure expression `(x => ...)`, the Scala compiler will automatically synthesize a corresponding object creation with the right signature.

Compared to the theoretical model, we need to worry about objects, traits, and classes in addition to lexical functions. These object-oriented constructs have a more complicated scope structure due to inheritance. Our current implementation is conservative and focuses primarily on the lexical level. Class definitions are treated like first-class functions and cannot access second-class values from their defining scope. The following code is thus illegal,

```
@local val log = ...
class Handler {
  def func() = log.println("A") // error
}
val a = new Handler; a.func()
```

but the same functionality can be implemented like this:

```
@local val log = ...
```

```
class Handler {
  def func(@local val log: File) = log.println("A")
}
val a = new Handler; a.func(log)
```

We plan to extend our implementation with a notion of `@local` classes, once all the implications are worked out. This would enable writing the same code snippet above as `@local class Handler`. In practice, we have not found the absence of such a facility limiting.

A key goal of this implementation was to investigate how well second-class values map to real world Scala code. To this end we conducted several case studies, described next.

2.5 Case study: Scala Collections

The cornerstone of the Scala standard library is its set of collection classes, supporting a variety of sequence data structures (`List`, `Array`, ...), as well as `Set`s, `Map`s and so on. Methods to traverse and transform collections use higher-order and first-class functions pervasively, making Scala Collections an excellent testbed to evaluate the expressiveness of our implementation of second-class values. The goal of this experiment is to assess how precisely we can model second-class behavior for functions passed as arguments. As described in Section 2.1, we would like a standard `List.map` call to treat its argument function in a second-class way, whereas a distributed or lazy collection would demand a true first-class function.

The key problem is that, for example, `List` is eager but `Stream` is lazy, and `Array` is sequential but `ParArray` is parallel. Yet, all the classes share the same base class hierarchy [141]. Most functionality is implemented only once, and reused among leaf classes. The Scala Collections library already has a large number of classes and traits (`GenTraversableOnce`, `IterableLike`, ...), so that adding another dimension to distinguish eager and lazy collections would not work well.

The solution we found makes crucial use of privilege parametricity. To handle lazy and eager collections in a uniform way, we use `@local[LT]`, where `LT` is an abstract

type parameter defined in a base class, that can be instantiated to `Nothing` or `Any` (first- or second-class) depending on the collection type. The corresponding code has been shown already in Section 2.3.2.

Note that method `foreach`, in contrast to `map` is eager for all collections. It uses `@local` directly instead of `@plocal`. Note further that we have omitted the return type of `map` above. In practice the situation is slightly more complicated, as transformer methods on collections use F-bounded polymorphism to return an instance of the same class (or a compatible one) as the object itself.

Evaluation We have achieved the abovementioned behavior without any code duplication or addition of new types, by changing <1%[2] of SLOC in the Scala Collections API, comprising 29310 SLOC total. Out of the 277 lines changed, over 75% are global search-replace that inserts `@local` annotations. The main challenge was to propagate the type-dependent type `LT` and deal with `*Proxy[Like]` traits (eventually removed as they are deprecated anyway). The modified code and scripts to reproduce the results are available as open-source[3].

2.6 Case study: Checked exceptions

Given our modified version of the Scala Collections library, whose higher-order traversal and transformer methods correctly track first-class and second-class arguments, we would like to put these facilities to some good use. We have already seen how we can model operations, like `println`, as second-class functions. These serve as capabilities and control when and where the associated operation and its side effect can happen. Thus, the question bears asking whether we can use the same model for more general classes of side effects.

We have extended the Scala Library further[3], with a notion of checked exceptions. Checked exceptions can be seen as an instance of a type-and-effect system [27], and in

[2]Only meaningful lines of code, i.e., not Scala docs, were counted.
[3]`https://github.com/losvald/scala/tree/esc`

fact, Java's support for checked exceptions is probably the only type-and-effect system in practical use today. The key idea is to include the side effects of an expression in its type. However, a fundamental trade-off between usefulness (larger, more precise types) and usability (smaller, more comprehensible types) makes such effect systems hard to use in practice.

In our case, exceptions might only be allowed to be thrown if an appropriate `throw` function is available, and we would like to enforce that this can only happen within a try/catch block. With our support for second-class values, we can define `try` blocks as follows:

```
def try[T](fn: (@local Exception => Nothing) => T): Option[T]
```

A realistic implementation would also contain a `catch` block, but here we content ourselves with returning `Option[T]` values. Given the definition of `fn`'s parameter as `local`, client code may use `try` as follows,

```
try { throw =>
  throw(new Exception)  // ok: throw cannot escape
}
```

but the function passed as argument to `try` cannot leak the value of `throw`. Inside such a `try` block we can use `throw` in other safe (i.e., second-class) positions but not in unsafe ones, where it could escape:

```
def safe(@local fn: () => Any): Int = ...
def unsafe(fn: () => Any): Int = ...
try { throw =>
  safe { () => throw(new Exception) }    // ok: safe
  unsafe { () => throw(new Exception) }  // not ok
}
```

Effect polymorphism It is easy to see that we have utilized the same pattern in `safe` as in the previous definition of `map` on `Lists`. In fact, the following code is perfectly legal:

```
try { throw =>
  map(xs) { x =>
    if (x > 0) x else throw(new Exception)
  }
}
```

As we would expect, we can use `throw` in nested second-class functions within the dynamic scope of `try` but not as a first-class value that might escape.

It is important to note that we are using the same `map` implementation independently of whether the function we are passing as argument may throw an exception or not. This would not be the case with monads or with Java's checked exceptions, where the following two different `map` declarations would be needed (example from Rytz [33]):

```
public <U> List<U> map(Function <T, U> f);
public <U, E extends Exception> List<U>
                    mapE(FunctionE<T, U, E> f) throws E;
```

Similar effect polymorphism can also be achieved in the context of type-and-effect systems but with significant effort [32, 33].

Implicit capabilities It is also worth noting that we do not have to use the object `throw` itself as a capability. We might as well define the `throw` method globally and have it require an additional argument of a designated capability type.

```
def throw(e: Exception)(implicit cap: CanThrow): Unit = ...
```

In fact, it has been proposed to use such a pattern for more flexible handling of side effects in general [17], for example:

```
def println(s: String)(implicit @local cap: CanIO): Unit = ...
```

As we will see below, this pattern is especially useful when the main object in question needs to be first-class for some other reason. In Scala, parameters declared as `implicit` will have the arguments resolved and inserted automatically by the compiler, so one can write

```
throw(new Exception)
```

and the Scala compiler would automatically insert `cap` as the missing capability argument for `throw` from the context.

In summary, scoping rules for second-class values ensure that such objects cannot be copied, stored, or escape by other means, which makes them ideally suited to serve as access tokens or capabilities. With effect capabilities as regular program values, specifying new classes of effects becomes almost trivial, an important benefit for expressive libraries and embedded DSLs (domain-specific languages).

Parallel collections A subtlety that arises from the inherently blocking nature of parallel operations has a rather unexpected implication with respect to effects. Since a blocking thread may be interrupted, it needs to handle an `InterruptedException`, which means that all parallel collection operations need the exception-throwing capability `CanThrow`. There are two choices: a pragmatic one, merely converting `InterruptedException` to `RuntimeException`; or the rigorous one, requiring a proper capability. We went with the latter, to investigate the effort of propagating exception capabilities, thus stress-testing our type system. To accommodate this without breaking the API, we exploit abstract types, type bounds and *implicit default arguments*:

```
type CanSeq  // non-parallel dummy capability
type CanPar <: CanSeq with CanThrow
trait GenIterable[A] {  // common super-trait
  type Cap >: CanPar
  def foreach[U](@local fn: A => U)(
      implicit @local cap: Cap)
}
trait Iterable[+A] extends GenIterable[A] {
  type Cap = CanSeq
  implicit val capDummy = new CanSeq {}
  override def foreach[U](@local fn: A => U)(
      implicit @local cap: CanSeq = capDummy) { ... }
}
trait ParIterable[+A] extends GenIterable[A] {
  type Cap = CanPar
  override def foreach[U](@local fn: A => U)(
  implicit @local cap: CanPar) { // note CanPar <: CanThrow
    ...
    doInterruptible(...)  // using cap as CanThrow
  }
}
```

The above implementation ensures that a (potentially) parallel method can only be called if the corresponding implicit `CanPar` capability is in scope, e.g.:

```
val coll: Iterable[Int] = ...
val collPar: ParIterable[Int] = ...
val collGen: GenIterable[Int] = collPar  // common base type
coll foreach { x => ... }     // ok (using default capDummy)
collPar foreach { x => ... }  // error: missing capability
collGen foreach { x => ... }  // error: could be parallel
```

Annotation overhead The default implicit arguments are essential, since they allow the compiler to insert `capDummy`s based on a scope of callee's (super)type rather

than leaving this burden at the call site. In the above case, putting capability arguments was the responsibility of non-parallel collections, rather than relying on callers to have them available in their scopes, which is fragile (prone to shadowing or ambiguity, and not resistant to passing other implicit arguments). For user functions we can alleviate this burden by providing an implicit dummy capability that can be imported as a first-class from a module. To show this eliminates overhead in dispatching capabilities, consider the following example:

```
def process[A](coll: GenIterable[A])(
  implicit @local cap: coll.Cap)
```

Note a path-dependent capability argument. It enables reuse of a *single* implementation for subtypes that require different levels of capabilities (forming a lattice), and subsumes optional capabilities. Our function works with both parallel and sequential collections, as the following snippet illustrates:

```
import CapDummy._
process(Range(0, 9))          // ok (using imported capDummy)
process(ParRange(0, 9))       // error (missing CanPar capability)
...
def parallelContext(implicit @local canPar: CanPar) {
  process(ParRange(0, 9))     // ok
}
```

Evaluation We modified the Scala compiler to signal all uses of checked exceptions according to the Java definition (excluding `Error`s and `RuntimeException`s) as compile errors, thus requiring the use of our `try` facility above. Additionally, `throw` markers were required for interfacing with Java methods, and finally the `no` unsafe hooks were used to comply to signatures of inherited Java methods.

We have evaluated the effort of using the above three facilities, as well as propagating our `CanThrow` (and `CanPar`) capabilities required for throwing exceptions, on the entire Scala standard library, comprising 43040 SLOC. Manual effort was due to the former and placing `Cap` type definitions in: a few Collection types (deep hierarchy) and many subtypes of mixins (shallow hierarchy). Adding capability parameters was largely automated (using a PERL-based regular expression engine), guided by compile errors. In total, $\sim$3% SLOC is affected, and the breakdown is as follows:

	try	throw	no	types	CanThrow	Cap
#	54	75	38	26	264	971

In the above effort breakdown, most `throw`s and `no`s come from code related to IO and processes (which exploits JVM). A high number of `try`s is due to a trade-off we needed to make to keep compatibility with user code; we could not require a capability in an `Any`'s core method such as `==` just because it might be comparable with a parallel collection.

2.7 Case study: Region-based memory

Most modern high-level languages run on managed runtimes such as the JVM, .NET CLR, or JavaScript VMs. All these platforms come with automatic memory management, garbage collection, and built-in memory safety. Sometimes it is, however, desirable to allocate memory outside the managed heap: to reduce garbage collection overhead, to address larger amounts of memory, or just to have more control over memory layout. Unfortunately, then the safety guarantees of the platform are invalidated and segfaults bound to happen.

We present a small off-heap memory library based on scoped capabilities that preserves memory safety by imposing a region-based object lifetime policy. Our implementation is inspired by a recent Scala library[4] by Shabalin et al. with much larger functionality, but without such guarantees.

Our implementation is based on two interfaces: `Data`, corresponding to an off-heap chunk of memory, and `Region`, from which such chunks can be allocated. We will discuss the role of the type parameter and the implicit arguments.

```
trait Data[T] {
  def size: Long
  def apply(i: Long)(implicit @local cc: T): Long
  def update(i: Long, x:Long)(implicit @local cc: T): Unit
}
trait Region {
  type Cap
  def alloc(n: Long)(implicit @local c: Cap): Data[Cap]
}
```

[4] `https://github.com/densh/scala-offheap`

The interface further provides a scoped method `withRegion` that can be used as follows:

```
withRegion[Long](1000) { region => implicit c =>
  val arr = region.alloc(300)  // type: Data[r.Cap]
  arr(0) = 1; println(arr(0))
  ...
}
```

The types ensure statically that data object `arr` cannot be used outside the scope of the `withRegion` call. Here is the implementation of `withRegion`:

```
abstract class F[B] { def apply(r: Region): r.Cap -> B }
def withRegion[T](n: Long)(f: F[T]): T = {
  object cap
  val r = new Region {
    type Cap = cap.type
    var data = malloc(n)
    var p = 0L
    def alloc(n: Long)(@local c: Cap) = new Data[Cap] {
      def size = n
      val addr = p
      p += n
      def apply(i: Long)(implicit @local c: Cap) =
        data((addr+i).toInt)
      def update(i: Long, x:Long)(implicit @local cc: Cap) =
        data((addr+i).toInt) = x
    }
  }
  try f(r)(cap) finally free(r.data)
}
```

For safety, all `Data` objects need to be guarded by their `Region`. On the other hand, we cannot mark the `Region @local`, because data objects actually need to store a reference to the region. The solution is to introduce external capabilities. The way `withRegion` is implemented, a region and its capability always obey the same scope.

As an extension, we might add bounds checking with the checked exceptions implementation from Section 2.6. Now, we need to use two scoped introduction forms:

```
withRegion[Long](1000) { r => c =>  try { throw =>  ... } }
```

Instead, we can just as well use the alternative form:

```
try { throw =>  withRegion[Long](1000) { r => c =>  ... } }
```

Region-based memory systems have also been proposed based on monads, phantom types, and rank-2 polymorphism [45]. These and other approaches based on (layered)

monads offer comparable guarantees, but they require users to rewrite their code in monadic style throughout, which has well-established shortcomings.

Systems that enforce a non-escaping property using rank-2 polymorphism do so by introducing additional type constraints, requiring the function passed to the `withRegion` equivalent to return a monad instance which is parameterized with the phantom type. By contrast, our `withRegion` blocks can return any type, and we just require capabilities to be present in the context.

Since types are flexible, we can independently define "checked" features like regions, exceptions, and IO, and use them together, whereas composition is more complicated even with monad transformers and has to be planned ahead. We have also no issues changing the order of our scoped constructs, which would lead to *different* monadic types.

2.8 Case study: Program generation

Multi-stage programming [5,110], a form of runtime code generation, is a popular way to implement high-performance DSLs [52–56, 142, 143] and specialized numeric kernels [144,145]. In Scala, we can provide a shallow DSL interface on top of low-level code generation facilities, so that users can write, for example,

```
genloop(200) { x => ... }
```

to emit corresponding C code:

```
for (int x37 = 0; x37 < 200; x37++) { ... }
```

This can be achieved by implementing `genloop` as follows:

```
case class Code[T](s: String)
def genloop[T](size: Code[Int])
  (@local body: (@local Code[Int]) => Code[T]) = {
  @local val x = Code(freshVar[Int])
  emit(s"for (int $x = 0; $x < $size; $x++) { ${body(x)} }")
}
```

Inside the body of `genloop(200) { x => ... }`, the variable `x` is a regular program value of type `Code[Int]`, representing the auto-generated identifier `x37`. Without the `@local` annotations, it could be stored into a variable and used to construct another piece of code that refers to `x37`, but where `x37` is not in scope. This situation is

known as scope extrusion in the literature on program generation, and elaborate type systems have been proposed to prohibit such pitfalls [13,146]. Here, we prevent scope extrusion using just three `local` annotations in the definition of `genloop`.

Note that there is a problem: we could not write
`genloop(200) { x => ... genloop(x) { y => ... }}`
because `genloop` requires a first-class `size` value. We cannot easily change the definition of `genloop`, either, because `size` actually escapes through code generation. In fact, we will encounter this issue anywhere we want to use `x`.

The solution is to leverage a split between interface and implementation traits, which already exists in popular code generation frameworks [5]:

```
trait Interface {
  type LT; type clocal = local[LT]
  def genloop[T](@clocal size: Code[Int])
    (@local body: (@clocal Code[Int]) => Code[T])
}
trait Impl extends Interface {
  type LT = Nothing
  def genloop[T](size: Code[Int])
    (@local body: (Code[Int]) => Code[T]) = {
    ... emit ...
} }
```

The argument to `genloop` can now be second-class in user-visible (as abstract type `LT` is unknown to be different from `Any`) but first-class in the implementation code.

Another potential downside is that we cannot store local `Code` objects in a data structure, even temporary, or return them from functions. Thus, we would rule out many useful generative programming patterns [57].

We can solve this final issue in a similar way to the region-based memory system in Section 2.7, by not making the code object itself `@local`, but instead adding a capability token. All operations on `Code` types require such a capability, which is specific to the enclosing region.

```
def genloop[T,L0](size: Code[Int,L0])(@local Cap[L0]): {
  type L1 >: L0
  def apply(body:Code[Int,L1]=>(@local Cap[L1] => Code[T,L1]))
}
```

The type bound `L1 >: L0` provides us with a notion of nested regions, ensuring that inner capabilities are more specific subtypes of outer capabilities.

2.9 Case study: Secure information flow

We utilize the concepts from Section 2.3.2 to bring safety to a whole new level: statically preventing leakage of confidential data to less confidential files. This is analogous to enforcing the "no read up" and "no write up" rules in the Bell-LaPadula (BLP) security model [147], which suggests read privileges `PubR <: SecR` and write privileges `SecW <: PubW`, respectively. Because of their inverse subtyping relationship with respect to secret (`Sec*`) and public (`Pub*`), we specialize `File` as follows:

```
class FileW(val n: String) { def print(@local s: String) ... }
class FileR[P](val n: String) {
  def read[U](@local fn: (@local[P] String) => U): U }
```

Second-class values and a callback in `read` ensure that read contents cannot be written outside the scope of the input file. The BLP security model assumes correct classification of objects, which correspond to files in our case, therefore we need to specialize our scoped file access (via four methods) such that the snippet below achieves the desired behavior:

```
withSecR(...) { fSecIn => fSecIn.read { sec =>
  withPubR(...) { fPubIn => ... }           // ok
  withSecW(...) { fSec => fSec.print(sec) } // ok
  withPubW(...) { fPub => fPub.print(sec) } // error
} }
withPubW(...) { fLeak =>
  withPubR(...) { f =>
    f.read { pub => fLeak.print(pub) }    // ok
    withSecR { fSec => fLeak.print(...) } // error
    @local[SecR] val sec =
      withSecR { f => f.read { s => s} }  // error
} }
```

Here the phantom type `P` in `local[P]` denotes a classification level. As explained in Section 2.3.2, in order for a closure to conform to a function annotated with `@local[P]`, its free variables need to annotated with `@local[T]` for some `T <: P` (`P` is `Any` by default and `Nothing` if the annotation is omitted). We then exploit this mechanics by combining read and write privileges into the lattice in Figure 2.4.

To achieve our goal, we need to disallow free references to secret input files from each closure that models the lifetime of a public output file, and vice versa. For the former, we define a union type `PubW|PubR` and use it to guard the closure; since

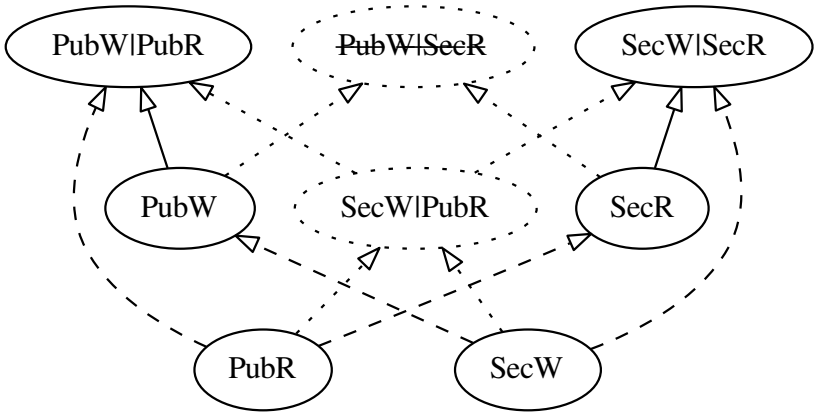


Figure 2.4. The privilege lattice for enforcing the BLP security model.

`SecR` is not a subtype of `PubW|PubR`, attempts to access file handles annotated with `@local[PubW|PubR]` (and declared in outer scopes) will trigger a type error:

```
def withPubW[U](n: String)(
    @local[PubW|PubR] fn: (@local[PubW] FileW) => U)
```

Symmetrically, we guard the closure in the latter case with a union type `SecW|SecR` to disallow free references to public output file handles, which are annotated with `@local[PubW]`:

```
def withSecR[U](n: String)(
    @local[SecW|SecR] fn: (@local[SecR] FileR) => U)
```

The two guards are depicted by solid arrows in the lattice. Finally, public reads and secret writes are harmless, so we allow outer second-class values to pass through those scopes:

```
def withPubR[U](...)(@local fn: (@local[PubR] FileR) => U)
def withSecW[U](...)(@local fn: (@local[SecW] FileW) => U)
```

Observe that closures passed to `with*` do have sufficient privilege to close over free references that bind read data in the enclosing scopes, since their guards use phantom types that are supertypes of `PubR`/`SecR`—the privilege of data introduced via `read`.

The above privilege lattice can be generalized further; more classification levels as well as categories as in the BLP model can easily be added via subtyping.

2.10 Conclusion

In this chapter, we have studied the interplay of modern first-class values with second-class values, as they were commonplace in the days of ALGOL. While second-class values have largely disappeared from modern languages, a process not unlike gentrification in urban development, we find that second-class values can provide important and practically relevant static guarantees, due to their statically bounded lifetimes. We have formalized type systems containing both first-class and second-class values, proving type soundness and lifetime properties with mechanized proofs in Coq. We have also implemented our system as an extension of the Scala language, and conducted several case studies. These demonstrate that ideas from the days of ALGOL complement and play well with cutting edge functional and object-oriented programming facilities such as path-dependent types. Our case studies underline the usefulness and practicality of our system and of second-class values as a programming model.

3 DATA VIEWS

Programmers often face a choice of how to structure their data, but some choices have long-standing consequences on the code design and, more seriously, performance guarantees. One such dilemma is array versus tuple of same-typed values. An array can be offset using raw pointer arithmetic or *sliced* in order to create subarrays in $\mathcal{O}(1)$ time with no or minimal runtime overhead in some languages, such as C and Go, respectively. A tuple is more syntax-friendly, but conversion to or from an array takes linear time and allocation, forcing a programmer to choose either and be stuck with it.

We consider a more general problem, the design and implementation of *views* on an (ordered) set of data chunks (variables or parts of arrays) without the need for rearranging data in a special way. It should be possible by design that a part of data is seen by multiple views, each providing its own logical layout, and we allow composing views into hierarchies for convenience, therefore our data views must be at least partially (ideally fully) *persistent*. (A persistent data structure supports changes without destroying its old versions, which can be at least accessed if the structure is *partially*-persistent, or even modified if it is *fully*-persistent.) A *purely functional* data structure is immutable and hence fully persistent, while the converse is not necessarily true [96].

3.1 Motivating examples

3.1.1 Interleaved vs split representation

In some numerical libraries that work with complex vectors, such as FFTW [148], Spiral [149] or the C++ STL, APIs expect either of two representations—an array

with alternating real and imaginary parts, or the complex and imaginary parts as separate arrays—yet their performance guarantees are sometimes in favor of one or the other. (For example, a null pointer or an array of half the size suffices for the imaginary part in the split representation if the vector is real or conjugate symmetric, respectively.) In those cases, users are forced to do the conversion by copying data, which takes linear time, wastes memory, and requires either provisioning of statically allocated memory for such conversion or paying overhead for a dynamic allocation.

As written in the FFTW documentation, the interleaved format is redundant but still in a widespread use, mostly because it is simpler to use in practice. We introduce an *interleaved* view to neatly provide this convenience without incurring overhead due to conversion between the representation. The index conversion is performed on the fly by division through bit shifting, which should not increase overhead on modern processors that perform both an addition and shifting in one cycle (at least for the cases when array subscripts do not otherwise require bit shifts). In C++, storing such a view as `array<T*, 2>` (i.e., a two-element array of pointers) enables the following implementation of ours for accessing at index i: access the first or the second array (pointer) without branching using subscript i & 1 (modulo 2), then access the element of type `T` at index $i >> 1$ (division by 2).

3.1.2 Excluding a slice or combining arrays

Some algorithms that work with arrays require certain elements to be excluded. Unfortunately, the concept of array slices fails to solve this elegantly because slices can be narrowed but not expanded nor catenated; therefore, one needs to maintain a pair of non-excluded slices instead. To illustrate why this is problematic, consider an algorithm for creating permutations which maintains a list of used elements—eventually a permutation—in array `prefix`, and at each step:

1. picks every unused element stored in array `unused`;

2. solves the problem recursively for modified `prefix` and `unused` with the picked element appended and excluded, respectively.

Observe that a typical implementation would incur $\mathcal{O}(n)$ time overhead to exclude the element by catenating the slices before and after the picked index. Instead, we provide a slice view that is catenable; i.e., two such views (e.g., before and after the excluded element) can be catenated in $\mathcal{O}(1)$ or $\mathcal{O}(\log n)$ time, depending on which guarantees for random access we require, as we are going to explain in Section 3.3. Additionally, we provide a split operation for our generalization of slice (i.e., a view) into two views, which also runs in *sublinear* asymptotic time. Splitting is especially useful for higher-dimensional views, since widespread representations, e.g., row/column-major (sparse) formats, require linear time.

In both cases, our data views provide the convenience and simultaneously solve the underlying algorithmic challenge of maintaining reasonably efficient, but perhaps irrelevant to the programmer, representation of the accessible data. In cases of catenation and split, the problem boils down to maintaining a balanced or shallow tree (or a forest) of portions, or even provide so-called *fingers* for more efficient localized access, as well as specialized iterators.

3.1.3 Sparse matrices

We show that it matters how views are composed together into hierarchies on the following seemingly toy example[1] of a sparse matrix, which actually comes from a collection of real-world sparse matrices SuiteSparse Matrix Collection [150].

Figure 3.1 shows a naive breakdown using horizontal then vertical catenation of 2-D array views. The sparse matrix comprises: the main diagonal on the left; and the ten parts on the right, each containing a full matrix (whose position vary) and a 3x3 diagonal matrix (at a fixed vertical position). As most elements are on the right, reading through or iterating over such a view involves traversing the view hierarchy

[1]linear programming problem, C.Meszaros test set (p0040)

of depth 2, and wastes space; i.e., 32 (1+1+10·(1+2)) views are used to represent a sparse 23x63 matrix.

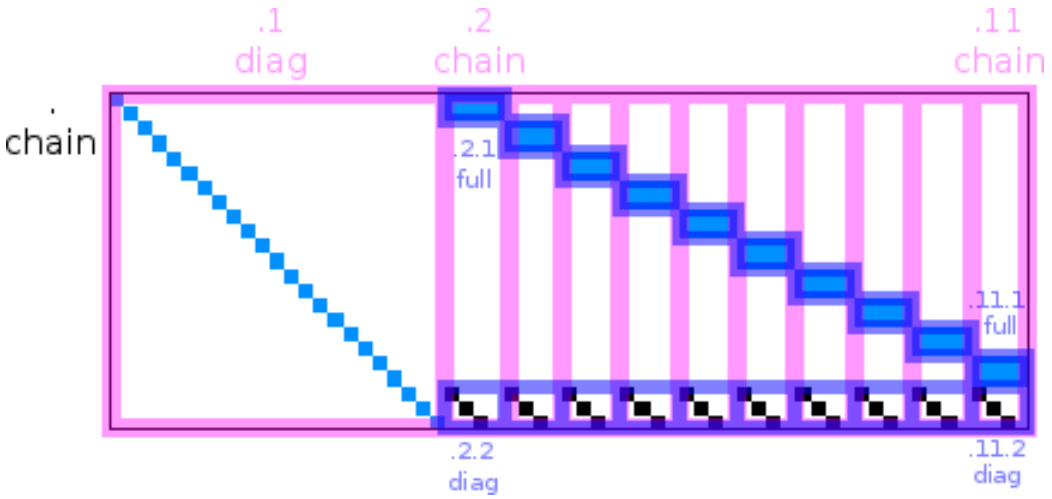


Figure 3.1. A naive view nesting; each block of the block-diagonal submatrix is catenated with a small diagonal below it, forming vertically nested views (dark blue) that are then horizontally catenated with the main diagonal (magenta) into the outermost view (black).

A more conservative approach is illustrated in Figure 3.2. Here, the space is saved by observing that full matrices in the top-right corner form a *block-diagonal* matrix; ˜50% fewer views are required compared to Figure 3.1 (15 instead of 32), albeit the small diagonals views are now nested one level deeper (raising the average nesting level from ˜1.83 to ˜2.05). Moreover, since the blocks are of fixed size (2x4), we are able to optimize away division on accesses within such blocks (given a row and/or column) through specialization; for block dimension of size that is a power of two, we do logical shift right (LSR), otherwise we multiply by a magic number that is precomputed statically using C++ templates (or dynamically compiled once on the JVM).

In fact, using a more advanced kind of 2-D array views we can achieve the same asymptotic complexity of random access and iteration, but decrease the level to 1. The idea is to support a view in which nesting is not necessarily along one dimension (i.e.,

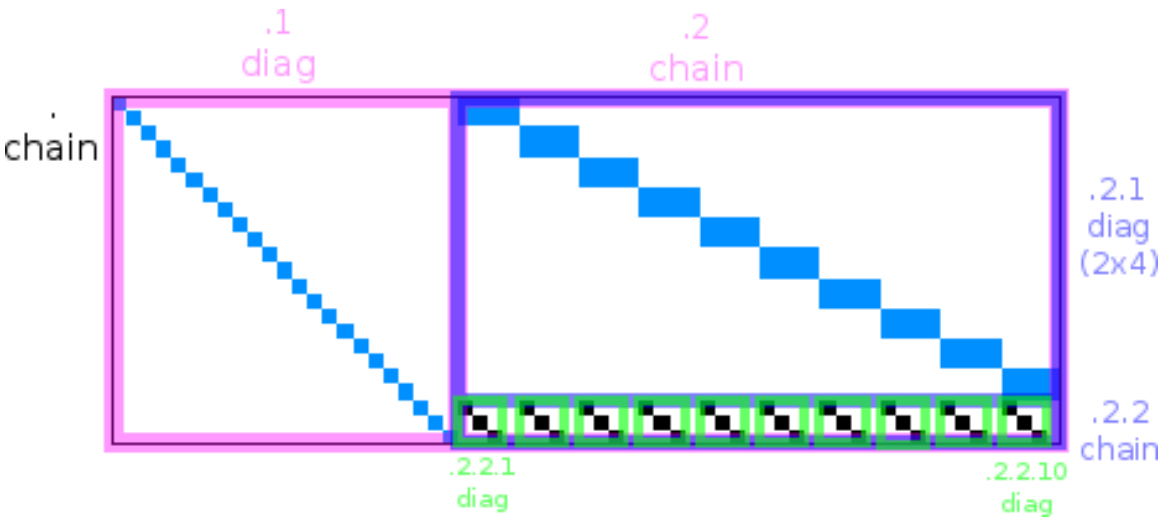


Figure 3.2. An obvious breakdown into the main diagonal and the rest (purple), which is vertically broken down into a block-diagonal matrix (topmost purple) and a horizontal "chain" of 3x4 matrices (green) with non-zero elements along their main diagonals (black).

horizontally or vertically) but may alternate as long as end coordinates of nested views behave as a *monotone* function—this enables binary search in either dimension based on a given row/column to locate the nested views efficiently. Figure 3.3 illustrates this kind of nesting via a so-called `Mono` view, resulting in only a single level of nesting and 13 views, which is indeed optimal.

3.2 View properties and taxonomy

Our data views have semantics similar to slices in Go (or the C++ Standard Template Library), except that they can be uniformly used with all built-in data structures such as arrays, plain variables, or even (hash) maps. In addition, we allow combining two or more existing views into a merger view, provided that the corresponding data types are compatible. Lastly, we discriminate between writable and read-only views. As an example of why the last property is desired, consider a

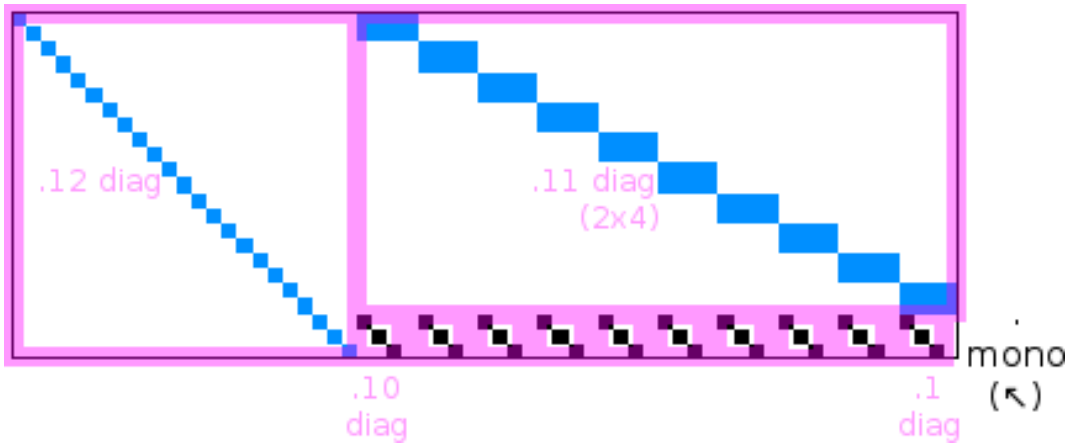


Figure 3.3. The optimal view nesting; all the subviews are catenated at a single level such that their maximal absolute coordinates never increase, in order: the small diagonal views .1 through .10 (the column decreases), the block-diagonal view .11 (the row decreases), and the view spanning the main diagonal (the column decreases).

view whose data is static, ordered and follows a pattern; in that case, we may use a read-only view that uses $\mathcal{O}(1)$ space and encodes the data using a function. If either of the merged views is read-only, the resulting merger is read-only as well.

Since views can be aliased (i.e., see the shared data), they require some sort of garbage collection. In order to avoid speculating when such resource handling of views is needed, we require that data is only referenced through views, not references nor pointers (i.e., all the variables are views). In that case, it is obvious that the data which can no longer be seen by any view can be deallocated. Conversely, data can be created by expanding a view from a thread; this is a generalization of appending to a slice in the Go programming language (which grows the underlying array). Finally, data can become shared only if another thread creates a view out of the view that uniquely sees it—we refer to such a view as *owner*.

3.2.1 Higher dimensions

Our views naturally extend to N dimensions, where we define the following kinds of view via C++ template parameters:

- `NestedArray<T, N>`, a wrapper around `array<T, N>` that provides access by coordinates and iteration along any dimension
- `Sparse<T, N>`, a generalization of a sparse matrix that requires $\mathcal{O}(\log S))$ time for random access, where S is the number of non-default elements (e.g., non-zeros)
- `Diag<T, BlockSizeT, S...>`, a generalization of block-diagonal matrix with $S_1 \times S_2 \times \ldots \times S_N$ blocks
- `Impl<T, N, Access, DimIterFactory>` (usually read-only), which uses $\mathcal{O}(1)$ space by using (stateful) functors (e.g., a closure) for random access and dimension iterator (via a specialized `get<I>` for each dimension I)
- `Chain<T, N, View, Along>`, which catenates views of type `View` into a chain along dimension `Along`; end coordinates for each chained view are required to allow for gaps and/or when dimensionality of nested view is less than $N - 1$
- `Mono<T, N, View>`, which catenates N-dimensional views with monotonically increasing/decreasing end coordinates

All the above family types provide access by coordinates via variadic operator (), as well as efficient iteration along any dimension. For `Diag<T, uint8_t, 2, 4>` as an example, random access involves 8-bit arithmetic operations, and dimension iterators maintain a counter which yields a diagonal element when a certain counter value is reached and a default element otherwise.

3.2.2 Mutable views

We also design and implement mutable ordered views, referred to as *list* views as opposed to array views. They support efficient in-place changes such as inserting or deleting contiguous (parts of) portions, just like a linked list with fingers (e.g., `list::iterator` in C++), in addition to catenation and split (which were fully persistent in the case of array views). Their benefits come in terms of both performance and simplicity of their implementation, albeit at the expense of introducing possible data races when the same view is mutated. We categorize these mutations as follows:

- *expansion*: grows a portion at either end to accomodate newly-allocated data
- *shrinkage*: shrinks a portion at either end to allow for memory reclamation
- *extension*: adds a portion to a view at either end or around a finger
- *restriction*: removes a portion from a view at either end or around a finger

If portions are inserted and removed only at the front/back, opposite or both ends, then it suffices to use a stack, queue or deque, respectively, to store them. Such view implementations need to use a growable array when random access through a view is necessary—at least for prefix sums of the portion sizes—so that we can efficiently find which portion (and where) covers the memory at a (relative) index via a binary search. Our implementation uses two hand-tuned skip lists that make at most $\frac{7}{4}\log_2 N + \frac{1}{4}\log_2\log_2 N$ steps, where N is the distance from the finger (or the closer end if none is provided), and is used throughout our C++ benchmarks. By growing the underlying `deque`/`vector`, on average $\mathcal{O}(1)$ links need to be adjusted when a portion is inserted or removed, which does not compromise the performance. In contrast, repeated catenation of immutable array views necessitates self-balancing trees and the like in order to reach the same asymptotic complexity; moreover, rebalancing algorithms have been empirically shown to be significantly slower than skip lists [94].

Another advantage of list views is that their slicing and splitting may be destructive, thereby avoiding accumulated performance overhead that is due to representing

subviews as wrappers of the original view with adjusted offsets. Consequently, many simpler and faster non-persistent data structures, often with significantly lower memory footprint, from the standard libraries are applicable.

3.2.3 Unordered views

To fulfil our promise of uniformly representing both arrays and (hash) maps, we introduce *bag* views. They can be mutable or immutable, and comprise portions (each being a memory segment or another view, as usual). However, their portions are unordered, although there is still a FIFO order imposed on portions associated with the same key, which serves as a finger. Such same-key portions form buckets, thus a mutable bag view is analogous to `std::unordered_multimap`[2] in C++. Iterating over portions within a bucket using a finger is equivalent to increasing the corresponding `local_iterator` after the bucket is located (i.e., finger obtained). Similarly, immutable bag views can be implemented in Scala via `Map[Key, Queue[Portion]]`.

The extension and restriction operations behave as finger-based insertion and removal in a multimap using `emplace_hint` and `erase` in C++, respectively, with the iterator at the end of a bucket–finger—provided as the argument. Finally, mutable expansion and shrinkage operations on bag views are equivalent to expansion and restriction (well-defined even for array subviews) on the first and last portion within the bucket designated by the key (finger), respectively. The first step runs in constant time once the finger is obtained; so do subsequent recursive steps because they modify only ends of ordered views, or behave as the first step if the argument is a key-value pair. The base case in this recursion is reached whenever an extended/shrunk portion is a non-nesting (leaf) view in which case the unordered view implementation simply delegates to a multimap (if the view is unordered) or an ordered-preserving container such as a deque.

[2] `http://en.cppreference.com/w/cpp/container/unordered_multimap`

The split operation is meaningless without an order, but we instead allow bag views to be *merged* in sublinear time using the disjoint-set data structure (also known as *union-find*), including the more recent variants with deletions [151, 152] and nesting [82, 153]. Merging is useful for representing (sub)records—or even objects and environments in object-oriented and functional settings—as bag views. For example, a superclass or a parent environment can either be represented as an older version of the view if immutable (persistent) bag views are used; otherwise, the overriding and shadowing is achieved by FIFO priority on same-named slots in the record.

3.3 View run-time

So far, it might have seemed as though views are little more than wrappers around arrays or references. In this section we show that views are, in fact, building blocks for creating self-optimizing data structures. Intuitively, this is possible because data views allow the programmer to specify how they want their data to be accessible and under which asymptotic time and space guarantees but without explicitly choosing a specific representation. Actually, the representation need not even be the same throughout a view's lifetime; e.g., data with the same value can be initially shared but lazily allocated and moved on writes by splitting each affected view into several (as in immutable data structures).

3.3.1 Representation

As ordered views are a generalization of slices, they need to store ordered metadata of memory chunks, i.e., triples (source object, begin index, and size or end index). In languages that allow raw memory access via pointers, a pair of virtual addresses unambiguously represents not only an array slice but also a view reference. Otherwise, dummy values for indices (or sizes) can be used but with considerable space overhead. A common base class is a good solution for languages that run in a VM, where virtual dispatch is cheap. Unordered views impose less restrictions, so their implementation

can better exploit optimization opportunities that are due to unspecified behavior, such as object layout in many object-oriented languages.

What about nesting? A simple solution is to allow the source object to be a view and use Run-Time Type Information (RTTI) to specially handle cases when a portion is actually a (part of a) view. This works particularly well on the JVM, since `instanceof` checks are very fast, but is neither efficient nor portable in C++; therefore, we use (variadic) template arguments and specialize the cases of array slice/pointers versus views.

3.3.2 Random access

Given an index i, the main question is how to efficiently find a portion that sees the i-th element in the imaginary flattened view. If the view is frozen, it might pay off to actually flatten it, and compute the prefix sums of the portion sizes; then the binary search on every random access takes $\mathcal{O}(\log i)$ time, provided that empty views are filtered out during the preprocessing. In the general case, however, a thread may create a view that contains many portions, but the actual amount of accesses through that view is largely dependent on the execution path, which may be much less. Therefore, a conservative choice is to not flatten by default but join the corresponding tree-like nesting hierarchies. Even so, the problem is essentially no different—a binary search along a binary or multi-way search tree may be used, which takes time proportional to the tree depth, especially a self-balancing one such as AVL, Red-Black, or B-tree. Among those tree variants, the AVL tree has the least depth, but in all variants it is straightforward to maintain the subtree size information (which is needed for binary search) without increasing the asymptotic time complexity.

3.3.3 Iteration

Supporting efficient iteration over a view is tricky because not only portions might be nesting views; they can be views of different kinds! The latter case is particularly

problematic because each kind of view has its own iterator type, which means that iteration over a nesting (outer) view requires iteration over nested (inner) views, yet the type of the nested views may change, since the nested view might nest another view, and so on. Therefore, the nested iterators need to be polymorphic. While this does not increase time in the asymptotic sense, it does incur overhead due to virtual dispatch. We forbid empty views, as the iterator's `next` method could otherwise take more than constant time; this way, iteration has the theoretically optimal asymptotic time complexity.

3.3.4 Split and exclusion

It is also instrumental to discuss the efficiency of a split operation, which excludes a portion of a view, or (recursively) breaks an existing portion into two portions (i.e., views, respectively). If the AVL trees are used, this operation might not be practically efficient due to a potentially large number of rotations—proportional to the tree height—required to rebalance the AVL tree after deleting a portion (i.e., an element). It has recently been shown by Sen, Tarjan, and Kim [86,87] that rebalancing need not be performed after the deletion, provided that the such a relaxed AVL tree is periodically rebuilt, without sacrificing logarithmic performance, albeit in terms of insertions in this case.

One of the primary use cases of splitting a view is to decrease or control the aliasing. E.g., if a thread no longer needs part of a view, it might split it at the boundary into two views (and the boundary), and destroy one of them (or the data on the boundary, respectively).

3.3.5 Catenation (join)

When two or more array views are catenated (i.e., merged in an order preserving manner), the underlying portion trees undergo a so-called *join* operation, where the indices of the subsequent view operands are increased by the size of the preceding

merger. For example, if a view on characters A and B is catenated with a view on character C, the index of C would change from 0 to 2 in the resulting view, but the indices of A and B would remain the same.

View catenation in $\mathcal{O}(1)$ worst case time is possible using persistent deques by Kaplan and Tarjan [99], which also support random access in logarithmic time (as observed by Okasaki [102]). Another data structure that has been shown effective in practice, albeit providing catenation in logarithmic time, is RRB vector [104].

3.4 Specializing data views

As illustrated by motivating examples, naively creating views can result in deep nesting. This is a problem because every random access requires traversal from the root of the corresponding tree down to a leaf, and traversal in general requires polymorphic iteration along the whole tree. In Section 3.3 we showed a general approach for the most dynamic and unpredictable creation of views, but here we show that we can do much better in many practical scenarios. As an example, consider a view that comprises three array slices of length 4, 3, and 1, respectively, which contains a nested view on the first two chunks as illustrated in Figure 3.4. (The nesting may have occurred unintentionally, or as a result of catenation for efficiency.) For an efficient access at position i, instead of going through a decision procedure starting from the root towards the leaves—which generally requires $\mathcal{O}(\log n)$ comparisons of i and subtree sizes—we generate a switch table, which is $\mathcal{O}(1)$.

If chunks are indeed statically known, it suffices to use C++ template specialization and metafunctions to create specialized methods for access and traversal of views. In fact, a similar approach is already taken by the Standard Template Library implementors; `vector<bool>` could be considered as a view with a finer granularity—unpacked bits instead of bytes—and `bitset<T, Size>` is specialized into a plain integral type for small sizes.

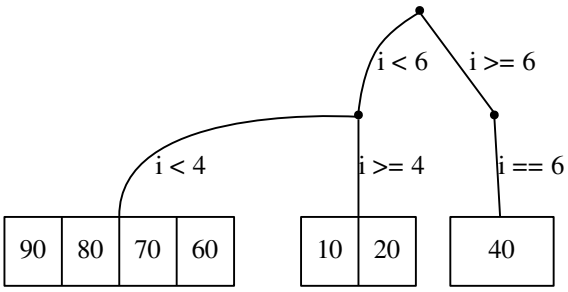


Figure 3.4. A nested array view with three portions, and decisions for random access through it.

Otherwise, we use the Lightweight Modular Staging (LMS) framework to specialize the code on the fly. Even though this is expensive, it eventually pays off as we increase the number of accesses to the views, since the specialized code is necessarily more efficient.

3.4.1 Static specialization (using C++ templates)

We show static specialization on our block-diagonal array view, which we specialize when block size in every dimension is 1, i.e., it is diagonal:

```
template<class T, typename BlkSizeT, BlkSizeT... S>
class Diag { NestedArray<T, BlkSize, S...>[] bs_;  /* ... */ }

// special case: 1 == S0 == S1 == ... == SN
template<class T, typename BlkSizeT, BlkSizeT S0, BlkSizeT... S,
         typename = enable_if_t<1 == S0 && Same(S0, S...)>>
class Diag<T, BlkSizeT, S0, S...> { T[] bs_;  /* ... */ }
```

In that special case, we use an array to store values along the diagonal, and the rest has some default value (e.g., 0), therefore the access method returns `Same(i...) ? bs_[i] : default_val_`, where `Same` is a variadic function template that checks if all arguments are equal without branching: it statically expands into $(i_0 == i_1)\&(i_1 == i_2)\&\ldots\&(i_{N-1} == i_N)$. In the general case when block sizes are $S_1, S_2, \ldots, S_N$, we store the blocks in a list $bs\_$ of nested arrays that support access by relative coordinates $(i_1, i_2, \ldots, i_N)$. We support random access by absolute coordinates via method `at` implemented as follows,

```
template<size_t Ix0, size_t... Ix, typename I0, typename... I>
T& at0(index_sequence<Ix0, Ix...>, I0&& i0, I&&... i) {
  auto k  =      i0 / get<Ix0>(kScaler);
  return Same(k, (i / get<Ix>(kScaler))...))
    ? bs_[k](i0 - k * get<Ix0>(kScaler),
             (i - k * get<Ix>(kScaler))...) : default_val_; }
static const tuple<DimScaler<S...>> kScaler;
template<typename... I>
enable_if_t<sizeof...(I)==sizeof...(S), T&> at(I... i) { return
  at0(make_index_sequence_for<I...>{}, forward<I>(i)...);   }
```

which is enabled only if the number of coordinates equals the number of dimensions, and delegates calls to `at0` (and the dummy index sequence $0, 1, \ldots, N$ that only exists at compile time). The `at0` method first computes the index of the block containing the coordinates, k, by dividing block size in any dimension; if quotients are not the same, a default off-diagonal value is returned. It then computes $i \mod S_i$ with a series of logical shifts and additions (instead of multiplications and divisions) in the overloaded operators `*` and `/` of the helper class `DimScaler`, which is able to specialize this computation because block size S_i is known statically.

We achieve modularity by employing a well-known Curiously Recurring Template Pattern (CRTP). Common functionality (i.e., methods and fields) is statically injected by inheriting one or more helper (base) class templates, each parametrized with an implementation (i.e., `DiagHelper<Derived, ...>`), providing implementation template in terms of `Derived` class. Such *static polymorphism* has no overhead, and helper classes can even access dependent types that may be different in each implementing `Derived` class.

3.4.2 Dynamic specialization (using Scala LMS)

A more flexible and user-friendly approach is taken in our implementation of 1-D and 2-D array views in Scala. The following snippet illustrates the creation of a view on catenation of (reversed) arrays from Figure 3.4:

```
val a = Array.range(0, 100, 10)  // 0, 10, 20, ..., 90
//  a --(implicit conversion with cache)-> ArrayView
val a9DownTo6And1To2And4V = ArrayView(
    a downTo 6,   a from 1 until 3,   a at 4)
```

Behind the scenes, the `ArrayView` type constructor is a *code generator factory* and its methods (e.g., for random access or iteration) are *lazy* fields that are compiled on first access. For example, reading at index i through the above view is specialized as follows:

```
if (i < 4) a(9-i) else if (i < 6) a(i-3) else a(4)
```

Compared to static specialization, implementation is much simpler because LMS does it automatically for execution paths that do not depend on future-stage values (typed as `Rep[*]`); for example:

```
class Diag[T](bs: Array[ Array[Array[T]] ]) {
  def at(i1: Int, i2: Int): T = atC(i1, i2)
  final lazy val atC = compile2(atS)   // lazily compiled once
  def atS(i1: Rep[Int], i2: Rep[Int]): Rep[T] { // staged
    val (k, k2) = ((i1 / bs(0).size), (i2 / bs(0)(0).size))
    if (k == k2) staticData(bs)(k)(i1 - k * bs(0).size)
                                  (i2 - k * bs(0)(0).size)
    else staticData(defaultVal)                              }
```

where current-staged values such as `bs(0).size` are known during dynamic compilation, so division is optimized away (as in C++).

3.5 Experimental results

We have implemented N-dimensional (N-D) array views with static specialization in C++, as well as 1-D and 2-D array views with dynamic specialization in Scala, as libraries named name *cppviews*[3] and *scalaviews*[4]. During the implementation the main challenge we identified is finding a balance between type refinement and runtime abstractions; the more properties of a view we encode as C++ template parameters or current-staged values (not typed as `Rep[*]` in Scala LMS), the more specialization we need to explicitly deal with. In the former case, apart from the complexity of doing compile-time computation in C++, there is a risk of code explosion. In the later case, not only the JVM may end up compiling too much at run-time, but the space for tuning may grow exponentially and become harder to optimize as well.

[3] `https://bitbucket.org/losvald/cppviews`

[4] `https://bitbucket.org/losvald/scalaviews`

3.5.1 Case study: Strassen algorithm (matrix multiplication)

The Strassen algorithm is an efficient divide-and-conquer algorithm for matrix multiplication in time $\mathcal{O}(N^{\log_2 7+o(1)}) \approx \mathcal{O}(N^{2.8074})$, which is faster than the naive $\mathcal{O}(N^3)$ algorithm. The asymptotic improvement in time is achieved by partitioning either square matrix (to be multiplied) into 4 equally sized block matrices—here is where our views come into play—and thus reducing the number of multiplications from 8 to 7. Our baseline is a fast C/C++ implementation by Cochran [154] in which partitioning is done in $\mathcal{O}(1)$ time by adjusting the access strides for the submatrices, but this makes the implementation verbose as both strides and offsets of block matrices need to be explicitly recalculated and carried around. Instead, we represent submatrices with views and split them (in $\mathcal{O}(1)$ time) at each step in the recursion. Table 3.1 presents the results, from which we can see that our convenient and conceptually simple approach has only 20% slowdown for sufficiently big matrices.

Table 3.1.
Running time in seconds of two implementations of the Strassen algorithm, a hand-optimized one that explicitly calculates strides as well as offsets (to avoid copying) and ours in which dense views are simply split, for multiplying two NxN matrices.

N	256	512	1024	2048	4096
strides & offsets	0.012	0.116	0.585	4.015	30.303
splittable views	0.017	0.136	0.704	4.827	36.660
relative slowdown	44%	16%	20%	20%	21%

3.5.2 Case study: Real-world sparse matrices

We have visually examined a huge collection of real-world sparse matrices from SuiteSparse [150], and observed that many can be represented using the same kind of views and with nestings of similar depths. We selected a matrix of sufficient

size (typically hundreds of thousands of elements) as a representative of each such equivalence class, as well as some of the atypical ones in order to stress test our methods. Details of the matrices can be found through the online search tool[5] by entering their unique names.

We were able to represent each of our sample matrices using 2-D array views defined in Section 3.2.1 with only a few levels of nesting (depicted as magenta, dark blue, green, red, respectively), after allowing ourselves to: waste a small fraction of space by overapproximating certain submatrices as dense by using full views, which is shown in Figure 3.5; or potentially give up some performance by using sparse views instead of fully exploiting a structure of a submatrix with complicated patterns, as illustrated in Figure 3.6.

To evaluate performance of reading sparse matrices through our views, we first wrote a GUI program (with an interface similar to the previous figures), which generates a JSON file that describes the user-created view hierarchy without the actual non-zero elements; i.e., which views cover which parts of the matrix and how they are nested into the top-level view. Then, we have a C++ code generator that outputs a header file in which views have many properties statically encoded using C++ template parameters, as shown in Figure 3.7, so that further compilation for the benchmark of a particular view specializes the code. For each 3rd-party library that we compare performance against, we wrote a template-specialized sparse matrix view facade, `SmvFacade<ThirdPartySparseMatrix>`, which allows for easy uniform and static treatment. The overhead of the facade layer is normally optimized away by the C++ compiler, since our classes use *static polymorphism* and their methods simply delegate parameters to the APIs of the underlying libraries. Figure 3.8 shows the part of our evaluation pipeline that produces `*.hpp` header files that declare an uninstantiated class template of a view-like object (each inheriting the corresponding facade), and Figure 3.9 shows the next stage in which the code is specialized (through

[5] `http://yifanhu.net/GALLERY/GRAPHS/search.html`

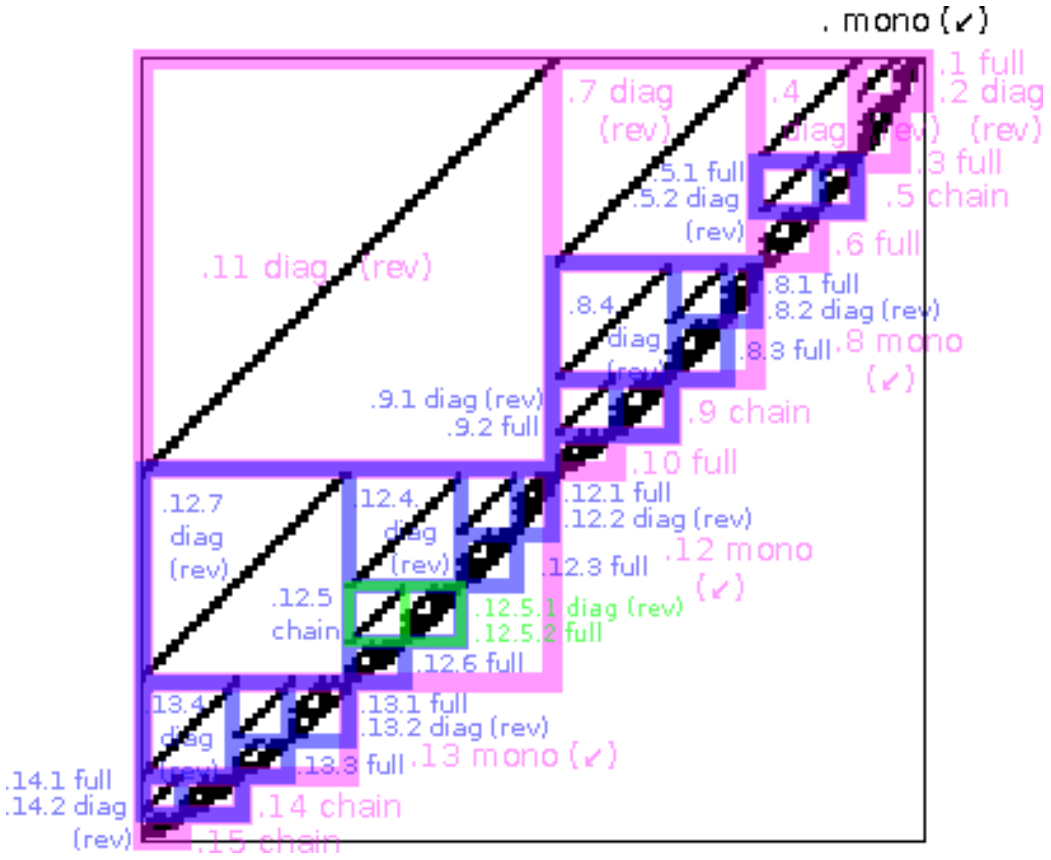


Figure 3.5. Simplical complexes from Homology from Volkmar Welker (n3c6-b7). The parts around the antidiagonal are represented via 16 full views (`NestedArrays`), each of approximate size as the rightmost green rectangle, although these small submatrices look similar to the whole matrix (i.e., have a fractal pattern).

template instantiation and specialization) based on a statically known view hierarchy (or properties of sparse matrices in case of 3rd-party libraries).

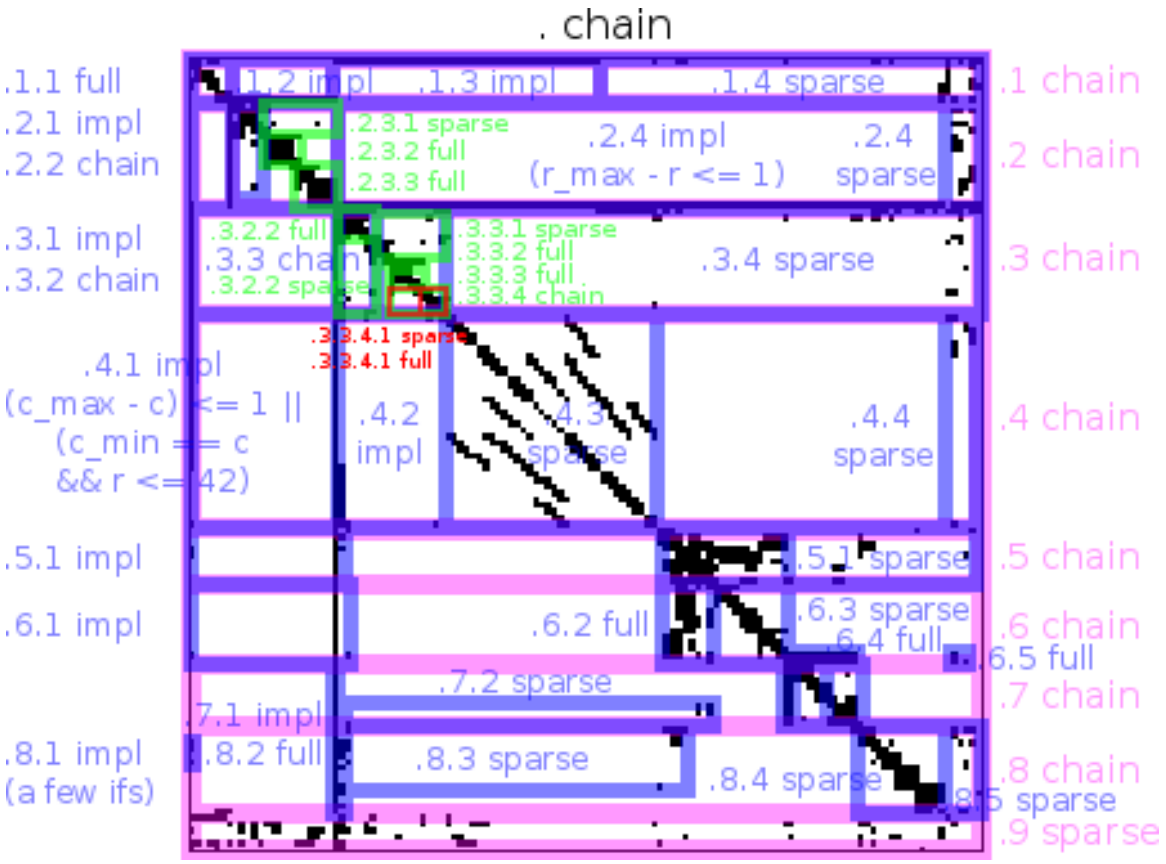


Figure 3.6. A circuit simulation problem (rajat01). The central region with diagonal-like submatrices—not even block-diagonal due to gaps (not visible)—is underapproximated by using a sparse view. This avoids the need of a nearly-(block-)diagonal kind of view.

Using our pipeline, we performed a series of microbenchmarks, random reading of zero and non-zero values, and iterating over non-zero values in a fixed order (consistent with iteration over the corresponding indices). We measured average times on 3–5 runs of these benchmarks on two matrices—containing 133 and 255004 non-zero elements (and 23 rows and 32 columns, and 60008 rows/columns, respectively)—such that a large number of candidate access coordinates are precomputed (typically 10^5–10^6 pairs), which are repeatedly shuffled and read in round-robin fashion sufficiently

```
#include "facade.hpp"
struct Figure3_2
#define SM_BASE_TYPE Chain<ArrayView<int, 2>, 1>  // CRTP for static
    : public SM_BASE_TYPE, public SmvFacade<Figure3> { // injection
  Figure3_2() : SM_BASE_TYPE(                          // of methods
#undef SM_BASE_TYPE
    ChainTag<1>(), PolyVector<ArrayView<int, 2>>()
    .Append([] {          // .1 MAIN DIAGONAL (diag)
      Diag<int, uint, 1, 1> v(ZeroPtr<int>(), 23, 23);
      for (uint i = 0; i < 23; ++i) v(i, i) = 1;
      return v; }())
    .Append(              // .2 VERTICALLY CHAINED RIGHT PART (BLUE)
      ChainTag<0>(), PolyVector<ArrayView<int, 2>>()
      .Append([] {        // .2.1 BLOCK-DIAGONAL PART (diag 2x4)
        Diag<int, uint, 2, 4> v(ZeroPtr<int>(), 20, 40);
        return v; }())
      .Append(            // .2.2 HORIZONTALLY CHAINED diags (GREEN)
        ChainTag<1>(), PolyVector<ArrayView<int, 2>>()
        .Append([] {      // .2.2.* diag (ONLY THE FIRST ONE SHOWN)
          Diag<int, uint, 1, 1> v(ZeroPtr<int>(), 3, 3);
          for (uint i = 0; i < 3; ++i) v(i, i) = -1669;
          return v;  }()) // ... 9 MORE Appends WITH ^value != 0
        , ZeroPtr<int>(), ChainOffsetVector<2>({
            {0, 0},       /* ... 8 MORE OFFSETS */       {0, 36}  })
        , 3, 40)
      , ZeroPtr<int>(), ChainOffsetVector<2>({{0, 0}, {20, 0}})
      , 23, 40)
    , ZeroPtr<int>(), ChainOffsetVector<2>({{0, 0}, {0, 23}})
    , 23, 63)
  {                       // VALUE INITIALIZATION (8 BLOCKS HIDDEN,
                          // ONLY FIRST&LAST 8-ELEMENT BLOCKS SHOWN)
    static int data[] = { -1, -1, -1, -1,  +1, +1, +1, +1,    // ...
                          -1, -1, -1, -1,  +1, +1, +1, +1, };
    static uint rows[] = { 0,  0,  0,  0,   1,  1,  1,  1,    // ...
                          18, 18, 18, 18,  19, 19, 19, 19, };
    static uint cols[] = {23, 24, 25, 26,  23, 24, 25, 26,    // ...
                          59, 60, 61, 62,  59, 60, 61, 62, };
    for (size_t i = 0; i < 80; ++i)
      (*this)(rows[i], cols[i]) = data[i];
  }
}
```

Figure 3.7. Generated C++ header code (except include guards) for the view in Figure 3.2. `Diag` views have their block sizes, 2x4 and 1x1, as template parameters, which enables shifts by a constant instead of divisions upon random access. Similarly, chaining dimensions are statically encoded via `ChainTag` for efficient iteration.

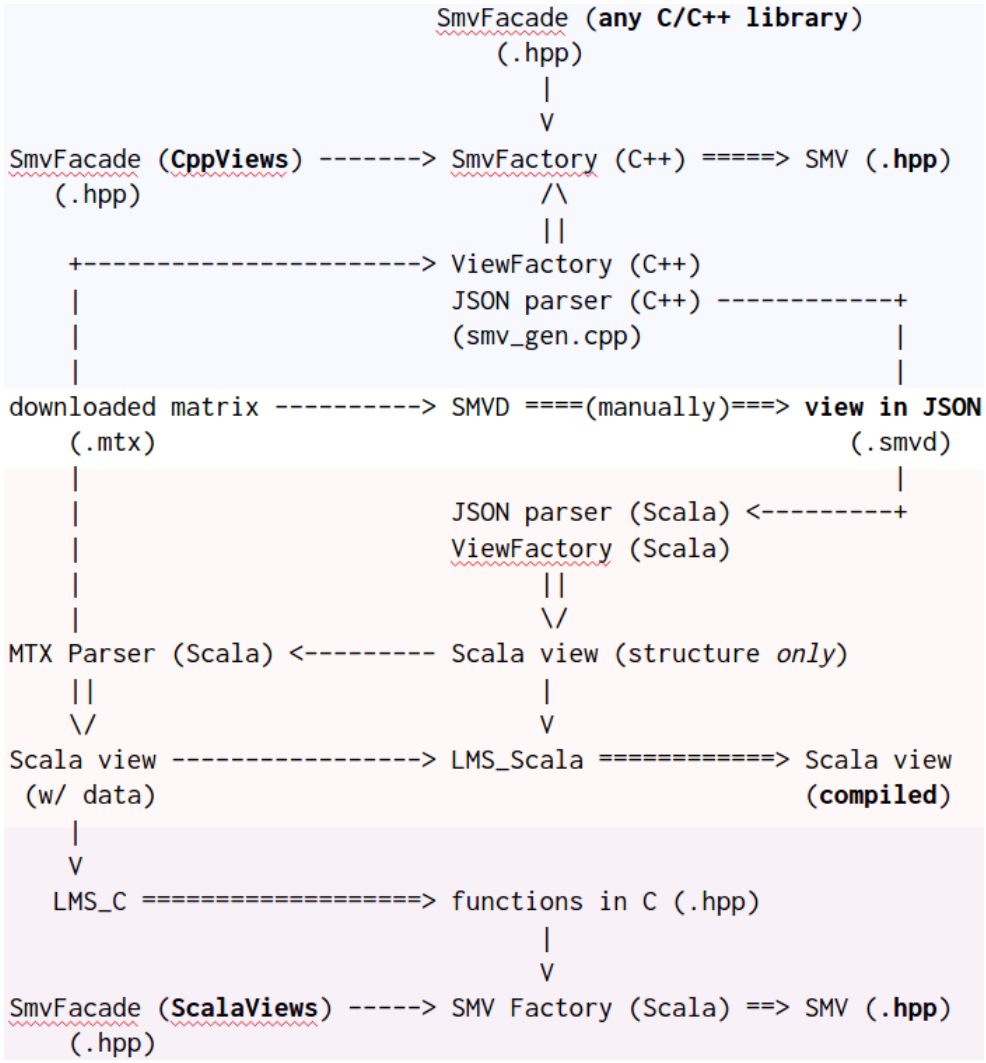


Figure 3.8. The pipeline for generating a specialized view of a sparse matrix: as a C++ header file out of its JSON representation, dynamically from Scala (lower part), or statically from C++ code (upper part) using cppviews or a third-party library that provides matrix-like data structure for which a facade should be written.

many times (10^6–10^9) in each run, so that the times are around a second. Table 3.2 and Table 3.3 show normalized results for these two matrices in millions of IO operations per second (IOPS). In a sufficiently large matrix, our random access of non-zero values is 658% and 14% faster than the one of sparse matrices in Armadillo [155] and

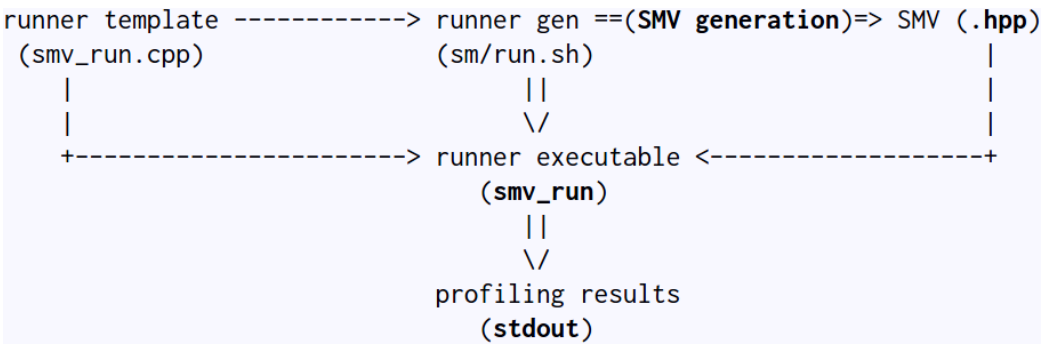


Figure 3.9. Evaluation pipeline for running experiments using the generated C++ header file (see Figure 3.8).

Eigen [156] libraries, and 77% faster than the C++ hash table, while the random access to zero values is still acceptable, 59% and 93% slower than in Armadillo and Eigen. Iteration over non-zero values is also several times slower, which is understandable since we do not statically eliminate nesting in our C++ implementation.

Table 3.2.
Performance of random reads and iteration in millions of IOPS (higher is better) for the small sparse matrix p0040. Two implementations of the view hierarchy as in Figure 3.2 were benchmarked, in which the sizes of chained diagonals as well as in-between gaps are both statically known or unknown, respectively. Creating either view using our GUI tool took 35–60 seconds on average.

	Random Read		Iteration
	0s	non-0s	non-0s
`arma::SpMat`	110.536	104.984	450.352
`Eigen::SparseMatrix`	71.180	39.701	1745.864
cppviews	44.389	34.972	34.620
	44.155	37.026	33.155
`std::map`	30.941	27.067	196.826
`std::unordered_map`	39.718	56.875	–

Table 3.3.
Performance of random reads and iteration in millions of IOPS (higher is better) on the large sparse matrix a5esindl, which we represent with `Diag`s nested up to 3 levels deep via `Chain` views. The view creation using our GUI tool took 3–5 minutes on average.

	Random Read		Iteration
	0s	non-0s	non-0s
`arma::SpMat`	51.259	2.757	172.665
`Eigen::SparseMatrix`	57.814	15.872	251.166
cppviews	29.997	18.149	45.075
`std::map`	2.023	1.497	55.155
`std::unordered_map`	8.937	10.252	–

3.5.3 Case study: Writing through a shared view

Motivated by the observation that our data views precisely capture aliasing, we evaluated the feasibility of a push-based memory model in which:

- each thread (or process if data is shared system-wide) keeps its own copy of the aliased (shared) data;
- writes are propagated to every thread (or a process), i.e., observer.

The former is practical in many cases, since it suffices that each observer creates views on a subset of data that it reads or modifies. (If this is a rather large subset, then parallelization is impractically slow regardless of the approach taken because of contention on shared locks or memory duplication from data versioning, for instance.)

In order to efficiently support the latter, we need to support identifying a set of affected observers on each write through a view. We assume that no data can be read or written except through some view—at the very least a *singleton* view that has the semantics of a reference—and that data is created via views, as explained at

the beginning of Section 3.2. Under this simplifying assumption, each observer (i.e., a thread or a process) can only see updates within memory segments covered by at least one of its views (referred to as view portions). Therefore, the runtime (or even compiler if we enforce static ownership semantics as explained in Chapter 4) needs to track such view portions, as well as the corresponding observer/view for each of them to be able to notify it when their data is modified.

This could be tracked globally, but it would require coordination between observers upon writes and/or reads unless the access involves reading a non-shared data (owned by a single view). A more lightweight approach that we evaluated is to let each observer track those intervals for their own views; that way, observers only need to synchronize acquisition/release of shared ownership of data during creation/destruction of such *overlapping* views. We efficiently compute the set of affected observers for writing through a view (or creating/destroying a view) by querying all overlapping intervals at an access point (or interval if a view is created/destroyed, respectively) using a data structure commonly known as *interval tree*, which is explained in Figure 3.10. By storing the overlapping intervals at the probing point of each interval tree node in a self-balancing binary search tree, we support (un)registration in $\mathcal{O}(\log P)$ time per view portion, where P is the total number of portions that an observer can see. Since the number of (un)registrations is typically negligible compared to the number of reads/writes, we chose the AVL tree over red-black tree for the interval tree itself because the upper bound on its height is 28% then less—approximately $1.44\log_2(N+2)-0.328$ instead of $2\log_2(N+1)$—resulting in 28% faster binary search per write; an overhead of $\mathcal{O}(\log P)$ rotations per (un)registration is acceptable.

In large programs, we expect many views to be live upon writing data, and thus the dominating factor in performance to be the accumulation of intersecting intervals at each node (see Figure 3.10) along the path of length $\log P$ during a binary search. This operation, referred to as `find-overlap`, clearly runs in $\mathcal{O}(R+\log P)$, where R is the number of accumulated intervals, i.e., overlapping view portions, since in-order (sorted) iteration over trees in each node takes amortized $O(1)$ time per

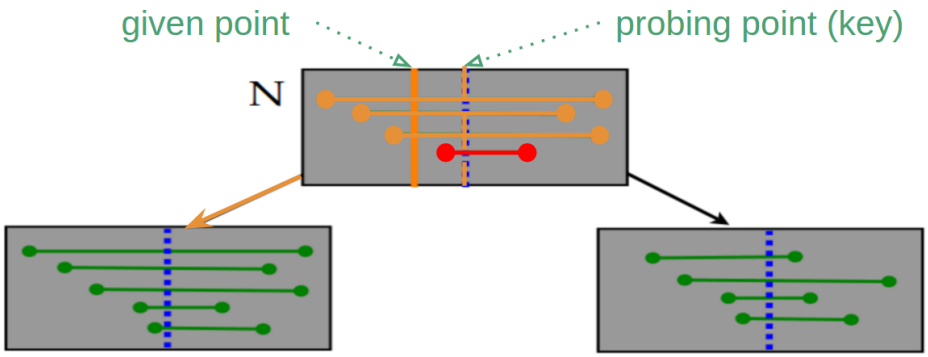


Figure 3.10. A (self-balancing) interval tree storing intervals that overlap a probing point in each node, both in ascending and descending order by their start and end points, respectively. This allows efficient enumeration of intervals that intersect any given point: include (reverse) sorted intervals as long as they overlap the given point, if the given point is less (or greater, respectively).

stored interval. We show that `find-overlap` is also *practically efficient* by profiling our implementation based on interval tree described above, whose code is shown in Figure 3.11, comparing it to the naive linear-time approach that checks overlap with every view with the following implementation[6]:

```
void findOverlap(Integer point, Collection<Interval> result) {
  for (Interval interval : intervals) {       // O(P) time total:
    if (interval.from.compareTo(point) <= 0   // if given point
        && point.compareTo(interval.to) <= 0) // overlaps, add it in
      result.add(interval);                   // O(1) amortized time
  }
}
```

The profiling was performed on Java 7, after making 10^5–10^6 calls in order to give enough time for methods to be inlined by JIT compilation. Similarly, the overlapping intervals (representing view portions) were pushed into a large preallocated `ArrayList`, which was cleared between calls (preserves capacity), to avoid bias due to amortized overhead when calling the `add()` method.

[6]The complete source code, including unit and performance tests, is available at the following URLs:
`https://github.com/losvald/sglj/tree/phd-thesis/src/main/java/org/sglj/util/struct`
`https://github.com/losvald/sglj/tree/phd-thesis/src/test/java/org/sglj/util/struct`

```
void findOverlap(K point, Collection<E> result) {
  final E pointInterval = this.traits.pointInterval(point); // given
  final Comparator<Object> pointComparator = this.getComparator();
  Node<E, K> node = (Node<E, K>)this.getRoot();   // Start from root
  while (node != null) {
    int cmp = node.compareKey(pointInterval, pointComparator);
    NavigableSet<E> overlapSet;  // 1. Use sorted set whose order is
    if (cmp < 0) {               //  ascending if given < probing,
      overlapSet = node.asc;  node = (Node<E, K>)node.left;
    } else if (cmp > 0) {        //  descending if given > probing
      overlapSet = node.desc; node = (Node<E, K>)node.right;
    } else {                     // (optimize if 2 points are equal
      if (node.asc.size() == 1) result.add(node.asc.first());
      else result.addAll(node.asc);
      break;                     //  by breaking early)
    }                            // 2. Update node accordingly
                                 // 3. Binary search (twice) in time
    E last = overlapSet.floor(pointInterval);      // O(1) amortized
    if (last != null) {          //  (optimize if overlap size <= 1)
      if (overlapSet.first() == last) result.add(last);     // == 1
      else {                     //  add O(1) amortized intervals
        overlapSet = overlapSet.headSet(pointInterval, true);
        result.addAll(overlapSet);  // in O(1) amortized time,
      }
    } // Note: amortized analysis is over # of overlap. intervals, R
} }
```

Figure 3.11. `find-overlap` in $\mathcal{O}(R + \log P)$ time using an interval tree in Java.

Figures 3.12, 3.13, and 3.14 show the execution times in nanoseconds for increasing number of view portions total (P) and those that overlap (R), from which it is visible that the interval tree implementation:

- takes hundreds of nanoseconds (ns), which is comparable to a system call[7];
- is faster than the naive approach, except slightly slower for no overlap;
- scales well and consistently with the analyzed $\mathcal{O}(R + \log P)$ asymptotic time.

It is worth emphasizing that the experiments were run on an old machine, so execution times would probably be significantly smaller (albeit the same order of magnitude) on a modern machine.

[7] `https://gist.github.com/jboner/2841832`

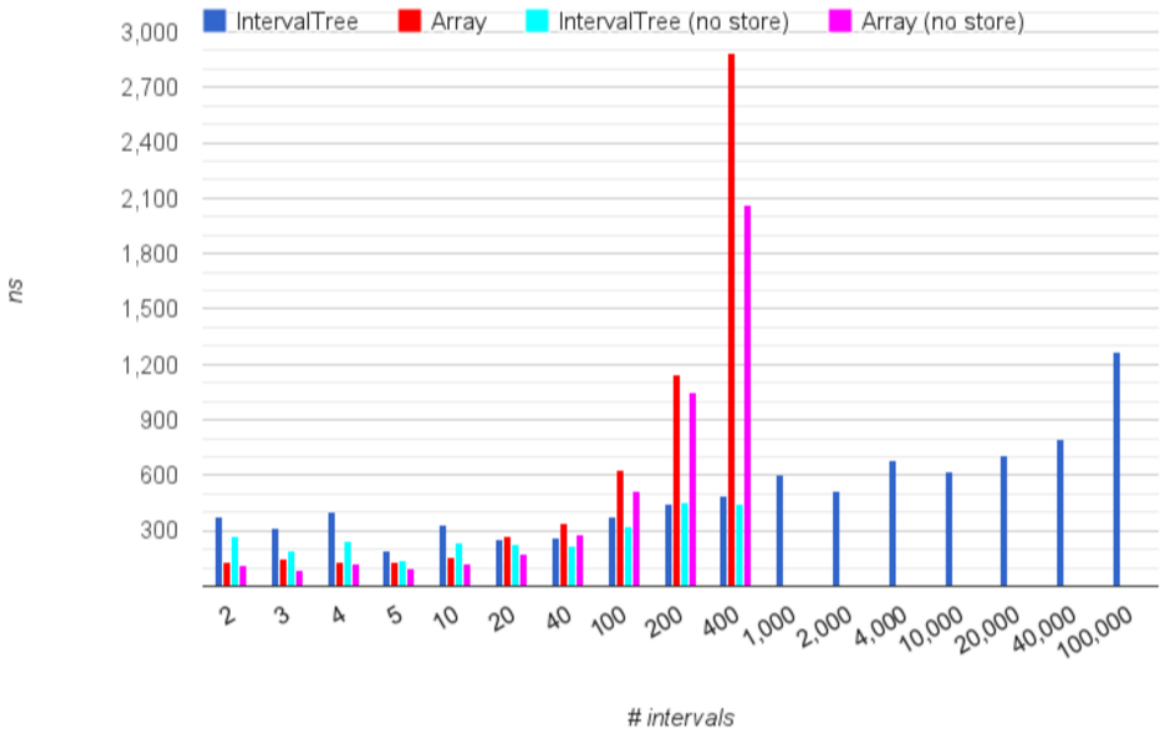


Figure 3.12. Performance of `find-overlap` with minimal sharing. Observe that the search time of our interval tree implementation is in the range of 300–1000 ±200 nanoseconds, which is only a few times slower than reading directly from RAM. This suggests that it is practical even for *real-time* systems. On the other hand, the naive search is linear in this case (because each portion (view) overlaps exactly one other view), and quadratic in general, which makes it impractical as soon as the number of shared views exceeds a few hundred or a thousand.

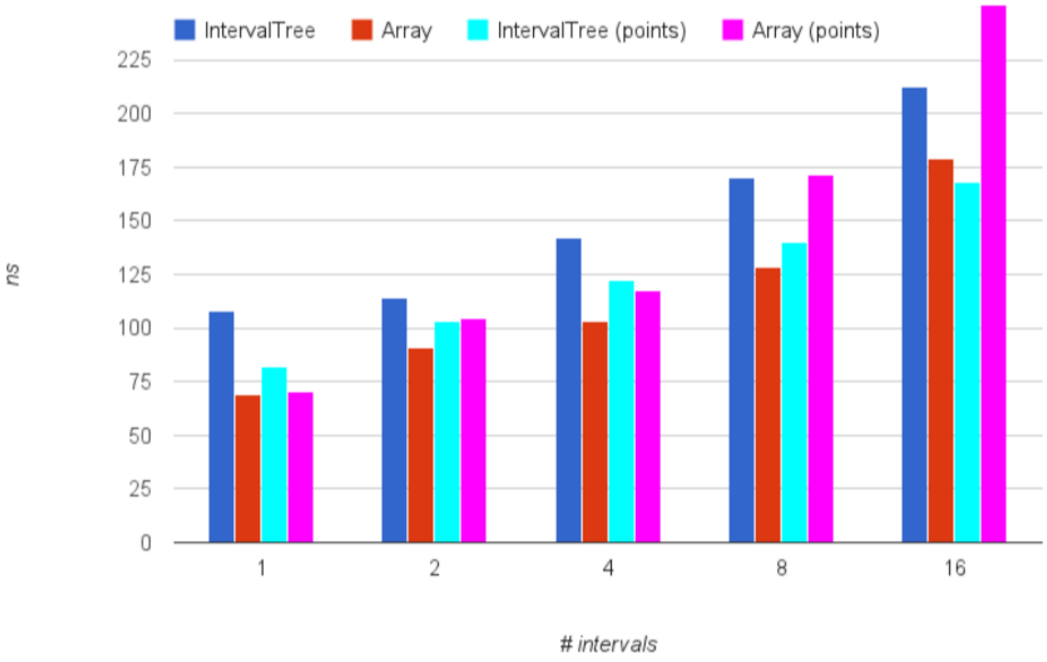


Figure 3.13. Performance of `find-overlap` when no view is shared. Observe that the *constant-time work* during the interval tree search is comparable with the one in a simple array-based search, regardless of whether we store endpoints of the interval (portion) as two integers (primitives) or a `Point` object in Java. The number of views is kept very low to avoid bias that is due to the difference in asymptotic time of the two algorithms.

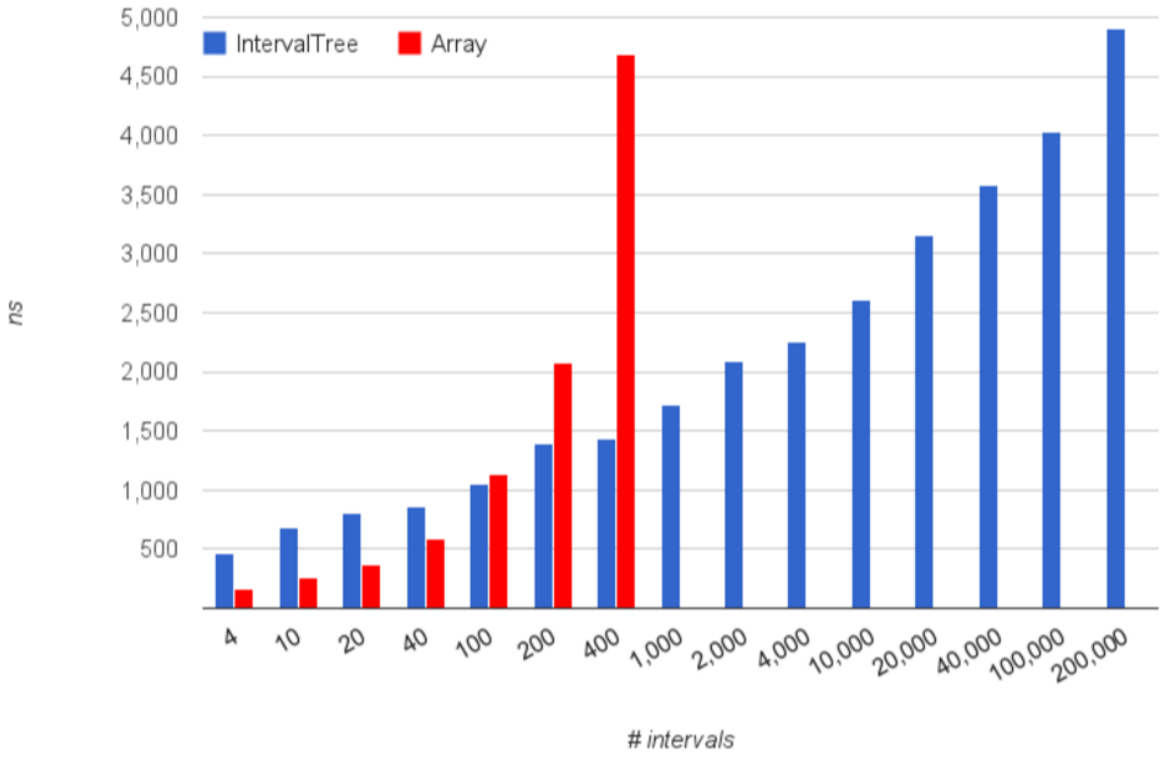


Figure 3.14. Scaling of the interval tree approach for shared view writes when each of P views is shared with another $\log_2 P$ views. Observe that the time for interval tree search (blue) approximately doubles when the number of shared views is squared, which empirically supports its logarithmic time complexity; e.g., consider comparing times when P equals 10 vs 100 as well as 200 vs $200^2 = 40000$. By contrast, the time for naive array-based search (red) grows quadratically; consider comparing time for 10 vs 100 or 100 vs 200 (time quadruples).

3.6 Conclusion

We design and implement data views that are more general than existing data structures, supporting efficient operations such as split/catenation in N dimensions. They allow not only finer-grained resource management, alias control and sharing; they shift the burden of picking the optimal representation from a programmer to the compiler. In C++, we compared our view performance and found it superior to optimized implementations of general-purpose data structures such as hash and sorted maps, on par with hand-tuned dense matrix representations, and not more than a few times slower than ad-hoc (domain-specific) representations implemented in state-of-the-art linear algebra libraries. For efficiency, we use static specialization, static polymorphism and other compile-time metaprogramming facilities. We also show feasibility of dynamic specialization through on-line compilation in Scala LMS, a multi-stage-programming framework, on our prototype library that hides the complexity of specialization from the user. Finally, we provide promising results for a memory model based on views in which all reads and writes go through a view and memory is managed by the views, which are obtained by simulating data synchronization operations that would be a bottleneck in a full-fledged system.

4 FLOW-INSENSITIVE RUST-LIKE REFERENCES

It is well known that *mutability* in imperative (as well as non-pure functional) languages may lead to obscure bugs, not only in a concurrent setting but also in sequential programs, so long as multiple *aliased* references exist at a program point. Examples of the former are data races, which occur whenever there are two concurrent accesses to the same memory location with at least one being a write, unless both are atomic (which incurs synchronization overhead). A perhaps unusual but important example of the latter is iterator invalidation, which occurs due to concurrent modifications of the traversed data structure through an aliased mutable reference (the other reference is needed for traversal):

```
val nats = scala.collection.mutable.MutableList(1)
for (nat <- nats) {
  println(nat)
  if (nat < 10) nats += nat + 1
}  // prints 1 and 2 (not 1 only, not 1 through 10)
```

To restrict mutability to cases when it is safe, different programming languages take different approaches. Two extreme approaches are to disallow mutability or aliasing whatsoever; these are arguably best witnessed by Haskell[1], a purely functional language, and R, which copies data whenever aliasing may occur (copy on write) [157]. Rust [3], a recent statically typed language by Mozilla, on the other hand, offers zero-cost abstractions and provides static guarantees that programs are free of data races, null pointer dereferencing, as well as certain bugs present even in sequential programs with garbage collection such as iterator invalidation. Its type system includes a *borrow checker* component that statically checks that the following rules hold:

- one or more references (`&T`) to a resource; and
- no more than one *live* mutable reference (`&mut T`).

[1] `https://www.haskell.org`

A live reference is roughly a reference that can be (safely) dereferenced. Immutable references are always live unless they are unreachable; i.e., their pointed-to values have been moved out due to *ownership* transfer. However, a (reachable) mutable reference can be *reborrowed* for a shorter lifetime, such as through a function call, during which time the reference is *not* live. (For example, it is safe to pass a mutable reference to a function while holding one at the call site.)

In this chapter, we show one way of retrofitting the concepts of borrowing and alias control from Rust into Scala. We describe a general method of adding *flow sensitivity* to Scala's type system, which is applicable to a broad set of flow-insensitive languages. More precisely, we describe a Scala extension that statically enforces the following as long as the variables are introduced via the appropriate wrappers:

1. lexical scoping of variables;
2. no creation of mutable variables out of other variables;
3. no assignments on immutable variables.

On a high level, we use second-class values from Chapter 2, subtyping and implicit conversions, and macros and virtualization, to enforce 1., 2., and 3., respectively. We exploit the extended Scala type system based on top of System $D_{<:}^{1/2}$, which we proved sound in Section 2.3.1, and show how to achieve some static guarantees as in Rust, namely stack-bounded lifetimes and exclusive mutability of references.

4.1 Motivation

Rust's type system is flow-sensitive. Consequently, an expression may have different types depending on control flow; e.g., the following snippet in Rust,

```
let mut x = 5;
let y = &mut x;
*y += 1;
println!("{}", x);
```

fails to compile, producing the following error message:

```
error: cannot borrow 'x' as immutable because it is
       also borrowed as mutable
                    println!("{}", x);
```

This is in contrast to Scala (and most other languages), where types are flow-insensitive. To be more specific, the `x` variable does not have the same "type" throughout the scope in the above snippet, which means programs in Rust (or another flow-sensitive language) are more complicated to reason about in general. The line at fault here is `let y = &mut x`, which restricts the "type" of `x` by disallowing any further accesses to it for the remainder of the scope. (Indeed, if the above snippet is rewritten as

```
let mut x = 5;
{
  let y = &mut x;
  *y += 1;
}
println!("{}", x);
```

it does compile, since limiting the mutable alias of `x` (via `*y`) to an inner scope makes `x` unaliased in the outer scope.) The printed error explanation is:

```
        let y = &mut x;
                    - mutable borrow occurs here
        *y += 1;
        println!("{}", x);
                       ^ immutable borrow occurs here
    }
    - mutable borrow ends here
```

For our goal of extending Scala with similar static checking capabilities, a direct translation of Rust's typing rules would therefore lead to a quite different, and certainly much more complicated language. Inspired by recent work on object capabilities in Scala [128], we observe that we can remove the dependence on flow sensitivity by introducing auxiliary scopes whenever flow-dependent information changes. This idea naturally leads to expressing programs in continuation-passing style (CPS) or monadic style. The example `let y = &mut x` becomes in Scala syntax:

```
bindMut(x) { y => ... }
```

Observe that the visibility of `y` is based on the `{ y => ... }` scope, thus it suffices to solve a simpler checking problem that is based on scopes. For clarity, we will use explicit monadic syntax throughout this chapter, but the transformation can easily be automated and hidden from users [7].

4.1.1 Dangling references and mutable aliasing

Each scope in Rust has a lifetime associated with it, and if it declares a variable, then the variable may not outlive that scope.

Data In Rust, the data is stack-allocated by default, and the type-checker statically enforces its non-escaping, so it can deallocated at the end of the scope. (Special functions for allocating objects on heap, and performing reference-counted garbage collection exist in Rust as well, but are not of interest here.) The lifetime of uniquely-owned data is extended beyond its defining scope through a *move* (destructive read).

Binding In Rust, several pieces of data can be bound to variables within the same scope using `let` statements, and these pieces are deallocated in the reverse order upon exiting the scope. However, this is just a syntactic sugar and the compiler inserts additional scopes to simplify type-checking.

Consider the following snippet in Rust:

```
let mut data = vec![1, 2, 3];
let x = &data[0];
data.push(4);
println!("{}", x); // dangling ref. (data may have been reallocated)
```

The above snippet is not safe because pushing an element into a vector that is at full capacity requires its reallocation, which involves deallocating the old memory region, yet reference `x` points to the old memory region. In fact, the above code does not type-check in Rust; it desugars to the following intermediate representation, in which lifetime of each scope `s` is denoted by `'s`:

```
'a: {
    let mut data: Vec<i32> = vec![1, 2, 3];
    'b: {  // 'b is as big as we need this borrow to be
           // (just need to get to "println!")
        let x: &'b i32 = Index::index::<'b>(&'b data, 0);
        'c: {  // Temporary scope because we don't need the
               // &mut to last any longer.
            Vec::push(&'c mut data, 4);
        }
        println!("{}", x);
    }
}
```

Lack of trust in Rust Despite a growing adoption of the Rust programming language—the first widely used industrial language because of its novel static safety guarantees (no mutable aliasing, no memory leaks or dangling pointers, no null-pointer dereferencing)—there are concerns about the soundness of its type system, especially with respect to borrow checking and lifetime inference. On a high-level, the Rust compiler comprise two components: *type checker* and *borrow checker*. These two seem to be well understood in isolation but not as a whole. In fact, some Rust programs that do not violate any of the safety rules that Rust should enforce in theory still fail to type-check. For similar reasons, the formalization is hard; there were only partial proofs of progress and preservation [130] at the time of publishing the work in this chapter (October 2017), although the relevant subset of Rust has been formalized since [131].

To model stack-bounded lifetimes in Scala, we use second-class values described in Chapter 2 but in a way that mutable references are subject to more restrictions. We do not enforce lifetimes of data—only variables—since that would require a full ownership model with the move semantics (which interferes with garbage collection).

4.2 Syntax and examples

In Scala, we introduce facilities named `bindMut` and `bindImm` to bind a literal or a variable to a newly introduced mutable or immutable variable, respectively, in the continuation-passing style (CPS).

To demonstrate the mechanics of our facilities for enforcing the aforementioned rules 1.–3. (page 97), consider the following snippet:

```
bindMut(42) { mut =>
  bindImm(mut) { imm => ... }        // error
  bindMut(mut) { mutAlias => ... }   // error
  val mutAlias = mut                 // error
  mut.value = 0                      // ok
}
```

First, the literal integral value 42 is bound to a mutable variable `mut` for the remainder of the snippet. In the second line, an attempt is made to rebind the mutable value as immutable, which fails as expected; if our system was to allow it, that would be unsafe because the same location that holds 42 would be *mutably aliased*: writeable through `mut` and readable through `imm`, two variables (aliases in our case). The third line fails for the same but stronger reason. The next line is yet another attempt to work around the type checker; it fails for a less obvious reason that will be described shortly, but intuitively it is because all variables in our system are introduced as second-class—parameters of closures in the CPS. (Being parameters, and thus values, their reassignment is disallowed in Scala by design.) Finally, we provide a means to mutate mutable variables by assigning new values to them via a setter method `value_=`, as demonstrated in the penultimate line.

Conversely, immutable variables can share their pointed-to values, e.g.:

```
bindImm(1) { imm =>
  bindImm(imm) { immAlias => ... }  // ok
  bindMut(imm) { mutAlias => ... }  // error
  imm.value = 0                     // error
}
```

In the snippet above, an immutable variable `imm` is initially assigned a 1 in the first line. In the second line, an alias to the same value (1) is created by binding `imm` to a new variable, `immAlias`; this is safe because the shared value (1) cannot be changed due to absence of any mutable variable, hence reading through either variable yields the same value. In the third line, however, this would be invalidated, therefore our system raises a compile-time error. The next line does not type-check either, due to the lack of a hidden but required implicit parameter in the setter method that is only available for mutable variables.

The burden of the CPS can be eliminated with the help of compiler plug-ins and macros; evidence that this is feasible is implementation of `break`/`return` as part of the Scala library, and polymorphic delimited continuations as a compiler plug-in [7].

4.3 Design

In order to simplify our implementation, as well as the reasoning behind it, we break our system into two levels: the *library* level, which enforces the rules but still provides escape hatches (similar to the usage of `unsafe` in Rust); and the *meta* level, which enforces that unsafe workarounds are prohibited. Of course, both of these are enforced statically, albeit using different facilities. In the former, we rely on an extended Scala type system that is proven to be sound, and enforce the CPS-style let bindings to make variable bindings explicit. In the latter, we use Scala Macros [158] to disallow certain syntactic patterns when binding to variables, and override the usual behavior of assignments using Scala-Virtualized [116] to confine the aliasing only to occur through the facilities introduced by our library.

4.3.1 Library-level design (core Scala)

For variables of type `T`, we introduce a wrapper type `Var[T,A]`, where `A` is either of following types: `Mut[T]` or `Imm[T]`, depending on whether the binding is mutable (and thus also reassignable) or immutable, as follows:

```
class Mut[T]
class Imm[T]
class Var[T,A](private var v: T) {
  def value = v
  def value_=(v2: T)(implicit ev: A =:= Mut[T]) = v = v2
}
```

In order to prevent mutability due to reassignments, we enable the setter method `value_=` only if type parameter `A` is `Mut[T]`; i.e., if the variable is mutable. (This is done by requiring an implicit evidence that `A` is the same type as `Mut[T]`, which exists in instantiation `Var[T,Mut[T]]`.)

Next, we introduce the `bind*` methods that bind a literal value or an existing variable to a new variable. In the case of immutable binding, it is safe to pass not only a literal value but also an existing immutable variable. However, aliased (shared) variable bindings are permitted only if none of them is mutable. Therefore,

we disallow conversion of variables from immutable to mutable by ensuring that access type parameter `A` is invariant and/or types `Mut[T]` and `Imm[T]` are unrelated, which invalidates the subtyping relationship `Var[V,Imm[S]] <: Var[V,Mut[T]]`, for all types `S`, `T` and `V`, and in turn disallows upcasting an immutable variable to a mutable one.

Additionally, we need to prevent creation of shared mutable variables, which we achieve by using second-class values [159]. Our second-class values cannot be stored in mutable variables, they cannot be returned from functions, and they cannot be accessed by first-class (named or anonymous) functions through free variables. These rules are statically enforced through our existing compiler plug-in, thereby ensuring that second-class values have stack-based lifetimes. The trick is to introduce mutable variables as second-class but require their sources to be first-class, as follows:

```
def bindMut[T, U](r: Var[T,Mut[T]])(
  @local f: Var[T,Mut[T]] -> U) = f(r)
def bindImm[T, U](@local r: Var[T,Imm[T]])(
  @local f: Var[T,Imm[T]] -> U) = f(new Var[T,Imm[T]](r.value))
```

The `A -> B` denotes a function in which the parameter of type `A` is second-class but the return type is first-class; i.e., `(@local A) => B`, where `@local` annotation denotes a second-class type. More specifically, introducing variables as second-class restricts their lifetime to the enclosing scope defined by the passed closure, e.g.,

```
bindImm(42) { x =>
  bindMut(0) { y =>
    y.value = x
  }
} // x cannot be returned/stored as a regular (1st-class) value
```

The function parameter must also be second-class to allow the usage of `x` in an inner closure, such as in the line `y.value = x`. (Informally, using free second-class values lifts the closure to second-class, and first-class values can be promoted to second-class values but not vice versa.)

Lastly, we introduce bridge methods to create variables out of literals so that the above snippet actually type-checks. To be as close to Rust as possible, we treat values as immutable by default, and mutable only when required to appease the type checker or explicitly requested. In Scala, this can be done automatically through unambiguous

implicit conversions from values of type `T` to variables of type `Var[T,A]`, such that the conversion to mutable variables has less priority. We achieve this by declaring an implicit conversion to a mutable variable in a supertype, which is searched after the corresponding `Var` companion object, as follows:

```
class LowPrioMut
object LowPrioMut {
  implicit def valToMut[T](v: T): Var[T,Mut[T]] =
          new Var[T,Mut[T]](v)                          }
object Var extends LowPrioMut {
  implicit def valToImm[T](v: T): Var[T,Imm[T]] =
          new Var[T,Imm[T]](v)                          }
```

4.3.2 Meta-level design (Scala Macros and Scala-Virtualized)

What remains to enforce that `ref.value` for any variable `ref` is not inadvertently passed to `bindMut` or `bindImm`, which would bypass the above type-checking rules in cases such as the following ones, respectively:

```
bindMut(123) { ref =>
  bindImm(ref.value) { imm => ... /* ouch */  }
}
bindImm("foo") { ref =>
  bindMut(ref.value) { mut => ... /* ouch */ }
}
```

To prevent this, we hide methods `bindImm` and `bindMut`, instead encouraging the usage of `let` and `letMut` macros. These macros statically check that either another variable or an r-value—such as `123` or `new StringBuilder()`—other than `ref.value`, for any `ref` of type `Var[_,_]`, is passed; otherwise, it raises a compile error. Such a syntactic inspection is performed by straightforward pattern matching on the AST of the first argument passed to `let(Mut)`. (The second argument is a closure, as in the case of `bind*`.) Similarly, we disallow assignment of non-wrapped values to local variables by overriding `__newVar` and `__assign` in Scala-Virtualized. Hence, none of the following attempts type-check anymore:

```
letMut(123) { ref =>
  letMut(ref.value) { mut => ... }  // error (Var.value as arg.)
  var indirect = mut.value          // error (Var.value in assign.)
  let(indirect) { imm => ... }      // error (not an r-value/Var)
}
```

4.4 Borrowing

In Rust, it is possible to temporarily use a value without necessarily transferring the ownership (e.g., passing it as a function argument) and regaining it afterwards (e.g., after returning a uniquely owned passed argument). This is called *borrowing*, and includes taking a reference.

With the above API in place, we can model borrowing; i.e., permit *temporary* aliasing for the duration of a method call (or an inner scope). It suffices to a turn function parameter (or a local variable) into a second-class variable wrapper, for example:

```
def doWithBorrowed[T](@local ref: Var[T,Mut[T]]) = ...
bindMut(new MutableObject()) { mut =>
  ...
  doWithBorrowed(mut)
  ...
  { @local val borrowed = mut  // requires @local to type-check
    ...
  }
}
```

Unlike Rust, errors are detected at declaration sites (i.e., within methods that borrow) instead of use sites (i.e., where the borrows occur) in our system. This suggests possible benefits in terms of ease of use similar to the ones of declaration variance in Scala vs use-site variance in Java.

For performance and convenience of a reduced number of changes to the existing functions when all parameters are immutable, we introduce wrappers,

```
def call[T, R](f: T -> R)(@local ref: Var[T,_])
def call[T1,T2, R](f: (T1,T2) -> R)(
  @local ref1: Var[T1,_])(@local ref2: Var[T2,_])
...
```

which unwrap the pointed-to values and pass them as second-class arguments to a function. Pure functions that do not store parameters can have all their parameters annotated as second-class, and our system can statically check that they indeed store no values and thus not create any permanent aliases, which could be unsafe (or unexpected) if the arguments are borrowed through a mutable reference, for example:

```
def storeMut(@local sb: StringBuilder,
             @local store: Store): String = {
  store.field = sb  // error (cannot store 2nd-class/borrowed)
  sb.toString
}
bindMut(new Store()) { storeThatLeaksMutable =>
  bindMut(new StringBuilder()) { sb =>
    val s = call(storeMut)(sb)(storeThatLeaksMutable)
    sb.value.append("brakes encapsulation")
    assert(s == storeThatLeaksMutable.field)  // would fail
} }
```

4.5 Conclusion and future work

We presented a minimalistic design for statically enforcing Rust-like notion of borrowing and alias control for references, which prevents various bugs in concurrent and sequential settings alike, but without putting a burden of appeasing a flow-sensitive type checking on the programmer as Rust does. Therefore, our system is both practical and integrates well with the existing context-insensitive type system with local type inference rules, in particular, Scala. Moreover, our approach requires only a minor extension to Scala's (or another context-insensitive) type system—a support for second-class values—and the code is mostly self-contained in this chapter (the only exceptions are the macros and Scala-Virtualized method overload in Section 4.3.2).

In the future, we plan to further investigate how to precisely model ownership (transfer) and lifetimes of bound values, perhaps using state-of-the-art capability-based approaches [124, 160] in conjunction with subtyping rules for a generalization of second-class values (i.e., a privilege lattice) [159]. Another promising direction is static resource management using ownership tracking, which would give rise to Scala Native and off-heap libraries, to avoid unnecessary performance overhead and latencies due to JVM garbage collection in the light of some previous approaches [118]. In either case, we would like to employ data views from Chapter 3 to support a fine-grained access control—Rust has it at the data structure level (i.e., a reference), while we could have it at the view level—thus statically enforcing safe decomposition patterns such as splitting a reference to a view into several references to subviews.

5 PERFORMANCE GAINS USING SECOND-CLASS VALUES

This chapter exploits the fact that second-class values have stack-bounded lifetimes in order to provide performance benefits by differentiating their value representation from the one of first-class values. More precisely, the idea is to allocate second-class values on the stack as opposed to the heap, which is the memory coordinated by a memory allocator (typically the `libc` library) or by the operating system directly (through system calls (`s`)`brk` and `memmap` on Linux). Previous studies show that such trade-offs yield noticeable gains not only for native-code compiled languages such as C [161] but also for object-oriented languages that run on the JVM [60, 61, 63–65] as well as the ones with closures [62].

5.1 Choosing the right Scala subset

One could imagine modifying the Java bytecode to support allocation of second-class values on the stack, and propagating this information through the Scala and Java compiler. Another, more feasible approach that we went with instead is to build upon a simpler compiler and/or virtual machine that supports only a small but powerful subset of Scala—MiniScala. The MiniScala language and compiler was implemented by Grégory Essertel based on the L_3 compiler developed by Michel Schinz from École polytechnique fédérale de Lausanne (EPFL). It supports features such as higher-order functions and subtyping with a limited sort of parametric polymorphism; the variance is defined only for built-in data types, and there are no user-defined types, objects or classes. Next, we describe our extension of MiniScala with second-class values, namely MiniScala2.

5.1.1 Syntax extensions

We extend the MiniScala syntax so that second-class annotations can be be provided in a (mutable) variable declaration, before a parameter name, or after a type name (as with annotations on types in Scala). For example,

```
<annotation> val constant = 42;
<annotation> var variable = constant;
def foo(<annotation> array: Array[Int])
def bar(<annotation> list: List[Array[Int] <annotation>])
```

where `<annotation>` is optional, defaulting to `@local[Nothing]` (first-class), but any phantom type (within brackets following `local`) other than `Nothing` denotes second-class (a proper subtype of `Any` denotes a weaker privilege, see Section 2.3.2). There is a subtlety in the last case; the annotation associated with the list element type is needed to distinguish a partially stack-allocatable second-class container with pointers to arrays on heap (`List[Array[Int]]`) from the one fully allocated on stack (`List[Array[Int] @local[Any]]`), for instance. As in Chapter 2, `@local` is a shorthand for `@local[Any]`.

Note: In our implementation, due to limitations of the preexisting MiniScala parser, we require the annotation *after* a variable's identifier instead of before the `var`/`val` keyword or parameter name, and we require the phantom type (denoting classiness) to be the annotation name; e.g., `var variable @Any = 42`. Nevertheless, we will henceforth be using the previously established syntax to be consistent with the more standard Scala annotations syntax used in the previous chapters.

5.1.2 Semantics of second-class constructs

Primitive types

Primitive types in MiniScala2 are `Int` (integer), `Char` (byte, ASCII character), `Boolean`, and `Unit`. A notable departure from Scala is that these types are never boxed; they are instead represented as *tagged values* and distinguished from pointers to allocated blocks of memory (used for `var`s and non-primitive types) by their unique

prefix with respect to least significant bits (LSBs) in a word. An integer always has its LSB set, thus it can represent only half the range of Scala's `Int`; its actual value is obtained by a single arithmetic shift right. Other primitive types use unique prefixes with the LSB unset, differing in the next few LSBs, but otherwise fit in a single byte.

Mutable variables

In the case of a `var` declaration, the second-class annotation applies to the reference (i.e., a data structure), not the referencing value. More precisely, we desugar variable stores and reads into operations on its reference (i.e., `ref.assign(rhs)` and `ref.get`, respectively), and we type-check the parameter and return type, respectively, as first-class by design. Consequently, no second-class value can ever be stored in a mutable variable, which establishes the same properties described in Chapter 2. (This is not limiting, since first-class values can always be coerced to the more general, second-class values.) However, the reference itself (`ref`) can be first-class or second-class (of arbitrary privilege), depending on how it is used.

An important case where a variable cannot be second-class is when it is used by returned/stored generators or mutable class-like objects. Here is a simple example:

```
def mkCounter(limit: Int, by: Int) = {
  var count = 0;
  def counter(): Boolean = {
    count = count + by;
    count != limit
  };
  counter  // returned (or stored into another var/array)
};
```

In the above snippet, if the `count` variable were second-class, then the `counter` function would have to be second-class, too (as `count` is free). However, if the `counter` is returned or ultimately stored in a mutable structure, then it cannot be second-class.

Arrays

Arrays remain mutable in MiniScala2, therefore the same restrictions as for mutable variable (`var`) apply; this can be intuitively justified by treating an array as multiple `var`s. The type checker disallows types such as `Array[(Int,Int) @local[Any]]`.

Fortunately, element types are often primitives, which renders arrays fully stack-allocatable in many cases. This will be seen in the `fannkuchredux` benchmark; even though it manipulates several arrays, all of them are arrays of integers (primitives) and thus allocated on stack in the optimized version (`fannkuchredux2`).

Pairs

A pair is an immutable data structure comprising two elements of possibly different types. If a pair is allocated on stack, then any of its two elements, `_1` and `_2`, can be first- or second-class, as pointers to data on heap may safely be stored on the stack. Conversely, a heap-allocated pair must not contain (pointers to) stack-allocated data because such a pair could escape the lexical scope and thus contain dangling pointers. For that reason, our type checker rejects any first-class value with second-class components (i.e., pointers to data on stack), including:

- `val pair: (Int @local, Int @local)` (pair itself is not second-class)
- `def fn(lst: List[(Int @local, Char) @local])` (list is not second-class)

Lists

Unlike `Array`, `List` is immutable in MiniScala2, thus it may be completely stack-allocated regardless of its element type. For instance, the snippet

```
@local val lst2: List[Array[Int] @local] =
  new Array[Int](2) :: new Array[Int](3) :: Nil;
```

ensures allocation of `lst2` and its elements on stack, but with the restriction that both its elements (via `.head`) and sublists (via `.tail`) are always typed as second-class. Conversely, if we omit the `local` annotation associated with the element type, then

the elements may also be typed as first-class (even though the `tail` sublist remains second-class), since we essentially have a stack-allocated data structure with pointers to heap-allocated array objects. Finally, omitting both annotations makes both the list and its elements allocated on the heap, but offers increased flexibility because its sublists are available as first-class values.

Strings

`String` is merely syntactic sugar for `Array`, thus string literals are ultimately translated into a series of array updates followed by a return of such an array. In MiniScala2, however, the desugaring has to be done after the type-checking phase in order to support second-class literals, e.g., `@local val const = "TEXT"`.

5.2 Experimental results

We ported a series of benchmarks from Benchmarks Game[1] and Scala Native[2]. The former is compelling because it was used in case studies in peer-reviewed literature [157, 162–165]. The latter is also justifiable because Scala Native shares much of the same goal as we do; supporting cheap allocation of objects on par with languages such as C or Rust, which compile to native code.

Each benchmark has a correctness-checking logic (typically comparing a hash of computed values) that has insignificant impact on the program performance (CPU time and memory used). We did not change the algorithms from the reference implementation, and—with the aim of being even more convincing—we did not choose a particular implementation because of its optimization opportunities. Nevertheless, we sometimes had to change the code style, such as rewriting a function that returns a value to pass it to a continuation as a parameter (i.e., inversion of control) or using recursion in place of loops, so that we can utilize second-class values, e.g.,

[1] `https://benchmarksgame.alioth.debian.org/`
[2] `https://github.com/scala-native/scala-native`

```
withBinarySearch(lo, hi) { result /* 2nd-class */ => ... }
```

with the definition:

```
def withBinarySearch(@local lo: Big, @local hi: Big)(
  @local ret: (@local Big) => U  // return continuation
): U = {  // find the biggest quotient lhs / rhs between lo and hi
  if (lo < hi) {
    with+(hi, one) { hiPlus1 =>  // intermediate results
      with+(lo, hiPlus1) {       // can be 2nd-class, too
        with/2(_) { mid =>
          with*(mid, rhs) { product =>
            if (product <= lhs)
              withBinarySearch(mid, hi)(ret)
            else with-(mid, one) { hiNext =>
              withBinarySearch(lo, hiNext)(ret)
            }
    } } } }
  } else ret(lo)  // pass the result back to the caller
}
```

The above snippet implements the same binary search algorithm that could be implemented with a while-loop. However, because intermediate arbitrary-precision integer values are never returned nor stored in a mutable variable, they can now be second-class and, in turn, *stack-allocated*. (A value that is cheap to allocate may still be returned from any `with*`-like block, including the one where the `result` is used.)

The breakdown of the topics covered by each benchmark is shown below:

suite	benchmark	topics
Benchmarks Game	fannkuchredux	arrays, primitive types
	pidigits	big-number arithmetic, arrays, recursion
Scala Native	bounce	higher-order functions, generators, factory
	list	lists, tail recursion
	storage	nested arrays (trees)
	towers	recursion, arrays

The ported benchmarks are available as open-source code[3] in the baseline version (i.e., without second-class value annotations) as well as the annotated version that is optimized for stack allocation without sacrificing the asymptotic running time.

[3]The modified code can be found at the following URLs:
`https://github.com/losvald/benchmarks-game/tree/phd-thesis`
`https://github.com/losvald/scala-native/tree/phd-thesis`

All measurements were made on a 4-core Intel i7-5600U machine, running at 2.6GHz with the GNU/Linux 4.4.0 (x86_64) kernel and the Java 1.8.0_151. (The MiniScala2 interpreter used to measure memory usage was written in and compiled with Scala 2.12.3, but that does not matter as we are executing the JVM bytecode.)

5.2.1 Memory allocation

In each benchmark, we measured the amount of both heap and stack memory used by the program on a medium-sized input (i.e., long enough that it takes a few seconds for a program to complete). The results are shown in Table 5.1. The two benchmarks where the improvement in memory usage is asymptotically significant are `pidigits` and `list`. In the former case, rewriting the code in recursive and continuation-passing style enabled all dynamic allocations of big integers to be replaced by allocation of second-class arrays of integers, which happens to be completely on stack despite the first-classiness of element types (integers) due to tagged value representation in MiniScala. In the latter case, the code was already recursive but manipulates immutable lists, which can be completely stack-allocated regardless of the actual element type, since a second-class value cannot escape through immutable stores. Another noteworthy result is for the `bounce` benchmark, in which neither the variables that are part of a (mutable) generator state nor the higher-order functions using them could be made second-class because the former are stored in an array, therefore we see only slight improvements there. The `storage` benchmark shows no improvements at all because of the same reason, although this is amplified by the frequency of such stores. Other benchmarks do not show noticeable gains.

Table 5.1.
Memory profile for baseline and annotated (suffix "2") benchmarks. Both heap and stack memory is represented as fractions, with each numerator and denominator being the total allocation amount in bytes (B) and count (#), respectively. Computing these fractions yields *average* allocation size, which is a valuable for predicting if the faster stack allocation may have sufficient impact on decreasing the CPU time. Another necessary precondition is that the number of allocations grows similarly ($\sim$) to the running time $T(N)$, where N is the input size, therefore this information is also presented using asymptotically tight bounds (Θ).

suite	benchmark	heap (B/#)	stack (B/#)	stack-alloc. fraction
Benchmarks Game	fannkuchredux	$\frac{60588}{15129} \sim \Theta(1)$	$0 \sim \Theta(1)$	0
	fannkuchredux2	$0 \sim \Theta(1)$	$\frac{60588}{15129} \sim \Theta(1)$	1
	pidigits	$\frac{30852336}{2302070} \sim \Theta(N)$	$0 \sim \Theta(1)$	0
	pidigits2	$\frac{500}{33} \sim \Theta(1)$	$\frac{30761576}{2279472} \sim \Theta(N)$	**0.99998**
Scala Native	bounce	$\frac{24232}{5557} \sim \Theta(N)$	$0 \sim \Theta(1)$	0
	bounce2	$\frac{4208}{553} \sim \Theta(N)$	$\frac{20024}{5004} \sim \Theta(N)$	0.82635
	list	$\frac{22732}{5655} \sim \Theta(N)$	$0 \sim \Theta(1)$	0
	list2	$0 \sim \Theta(1)$	$\frac{42408}{2144} \sim \Theta(N)$	**1**
	storage	$\frac{139060}{8192} \sim \Theta(N)$	$0 \sim \Theta(1)$	0
	storage2	$\frac{117220}{6827} \sim \Theta(N)$	$\frac{21844}{1336} \sim \Theta(N)$	0.15708
	towers	$\frac{84}{6} \sim \Theta(1)$	$0 \sim \Theta(1)$	0
	towers2	$0 \sim \Theta(1)$	$\frac{84}{6} \sim \Theta(1)$	1

5.3 Conclusion and future work

We empirically show that differentiation between first- and second-class values also yields performance gains in a subset of Scala: our enhanced version of MiniScala. First, we have extended its type checker according to the semantics of second-class values presented in Chapter 2, including their generalization to a 3-level privilege lattice (`@local[Nothing]`, `@local[Any]`, and `@local[P]` for $\texttt{P} \notin \{\texttt{Nothing}, \texttt{Any}\}$). Second, we have propagated this information to the CPS interpreter and modified the allocation scheme accordingly; memory blocks that hold mutable variables and non-primitive types (functions, arrays, lists and pairs) may now be stack-allocated as opposed to their default allocation on heap if they are first-class. Finally, we demonstrate that such gains are significant by measuring the amounts and ratios of heap-allocated memory that can be traded off for the less expensive stack memory, once the appropriate values are classified as second-class.

5.3.1 Future work

Measuring CPU time If we compile MiniScala to C or assembly code, then the allocations for keeping second-class values in memory are significantly faster as they do not go through a memory allocator (and the operating system). Consequently, programs which trade heap allocations for stack allocations are expected to be faster, albeit not significantly unless their heap memory usage is comparable to their running time.

The best example of this is the `pidigits` benchmark—which computes the first N digits of π—and performs $\Theta(N)$ additions, multiplications and divisions on integers with $\Theta(N)$ digits. Each addition runs in $\Theta(N)$ time and allocates $\Theta(N)$ memory. The multiplication runs in $\mathcal{O}(N^2)$ time but only consumes $\Theta(N)$ heap. However, since the division is implemented as a binary search involving multiplications and additions, it consumes $\Theta(N \log N)$ heap and is dominant in each iteration of the algorithm. Since N is only up to a few thousands, the asymptotic ratio of running time and heap

allocation size is not significant enough to hide improvements in CPU time saved by performing stack allocation in place of those heap allocations. (Alternatively, we could rewrite the multiplication using the Karatsuba's algorithm and division using the Burnikel and Ziegler's algorithm [166], which would lower the ratio asymptotically and, if implemented carefully, result in bigger performance gains [167]. This is not against the rules of the Benchmarks Game; in fact, the fastest implementations of this benchmark use the GNU Multiple Precision arithmetic library (GMP), which implements these algorithms.)

6 SUMMARY

In Chapter 2, we have formalized the second-class values—a generalization of second-class functions from Algol and Pascal—and developed a programming model where first- and second-class objects objects of the same kind can peacefully coexist, demonstrating that such provides extended static checking for a number of challenging and diverse programming tasks. In Chapter 3, we have generalized the concept of the reference and decoupled it from the underlying data structure, providing uniform treatment between substructures and different representations while keeping the performance of highly-optimized (often domain-specific) data structures. The reference has been further improved in Chapter 4 with respect to restrictions on unsafe mutable aliasing that we learned from Rust and ownership systems with borrowed references, but without bringing the disadvantages of flow-sensitivity, thus enabling statically safe but simpler usage and debugging in other widely used industrial languages such as Scala. Ultimately, we have modified the memory allocation scheme to enable the cheaper allocation on stack (as opposed to heap) for second-class values in a realistic subset of Scala (featuring higher-order functions, mutual recursion, parametric polymorphism and variance with built-in and universal types, lists, mutable variables, arrays, etc.) in Chapter 5, and empirically shown that the gains are significant through measurements on state-of-the-art benchmarks that represent practical workloads.

The work in this dissertation has been rigorously peer-reviewed; Chapters 2–4 are only slightly expanded (elaborated) versions of the scholarly articles published in prominent conference and workshop proceedings. Not only was Chapter 2 published and presented in the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'16); its experimental evaluation won OOPSLA's Distinguished Artifact Award. Chapter 3 was published and presented in the International Workshop on Libraries, Languages, and Compilers for Array Pro-

gramming (ARRAY'17) [168]. Chapter 4 was published and presented[1] in the International Symposium on Scala (Scala'17) [169]. Chapter 5 is unpublished at the time, but it nonetheless does build upon solid foundations: mechanically proved properties of second-class values [159] as well as methodologies used in my previously published evaluation of R (another functional language) on the same Benchmark Game suite in the European Conference on Object-Oriented Programming (ECOOP'12) [157, 162].

We hope that this work will be useful for advancing the state-of-the-art research not only in programming languages and compilers but also real-world applications, engineering and experimental analysis of algorithms. Around the time of writing, authors of several related works [170–172] have expressed interest in our latest publications on data views and Rust-like borrowed references in Scala, and the earlier work on second-class values has just recently started to accrue citations. Based on the citation count of the paper on evaluating the performance of R, we have reasons to believe that publishing the last chapter would be impactful due to the similarity of R and Scala, especially given that Scala is becoming more popular than R nowadays.

[1] `https://www.youtube.com/watch?v=sIan12EQoFM`

REFERENCES

REFERENCES

[1] John Hogg. Islands: Aliasing protection in object-oriented languages. In *Proceedings of the sixth ACM Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '91, pages 271–285, New York, NY, USA, 1991. ACM.

[2] Karl Naden, Robert Bocchino, Jonathan Aldrich, and Kevin Bierhoff. A type system for borrowing permissions. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 557–570, New York, NY, USA, 2012. ACM.

[3] Nicholas D. Matsakis and Felix S. Klock, II. The Rust language. *Ada Letters*, 34(3):103–104, October 2014.

[4] Tiark Rompf and Nada Amin. Type soundness for dependent object types (DOT). In *Proceedings of the 31st ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '16, pages 624–641, New York, NY, USA, 2016. ACM.

[5] Tiark Rompf and Martin Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. *Communications of the ACM*, 55(6):121–130, 2012.

[6] Alexis Beingessner and Steve Klabnik. The Rustonomicon: The dark arts of advanced and unsafe Rust programming. `https://doc.rust-lang.org/nomicon/references.html`, 2016.

[7] Tiark Rompf, Ingo Maier, and Martin Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 317–328, New York, NY, USA, 2009. ACM.

[8] Christopher Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(1/2):11–49, 2000.

[9] Joel Moses. The function of function in LISP or why the funarg problem should be called the environment problem. *ACM Sigsam Bulletin*, (15):13–27, 1970.

[10] Joseph Weizenbaum. The funarg problem explained. Technical report, MIT, Cambridge, MA, 1968.

[11] Anindya Banerjee and David A. Schmidt. Stackability in the simply-typed call-by-value lambda calculus. *Science of Computer Programming*, 31(1):47–73, 1998.

[12] John Hannan. A type-based escape analysis for functional languages. *Journal of Functional Programming*, 8(3):239–273, 1998.

[13] Walid Taha and Michael F. Nielsen. Environment classifiers. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pages 26–37, New York, NY, USA, 2003. ACM.

[14] Éric Tanter. Beyond static and dynamic scope. In *Proceedings of the fifth Symposium on Dynamic Languages*, DLS '09, pages 3–14, New York, NY, USA, 2009. ACM.

[15] Mark S. Miller. The E language. `http://erights.org/elang/index.html`, 1998.

[16] Tomas Petricek, Dominic A. Orchard, and Alan Mycroft. Coeffects: A calculus of context-dependent computation. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 123–135, New York, NY, USA, 2014. ACM.

[17] Martin Odersky. Scala - Where it came from, where it is going. `http://www.slideshare.net/Odersky/scala-days-san-francisco-45917092`, 2015.

[18] David Gay. *Memory management with explicit regions*. Ph.D. dissertation, Department of Computer Science, Stanford University, Berkeley, CA, USA, 1997.

[19] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 47–57, New York, NY, USA, 1988. ACM.

[20] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James W. O'Toole. Report on the FX programming language. Technical report, MIT/LCS/TR-531, 1992.

[21] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2:245–271, June 1992.

[22] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. In *Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science*, LICS '92, pages 162–173, 1992.

[23] Ben Lippmeier. *Type Inference and Optimisation for an Impure World*. Ph.D. dissertation, Australian National University, 2010.

[24] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, pages 188–201, New York, NY, USA, 1994. ACM.

[25] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld, and Olesen Peter Sestoft. Programming with regions in the MLKit (revised for version 4.3.0). Technical report, IT University of Copenhagen, January 2006.

[26] Jeremy G. Siek, Michael M. Vitousek, and Jonathan D. Turner. Effects for funargs. *Computing Research Repository*, abs/1201.0023, 2012.

[27] David K. Gifford and John M. Lucassen. Integrating functional and imperative programming. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pages 28–38, New York, NY, USA, 1986. ACM.

[28] James Gosling, Bill Joy, Guy L. Steele, Jr., Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 7 Edition*. Addison-Wesley Professional, 2013.

[29] David J. Pearce. JPure: A modular purity system for Java. In *Compiler Construction*, volume 6601 of *Lecture Notes in Computer Science*, pages 104–123, Germany, 2011. Springer Berlin Heidelberg.

[30] Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. Semantics of transactional memory and automatic mutual exclusion. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 63–74, New York, NY, USA, 2008. ACM.

[31] Daniel Marino and Todd Millstein. A generic type-and-effect system. In *Proceedings of the fourth International Workshop on Types in Language Design and Implementation*, TLDI '09, pages 39–50, New York, NY, USA, 2009. ACM.

[32] Lukas Rytz, Nada Amin, and Martin Odersky. A flow-insensitive, modular effect system for purity. In *Proceedings of the 15th Workshop on Formal Techniques for Java-like Programs*, FTfJP '13, pages 4:1–4:7, New York, NY, USA, 2013. ACM.

[33] Lukas Rytz, Martin Odersky, and Philipp Haller. Lightweight polymorphic effects. In *ECOOP 2012 — Object-Oriented Programming: 26th European Conference. Proceedings*, volume 7313 of *Lecture Notes in Computer Science*, pages 258–282, Germany, 2012. Springer Berlin Heidelberg.

[34] Flemming Nielson and Hanne Riis Nielson. Type and effect systems. In Ernst-Rüdiger Olderog and Bernhard Steffen, editors, *Correct System Design*, volume 1710 of *Lecture Notes in Computer Science*, pages 114–136. Springer Berlin Heidelberg, Germany, 1999.

[35] Martin Odersky and Tiark Rompf. Unifying functional and object-oriented programming with Scala. *Communications of the ACM*, 57(4):76–86, 2014.

[36] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, July 1991.

[37] Philip Wadler. The essence of functional programming. In *Proceedings of the 19th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, pages 1–14, New York, NY, USA, 1992. ACM.

[38] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, pages 71–84, New York, NY, USA, 1993. ACM.

[39] Edwin Brady. Programming and reasoning with algebraic effects and dependent types. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, New York, NY, USA, 2013. ACM.

[40] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 333–343, New York, NY, USA, 1995. ACM.

[41] Philip Wadler. The marriage of effects and monads. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, pages 63–74, New York, NY, USA, 1998. ACM.

[42] Robert Atkey. Parameterised notions of computation. *Journal of Functional Programming*, 19(3-4):335–376, 2009.

[43] Ross Tate. The sequential semantics of producer effect systems. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 15–26, New York, NY, USA, 2013. ACM.

[44] Oleg Kiselyov and Chung-chieh Shan. Lightweight static capabilities. *Electronic Notes in Theoretical Computer Science*, 174(7):79–104, 2007.

[45] Oleg Kiselyov and Chung-chieh Shan. Lightweight monadic regions. In *Haskell*, pages 1–12, New York, NY, USA, 2008. ACM.

[46] Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming*, 84(1):108–123, 2015.

[47] Gordon D. Plotkin and Matija Pretnar. Handling algebraic effects. *Logical Methods in Computer Science*, 9(4), 2013.

[48] Philip Wadler. Linear types can change the world! In *Proceedings of the IFIP Working Group 2.2/2.3 Working Conference on Programming Concepts and Methods*. North-Holland, 1990.

[49] Erik Barendsen and Sjaak Smetsers. Conventional and uniqueness typing in graph rewrite systems. In *Foundations of Software Technology and Theoretical Computer Science*, volume 761 of *Lecture Notes in Computer Science*, pages 41–51, Germany, 1993. Springer Berlin Heidelberg.

[50] Tachio Terauchi and Alex Aiken. Witnessing side-effects. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming*, ICFP '05, pages 105–115, New York, NY, USA, 2005. ACM.

[51] Daan Leijen. Koka: A language with effect inference. `http://research.microsoft.com/en-us/projects/koka/2012-overviewkoka.pdf`, April 2012.

[52] Kevin J. Brown, Arvind K. Sujeeth, HyoukJoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. A heterogeneous parallel framework for domain-specific languages. In *20th International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 89–100, Washington, DC, USA, 2011. IEEE Computer Society.

[53] Arvind K. Sujeeth, HyoukJoong. Lee, Kevin J. Brown, Tiark Rompf, Michael Wu, Anand R. Atreya, Martin Odersky, and Kunle Olukotun. OptiML: an implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on Machine Learning*, ICML, 2011.

[54] HyoukJoong Lee, Kevin J. Brown, Arvind K. Sujeeth, Hassan Chafi, Tiark Rompf, Martin Odersky, and Kunle Olukotun. Implementing domain-specific languages for heterogeneous parallel computing. *IEEE Micro*, 31(5):42–53, 2011.

[55] Arvind K. Sujeeth, Tiark Rompf, Kevin J. Brown, HyoukJoong Lee, Hassan Chafi, Victoria Popic, Michael Wu, Aleksander Prokopec, Vojin Jovanovic, Martin Odersky, and Kunle Olukotun. Composition and reuse with compiled domain-specific languages. In *ECOOP 2013 — Object-Oriented Programming: 27th European Conference. Proceedings*, volume 7920 of *Lecture Notes in Computer Science*, pages 52–78, Germany, 2013. Springer Berlin Heidelberg.

[56] Arvind K. Sujeeth, Austin Gibbons, Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Martin Odersky, and Kunle Olukotun. Forge: Generating a high performance DSL implementation from a declarative specification. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*, GPCE '13, New York, NY, USA, 2013. ACM.

[57] Tiark Rompf, Kevin J. Brown, HyoukJoong Lee, Arvind K. Sujeeth, Manohar Jonnalagedda, Nada Amin, Georg Ofenbeck, Alen Stojanov, Yannis Klonatos, Mohammad Dashti, Christoph Koch, Markus Püschel, and Kunle Olukotun. Go meta! A case for generative programming and DSLs in performance critical systems. In *First Summit on Advances in Programming Languages — SNAPL 2015*, volume 32 of *Leibniz International Proceedings in Informatics (LIPIcs '15)*, pages 238–261. Schloss Dagstuhl — Leibniz-Zentrum für Informatik, 2015.

[58] Tiark Rompf. *Lightweight Modular Staging and Embedded Compilers: Abstraction Without Regret for High-Level High-Performance Programming*. Ph.D. dissertation, EPFL, Lausanne, Switzerland, 2012.

[59] Benjamin Goldberg and Young Gil Park. Higher order escape analysis: Optimizing stack allocation in functional program implementations. In *ESOP '90 — Programming: Third European Symposium. Proceedings*, pages 152–160, Germany, 1990. Springer Berlin Heidelberg.

[60] David Gay and Bjarne Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *Compiler Construction*, CC '00, pages 82–93, London, UK, 2000. Springer-Verlag.

[61] Robert Fitzgerald, Todd B. Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi. Marmot: an optimizing compiler for Java. *Software: Practice and Experience*, 30(3):199–232, March 2000.

[62] Andrew W. Appel and Zhong Shao. Empirical and analytic study of stack versus heap cost for languages with closures. *Journal of Functional Programming*, 6(1):4774, 1996.

[63] Xiaoliang Xu and Jiang Shen. Research on stack-allocation based JVM garbage collection. In *Proceedings of the third International Conference on Advanced Computer Theory and Engineering*, volume 2 of *ICACTE '10*, pages 346–349. IEEE, August 2010.

[64] Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. Stack allocation and synchronization optimizations for Java using escape analysis. *ACM Transactions on Programming Languages and Systems*, 25(6):876–910, November 2003.

[65] Peter Molnar, Andreas Krall, and Florian Brandner. Stack allocation of objects in the CACAO virtual machine. In *Proceedings of the seventh International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 153–161, New York, NY, USA, 2009. ACM.

[66] Oren Avissar, Rajeev Barua, and Dave Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM Transactions on Embedded Computing Systems*, 1(1):6–26, November 2002.

[67] Andreas Sewe, Mira Mezini, Aibek Sarimbekov, Danilo Ansaloni, Walter Binder, Nathan Ricci, and Samuel Z. Guyer. new Scala() instance of Java: A comparison of the memory behaviour of Java and Scala programs. In *Proceedings of the 2012 International Symposium on Memory Management*, ISMM '12, pages 97–108, New York, NY, USA, 2012. ACM.

[68] Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder. Da Capo con Scala: Design and analysis of a Scala benchmark suite for the Java Virtual Machine. In *Proceedings of the 26th ACM International Conference on Object-Oriented Programming Systems Languages, and Applications*, OOPSLA '11, pages 657–676, New York, NY, USA, 2011. ACM.

[69] Martin Weiser and Gary Powell. The View Template Library. In *In First Workshop on C++ Template Programming*, 2000.

[70] Thomas Becker. Smart iterators and STL. *C/C++ Users Journal*, 16(9):39–45, September 1998.

[71] Jay Black, Paul Castro, Archan Misra, and Jerome White. Live data views: Programming pervasive applications that use "timely" and "dynamic" data. In *Proceedings of the sixth International Conference on Mobile Data Management*, MDM '05, pages 294–298, New York, NY, USA, 2005. ACM.

[72] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989.

[73] Gerth Stølting Brodal. *Worst Case Efficient Data Structures*. Ph.D. dissertation, Department of Computer Science, Aarhus University, Denmark, 1997.

[74] Friedrich L. Bauer and Hans Wössner. The PlankalkÜL of Konrad Zuse: A forerunner of today's programming languages. *Communications of the ACM*, 15(7):678–685, July 1972.

[75] David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pages 58–71, New York, NY, USA, 2003. ACM.

[76] Perry Cheng and Guy E. Blelloch. A parallel, real-time garbage collector. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 125–136, New York, NY, USA, 2001. ACM.

[77] Fridtjof Siebert. Eliminating external fragmentation in a non-moving garbage collector for Java. In *Proceedings of the 2000 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '00, pages 9–17, New York, NY, USA, 2000. ACM.

[78] Filip Pizlo, Lukasz Ziarek, Petr Maj, Antony L. Hosking, Ethan Blanton, and Jan Vitek. Schism: Fragmentation-tolerant real-time garbage collection. In *Proceedings of the ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation*, PLDI '10, pages 146–159, New York, NY, USA, 2010. ACM.

[79] Jennifer B. Sartor, Stephen M. Blackburn, Daniel Frampton, Martin Hirzel, and Kathryn S. McKinley. Z-rays: Divide arrays and conquer speed and flexibility. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 471–482, New York, NY, USA, 2010. ACM.

[80] Christopher L. Kuszmaul. Efficient merge and insert operations for binary heaps and trees. Technical report, NASA, 2000. `https://www.nas.nasa.gov/assets/pdf/techreports/2000/nas-00-003.pdf`.

[81] Mohamed Y. Eltabakh, Wing-Kai Hon, Rahul Shah, Walid G. Aref, and Jeffrey S. Vitter. The SBC-tree: An index for run-length compressed sequences. In *Proceedings of the 11th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '08, pages 523–534, New York, NY, USA, 2008. ACM.

[82] Daniel H. Larkin. *Compressing Trees with a Sledgehammer*. Ph.D. dissertation, Department of Computer Science, Princeton University, USA, 2016.

[83] Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, SFCS '78, pages 8–21, Washington, DC, USA, 1978. IEEE Computer Society.

[84] Georgy M. Adelson-Velskii and Evgenii M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3:1259–1262, 1962.

[85] Mahdi Amani, Kevin A. Lai, and Robert E. Tarjan. Amortized rotation cost in AVL trees. *Information Processing Letters*, 116(5):327–330, 2016.

[86] Siddhartha Sen and Robert E. Tarjan. Deletion Without Rebalancing in Balanced Binary Trees. In *Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '10, pages 1490–1499, Philadelphia, PA, USA, 2010. SIAM.

[87] Siddhartha Sen, Robert E. Tarjan, and David Hong Kyun Kim. Deletion without rebalancing in binary search trees. *ACM Transactions on Algorithms*, 12(4):57:1–57:31, September 2016.

[88] Christos Levcopoulos and Mark H. Overmars. A balanced search tree with O(1) worst-case update time. *Acta Informatica*, 26(3):269–277, 1988.

[89] Paul Dietz and Rajeev Raman. A constant update time finger search tree. In *Advances in Computing and Information — ICCI '90: International Conference on Computing and Information. Proceedings*, volume 468 of *Lecture Notes in Computer Science*, pages 100–109, Germany, 1990. Springer Berlin Heidelberg.

[90] Ralf Hinze and Ross Paterson. Finger trees: A simple general-purpose data structure. *Journal of Functional Programming*, 16:197–217, 2006.

[91] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Addison-Wesley Professional, Redwood City, CA, USA, 1998.

[92] Daniel D. Sleator and Robert E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, July 1985.

[93] Yehuda Afek, Haim Kaplan, Boris Korenfeld, Adam Morrison, and Robert E. Tarjan. The CB tree: A practical concurrent self-adjusting search tree. *Distributed Computing*, 27(6):393–417, 2014.

[94] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, June 1990.

[95] Paul F. Dietz and Daniel D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, STOC '87, pages 365–372, New York, NY, USA, 1987. ACM.

[96] Gerth Stølting Brodal, Christos Makris, and Kostas Tsichlas. Purely functional worst case constant time catenable sorted lists. In *Algorithms — ESA 2006: 14th Annual European Symposium. Proceedings*, volume 4168 of *Lecture Notes in Computer Science*, pages 172–183, Germany, 2006. Springer Berlin Heidelberg.

[97] Paul F. Dietz. Fully persistent arrays (extended array). In *Proceedings of the Workshop on Algorithms and Data Structures*, WADS '89, pages 67–74, London, UK, 1989. Springer-Verlag.

[98] James R. Driscoll, Daniel D. Sleator, and Robert E. Tarjan. Fully persistent lists with catenation. *Journal of the ACM*, 41(5):943–959, 1994.

[99] Haim Kaplan and Robert E. Tarjan. Persistent lists with catenation via recursive slow-down. In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing*, STOC '95, pages 93–102, New York, NY, USA, 1995. ACM.

[100] Haim Kaplan and Robert E. Tarjan. Purely functional, real-time deques with catenation. *Journal of the ACM*, 46(5):577–603, 1999.

[101] Chris Okasaki. *Purely Functional Data Structures*. Ph.D. dissertation, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, 1996.

[102] Chris Okasaki. Purely functional random-access lists. In *Proceedings of the seventh International Conference on Functional Programming Languages and Computer Architecture*, volume 33 of *FPCA '95*, pages 86–95, New York, NY, USA, 1995. ACM.

[103] Haim Kaplan, Chris Okasaki, and Robert E. Tarjan. Simple confluently persistent catenable lists. In *Proceedings of the sixth Scandinavian Workshop on Algorithm Theory*, SWAT '98, pages 119–130, London, UK, 1998. Springer-Verlag.

[104] Nicolas Stucki, Tiark Rompf, Vlad Ureche, and Phil Bagwell. RRB vector: A practical general purpose immutable sequence. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP '15, pages 342–354, New York, NY, USA, 2015. ACM.

[105] Bjarne Stroustrup. Parameterized types for C++. In *USENIX C++ Conference*, pages 1–18, Denver, CO, USA, 1988.

[106] Bjarne Stroustrup. A history of C++: 1979–1991. In *The Second ACM SIGPLAN Conference on History of Programming Languages*, number 1 in HOPL-II, pages 271–297, New York, NY, USA, 1993. ACM.

[107] Jeremy Siek and Walid Taha. A semantic analysis of C++ templates. In *ECOOP 2006 — Object-Oriented Programming: 20th European Conference. Proceedings*, volume 4067 of *Lecture Notes in Computer Science*, pages 304–327, Germany, 2006. Springer Berlin Heidelberg.

[108] Martin J. Cole and Steven G. Parker. Dynamic compilation of C++ template code. *Scientific Programming*, 11(4):321–327, 2003.

[109] Walid M. Taha. *Multistage programming: Its theory and applications*. Ph.D. dissertation, Oregon Graduate Institute of Science and Technology, USA, 1999.

[110] Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2):211–242, 2000.

[111] Krzysztof Czarnecki, John T. O'Donnell, Jörg Striegnitz, and Walid Taha. DSL implementation in MetaOCaml, Template Haskell, and C++. In Christian Lengauer, Don Batory, Charles Consel, and Martin Odersky, editors, *Domain-Specific Program Generation: International Seminar. Revised Papers*, volume 3016 of *Lecture Notes in Computer Science*, pages 51–72. Springer Berlin Heidelberg, Germany, 2004.

[112] Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. Implementing multi-stage languages using ASTs, gensym, and reflection. In *Generative Programming and Component Engineering: Second International Conference — GPCE '03. Proceedings*, volume 2830 of *Lecture Notes in Computer Science*, pages 57–76, Germany, 2003. Springer Berlin Heidelberg.

[113] Tim Sheard and Simon L. Peyton Jones. Template meta-programming for Haskell. *SIGPLAN Notices*, 37(12):60–75, 2002.

[114] Jacques Carette, Oleg Kiselyov, and Chung-Chieh Shan. Finally tagless, partially evaluated. In *Programming Languages and Systems: Fifth Asian Symposium — APLAS '07. Proceedings*, volume 4807 of *Lecture Notes in Computer Science*, pages 222–238, Germany, 2007. Springer Berlin Heidelberg.

[115] Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. Polymorphic embedding of DSLs. In *Proceedings of the seventh International Conference on Generative Programming and Component Engineering*, GPCE '08, pages 137–148, New York, NY, USA, 2008. ACM.

[116] Adriaan Moors, Tiark Rompf, Philipp Haller, and Martin Odersky. Scala-virtualized. In *Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation*, PEPM '12, pages 117–120, New York, NY, USA, 2012. ACM.

[117] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In *ECOOP 1998 — Object-Oriented Programming: 12th European Conference. Proceedings*, volume 1445 of *Lecture Notes in Computer Science*, pages 158–185, Germany, 1998. Springer Berlin Heidelberg.

[118] Tian Zhao, Jason Baker, James Hunt, James Noble, and Jan Vitek. Implicit ownership types for memory management. *Science of Computer Programming*, 71(3):213–241, 2008.

[119] Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. Ownership types: A survey. In Dave Clarke, James Noble, and Tobias Wrigstad, editors, *Aliasing in Object-Oriented Programming*, volume 7850 of *Lecture Notes in Computer Science*, pages 15–58. Springer Berlin Heidelberg, Germany, 2013.

[120] Dave Clarke and Tobias Wrigstad. External uniqueness is unique enough. In *ECOOP 2003 — Object-Oriented Programming: 17th European Conference. Proceedings*, volume 2743 of *Lecture Notes in Computer Science*, pages 176–200, Germany, 2003. Springer Berlin Heidelberg.

[121] Tobias Wrigstad. *Ownership-Based Alias Management*. Ph.D. dissertation, Royal Institute of Technology, Stockholm, Sweden, 2006.

[122] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, pages 311–330, New York, NY, USA, 2002. ACM.

[123] John Boyland. Alias burying: Unique variables without destructive reads. *Software: Practice and Experience*, 31(6):533–553, 2001.

[124] Philipp Haller and Martin Odersky. Capabilities for uniqueness and borrowing. In *ECOOP 2010 — Object-Oriented Programming: 24th European Conference. Proceedings*, volume 6183 of *Lecture Notes in Computer Science*, pages 354–378, Germany, 2010. Springer Berlin Heidelberg.

[125] Philipp Haller. *Isolated Actors for Race-Free Concurrent Programming*. Ph.D. dissertation, EPFL, Lausanne, Switzerland, 2010.

[126] Werner Dietl, Sophia Drossopoulou, and Peter Müller. Generic universe types. In *ECOOP 2007 — Object-Oriented Programming: 21st European Conference. Proceedings*, volume 4609 of *Lecture Notes in Computer Science*, pages 28–53, Germany, 2007. Springer Berlin Heidelberg.

[127] Peter Müller and Arsenii Rudich. Ownership transfer in universe types. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications*, OOPSLA '07, pages 461–478, New York, NY, USA, 2007. ACM.

[128] Philipp Haller and Alex Loiko. LaCasa: Lightweight affinity and object capabilities in Scala. In *Proceedings of the 31st ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '16, pages 272–291, New York, NY, USA, 2016. ACM.

[129] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. Deny capabilities for safe, fast actors. In *Proceedings of the fifth International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE! 2015, pages 1–12, New York, NY, USA, 2015. ACM.

[130] Eric Reed. Patina: A formalization of the Rust programming language. `ftp://ftp.cs.washington.edu/tr/2015/03/UW-CSE-15-03-02.pdf`, 2015.

[131] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the foundations of the Rust programming language. In *Proceedings of the 45th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '18, pages 66:1–66:34, New York, NY, USA, 2018. ACM.

[132] Trevor Jim, Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 275–288. USENIX, 2002.

[133] Tomas Petricek, Dominic Orchard, and Alan Mycroft. Coeffects: Unified static analysis of context-dependence. In *Automata, Languages, and Programming: 40th International Colloquium — ICALP '13. Proceedings*, volume 7966 of *Lecture Notes in Computer Science*, pages 385–397, Germany, 2013. Springer Berlin Heidelberg.

[134] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the ninth USENIX conference on Networked Systems Design and Implementation*, NSDI, 2011.

[135] Heather Miller, Philipp Haller, and Martin Odersky. Spores: A type-based foundation for closures in the age of concurrency and distribution. In *ECOOP 2014 — Object-Oriented Programming: 28th European Conference. Proceedings*, volume 8586 of *Lecture Notes in Computer Science*, pages 308–333, Germany, 2014. Springer Berlin Heidelberg.

[136] Jeremy Siek. Type safety in three easy lemmas. `http://siek.blogspot.ch/2013/05/type-safety-in-three-easy-lemmas.html`, 2013.

[137] Erik Ernst, Klaus Ostermann, and William R. Cook. A virtual class calculus. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 270–282, New York, NY, USA, 2006. ACM.

[138] Nada Amin, Adriaan Moors, and Martin Odersky. Dependent Object Types. In *19th International Workshop on Foundations of Object-Oriented Languages*, FOOL '12, New York, NY, USA, 2012. ACM.

[139] Tiark Rompf and Nada Amin. From F to DOT: Type soundness proofs with definitional interpreters. Technical report, Purdue University, July 2015. http://arxiv.org/abs/1510.05216.

[140] John Boyland. Checking interference with fractional permissions. In *Static Analysis*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72, Germany, 2003. Springer Berlin Heidelberg.

[141] Aleksandar Prokopec, Phil Bagwell, and Tiark Rompf abd Martin Odersky. A generic parallel collection framework. In *Proceedings of the 17th International Conference on Parallel Processing*, Euro-Par '11. Springer Berlin Heidelberg, 2011.

[142] H. Chafi, Z. DeVito, A. Moors, T. Rompf, A. K. Sujeeth, P. Hanrahan, M. Odersky, and K. Olukotun. Language virtualization for heterogeneous parallel computing. In *Proceedings of the 9th Conference for new ideas, new paradigms, and reflections on everything to do with programming and software*, Onward!, 2010.

[143] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun. A domain-specific approach to heterogeneous parallelism. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 35–46, 2011.

[144] M. Püschel, J. M. F. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE*, 93(2):232–275, February 2005.

[145] Georg Ofenbeck, Tiark Rompf, Alen Stojanov, Martin Odersky, and Markus Püschel. Spiral in Scala: Towards the systematic construction of generators for performance libraries. In *Generative Programming: Concepts & Experiences*, GPCE '13, pages 125–134, New York, NY, USA, 2013. ACM.

[146] Kedar N. Swadi, Walid Taha, Oleg Kiselyov, and Emir Pasalic. A monadic approach for avoiding code duplication when staging memoized functions. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '06, pages 160–169, New York, NY, USA, 2006. ACM.

[147] John Rushby. The Bell and La Padula security model. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, USA, 1984. http://www.csl.sri.com/~rushby/papers/blp86.pdf.

[148] Matteo Frigo. A fast Fourier transform compiler. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 169–180, New York, NY, USA, 1999. ACM.

[149] Markus Püschel, Franz Franchetti, and Yevgen Voronenko. Spiral. In David Padua, editor, *Encyclopedia of Parallel Computing*. Springer, 2011.

[150] Timothy A. Davis and Yifan Hu. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software*, 38(1):1:1–1:25, 2011.

[151] Haim Kaplan, Nira Shafrir, and Robert E. Tarjan. Union-find with deletions. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, pages 19–28, Philadelphia, PA, USA, 2002. SIAM.

[152] Stephen Alstrup, Mikkel Thorup, Inge Li Gørtz, Theis Rauhe, and Uri Zwick. Union-find with constant time deletions. *ACM Transactions on Algorithms*, 11(1):6:1–6:28, August 2014.

[153] Daniel H. Larkin and Robert E. Tarjan. Nested set union. In *Algorithms — ESA 2014: 22th Annual European Symposium. Proceedings*, pages 618–629, Germany, 2014. Springer Berlin Heidelberg.

[154] Wayne O. Cochran. Program that tests the performance of the Volker Strassen algorithm for matrix multiplication. `https://ezekiel.encs.vancouver.wsu.edu/~cs330/lectures/linear_algebra/mm/`, 2011.

[155] Conrad Sanderson and Ryan Curtin. Armadillo: A template-based C++ library for linear algebra. *The Journal of Open Source Software*, 1:26, 2016.

[156] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. `http://eigen.tuxfamily.org`, 2010.

[157] Floréal Morandat, Brandon Hill, Leo Osvald, and Jan Vitek. Evaluating the design of the R language: Objects and functions for data analysis. In *ECOOP 2012 — Object-Oriented Programming: 26th European Conference. Proceedings*, Lecture Notes in Computer Science, pages 104–131, Germany, 2012. Springer Berlin Heidelberg.

[158] Eugene Burmako. Scala Macros: Let our powers combine! On how rich syntax and static types work with metaprogramming. In *Proceedings of the fourth Workshop on Scala*, SCALA '13, pages 3:1–3:10, New York, NY, USA, 2013. ACM.

[159] Leo Osvald, Grégory Essertel, Xilun Wu, Lilliam I. González Alayón, and Tiark Rompf. Gentrification gone too far? Affordable 2nd-class values for fun and (co-)effect. In *Proceedings of the 31st ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '16, pages 234–251, New York, NY, USA, 2016. ACM.

[160] John Boyland, James Noble, and William Retert. Capabilities for Sharing. In *ECOOP 2001 — Object-Oriented Programming: 15th European Conference. Proceedings*, pages 2–27, Germany, 2001. Springer Berlin Heidelberg.

[161] David R. Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Software: Practice and Experience*, 20(1):5–12, January 1990.

[162] Tomas Kalibera, Petr Maj, Floréal Morandat, and Jan Vitek. A fast abstract syntax tree interpreter for R. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '14, pages 89–102, New York, NY, USA, 2014. ACM.

[163] Keith Adams, Jason Evans, Bertrand Maher, Guilherme Ottoni, Andrew Paroski, Brett Simmers, Edwin Smith, and Owen Yamauchi. The HipHop Virtual Machine. In *Proceedings of the 29th ACM International Conference on Object-Oriented Programming Systems Languages, and Applications*, OOPSLA '14, pages 777–790, New York, NY, USA, 2014. ACM.

[164] Jose M. Redondo and Francisco Ortin. Efficient support of dynamic inheritance for class- and prototype-based languages. *Journal of Systems and Software*, 86(2):278–301, 2013.

[165] Prodromos Gerakios, Nikolaos Papaspyrou, and Konstantinos Sagonas. Static safety guarantees for a low-level multithreaded language with regions. *Science of Computer Programming*, 80:223–263, 2014.

[166] Christoph Burnikel and Joachim Ziegler. Fast recursive division. Technical report, Max Planck Institute for Informatics, Saarbrücken, Germany, 1998. `https://www.mpi-inf.mpg.de/~ziegler/TechRep.ps.gz`.

[167] Karl Hasselström. Fast division of large integers: A comparison of algorithms. M.S. thesis, Royal Institute of Technology, Stockholm, Sweden, February 2013.

[168] Leo Osvald and Tiark Rompf. Flexible Data Views: Design and Implementation. In *Proceedings of the fourth ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY '17, pages 25–32, New York, NY, USA, 2017. ACM.

[169] Leo Osvald and Tiark Rompf. Rust-like borrowing with 2nd-class values (short paper). In *Proceedings of the eighth ACM SIGPLAN International Symposium on Scala*, SCALA '17, pages 13–17, New York, NY, USA, 2017. ACM.

[170] Juliana Franco, Martin Hagelin, Tobias Wrigstad, Sophia Drossopoulou, and Susan Eisenbach. You can have it all: Abstraction and good cache performance. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2017, pages 148–167, New York, NY, USA, 2017. ACM.

[171] Amir Kamil, Yili Zheng, and Katherine Yelick. A local-view array library for partitioned global address space C++ programs. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY'14, pages 26:26–26:31, New York, NY, USA, 2014. ACM.

[172] Jonathan I. Brachthäuser and Philipp Schuster. Effekt: Extensible algebraic effects in Scala (short paper). In *Proceedings of the eighth ACM SIGPLAN International Symposium on Scala*, SCALA '17, pages 67–72, New York, NY, USA, 2017. ACM.

VITA

VITA

Leo is a computer scientist and practitioner interested in programming languages and algorithms, born in 1989. His research focuses on improving high-level languages, in particular object-oriented and functional ones. He has worked on extending their type systems as well as optimizing and understanding their run-time performance, often by exploiting advanced type system features and customized data structures.

Leo has a diverse background in algorithms and software engineering, and is an advocate of reproducible research via open-source tools. Among his earliest accomplishments are awards at national and international competitions in computer programming such as TopCoder[2] and the American Computer Science League, during which algorithmic problems are solved in limited time and memory. More recently, he had the honor to receive the Distinguished Artifact Award for the paper "Gentrification Gone too Far? Affordable 2nd-Class Values for Fun and (Co-)Effect", published by ACM in the OOPSLA'16 proceedings.

Outside academia, Leo has enhanced infrastructure and tools at Facebook and Yelp, and developed a scalable and robust algorithm for enhancing aerial images at Google. In addition to computer vision, distributed systems and networking, he has also worked on constrained optimization problems. He received a master's degreee from Purdue University (West Lafayette, IN), USA in 2013 and a bachelor's degree from FER, University of Zagreb, Croatia in 2011, both in computer science.

His passion is applying novel (meta)programming techniques and ideas to develop elegant yet practical tools and libraries, which he releases as open-source on GitHub[3] when they become mature.

---

[2] `https://www.topcoder.com/community/competitive-programming/`

[3] `https://github.com/losvald/`