Domain Specific Memory Management for Large Scale Data Analytics

by

P.C. Shyamshankar

A dissertation submitted to The Johns Hopkins University

in conformity with the requirements for the degree of

Doctor of Philosophy

Baltimore, Maryland

April 11, 2018

Abstract

Hardware trends over the last several decades have lead to shifting priorities with respect to performance bottlenecks in the implementations of dataflows typically present in large-scale data analytics applications. In particular, efficient use of *main memory* has emerged as a critical aspect of dataflow implementation, due to the proliferation of multi-core architectures, as well as the rapid development of faster-than-disk storage media. At the same time, the wealth of static domain-specific information about applications remains an untapped resource when it comes to optimizing the use of memory in a dataflow application.

We propose a compilation-based approach to the synthesis of memory-efficient dataflow implementations, using static analysis to extract and leverage domain-specific information about the application. Our program transformations use the combined results of type, effect, and provenance analyses to infer time- and space- effective placement of primitive memory operations, precluding the need for dynamic memory management and its attendant costs. The experimental evaluation of implementations synthesized with our framework shows both the importance of optimizing for memory performance, as well as significant benefits of our approach, along multiple dimensions.

Finally, we also demonstrate a framework for formally verifying the soundness of these transformations, laying the foundation for their use as a component of a more general implementation synthesis ecosystem.

Readers: Yanif Ahmad (advisor), Scott Smith, Randal Burns.

Ammy, Appy & Bojo

Preface

This has been a journey of discovery — primarily the discovery of how much I have yet to learn.

While this has led to many disconcerting moments, it has been, in sum, an incredible journey.

I would like to make some acknowledgements:

- My advisor — Dr. Yanif Ahmad — for giving me the opportunity, and guiding me through it.
- My family — my parents, and my sister — for keeping me sane.
- My friends and colleagues — far too many to list — for keeping it interesting.
- And you — the reader.

With that, let's begin.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Generalization and *specialization* are staple concepts of software system design; each describes a relationship between a problem in one domain, and a *class* of problems in another.

In the context of programming languages and systems, generalization (or *abstraction*) is the idea that a single program written in a more general, *higher-level* (or *more expressive*) language or framework may simultaneously be used to solve the same problems described by an entire set of programs at a *lower-level* (in a *less expressive* language or framework). This is typically modeled by *parameterization*: the high-level specification involves a set of parameters, with each configuration of the parameters representing one of the low-level programs being generalized over.

Consider the extremely simple example in Listing 1.1. Despite the apparent low-level-ness of the C-like language, the `add` function written here generalizes across a large set of programs which might be written in a lower-level language like assembly, over a number of different dimensions, including, but not limited to:

1. It abstracts over *all possible additions* of two integers; indeed, depending on the values passed into `add` as `X` and `Y`, any one of a large set ($||\texttt{int}||^2$) of possible programs which might have

```
int add(int a, int b) {
  return a + b;
}

// ...

add(X, Y);
```

Listing 1.1: Addition in C

```python
def add'(a, b):
  return a + b

# ...

add'(a, b)
```
Listing 1.2: Addition in Python

been written in any particular assembly language.

2. `add` is also agnostic to how the data corresponding to `x` and `y` are copied or otherwise moved around memory, cache or registers in service of its computation. It instead relies on the compiler to determine one such scheme — hopefully, an efficient one.

Despite the numerous dimensions along which `add` is more general than a corresponding assembly program, it could still generalize further. The choice not to do so addresses the second, related concept of specialization.

Specialization (or *concretization*) is a counterpart to generalization, and expresses the idea that if a high-level specification represents a parameterized class of low-level programs, that specification may be *refined* to obtain a narrower specification, by partially evaluating it with respect to a specific subset of its parameters. This narrower specification would then be a generalization over a subset of the programs represented by the original, potentially even just one.

The `add` function described above might have been written as in Listing 1.2, in an even higher level language. This latter `add'` function generalizes further over the *types* of its arguments; in addition to addition over integers, `add'` also generalizes addition over any type for whom the `+` operator is supported. Alternatively phrased, the former `add` function represents only the subset of additions represented by `add'`, which are performed over integers.

There are may be many reasons to specialize a high-level specification to a lower-level, the most common being *performance*. Any layer of generalization or abstraction which confers a benefit of expressivity typically requires a penalty paid in the performance of the resulting program; specialization reverses this trade-off, giving up expressivity and convenience in exchange for performance.

The precise definition of performance may vary across a number of metrics; common metrics include running time, resource usage (memory, network, disk), binary size, or even power consumption [77]. Many specializations trade performance in one metric for another; for example, memoization trades increased memory usage for decreased running time, while underclocking trades slower running time for reduced power consumption.

As with any design technique, generalization and specialization are also prone to be *abused*: overgeneralization leads to a loss in performance compared to the necessary expressivity requirements of the domain, while overspecialization leads to architectural mismatches between problems and programs.

1.1 Generalization and Specialization in Large-Scale Data Analytics

In this work, we focus on overcoming the problems of overgeneralization and overspecialization in the context of large-scale data analytics.

The core of any data analytics framework is the *dataflow* abstraction: a description of how data is represented and moved between logical actors in a system (such as operators in a query plan), and how that representation and movement maps down to the physical level (such as among nodes in distributed system). Each dataflow implementation is a generalization over a class of applications; those whose data representation and movement patterns conform to the dataflow specification.

Contemporary data analytics platforms implement a variety of different dataflows, including map-reduce [22], iterative bulk synchronous [50], iterative asynchronous [48], cyclic [58, 1], array processing [71], and others. Each inhabits their own point in the performance trade-off space, along dimensions such as elasticity, throughput, latency, as well as the conventional metrics listed above.

The choice of which dataflow implementation framework to use for any given application rests on the architectural compatibility between the application and the framework, and the specialization capability of the framework itself.

Architectural compatibility relates to how well the framework captures the dataflow characteristics of the application; incompatibilities — or *architectural mismatches* [29, 30] — indicate that the resulting implementation will be a poor encoding of the application (e.g. with excessive data movement, or redundant computation or communication).

For example, map-reduce implementations such as Hadoop [32] have been used for a wide variety of applications, not all of them being the best match of application to dataflow [47]. Typically, a map-reduce programmer must only write application-specific implementations for `map` and `reduce` functions, other components — including input/output, and distributed scheduling — are provided by the framework. A map-reduce framework in turn makes key semantic assumptions about the programmer-supplied `map` and `reduce` implementations which, if not satisfied, may impact correctness or performance [84]. These include general assumptions of *determinism* and *statelessness* of both the `map` and `reduce` functions to ensure correctness, and *commutativty*, *associativity* and *selectivity* to improve performance.

Implementations such as Hadoop use the information guaranteed by the above assumptions to make implementation decisions which benefit the target demographic of programs: those conforming to the map/reduce programming paradigm. In this context, we can examine the ability of the implementation to specialize, and to generalize.

The ability of the framework to specialize deals with the incorporation of *domain-specific information* to tailor the dataflow implementation to the specifics of the application being supported. Types of domain specific information include high-level application invariants, data access patterns, data locality observations, distributed deployment topologies and hardware configurations.

Many existing dataflow implementations perform specialization to varying degrees: an RDBMS takes integrity constraints attached to a schema definition into account when evaluating heuristics for index selection, as well as using statistics to inform query planning. Distributed graph-processing engines take into account the structure of an input graph as well as network topology to determine an efficient partitioning scheme, optimizing for balanced usage of resources, and skew reduction. Proper use of available information can lead to orders-of-magnitude improvements across multiple dimensions.

The sheer volume of domain-specific information that can be brought to bear on performance-sensitive implementation decisions suggests that making the best use of this information is beyond the credible capability of individual programmers. In this thesis, we adopt a *compilation-centric* approach, focusing on the use of compile-time static analysis to tackle the problem of *memory efficiency* in large scale data analytics.

1.2 Memory Management in Large-Scale Data Analytics

With the increased cost-efficiency of RAM spurring the adoption of main-memory data analytics systems, managing that memory has proven to be one of the critical bottlenecks in high-performance data analytics. Research [62] shows that contrary to popular belief, CPU — not disk or network — is the dominant factor in main-memory architectures, and that memory management — allocation, deallocation, and copying — accounts for a considerable part of that cost. Ongoing work in systems such as Spark [87] and Flink [16] demonstrate the importance of optimizing memory management for performance gains [63, 3].

Main memory architectures exhibit two pervasive patterns. The first is *segmented* memory management, where memory management responsibilities are delegated throughout main-memory subsystems and modules. Each subsystem — the storage manager, query and dataflow engine, recovery manager, task scheduler, etc. — implements its own specialized memory management strategy to fit its tasks, responsibilities and workload assumptions (block-oriented storage, relational data-flow, etc.). This is exacerbated in hybrid engines [38, 68] and polystores [24] that couple multiple subengines (e.g., row, column, array, and stream-oriented engines among others), yielding memory architectures composed of a non-trivial number of regions in industrial grade systems.

As a second pattern, an increasingly important segment for main-memory systems is the *unmanaged* segment for expressive, unrestricted application-specific code (UDFs, vertex programs, etc.), where memory management is delegated to the underlying language compiler or virtual machine, either manually, or with garbage collection.

However, the fundamentally holistic nature of memory management presents several compelling reasons to adopt a whole-system model. Architectural mismatches manifest as suboptimal handling of memory resources between segments; this can lead to surprising performance anomalies due to unexpected memory operations. Developer efforts to achieve more predictable performance often leads to an over-reliance on the unmanaged segment of memory.

A system-wide, compilation-driven approach would expose every aspect of the system to a common optimizer, allowing architectural mismatches to be handled ahead-of-time, reducing the amount of work to be performed by the runtime system.

1.2.1 Language-Level Approaches to Memory Management

Memory management at the framework level builds on top of the techniques used at the language level; these may be broadly classified into two: *manual-but-static*, and *automatic-but-dynamic*.

Manual-but-static memory management requires programmer annotations to specify when memory should be acquired, released, and how data should be moved from one location in memory to another. In exchange for the burden of annotation, manual memory management affords the greatest degree of control over memory and related operations, and therefore the greatest degree of performance and attunement to workload characteristics.

Programmers may use domain-specific information available to them to reason about the lifetimes of objects and conflicts between them, making optimal use of the available memory. This benefit is balanced by the potential for errors in the memory management related parts of the program, which may cause dangerous and difficult-to-diagnose errors. The need (and ability) to perform manual memory management is a hallmark of systems languages.

In contrast, automatic-but-dynamic memory management is a characteristic feature of high-level languages; the lack of any need for annotations translates to a variety of software engineering benefits such as readability. Memory management in such languages is done *at runtime* by a separate entity — a *garbage collector* [55] — which executes alongside the user program, monitoring its memory usage, and acquiring and releasing memory as necessary or possible.

Contemporary main-memory data systems are predominantly written in languages with heaps managed by garbage collectors, with dedicated memory segments managed separately as described above. However, this leaves a glaring inefficiency at the boundary between the core data flow and user-defined code, an interface becoming increasingly more important. As the user-code is managed by the underlying language implementation's garbage collector, this leads to considerable performance losses [8, 7].

1.2.2 Holistic, Automatic, Static Memory Management

Focusing on compilation allows us to take a third approach to memory management: *automatic, and static*. We use the static analyses present in our compiler to determine an efficient memory management strategy, amounting to the inference of the annotations a programmer would otherwise have been required to place manually. This frees us from the overhead of garbage collection, while being able to specialize the use of memory to the program being compiled, resulting in performance similar to a corresponding hand-written implementation in a low-level language. The use of a compiler also allows for a more *holistic* memory management strategy, trading compilation time for improved performance comparable to hand-written implementations.

The notion of *compile-time garbage collection* is not new, but as with other compilation-related techniques, is staging a renaissance of its own. The idea was first formulated in the context of Lisp [9], and focused on the use of static information to perform some tasks at compile time, which would otherwise have been performed at runtime anyway. More recently, static memory management has been applied to very high-level languages such as Mercury [70, 54], whose compiler may take advantage of the regular structure of programs to synthesize efficient memory management schemes.

In our work, we make particular use of *move semantics*, a convention of *ownership transfer* of data between program variables, as an intermediate between copying and aliasing. While move semantics have existed in the programmer's collective consciousness for decades, they have only recently been given first-class status in contemporary languages (such as C++ and Rust) as an alternative to copy semantics.

We look at how move semantics may be incorporated into the synthesis of a memory management scheme for a program, alongside copying and aliasing. In particular, we use static type, effect and provenance analysis to determine whether a move operation is safe to apply at any given point, preserving an "intended" copy semantics while performing better.

Automated inference of move semantics provides a key bridge between the inefficient-but-safe use of copy semantics, and the efficient-but-complex use of reference semantics, and may lead to a considerable reduction of runtime memory management overhead by static partial evaluation.

1.3 A Renaissance of Compilation

A compilation-centric view of data analytics is part of a more general trend towards declarative programming, as well as a renaissance of compilation. Declarative methods and pervasive use of static analysis for optimization were particularly popular with the advent of SQL [19] and Prolog [20]; a great deal of work has been put into the compilation and optimization of Lisp [81] and other functional languages.

However, many of these techniques lost out in favor of the much simpler brute-force approach to performance optimization: using better — or more — hardware. The broader phenomenon known as Moore's Law [57] provided for a scenario in which the same program or workload, if left unchanged for a period of months or a year, would perform at a significantly better level simply by virtue of improved hardware performance. These hardware advances were largely transparent to the programmer, who benefited from them without needing to update their programs in any major way, allowing them to focus instead on qualitative improvements such as application features.

Evidence suggests that we are now trending towards a *post Moore's Law* computing landscape [67], where hardware advances no longer benefit the programmer as much, or — more importantly — as transparently. Leveraging the benefits of multi-core, GPU and FPGA architectures often requires a non-trivial redesign of core algorithms; integration with cloud computation platforms such as serverless architectures (such as AWS Lambda, Google Cloud Functions or Azure Functions) requires optimization along a different set of dimensions compared to traditional optimization.

Compilers are already moving to occupy a central position in this computing landscape, including multi-target code generation backends [17, 72], memory hierarchy abstraction [27], and distributed communication abstraction [33]. The growth of this already complex space only reinforces the fact that compilation will remain a dominant approach to achieving optimal software performance.

1.4 This Thesis

This thesis is a report on the design and implementation of K3, a language which aims to be a vehicle for incorporating domain-specific information in the synthesis of efficient implementations of large-scale data analytics tasks. In particular, this document focuses on using static analysis to synthesize efficient memory management schemes in K3 using move semantics, avoiding the need for a runtime garbage collector.

1.4.1 An Overview of K3

Chapter 2 provides an overview of the design of K3. We describe the overarching design philosophy of K3: a combination of systems declarativity and optimization-friendly semantics, and explain the set of trade-offs made and their relation to typical data analytics tasks.

We then present the syntax of K3, and present number of programs exemplifying its semantics — both at the local, and distributed levels. We close with a discussion of its *annotation* system, the primary vehicle for conveying domain specific information from the programmer to the compilation toolchain.

1.4.2 The Compiler Frontend

Chapter 3 covers the front-end of the K3 compiler toolchain, including the following:

Annotation Expansion: The implementation of the annotation system described in Chapter 2 comes in the form of an expansion system, analogous to a metaprogramming framework in contemporary programming languages, but with one critical distinction — access to the results of static analysis.

Type Analysis: K3's type system is an extension of a standard Hindley Milner type system, with added support for typechecking collection annotations. We describe the type syntax and inference algorithm, as well as the changes made to support annotation typechecking.

Effect Analysis: As with the type system, K3's effect analysis is an extension of typical effect systems, particularly with regards to higher-order functions. We describe in this section the distinction between *immediate* and *deferred* effects, their representation, inference, and usage.

We also describe the use of *effect signatures*, an important part of how K3 integrates with foreign code, without sacrificing static analysis utility.

Provenance Analysis: An integral part of the utility of effect analysis is the ability to specify in detail the nature of the target of any given effect: for example, in the presence of record data types, some optimizations depend on being able to distinguish between modifications of one field of the record and another, and the record as a whole. This information is inferred and attached in the form of *provenance terms*, describing the relationships between data items in a K3 program.

In this section, we describe the expressivity of our provenance encoding, the inference procedure, and how provenance information interacts with the results of effect analysis.

Source Level Optimization: K3 leverages a variety of well-known optimizations in the literature; however, many of these optimizations must be augmented with additional smarts in order to apply in the presence of side-effecting computation. In this section, we describe how effect and provenance information is used to permit traditionally pure optimizations such as stream fusion to be applicable to effectful computations, while preserving semantic equivalence.

1.4.3 The Compiler Backend

Chapter 4 covers the back-end of the K3 compiler toolchain, dealing primarily with the low-level decision making process necessary to synthesize efficient C++ implementations for K3 programs. In particular, we address the following:

Materialization Analysis: The assumption of optimization-friendly copy-semantics makes K3 extremely amenable to optimizations such as automatic parallelization, but have the unfortunate consequence that naïve implementations contain a large amount of unnecessary memory operations.

Materialization analysis is a novel technique we have developed to use static information — including type, provenance and effect information described above — to make decisions on how to perform any given low-level assignment operation.

In particular, we leverage *move semantics* — low-level transfer of resource ownership — to circumvent limitations of aliasing when dealing with variable isolation, maintaining independence and allowing related optimizations to proceed, while avoiding the use of a garbage collector.

We describe the value materialization problem in K3, and our formulation as a constraint satisfaction problem. We then present our materialization constraint solver, and the criteria it uses to judge efficient solutions to the problem.

Code Generation: The choices made by materialization analysis influence the code generation process. In this section, we describe how we translate our high-level materialization decisions into low-level use of C++ features to achieve performant — but entirely static — memory management.

Runtime System: A large part of the promise of declarativity in K3 is fulfilled by dispatching systems concerns to specific low-level implementations; this is handled by the runtime system and standard library. In this section, we explain how the C++ code generated from K3 programs interact with the communication and collection subsystems.

1.4.4 Evaluation

In Chapter 5, we present an experimental evaluation of implementations synthesized in K3 for a select set of workloads. The workloads covered are primarily SQL analytics queries drawn from the TPC-H benchmark, and provide a good coverage of language features and complexity. We also look at a smaller set of graph and iterative (machine-learning) workloads.

Our evaluation focuses primarily on two metrics: running time and memory use. In particular, we look at how these metrics interact, and the impact of managing memory on running time. In addition we present a novel metric for studying the memory impact of optimization — *memory load* — as a combination of object sizes and lifetimes.

1.4.5 Formalizing Value Materialization

Chapter 6 takes a deeper look at the value materialization problem presented in Chapter 4 — more specifically what it means to incorporate move semantics into an existing program, while preserving equivalence to copy semantics.

We present a simplified core calculus — System $\mathcal{M}$ — derived from K3 and intended to model just the aspects of the language relevant to the problem of determining assignment methods. We establish a formal framework within which we can prove equivalence between programs which differ only in how they perform assignments, and demonstrate a conservative, but efficient, approximation to this notion of equivalence.

Finally, we outline an algorithm which would systematically relax copy operations into moves, and discuss how this framework can be extended to handle the traditionally difficult problem of conservative alias analysis.

1.5 Who This Work Is For

The work in this thesis should appeal to a number of different audiences.

The use of K3 is primarily targeted towards domain specialists, who would like to develop an application for their domain at their chosen level of abstraction — being able to specify information about their domain to improve performance, while not needing to make low-level implementation decisions.

In contrast, systems designers and implementors are best positioned to make use of K3's specialization framework to create the abstractions used by domain specialists to realize their applications. The modular design of the language and compiler make it possible and relatively straightforward for these groups of people to be independent, focusing on their respective strengths.

Finally, the internals of the K3 compiler will be of interest to practitioners of static analysis tools, as a study of how complex and holistic static analyses may be put to use in practical scenarios, at scale. K3's approach espouses the belief that program analyses do not need to be *perfect* in a theoretical sense to be useful, and that sound — if incomplete — analyses provide value to the compilation and optimization of large-scale systems.

Chapter 2

An Overview of K3

The motivation behind the design of K3 is to provide a platform for expressing a variety of large scale distributed data processing workflows at a high-level, while abstracting away the process of making low-level systems decisions.

In the absence of explicit programmer direction, the burden of making these performance sensitive decisions rests with the compiler. The K3 compiler's approach to making these decisions may be summarized by two main design principles: *systems declarativity* and *optimization-friendly semantics*.

2.1 Systems Declarativity

As outlined in Chapter 1, a major theme grappled with during any language design process is the trade-off between *specificity* and *generality*.

At one extreme, assembly language requires programmers to completely specify exactly what the computer is to do during execution. To a lesser degree, systems languages such as C and C++ abstract away some of the more arduous tasks — register allocation, interaction with the kernel, etc. — while still requiring a great deal of explicit direction from the programmer.

At the other extreme, high-level languages such as SQL [19], Prolog [20], Python [64], Ruby [66], and most functional programming languages only require that the programmer describe *what* is to be computed, leaving the *how* up to a compiler. Tasks such as memory management, execution ordering and parallelism, and others are handled automatically, making high-performance computing accessible to programmers without low-level experience.

At the data processing platform level the focus is more on tailoring design decisions towards specific workload types; for example, relational database engines are designed to excel at processing relational workloads of the form that can be expressed in SQL, but have relatively poor support for iterative algorithms. Most platforms and frameworks support user-defined code in some form; the interaction between the core dataflow and user-defined functions may however be a source of inefficiency, or a way to break semantic assumptions.

K3's position in the space of trade-offs is informed by its purpose of broadly supporting large-scale distributed data processing tasks; as such, we can classify the declarativity of K3 into three kinds: *data declarativtiy*, *control declarativity* and *ensemble declarativity*.

2.1.1 Data Declarativity

The promise of data declarativity is the ability of a programmer to express their workloads without needing to make specific data-centric implementation decisions.

At the high-level, the most common data-centric implementation decision in K3 is the separation between data structure interface and implementation, and the freedom of the programmer to use a given interface, without choosing the implementation of that interface.

Most contemporary programming languages exhibit this form of declarativity using some kind of ad-hoc polymorphism model, such as the principle of encapsulation in object-oriented programming. K3's approach follows closely the techniques used in *aspect-oriented programming* [39] and *mixins* [28], which allow the compiler greater flexibility in choosing implementations by way of a fine-grained interface specification, typically at the per-method level.

At the low-level — and the topic of much of the work in this thesis — is the ability to program without needing to specify an explicit memory management scheme. Most languages which accomplish this goal do so by way of a *garbage collector*, a runtime mechanism for determining if and when allocated memory may be safely reclaimed. Many large-scale data processing frameworks have been implemented in such languages with dynamic memory management; however, the runtime overheads of memory management are non-trivial, and are often the first focus of any performance tuning efforts [62, 8, 34].

The goal of achieving a purely static memory management scheme without the use of a garbage collector, but with comparable performance, and in the absence of manual annotations, is still a topic of research.

2.1.2 Control Declarativity

The counterpart to data declarativity is control declarativity, the notion that programmers can express their workload at the algorithmic level, without needing to specify control-flow related implementation details. This too can be broken up into high and low level decisions.

For distributed data processing, high-level control-flow decisions correspond to choices about how local tasks are distributed among a multitude of distributed worker nodes, and how communication between them is coordinated. Data processing frameworks are typically built around a single communication pattern — e.g. Bulk synchronous processing in Pregel [50], or asynchronous in GraphLab [48].

At the low-level, an important aspect of performance in the contemporary computing landscape is local multi-core parallelism, which requires a program to be properly segmented into fine-grained, independent units. Inference of these units is traditionally extremely difficult, and potentially undecidable. K3 aims to have this information available by construction, requiring the programmer to specify their tasks in an event-driven paradigm (as in Erlang [26]), which may then be scheduled automatically.

2.1.3 Ensemble Declarativity

Both data and control declarativity emphasize trading some freedom of the programmer's expression of how tasks are to be done, in exchange for an automatic, potentially more efficient and thorough compilation-driven process. This allows programmers to focus on the conceptual part of their application, and not on the specifics of how that application is to be deployed at scale.

A third form of declarativity is in abstracting over implementations of common systems tasks. These are more qualitative in nature, combining parts of both control and data abstractions.

For example, a common task in many data processing systems is *logging*. This corresponds to keeping records of events occurring during execution and tasks being performed, either for debugging purposes, or for recovery after failure. There are multiple possible approaches of solving the same general task of logging, differing in the granularity and frequency of information logged, as well as the location and durability of the logs themselves. Each of these approaches in turn consist of a set of data structures — whose implementations themselves might be abstracted over — and a set of event handlers — whose control flow may also be abstracted over.

Ensemble declarativity in K3 refers to the ability of the programmer to specify the qualitative requirements of an implementation for a specific task — e.g. per-query, per-minute logging durable for at least a day — without having to specify exactly how such requirements are fulfilled.

2.2 Optimization-Friendliness

While the concept of declarativity focuses on the separation of concerns between the programmer and the compiler, *optimization friendliness* is more concrete — the property of the syntax and semantics of a language which lend it to optimization.

Several of the promises made above — automatic memory management, parallelism, etc. — rely on the presence of rich static analysis information. Inferring the necessary information, in a general purpose programming language, without any programmer annotations is by no means a solved problem — indeed, in some cases, the problem might well be undecidable [44]. However, there are methods available to language designers to ensure that relevant static information is present *by construction*: syntactically prohibiting a class of programs which do not exhibit characteristics favorable to optimization. K3 makes a number of design decisions which make its semantics more amenable to static analysis and optimization.

Independence through by-default copy semantics: Data independence information — the ability to determine that the data referred to by two different names in a program are independent — is extremely useful for a variety of optimizations. Correspondingly, the general case of independence analysis in a language with aliasing is intractable with perfect accuracy, though many conservative approximations have been proposed. K3 sidesteps this problem by requiring that *all* variables be independent by construction, forcing any assignment — either explicit, or implicit due to function calls — to be performed by copy. A naïve implementation of copy semantics is extremely inefficient, but this performance is recovered through a separate static procedure — *materialization analysis*, the topic of Section 4.1. We discuss the importance and impact of this design decision in further detail in that section.

Fine-grained units of execution through event-driven programming: The degree to which the K3 compiler is able to parallelize or otherwise efficiently schedule execution of a program is dependent on its ability to ensure that programs are duly broken up into manageable pieces, each of which represents a non-trivial amount of computational work which may be independently executed and scheduled. This assumption is complicated in the presence of recursion, which K3 avoids similarly as above by disallowing recursion at the function-level; any recursive control flow is done asynchronously through use of the message-passing framework. K3's execution model follows in the vein of those found in Erlang [26], and implemented in Cloud Haskell [25].

Built-in collection manipulation with in-place mutation: Core to K3's promise as a framework for data processing tasks is its ability to manipulate collection data structures; K3 provides a primitive set of operations to create, access, transform and modify collections within the language itself. This interface is decoupled from their implementations to main declarativity, but are distinguished from generic function calls at the static analysis and optimization level. K3's collection infrastructure is modeled after the theory of complex objects and the nested relational algebra [13].

2.3 The K3 Programming Language

$$
\begin{array}{rcll}
d & ::= & d_c \mid d_e & \textit{declarations} \\
d_c & ::= & \texttt{declare}\, id : \tau = e \mid \texttt{trigger}\, id : \tau = e & \\
d_e & ::= & \texttt{source}\, id : \tau = e \mid \texttt{sink}\, id : \tau = e \mid \texttt{feed}\, id \rightarrow id & \\
e & ::= & c \mid v \mid e_1\, \theta\, e_2 \mid e_c \mid e_d \mid e_x \mid e\, \texttt{@}\, \mathcal{A}_E & \textit{expressions} \\
e_d & ::= & \texttt{none} \mid \texttt{some}\, e \mid (e^+) \mid \{(id : e)^+\} & \textit{data constructors} \\
e_x & ::= & \texttt{let}\, e_1 = e_2\, \texttt{in}\, e_3 \mid \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3 \mid e\, .\, id \mid & \textit{control flow} \\
 & & \lambda\, id \rightarrow e \mid e_1\ e_2 \mid id_1 = e \mid e_1 \leftarrow e_2 \mid e_1\ ;\ e_2 & \\
e_c & ::= & \texttt{empty}\, \texttt{@}\, \mathcal{A}_C \mid \{e^+\} \mid [e^+]\, \texttt{@}\, \mathcal{A}_C \mid e_t \mid e_m & \textit{collections} \\
e_t & ::= & e_1\, .\, \texttt{ext}\, e_2 \mid e_1\, .\, \texttt{fold}\, e_2\ e_3 \mid e_1\, .\, \texttt{zip}\, e_2 \mid e_1\, .\, \texttt{sort}\, e_2 & \textit{transformers} \\
e_m & ::= & e_1\, .\, \texttt{insert}\, e_2 \mid e_1\, .\, \texttt{update}\, e_2\ e_3 \mid e_1\, .\, \texttt{delete}\, e_2 \mid & \textit{mutators} \\
\tau & ::= & \tau_p \mid \tau_c & \textit{types} \\
\tau_p & ::= & \texttt{bool} \mid \texttt{int} \mid \texttt{real} \mid \texttt{string} & \textit{primitive types} \\
\tau_c & ::= & (\tau^+) \mid \{(id : \tau)^+\} \mid \texttt{option}\, \tau \mid \tau \rightarrow \tau \mid \texttt{collection}\, \tau\, \texttt{@}\, \mathcal{A}_T & \textit{complex types}
\end{array}
$$

Figure 2.1: K3 Language Grammar — Declarations, Expressions and Types

The syntax of the K3 programming language is presented in Figure 2.1, and consists of four main parts: the declaration language, the expression language, the type language, and the annotation language.

2.3.1 The Declaration Language

At the top level, a K3 program consists of a set of declarations, which may refer to each other (cyclic scope). Declarations perform no computation, merely define (or just declare) one or more entities for use during execution. The forms of declarations are described below.

Data Declarations

The most common form of declaration is a *data declaration*. Similar to global data declarations in languages such as C++, a data declaration of the form `declare` $id : \tau = e$ defines a global variable named id, with type τ, and initial value computed from the *initializer expression* e.

The type annotation in the declaration is mandatory, whereas the initializer expression is not; if omitted, a well-known default value for the type will be used. If present, the initializer expression will be evaluated on program startup, in source order.

In select cases, a global declaration may also specify an *effect signature*, denoting the set of computational effects which would occur if the initialization expression were to be evaluated, or — in the case of functions — invoked. This feature is used particularly when writing interface specifications for foreign functions, and is discussed further in Section 3.4.

Trigger Declarations

The primary unit of computation is the *trigger*, and represents an independently schedulable piece of code which handles a single event in the form of a message. Triggers are purely side-effecting — they have no return value. A trigger declaration of the form `trigger` $id : \tau = e$ defines a trigger named id, with message argument type τ, and body e. The body expression e is constrained to have the type $\tau \rightarrow$ `()`.

The lack of a return value reinforces the absence of general recursion as described above; complex local control flows are effected through asynchronous message passing between triggers, detailed below.

Triggers are modeled after the eponymously named construct from relational databases; in the context of an RDBMS, a trigger is an event handler which responds to changes to a table, such as the addition or deletion of a row. Analogously, the entirety of a K3 program may be viewed as a collection of event handlers which react to changes to the program's global state.

Message Graph Declarations

The control flow of a K3 program, as with any message-passing paradigm, may be viewed as a message-flow graph, with triggers forming the nodes of the graph, and message-passing operators forming the edges. The `source`, `sink`, and `feed` declaration forms are used to construct the entry and end points of this local control flow graph.

The declaration `source` $id : \tau = e$ defines a source id of messages, each of type τ, given by e.

Correspondingly, the declaration `sink` $id : \tau = e$ defines a sink to which messages may be sent, each of type τ.

Sources and sinks are connected using a *feed declaration* `feed` $id \rightarrow id$; this establishes the initial entry point to the K3 program. The exact nature of sources and sinks are defined by the runtime system; the K3 standard library defines multiple sources and sinks for common tasks such as basic input/output and data ingestion.

2.3.2 The Expression Language

The body of functions and triggers are written in a straightforward functional/imperative hybrid expression language. We build on existing languages such as OCaml [45], Scala [61] and Python [78]; the various expression forms are described below.

The expression syntax is based on that of OCaml, and at its most primitive includes constants c, variables v and binary operations $e_1\, \theta\, e_2$.

K3 supports a standard set of control flow forms typical in functional programming languages: lexically scoped `let`-bindings, conditionals, anonymous functions (λ-forms) and application.

On the imperative side, the primary effectful expression forms in K3 are sequencing (e_1 ; e_2), in-place assignment ($id = e$), and message passing ($id \leftarrow e$). These latter two are relatively uncommon in functional programming languages, and merit further explanation.

In-Place Assignment: Variables in K3 may be marked as mutable or immutable, similar to `ref` types in OCaml. However they are implemented more similarly to variables in C++ rather than pointers, not requiring dereferencing to manipulate. A K3 variable marked as mutable may be assigned in-place using the assignment operator, this is a pure side-effectful operation which may then be sequenced with other effects. A common K3 idiom is to combine assignment with functional collection manipulation, described below. The distinction between imperative and functional approaches to mutable state are discussed further in [52].

Message Passing: The message passing operator forms the basis of the macro-level inter-procedural control flow in K3. The send operator takes two arguments, a message target on the left-hand side, and the message contents itself on the right. The target of a message must be either the name of a sink, the name of a trigger, or a pair of trigger name and distributed identifier if the message handler is to be on another distributed worker.

Rich Data Types in K3

K3 provides a number of built-in primitive and composite data types, several of them common in functional programming languages.

Primitive data types in K3 include the standard array of integers, booleans strings and real numbers. In addition, the unit type `()` represents the prototypical "void" type of pure side-effects. Operators for manipulating these built-in types, including numeric operators, comparisons, boolean logic and string functions are provided in the standard library.

Option values are constructed as either `none` or `some` e, and functionally represent potentially failing computations. These are used heavily in conjunction with the built-in collection operations.

Tuples and records provide ordered and labeled composite types respectively; both forms may be destructured in their entirety at let-binding constructs; additionally, records may use the dot-operator to project selected fields by label name.

Collection Operations

A critical aspect of the K3 expression language is the ability to manipulate arbitrary-sized collections directly with built-ins to the language. The K3 collection sub-language is modeled after the theory of complex objects and the nested relational algebra [13], and consists of three sets of interactions — constructors, mutators, and iterators.

Constructors: The primary way to construct a collection is to create an empty collection with the appropriate properties. In this context "properties" refer to desired characteristics of the backing implementation, specified in the form of *annotations*. The expression `empty @` $\{\mathcal{A}\}$ (often abbreviated as `[]` in examples) constructs a zero-sized collection with a backing implementation which covers the set of properties described in $\{\mathcal{A}\}$.

Our implementation of K3 additionally provides syntactic sugar for constructing literal collections of statically known size; these desugar into an empty construction followed by a sequence of in-place insertions.

Mutators: In-place mutations on collections are performed through the use of `insert`, `update` and `delete`; these comprise the basic collection interface, their implementations are determined by the annotations specified in the type of the collection.

```
// Global State Declarations
declare numbers: [int] = []
declare result: real = 0.0
declare count: int = 100

// Main Computation Trigger
trigger fibonacci: {a: int, b: int, n: int} = \r ->
  if r.n == 0
    // Proceed to next stage of computation
    then (compute_sum, me) <- ()
    else (
      // Insert result into global aggregation,
      // and signal recursive call
      numbers.insert(r.a);
      (fibonacci, me) <- {a: r.b, b: r.a + r.b, n: r.n - 1})

// Simple follow-up computation,
// compute and print partial sums
trigger compute_sum: () = \_ -> (
  result = (
    numbers.fold (\acc -> i ->
      ((stdout, me) <- (i, acc); i + acc)
    )
    0.0
  )
)

// Program entry point
source start : {a: int, b: int, c: int }
  = { a: 0, b: 1, n: count }
feed start |> fibonacci
```
Listing 2.1: A K3 program to compute partial sums of the Fibonacci Sequence

Iterators: Computations are performed over a collection using a built-in set of collection iterators, these include `ext` (concatenating map), `fold`, `zip` and `sort`. The exact behavior of these iterators are also determined on a per-implementation basis, as determined by the attached annotations.

2.3.3 Anatomy of a K3 Program

Listing 2.1 shows a basic K3 program which performs a number of simple computations relating to the fibonacci sequence.

The core of the program is the `fibonacci` trigger, which takes as its message a single record with three fields: `a`, `b` and `n`. `n` represents the counter for the computation; having reached zero, a message is sent to the `compute_sum` trigger to continue the follow-up computation. Otherwise, the current element of the sequence `r.a` is inserted into a global collection data structure `numbers`, and the trigger sends an updated message to itself, K3's encoding of recursion.

The follow-up computation in `compute_sum` performs an aggregation over the now-complete `numbers` collection; the `fold` member of `number`'s collection interface is a function of three arguments: an aggregation function, an initial aggregand, and the collection itself. For each element of the collection, the aggregation function sends a message to `stdout` with the current value, as well as the partial sum to-date.

The `me` variable mentioned in each message-passing statement is a global identifier representing the address of the current node in a distributed execution context; this variable is populated by the runtime system on initialization of the program.

In this example, all execution is local; in a distributed execution, an additional variable — `peers` — is also provided by the runtime system, representing a collection of identifiers to all of the nodes known to the deployment. This allows the separation of control-flow and deployment concerns; messaging disciplines may be specified abstractly over a logical set of distributed workers, to be concretized just prior to execution with an actual list of actors.

2.4 The K3 Annotation System

Annotations — K3's program specialization mechanism — are tags attached at various points throughout a program, to customize the behavior of code or data structures according to domain-specific information.

As with our general discussion of declarativity, there are two main kinds of annotations — *data* annotations, which customize the interface and implementation of data structures in K3, notably collections — and *control* annotations, which customize code fragments within the context of a larger program, by specializing them to a specific execution context. Both of these forms are described below.

2.4.1 Implementing Collections with Data Annotations

The motivating intention of the K3 collection system is to provide an interface against which programmers may write their algorithms, while remaining agnostic to the specific implementation eventually used. However, it is often the case that an interface specification is not sufficient to express the set of semantics properties required of a data structure: for example, both sorted and unsorted linked lists may implement the same list interface; the sorted-ness property of the former is not reflected in the interface itself, only in an implicit assumption of the ordering of elements during iteration.

Most languages permit the programmer to specify the requirement of sortedness by choosing an underlying data structure implementation; this however forces the remaining implementation decisions to also become fixed. The ability to choose sortedness as an interface property, to be satisfied by the most efficient implementation which exhibits that property, is not permitted.

The Collection Annotation

At the most basic level, data annotations are quite similar to interface specifications in Java, pure virtual classes in C++, or typeclass definitions in Haskell. However, these interfaces are not nominally fulfilled by an implementation also written by the programmer, instead, the interface specification is used as a constraint, to be satisfied by the compiler.

Listing 2.2 shows part of the main `Collection` annotation in K3, containing a subset of the core primitives. When attached to a collection constructor (e.g. `empty`), the `Collection` annotation adds the interface specification to the type of the collection object; this type is then used during type analysis to ensure soundness. Afterwards during code generation, the set of annotations on a given collection are themselves collected, and an implementation satisfying all the required properties is chosen, or an error is returned.

2.4.2 Delimited Compilation with Control Annotations

$$\begin{aligned} d &::= \ \ldots \ \texttt{|}\ \texttt{compiler}\ id\ [c_{param}]\{{c_{rule}}^{+}\} \\ c_{param} &::= \{id\ \texttt{:}\ \tau^{\,+}\} \\ c_{rule} &::= p\ (\texttt{=>}|\texttt{*>})\ e\ \texttt{+>}\ \{d^{+}\}^{?} \end{aligned}$$

Figure 2.2: Syntax of a K3 Delimited Compiler

A more expansive form of program customization is K3's control annotation system, also called *delimited compilation*.

A delimited compiler (DC) is a set of program transformations which are attached to specific program points. Each transformation contains a rewrite rule which is matched against and applied to the attached expression, and a set of declarations which are introduced to the global scope if the rule fires. The ability to couple local expression rewrites with global declarations facilitates system specialization techniques such as physical database design, scheduling (synchronous or asynchronous) computation, and single-site to distributed transformations.

```
annotation Collection given type a, b, c, d {
  provides lifted size : () -> int
    with effects \_ -> R[self]

  @:CArgs 2
  provides lifted peek : (() -> a) -> (content -> a) -> a
    with effects \f -> \g -> [R[self]; f content; g content]

  provides lifted at : int -> content
    with effects \i -> [R[i]; R[self]]

  provides lifted insert : content -> ()
    with effects \elem -> [R[self]; R[elem]; W[self]]

  provides lifted extend : self -> ()
    with effects \col -> [R[self]; R[col]; W[self]]

  @:Transformer
  provides lifted combine : self -> self
    with effects \other -> [R[self]; R[other]]

  @:Transformer
  provides lifted map :
    (content -> a) -> collection {elem : a} @Collection
    with effects \mapF ->
      [R[self]; ([R[content]; mapF content])*]

  @:Transformer
  provides lifted filter :
    (content -> bool) -> self
    with effects \filterF ->
      [R[self]; ([R[content]; filterF content])*]

  @:{Transformer, CArgs 3}
  provides lifted group_by :
    (content -> a) -> (b -> content -> b) -> b
    -> collection { key : a, value : b } @Collection
    with effects
      \gbF -> \gaccF -> \z ->
        [ R[self]; R[z]
        ; ([R[content]; gbF content; ((gaccF z) content)])*
        ]
}
```

Listing 2.2: A fragment of the K3 collection annotation

The syntactic structure of a DC is shown in Figure 2.2, extending the declaration grammar in Figure 2.1. DCs define one or more rules (${c_{rule}}^+$), each rule consisting of three parts: the pattern p to match, the expression e to rewrite it to, and the declarations d associated with the rule. The pattern p can capture sub-expressions, types, and other pieces of information from the target expression, which can be used in both the rewrite rule and the declaration sections.

Listing 2.3 shows a real-world example of the use of delimited compilation in K3.

The `CollectGroupBy` delimited compiler provides a program rewrite which transforms a simple, local `groupBy` operation into a distributed version with hierarchical partial aggregation. The implementation consists of four separate rewrite transformations.

The `CollectGroupBy` transformation itself merely performs the top-level pattern match dispatch, and proceeds to attach two helper transformations: `PartialAggregation` and `Collect`.

`PartialAggregation` performs no in-place rewrite, but adds a pair of global declarations to each program: a data structure of the appropriate type (passed in from the attachment) to hold the partial aggregations, and a new merge trigger declaration to perform the per-node side of the aggregation computation. Each invocation of the merge trigger is *synchronized* with the use of a `Barrier` transformation.

The `Collect` transformation replaces the local computation with the initiation of the distributed computation, and adds the necessary global declarations to finalize the protocol after the last stage of the hierarchical aggregation completes. It takes as parameters the name of the computation itself, and functions to finalize the aggregation (in the case of batched aggregation such as average) and a continuation trigger to send the final result to.

While `CollectGroupBy` alters the control flow of the expression to which it is attached, it does not change its semantics — K3 programs are expected to perform correctly independently of attached DCs; the DCs are intended to *specialize* the program by making use of workload or deployment specific information.

This usage of DCs to represent cross-cutting concerns is the main focus of aspect-oriented programming, and is an important part of the process of specializing large systems codebases. Writing a whole-program specialization/optimization pass is typically a time-consuming process, whereas focusing on a specific component in the larger system, and being able to influence the compilation process of that component in isolation is a more efficient, incremental approach to specialization.

```
// Transform a flat groupBy to use aggregation tree.
compiler CollectGroupBy(mergeF):
  !d <- (!c.groupBy !g !f !z : !t) =>
    // Ignore consumes its argument and returns () to
    // proxy pattern parameters, preserving types
    (ignore ($[c].groupBy $[g] $[f] $[z])) @ [
      PartialAggregation(t, mergeF, [# comm_cont]),
      Collect(t, [$ \a -> $[d] <- a], [# merge]) ]

// Setup declarations for partial aggregation.
compiler PartialAggregation(aggT, mergeF, commContL):
  !e => $[e]
  +> declare agg: $[aggT]
     // Partial aggregation at each intermediate node.
     trigger merge: $[aggT] = \partial ->
       agg = $[mergeF] partial agg;
       ($[commContL] agg) @Barrier([$ children])

// Gather results using custom topology.
compiler Collect(commT, finalizeF, forwardL):
  ignore !e => comm_next me $[e]
  +> // Routing Table (construction omitted)
     declare comm_route : addr -> addr = \src -> ...
     // Forwarding step.
     declare comm_next  : addr -> $[commT] -> () =
       \n v -> ($[forwardL], n) <- v
     // Final step, invoke original continuation.
     declare comm_cont  : $[commT] -> () = \a ->
       let next = comm_route me in
       if next == me then $[finalizeF] a
         else comm_next next a

// Coordination barrier helper.
compiler Barrier(count):
  !e => counter += 1;
        if counter == count then $[e]
     +> declare counter: mut int = 0
```
Listing 2.3: The `CollectGroupBy` delimited compiler for transforming a flat to hierarchical `groupBy`.

Chapter 3

The Compiler Frontend

In this section, we discuss the front-end of the K3 compilation toolchain, which consists of three conceptual parts: delimited compilation/metaprogram expansion, static analysis, and source-level optimization. Each of these phases are interspersed with each other; optimization and metaprogram expansion both depend on the results of static analysis, which must be repeated after each transformation of the program.

3.1 Delimited Compilation: Expansion

As described in Section 2.4.2, K3's delimited compilation framework consists of a rewrite engine, augmented with the introductions of global state declarations at the top-level. Rewrite rules are matched against attachments at arbitrary points in the program, as well as the syntactic pattern of K3 to which annotations themselves are attached.

DCs are expanded by the K3 compiler following a well-defined procedure designed to avoid the staging problems associated with common metaprogramming systems. A brief pseudocode description of this procedure is shown in Algorithm 1. The procedure demonstrates two different ways in which DCs can be *composed* with each other — *nesting* and *chaining*.

DC **chaining** provides programmers the ability to attach multiple DCs to an expression. A common use-case for chaining is to apply several orthogonal transformations to the same expression. Chained DCs are evaluated in the order they were specified by the programmer; at lines 4–6 of Figure 2.3, `PartialAggregation` and `Collect` are chained together to provide merge and communication logic respectively. In an attachment `e@[A, B, ...]`, `e` is first expanded with `A`, and the expression resulting from that rewrite is then expanded with `B`, and so on.

Algorithm 1 Delimited Compiler Expansion Procedure

```
1: procedure EXPAND-DC-STAGE($program$)
2:     $newDecls \leftarrow \{\}$
3:     for each declaration $d \in program$ do
4:         for each expression $e \in d$ in post-order do
5:             for each DC $C$ attached to $e$ in attachment order do
6:                 $(e', d') \leftarrow$ EXPAND-DC$(c, e)$.
7:                 $newDecls \leftarrow newDecls \cup d'$
8:                 $e \leftarrow e'$
9:             end for
10:            $program \leftarrow program \cup newDecls$
11:        end for
12:    end for
13: end procedure
14: procedure EXPAND-DC-PROGRAM($program$)
15:    repeat
16:        $program \leftarrow$ ANALYZE($program$)
17:        $program \leftarrow$ EXPAND-DC-STAGE($program$)
18:    until No DCs were expanded in current stage.
19: end procedure
```

In contrast, DC **nesting** is when the expansion of a DC produces an expression which itself has DCs attached to it. Due to the staged nature of the expansion algorithm, nested DCs are expanded during the stage following the one in which they were produced, allowing them access to the global declarations and static information generated during that stage. Nested DCs promote a modular approach to specialization; the `CollectGroupBy` example uses nesting to separate the aggregation and communication logic, as well as the messaging barrier used at line 14 in Figure 2.3 and implemented by the `Barrier` DC.

The core set of static analyses — types, effects and provenance — are re-evaluated at the end of each stage, making them available for pattern-matching by DC expansions during the next stage. Global declarations introduced by DCs expanded at a given stage of the procedure are held until the end of the stage, when they are added en masse to the program's global environment. All the artifacts of DC expansion are therefore stored in the program tree, no internal state is carried by the expansion algorithm between stages.

3.2 Type Analysis

The first of the static analyses in the K3 compiler is the type system. This analysis is broken up into separate parts, corresponding to the separate so-called sublanguages within K3 — the declaration, expression, and annotation languages.

Declaration Typesystem: Declarations in K3 are one of the few places where type signatures are mandatory. Data declarations must specify a signature which matches the type of the initializer expression, while trigger declarations must specify the message type which the trigger expects to process. The body of the trigger must then correspond to a function which takes as argument a message of the specified type, and returns `()`.

Control-flow declarations such as `source`, `sink` and `feed` are also ascribed types; sources and sinks must specify the type of the message being generated or accepted respectively, while feed declarations may only connect entities in the message graph which produce and consume compatible message types.

Type-checking of declarations is a straightforward application of techniques common in the literature; special care is taken to ensure that the cyclic scope of global declarations in a K3 program are accounted for.

Expression Typesystem: The expression typesystem corresponds to a traditional Hindley--Milner typesystem in the vein of [40]. The inferred type of the body of a trigger or global declaration is matched against the type specified in the type signature.

Of note in the expression language, the message passing operator $\leftarrow$ is checked against the set of known triggers or sinks, the message being passed must match the declared type of the message accepted by the recipient, regardless of that entity's physical location.

Annotation Typesystem: The purpose of the data annotation typesystem is to ensure a proper matching between *provisions* and *requirements* between interface specifications and implementations which provide members thereof.

3.3 Provenance Analysis

The first of K3's suite of analyses is *provenance analysis*, directed at identifying the *values* in a program and the dependencies between them. Informally, a *value* is the result of any expression, named or a temporary, a free-standing entity or a subcomponent of another value, and can have arbitrarily long or short lifetimes within the program.

In K3, we are mainly interested in the notion of *value equivalence* — determining if two values are guaranteed to contain the same data — based purely on static program information. At any given point in a program, knowing that two values are equivalent (from provenance) and will remain equivalent (from effects) allows us to elide the memory operations related to value initialization (in materialization).

3.3.1 Formulation

Provenance analysis attributes to every expression a provenance term, denoting the relationships between the value representing the result of that expression, and other values in the program. Unlike lineage implementations in systems such as Spark, K3 provenance does not keep all the information required to recompute the value itself — rather, it stores only the values on which the computation depends. The purpose of provenance is to assist in designating memory management operations, not to assist in fault-tolerance.

The grammar of provenance terms in K3 is given in Figure 3.1, and indicates the type of value relationships tracked in the analysis — *variables*, *roots*, and *structure*.

$$\begin{array}{lcll}
p & ::= & v \mid r \mid d \mid s & \textit{provenance} \\
v & ::= & V_f^{\mathbb{Z}} \mid V_b^{\mathbb{Z}} & \textit{variables} \\
r & ::= & \texttt{G}\,(id) \mid \texttt{C} & \textit{roots} \\
d & ::= & \texttt{R}\,(id, p) \mid \texttt{T}\,(\mathbb{N}, p) \mid \texttt{O}\,(p) & \textit{data} \\
s & ::= & (p^{+}) \mid \{p^{+}\} \mid \lambda\,(id, \{p\}, p) \mid \oplus\,(p, p) & \textit{structure}
\end{array}$$

Figure 3.1: K3 Provenance Grammar

A *variable* provenance indicates that the value is specifically a named variable in scope of the expression. Variables can be either *free* — occurring inside the body of a function wherein it was the argument — or *bound* — occurring inside a `let`-binding or other destructuring expression. Both variable provenances track the expression at which they were bound, allowing the optimizer to jump from the provenance term back into the expression tree where necessary. *Root* provenances are the other form of terminal provenance, indicating that a value is either declared as a global, or is a constant with no further provenance.

Data provenances model super- and sub-structure value relationships, the shape of these provenance terms mimic the shape of the value's type. K3 tracks record `R`, tuple `T` and option `O` provenances, with additional information in the case of records and tuples to determine the location of the subcomponent. From an optimization perspective, these are some of the most commonly queried provenance relationships, as they allow for many of the copy elision optimizations discussed in Section 4.1.

Structure provenances represent the more complex value relationships. *Derived* provenances (p^{+}) indicate general derived-from relationships — A value having a derived provenance (p_1, p_2, p_3) was the result of an expression which used *all* of those values, in some computation. *Set* provenances are an instance of non-determinism in the provenance representation, and are used to model conditional statements. The expression `if` q `then` t `else` e has a provenance $\{p_q, (p_t, p_e)\}$. This level of non-determinism allows K3 to be a conservative approximation in the worst case, during the optimization.

The provenance of a *function* is represented as $\lambda\,(id, p)$, indicating the name of the function's formal argument, and the provenances of all closure-captured values, as well as the body of the expression. *Applications* on the other hand store just the constituent provenances of function and argument.

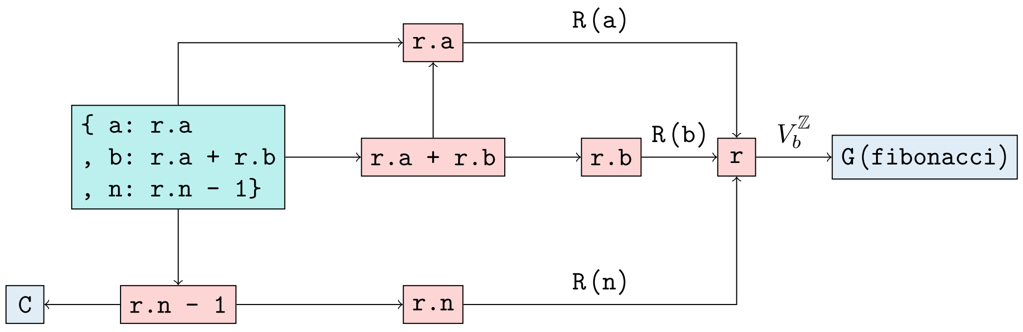


Figure 3.2: Provenance graph for record message in Fibonacci example (Listing 2.1)

Figure 3.2 shows the provenance term for the recursive record message from the Fibonacci example in Listing 2.1. The message's provenance (in green) is derived from the provenances of all three of its fields. `r.a` is derived from its source record `r` by projecting field `a`; `(r.a + r.b)` is a composite expression derived through computation from `r.a` and `r.b`, which in turn is derived from `r`; `(r.n - 1)` is also computationally derived from `r.n` (also a field of `r`), and a constant, whose exact value (1 in this case) we do not track. `r` itself is a bound variable, introduced at the global trigger `fibonacci`.

3.3.2 Provenance Queries

Value Equivalence

The notion of value equivalence motivated above can be expressed as term equivalence over the inferred provenance terms. This in itself is used heavily throughout the K3 compiler, and in effect and materialization analysis in particular. For example, the query "is the value x read-only within the expression e" requires two pieces of information: how to identify x, and how to determine if a given effect is actually on x.

Value Subsumption

We can extend the notion of value equivalence to something more general. In the presence of data substructure relations such as options, tuples and records, simple value-equivalence is not sufficient. Listing 3.1 and Listing 3.2 show two examples, each defining a record `r`, and using a combination of it and its fields.

Listing 3.1: Superstructure Relevance

```
let r = {a: [1.5, 4.5], b: [1, 2]} in (
  r = {a: [1.0, 2.0], b: [6, 7, 8, 9]};
  f(r.a)
)
```

Listing 3.2: Relevance of Sibling Substructure

```
let r = {a: [1.5, 7.5], b: [1, 2]} in (
  r.b.insert(10);
  f(r.a)
)
```

In both cases, the optimizer will inspect the call `f(r.a)` to determine the best calling convention and in doing so, will initiate a provenance search on `r.a`. However, value equivalence alone is insufficient in *both* cases, since `r.a` itself is not used anywhere else.

In Listing 3.1, the entire record `r` is reassigned, and therefore the value of `r.a` would change, impacting the optimizer's decision at the subsequent callsite. To handle this scenario, we need a notion of *value subsumption*, representing the superstructure-substructure relationship between a record and its fields. Since provenance trees are already represented as tree-structured terms, value subsumption equates nicely with a *subtree query*. The provenance of `r` is a subtree of the provenance of `r.a`, indicating that a potential effect on the former might also be an effect on the latter. We also refer to this kind of query as an *occurs-in* query.

A dual to this scenario is shown in Listing 3.2 where only `r.b` is modified before the call `f(r.a)`. In this case, a search for `r.a` *would not* turn up any matches, since it does not occur in `r.b`, only in `r`.

Value Liveness

A related query on provenance is value liveness and escape, which has multiple forms:

- What is the scope of value x?
- Does x strictly outlive y?
- Does x escape the function y?

Unlike Rust, K3 does not directly track lifetimes through explicit annotation of lifetime variables or regions, all value lifetimes in K3 are inferred from provenance. Lifetime information is used in practice to achieve better memory performance, usually in relation to allocation placement and garbage collection. K3 extends its use to value initialization, eliding copies if it is known that a value will both remain constant and be in-scope for the duration of the program where the copy would otherwise be required.

3.3.3 Contemporary Approaches to Provenance Analysis

Provenance and lineage are well-studied concepts in systems literature; databases use logging to track where the results of a transaction came from, while distributed data processing systems such as Spark store the lineage of their intermediate result datasets to facilitate their reconstruction in the event of a node failure. In both cases, provenance is used to support failure recovery, and the information stored is capable of reconstructing the output result exactly.

As K3 uses provenance at compile-time rather than run-time, its role and structure are different from existing implementations; K3 only stores data dependency relations (and not exact values), but to a greater level of detail. K3's goal isn't to be able to reconstruct a value in the event of a failure, but to be able to tell which values will be necessary to construct it ahead of time, and to be able to optimize that construction.

3.4 Effect Analysis: Identifying Value Conflicts

While effect analyses are well studied in programming language literature [49], K3 extends the classical treatment to *immediate* and *deferred* effects, as well as using *effect signatures* to interface with external code.

Effect analysis is crucial to K3's ability to perform the kinds of optimizations it does on impure code — program transformations such as dead-code elimination are straightforward in pure functional languages. In K3, the optimizer must ensure that its transformations preserve the source program's observable effects, using information it gains from effect analysis.

3.4.1 Representation

K3's effect grammar is described briefly in Figure 3.3. Every expression is attributed a *pair* of effects f, representing their *immediate* and *deferred* effects. Immediate effects would occur when the expression is evaluated, while deferred effects would occur when the *result* of the expression is evaluated, as is the case with higher-order functions.

Primitive effects are the leaves of the effect tree, denoting concrete effects on actual values. The values which are the targets of the effects are represented using the provenance terms described above; this allows the optimizer to perform occurs-in checks to determine if an effect on one value is reflected on a related value. K3 tracks the two main primitive effects on values — reads and writes — as well as a catch-all input/output effect for all other unknown operations such as printing to the screen, or sending a message.

$$
\begin{array}{rcll}
f & ::= & f_p \mid f_v \mid f_s \mid f_\ell & \textit{effects} \\
f_p & ::= & \texttt{Read}\,(p) \mid \texttt{Write}\,(p) \mid \texttt{IO} & \textit{primitives} \\
f_v & ::= & V_f^{\mathbb{Z}} \mid V_b^{\mathbb{Z}} & \textit{variables} \\
f_s & ::= & \phi \mid f\ ;\ f \mid [f] \mid \{f^+\} \mid \lambda\,(f_1, f_2, f_3) \mid \oplus\,(f_1, f_2) & \textit{structure}
\end{array}
$$

Figure 3.3: K3 Effect Grammar

Effect variables (free $V_f^{\mathbb{Z}}$, and bound $V_b^{\mathbb{Z}}$) are used as in provenance, to be able to represent the effects of an unknown quantity such as a higher-order function as the argument to another. Unlike provenance however, effect variables are mostly degenerate, and can therefore be inlined where necessary.

Effect structures are where K3's effect system differs from conventional approaches; in addition to the basic nullary effect ϕ and effect sequencing ;, K3 also supports a number of more complex effect composites. Effect *loops* $[f]$ represent effects that are repeated, as with operations within collection transformers, K3's only form of iteration. Effect *sets* are similar to their provenance counterpart, non-deterministically indicating a set of possible effects.

Function effects are used to model K3 functions in three components — creation, execution and result. Creation effects occur when the function is declared, and generally only consist of the effects of closure construction — read effects on all closure-captured values. Execution effects are the effects of the body expression of the function, which is able to access effect variables corresponding to its arguments (free) and closure (bound). Finally, result effects represent the stored effects of the return value of the function, necessary to properly characterize higher-order functions, *Application effects* correspondingly track the effects of their constituent function and argument.

3.4.2 Effect Schedules

Even though K3 effects are tree-structured terms, it is useful to represent them as linear *schedules*, as in the transaction processing literature. Schedules are obtained by performing an in-order traversal of the effect tree, and effect queries may be made in the form of subsequence queries on the schedule. The representation we adopt for effect schedules is simple; Primitive effects on values are written as $R[x]$ and $W[x]$, with $I[x, y]$ indicating distinctly, the initialization of x *from* y. For simplicity, if x is initialized from a constant, the initialization effect is further shortened to $I[x]$. Both x and y are represented in practice by provenance terms, the specific structure of the provenance is omitted for brevity. Effects are sequenced with , and repetition of effects is denoted by a Kleene star; e.g. $(I[x, y], R[x], W[x], W[y])*$ represents the repeated initialization of x from y, followed by a read on x and a write on x and y respectively.

3.4.3 Effect Queries

The schedules produced by effect analysis are queried throughout the rest of the compiler, in other analyses and optimization passes. While free-form traversal of the effect schedules (and the original effect terms) is permitted, a small set of queries generally suffices for most scenarios.

The most common queries made on the K3 effect system are also the simplest — `hasRead` and `hasWrite`. These simply check if an expression contains a read (or a write) on a given value, identified by its provenance. Given a schedule $\mathcal{S}$, this is equivalent to asking if $R[x] \in \mathcal{S}$ (or $W[x]$ respectively). Narrowing the scope of the effect query is done by generating the schedule on the fly for a specific part of the program, usually the body of a function, or the continuation of a specific expression.

Moveability

Apart from the simplest of queries, the most common pattern of access on the effect structures is to determine *moveability*. Moveability is the property of a source value, that indicates that its resources may be transferred to the target value without breaking observable isolation. Intuitively, a value is moveable if it is never inspected after the move — since moving a value vacates its original contents, the observable semantics of the program would be violated if its (unspecified) contents were subsequently observed.

Some implementations of move semantics [53] adopt this strict view of moveability; K3 however takes a less stringent interpretation, where the source value *can be inspected*, as long as it will have

```
declare group_by:
 (content -> a) -> (b -> content -> b) -> b -> [(a, b)]
 with effects \keyF -> \accF -> \z ->
  R[self]; R[z]; (R[content]; gbF content; (gAccF z content))*
```
Listing 3.3: Effect Signature of a K3 Group-By Function

been reinitialized before any such inspection.

For example, consider the effect sequence $\{I[x], I[y], I[x,y], W[y], R[y]\}$. Here, the initialization of x from y can be performed by move, since the invalid contents of y are never observed — the final read on y follows a write, which reinitializes y.

Since ownership transfer in practice involves the exchange of resources between source and targe values, the initialization $I[x,y]$ of x from y itself induces $R[y], W[x], W[y]$. If there were a $R[y]$ before a $W[y]$ following the initialization, it would result in a *write-read* conflict on y. We can formalize the definition of moveability as follows: "The initialization $I[x,y]$ at expression e can be performed by `Moved` if no *downstream* expression d of e contains the effect subsequence $R[y], W[y]$." An expression d downstream from e if it would be executed chronologically after e; that is, it is a subexpression of the *continuation* of e.

3.4.4 Effect Signatures

While effect analysis can characterize code written in K3, additional measures must be taken in order to interoperate with external library code. K3 provides a surface syntax for the effects of an external declaration, in order to give it a *signature* which can be used in its place during effect analysis. This allows optimizations to function correctly in the presence of arbitrary external code; their effects will be taken from their user-specified signature.

For example, a simple external hash function with the type signature `hash: a → int` would have the effect signature $\lambda x.R[x]$; `IO`, indicating the read on the value to be hashed, and possible external effects necessary to access the random number generator. For a more complex function such as K3's built-in group-by operator, the declaration would appear as in Listing 3.3.

The terms `self` and `content` are keywords used to refer to a collection and a representative element, and are necessary to provide arguments to the application of the effect functions `keyF` and `accF` in the effect signature.

3.4.5 Contemporary Approaches to Effect Analysis

The notion of effects — as reads and writes on data items — is also well known in the systems literature; transaction schedulers use the notion of serializability of schedules to predict data conflicts between transactions, in order to avoid inconsistency in the database during concurrent execution. Due to the simple structures of transactions however, databases do not need to store any representation more complicated than a linear schedule, and the queries necessary to show if the schedules are serializable under a desired definition are straightforward.

K3 uses its effect analysis for much the same purpose, but over a more expressive underlying structure, and allowing a richer query interface to the gathered information. This goes hand-in-hand with optimizations such as automatic parallelization, which may need to identify conflict-freedom using effect information for arbitrarily shaped program fragments, at varying granularities.

The application of effect analysis to optimize UDFs is also a first in K3. Existing systems with UDF support generally take one of two approaches to integrating UDF code with the core dataflow. If the structure of the UDF is sufficiently restricted where properties such as commutativity and associativity of effects are true by construction, no special care needs to be taken to ensure that the semantic requirements of the program are met (systems such as Tupleware [21], etc. — most compilation-based approaches to date); alternatively, the system might eschew guarantees of semantic correctness, and assume that the user understands the possible consequences (Spark, Impala [42], most interpretation-style approaches).

Through its use of effect signatures, K3 maintains its ability to maintain the correctness guarantees of the former approach without its restrictions; programmers exercise exactly how much control they desire over the integration of their external functions into the rest of K3's infrastructure.

3.5 Source-Level Optimization

The combination of the type, provenance and effect analyses described above allow the K3 compiler to undertake a wider variety of optimizations than would otherwise be possible. The most prominent such optimization is *fusion* or *deforestation* — an optimization typically used in functional languages to reduce or eliminate the creation of intermediate terms. In select cases, K3 can use the results of provenance and effect analysis to perform fusion in effectful contexts, when those effects are self-contained and preserved through the transformation. One such instance — *fold fusion* — is described in this section.

3.5.1 Fold Encoding and Fusion

K3's fusion optimization to remove intermediate collections for pipelined data processing uses the following two-step approach: i) we *encode* all collection transformers as `fold` transformers; ii) we *fuse* the accumulating functions of chained fold operations through function composition. We can compose two functions if at most one of these functions exhibits a `write` or `io` effect as obtained from our effect analysis. Consider the following select-project-aggregate example:

```
c.filter (\x->x>5).map (\x->x+1).fold (\acc x->acc+x) 0
```

Fusion Step 1: Encode transformers as folds

```
c.fold (\acc x -> if x>5 then acc.insert x; acc) empty
 .fold (\acc x -> acc.insert x + 1; acc) empty
 .fold (\acc x -> acc + x) 0
```

Fusion Step 2: Compose fold functions

```
c.fold (\acc x -> if x > 5 then acc + x + 1 else acc) 0
```

K3 also encodes a `groupBy` transformer as a `fold` with a hash table or array accumulator. We can fuse chained group-bys (`c.groupBy` g_1 a_1 z_1`.groupBy` g_2 a_2 z_2) if the following properties hold over the pair of grouping (g_1, g_2) and accumulation (a_1, a_2) functions:

1. g_2 produces a *subkey* of g_1, and the accumulating functions a_1 and a_2 both append to a (nested) collection. This occurs in the exchange operator when partitioning by keys and then by peer, where keys are associated with peers.
2. g_2 produces a *subkey* of g_1, and the functions a_1 and a_2 support the generalized distributive law, or are associative.

In these instances, we can generate a single fused `groupBy` that buckets by g_2, and applies a fused accumulator $a_1 \circ a_2$ that builds a nested collection (case 1) or aggregated collection (case 2).

3.5.2 Ad-Hoc Fusion with Delimited Compilation

As an alternative to performing fusion within the compiler, we may leverage the rewriting capabilities of the delimited compilation framework to achieve local rewrites as well. Listing 3.4 demonstrates one such example, specifically matching against a pattern of `filter` → `group_by` → `fold` → `group_by`, common in distributed join implementations.

This approach allows compiler rewrites to be packaged outside of the compiler itself in library-form.

```
((((((( ?e )
  .filter   ?filterF)
  .group_by ?gbF1 ?accF1 (?z1 : ?accT1))
  .fold     ((?foldF1) @:Accumulate) ?fz1)
  .group_by
    ((?gbF2) @:Projection)
    ((?accF2) @:Accumulate) ?z2)
) @:Fuse
) : collection ?t @Map
=> ( ($.[e].fold @:AccumulatingTransformer)
  (\acc2 -> \v ->
   if $.[filterF] v then
   ( let k  = $.[gbF1] v in
     let k2 = $.[gbF2] {key: k, value: $.[z1]} in
     ((acc2.upsert_with {key: k2, value: $.[fz1]}
      (\_ -> let ncz1 = $.[fz1] in
        ((ncz1.insert
          {key: k, value: (($.[accF1] $.[z1]) v)});
         {key: k2, value: ncz1}))
      (\acc -> ((acc.value.upsert_with
        {key:k, value: $.[z1]}
        (\_   -> {key: k, value: (($.[accF1] $.[z1]) v)})
        (\old -> {key: k, value: (($.[accF1] old.value) v)}));
      acc)));
     acc2))
   else acc2)
 (empty $::[t] @Map)
)
```
Listing 3.4: Manual Fusion through Delimited Compilation

Chapter 4

The Compiler Backend

The backend of the K3 compiler deals with the transformation of K3 code into an executable, via code-generation to C++. Primarily, this consists of *materialization*, the task of determining whether and how copy operations may be elided during code generation, while preserving the intended copy-semantics expected by the K3 programmer. The conceptualization and implementation of this analysis presented here — and its formalization in Chapter 6 — form the bulk of the novel research work in this thesis.

4.1 Materialization Analysis

Materialization analysis, addresses the optimization of *value initialization*. This looks at reducing the number of times data is copied between values, by determining if the level of isolation provided is necessary given the context in which the values are used. Proper materialization analysis allows K3 code to achieve a level of memory efficiency comparable to either pure functional immutable code (where all data is referenceable) or imperative code with in-place modification.

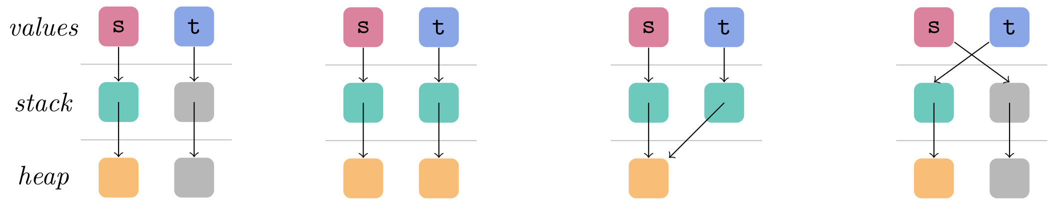
4.1.1 The Value Materialization Problem

Figure 4.1 shows the `compute_sum` trigger from Listing 2.1, annotated with the program points at which K3 would infer materialization decisions. Each point corresponds to a value initialization, and will be assigned one of the primitive materialization methods discussed below.

A naïve implementation of K3's copy semantics would result in a copy being performed at each of the marked materialization points. This is both extremely inefficient and unnecessary — fidelity to copy semantics does not require an actual implementation of copy semantics — depending on the

```
trigger compute_sum: () = \_1 -> (
  result =2 (numbers3.fold
            (\acc4 -> i5 -> ((stdout, me) <-8 (i6, acc7); i + acc))
            0.09)
)
```

Figure 4.1: Materialization decision points in a simple trigger (from Listing 2.1)



(a) Before (b) `Copied` (c) `Referenced` (d) `Moved`

Figure 4.2: Effect of primitive materialization methods on value locations and memory contents.

program, and the actual observation of variables' values in the program, copy operations may be relaxed to cheaper alternatives.

Primitive Initialization Methods

We support four primitive initialization methods, based on the standard parameter passing options defined as of the C++11 standard [14]. These methods each have different distinct semantic and performance characteristics, as described below.

The effects of each of these methods on a value's stack and heap memory allocations in the C++ memory model are shown in Figure 4.2.

The `Copied` (4.2b) method performs a deep copy of the source value into the target value. It incurs a computational cost proportional to the runtime size of the value, which can be arbitrarily large in the case of a collection. While the most expensive method of value initialization, it is also the safest, since it strictly isolates the target value from the source.

The `Moved` (4.2d) method transfers ownership of the source value's resources to the target. How this transfer is effected is generally implementation-specific, but the cost of the operation is generally proportional to the size of the *type* of the value, rather than the value itself. Moving resources from one value to another also satisfies the requirements of isolation between the source and target values, but generally leaves the source in a "valid but unspecified" state. The source value may however be reinitialized, and used thereafter in parallel with the target value.

The `Referenced` (4.2c) method indicates that the target is an *alias* to the source value; no extra allocation or copying takes place. This method has a constant computational cost to realize (the construction of the alias representation, usually a single pointer), but has the disadvantage that it provides no isolation whatsoever between the source and target values. Any effects on the source will be reflected on the target, and vice versa.

The `Forwarded` method is a utility, indicating that the target value itself is not used in any way, other than to initialize a third value in turn. Therefore, the initialization method is represented as the transitive initialization of source to forwarded value, through the target value.

4.1.2 Formulation

Materialization analysis in K3 is formulated as a constraint-satisfaction problem. Every program point giving rise to a new value is associated with a *materialization constraint*, which defines the set of viable methods available to initialize that value. Based on the K3 language grammar in Figure 2.1, this includes global declarations, data constructors, let-bindings, function creation (closure construction), function application assignment and message passing.

The constraints inferred at each program point are constructed from a language based on the boolean algebra, which is augmented with the ability to depend on the methods chosen at other program points. The grammar for the constraint language is shown in Figure 4.3.

$$
\begin{array}{lcll}
a & ::= & \texttt{Copied} \mid \texttt{Moved} \mid \texttt{Referenced} \mid \texttt{Forwarded} & \textit{atom} \\
v & ::= & (\mathbb{Z}, id) & \textit{variable} \\
e & ::= & a \mid v \mid \texttt{if}\, p\, \texttt{then}\, e\, \texttt{else}\, e & \textit{expression} \\
p & ::= & \texttt{true} \mid \texttt{false} \mid \neg p \mid p \land p \mid p \lor p \mid e \in \{a\} & \textit{predicate} \\
c & ::= & v \mapsto e & \textit{constraint}
\end{array}
$$

Figure 4.3: Materialization Constraint Grammar

4.1.3 Inference

Materialization inference proceeds through three steps; constraint *generation*, *simplification* and *solving*. A brief outline of the algorithm is shown in Algorithm 2.

Constraint Generation

Given a fully provenance- and effect-analyzed program, the compiler walks the syntax tree and constructs constraints of the form $v \mapsto e$, where v represents the program point (pair of value name and expression), while e represents a materialization expression which evaluates to the desired method. The primary structural component of e is a conditional, which allows it to decide on a

Algorithm 2 Materialization Inference

```
1: procedure InferMaterialization($\mathcal{E}, \mathcal{P}, \mathcal{F}$)
2:     constraints ← $\{e \mapsto$ Constrain$(e, \mathcal{P}, \mathcal{F}) \mid e \in \mathcal{E}\}$
3:     variable-graph ← DependencyGraph(constraints)
4:     components ← SCC(graph)
5:     decisions ← $\phi$
6:     for each component $c \in$ components in topological order do
7:         for each constraint $\{e \mapsto m\} \in c$ in bottom-up program tree order do
8:             if $m$ has no pending dependencies then
9:                 decisions ← decisions $\cup \{e \mapsto$ Solve$(m)\}$
10:            else
11:                decisions ← decisions $\cup \{e \mapsto$ Copied$\}$
12:            end if
13:        end for
14:    end for
15:    result ← Codegen(program, decisions)
16: end procedure
```

method, conditioned on a predicate p. e may also be a primitive method as discussed above, or a variable, referring to the method chosen at a different program point.

Constraints themselves are constructed by inspecting the provenance and effects of the values they constrain. Although there is no built-in restriction on the expressivity of the constraint expressions, the K3 compiler follows a set of basic patterns based on provenance and effect queries.

Application The materialization decision made at an application site determines half of how the formal argument of the function will be initialized. The call-site is responsible for indicating whether or not the actual argument's resources may be transferred into the function, or if manual isolation measures must be taken. This constraint at the application site represents the lifetime of the argument value — if it is no longer alive after the call in the caller's context, its resources may be moved to the callee.

Functions The other half of the function call materialization responsibility lies within the function, which must indicate its own *requirements* — whether it must have an isolated copy to execute its function body without violating observable effects, or if a read-only instance is sufficient.

Functions also introduce a number of other values, corresponding to the closure of the function. Each of these values must be initialized, but when the function is constructed, rather than when it would eventually be called. K3 models each closure variable as an immediate function construction/application. The actual argument in this case is implicitly in scope when the function is constructed, and the usage of the formal argument is that of the corresponding closure variable.

Data Constructors and Collection Mutators Like function application, data constructors are also an expression form where values are passed from one context into another. Indeed, most functional languages model them exactly the same as native functions. From a materialization perspective, data constructors are distinct from functions in that they *always*[1] require an isolated copy of their argument; that argument therefore cannot be initialized by reference. Moving the value then becomes the most efficient materialization method, which is dependent on the context of the data construction. Collection mutators are equivalent, since insertion into a collection requires a copy of the element, however it is initialized.

Consider materialization point 3, from the example in Figure 4.1. The decision made at this point determines how the `numbers` collection is passed into the `fold` transformer. Traditional built-in implementations of collection traversals always pass-by-reference, under the assumption that (i) the accumulating function of the fold is generally not side-effecting, and (ii) if it was, any copies necessary would be taken inside the function, rather than at the invocation of the fold itself.

The materialization constraint set corresponding to the `compute_sum` trigger is shown in Figure 4.4. Some positions are degenerate decisions, where the value being introduced is either not used (v_1), or can be blindly moved as the initializer is a computational temporary (v_2, v_8, v_9). The remaining decisions are conditioned on others within the same constraint set; for example at v_3, the `numbers` collection should be moved into the fold if an isolated copy is required of the element drawn at each iteration of the fold (v_5). In turn, the constraint at v_6 indicates that such a copy is required, since the value's ownership is to be given to the tuple constructor of the message value. v_8 then specifies that the tuple may be moved directly into the messaging operator, as it is an unnamed temporary.

Through this sequence of constraints, the compiler has inferred that memory ownership may be transferred from the global collection `numbers` (v_3) to the loop index variable `i` (v_5), to the message tuple (v_6), and finally into the runtime through the message operator at v_8. This entire sequence of data movement would have been accomplished using only move operators, with no copy operation at all. In the example given, the data types are simple enough that a copy would not be substantially less efficient. This does however generalize to the case where the values themselves are large collections, where a copy elision would have a cascading effect throughout every iteration of the fold loop.

[1]Not quite, although an even more complex form of lifetime analysis (such as region types) is required to keep track of the provenances of individual distinct fields of a composite data type.

$$\begin{aligned}
v_1 &\mapsto \texttt{Referenced}\\
v_2 &\mapsto \texttt{Moved}\\
v_3 &\mapsto \texttt{if}\, v_5 \in \{\texttt{Moved}\}\,\texttt{then}\,\texttt{Moved}\,\texttt{else}\,\texttt{Referenced}\\
v_4 &\mapsto \texttt{if}\, v_7 \in \{\texttt{Moved}\}\,\texttt{then}\,\texttt{Moved}\,\texttt{else}\,\texttt{Referenced}\\
v_5 &\mapsto \texttt{if}\, v_3 \in \{\texttt{Moved}, \texttt{Copied}\}\,\texttt{then}\,\texttt{Moved}\,\texttt{else}\,\texttt{Referenced}\\
v_6 &\mapsto \texttt{if}\, v_5 \in \{\texttt{Moved}, \texttt{Copied}\}\,\texttt{then}\,\texttt{Moved}\,\texttt{else}\,\texttt{Copied}\\
v_7 &\mapsto \texttt{if}\, v_4 \in \{\texttt{Moved}, \texttt{Copied}\}\,\texttt{then}\,\texttt{Moved}\,\texttt{else}\,\texttt{Copied}\\
v_8 &\mapsto \texttt{Moved}\\
v_9 &\mapsto \texttt{Moved}
\end{aligned}$$

Figure 4.4: Materialization constraint set for the trigger `compute_sum` (from Figure 4.1)

Constraint Simplification

Once all the materialization constraints for a program have been accumulated into a global constraint set, K3 performs a simplification pass before continuing to constraint solving. Simplification is performed for two primary reasons — performance, and cycle resolution.

Constraint simplification in K3 consists of basic constant propagation and short-circuit evaluation of the boolean algebra. From a performance perspective, simplification greatly reduces the size of the constraint set by removing redundant predicates. Failed occurs-in checks can be completely pruned; if x does not occur in y, then it is not necessary to keep any materialization expressions corresponding to if they did. By pruning entire subtrees of the constraints, spurious dependencies are also removed — again, if x does not occur in y, the initialization of any value from x cannot depend on how y itself was initialized. This allows constraint generation to produce cyclic constraints, having most of them removed before solving.

Constraint Solving

While simplification can reduce most constraints to trivially solvable forms, some program patterns can result in legitimate dependency chains or cycles. K3 first sorts the undecided materialization variables in topographic order according to the dependency graph. If a cycle is encountered, K3 breaks the dependency using the heuristic that strict isolation methods (`Copied`, `Moved`) should be performed as deep as possible into the program. This heuristic is based on the assumption that if isolation can be established at the lower levels, the higher levels of the expression tree (which contain potentially larger structures) can be initialized by reference, leading to an overall lower cost strategy. Other heuristics exist; determining if a closed-form solution for the optimal cost of

```
f = [] (auto&& _0) {
  auto& x = std::forward<decltype(_0)>(_0);
  return x + 1;
};
```

Listing 4.1: Generated code for a read-only formal parameter function

initialization across the entire program is a topic of active research.

4.2 Code Generation

The final stage of K3 compilation is code generation, wherein the fully analyzed and annotated K3 program is transformed into C++ code with appropriate low-level decisions made by the compiler. This section describes the salient parts of this process, including the reification of materialization decisions, the generation of ad-hoc record types, and that of the entire application shell.

4.2.1 Reifying Materialization Decisions

Once the materialization decisions for each value in the program have been obtained, they are attached to the program tree prior to code generation. At each expression form where a new value is introduced, the decision for that value is inspected to determine the necessary decoration for the initialization in C++.

A key part of the code generation process is reliance on *perfect forwarding*, a feature introduced in C++14 which allows the programmer to construct *forwarding references*. These are special types of references which act as a materialization proxy: if a program contains the sequence $[I(y,x), I(z,y)]$ and y is not otherwise used, then it may be declared as a forwarding reference, proxying its initialization from x to that of z from itself. While this is still a C++ compile-time determination, it greatly reduces the amount of code which must be explicitly generated by K3 to support all the desired materialization scenarios.

Forwarding references are further used in combination with C++14 generic lambdas to support perfectly forwarded anonymous functions. For example, the K3 function $\lambda\, x \to x + 1$ would generate the C++ code in Listing 4.1, while the function $\lambda\, x \to c.\,\texttt{insert}\, x$ (for some global c) would result in Listing 4.2.

In both functions, the formal parameter of the generated function is a forwarding reference (declared as `auto&&`), signifying that the function can accept *any* form of argument (l-value, r-value, etc.). During C++ compilation, a new template will be instantiated for each argument type deduced.

```
g = [] (auto&& _0) {
  auto&& x = std::forward<decltype(x)>(_0);
  return c.insert(std::forward<decltype(x)>(x));
};
```

Listing 4.2: Generated code for a read-write formal parameter function

The first line of the function body defines the original function parameter, initialized by forwarding the reference. In the first example, the function body $x + 1$ does not modify x, nor does it require an isolated copy of it. Therefore, x is initialized as a simple reference, and used directly in the rest of the function. A call to `f(x)` will never copy x, since it is handled by reference at all points.

In the second example, the function body uses x only in order to pass on to another function `insert`. x is itself declared as a forwarding reference, and passed transparently onwards. If g is called as `g(x)`, x will be inferred as an l-value reference and would eventually be copied inside the call to `insert`. If however g is called as `g(std::move(x))`, the forwarding references would carry over this intent to move, resulting in a move inside `insert`. This allows code generation to respect the separation between the caller's ability to provide a value, and the callee's requirements of it.

Several barriers were encountered in the implementation of this process; primary among them was the massive size of the constraint set for larger programs. Every value introduces a new constraint, and the size of that constraint grows approximately with the size of the value's provenance and the expression itself where it is introduced. This means that some constraints could potentially be as large as the program itself. Pushing simplification into the constraint generation process helped greatly with this problem, both in the direct reduction in the size of constraints, and in the indirect yet equally important stripping out of unnecessary dependencies.

4.2.2 Generation of Ad-Hoc Record and Collection Types

One of the problems tackled during code-generation is the reification of ad-hoc record and collection types used throughout a K3 program. Since the use of record types are not declared before use in the source program, these must be inferred and collected during type-analysis, for subsequent reification.

For each distinct record type (independent of field ordering), a separate class is generated; we rely heavily on C++ template metaprogramming to handle the bulk of the low-level expansion. In particular, we use the automatic generation of copy and move constructors to ensure that the materialization decisions made above may be safely effected regardless of the type of object being materialized.

4.2.3 The Low-Level Structure of a K3 Program

The end result of the code generation process is a single C++ file, consisting of a set of data type declarations and associated behaviors corresponding to the ad-hoc types described above, and a single application class definition representing the entire K3 program.

The generated application class for each K3 program is self-contained; each object of the class represents an independently executable instance with its own isolated data members. The data members of the class correspond to the global data declarations in the K3 program, while the trigger definitions are translated into member methods of the appropriate type. The cyclic scoping of members within a C++ class support a painless encoding of K3 namespacing into C++.

For each application class, a *dispatch* function is generated to efficiently route messages received from the runtime *engine*. The dispatcher uses type information from the underlying K3 program to deserialize messages from the network, and cast them to the appropriate message type.

4.3 The Runtime System

In this section, we provide an overview of the K3 runtime system — approximately 15.5ᴋLOC of support infrastructure written in C++, which handles low-level specifics of running K3 programs.

4.3.1 Control Infrastructure

The job of the control infrastructure of the K3 runtime system is to use the application class created during code generation and create the final usable executable. This is done by integrating the application class with an *engine*, a software component which handles a variety of low-level tasks.

Since the top-level of K3 is purely declarative, the result of code generation does not specify how a given program begins or ends. While termination might be defined intuitively as the state in which no further messages are to be processed, the application still requires an initial message. In addition, deployments may wish to customize the values of various global variables, such as the network identifiers of the master, or other peers in a distributed environment. This is done through a command-line interface, which reads deployment information and populates the relevant fields in the application object.

Our experimental evaluation (described in Chapter 5) uses multiple K3 workers per physical machine; this level of concurrency manifests as multiple instances of the same application object running in different threads within the same container process. The engine is responsible for maintaining these application threads, as well as a common network thread which receives messages for any of the application instances, and dispatches them appropriately.

4.3.2 Data Infrastructure

K3's data infrastructure consists of a set of backing implementations for the various collection interfaces described in Chapter 2.4.1, as well as the necessary declarations to integrate them with the engine and network transport.

These implementations are currently hand-written fulfillments of the collection interface specification, and include list, set, bag and map variants. During the final phase of C++ compilation, each requirement of a concrete implementation in the K3 application object is satisfied by a corresponding implementation in the K3 standard collection library.

Chapter 5

Experimental Evaluation

In this chapter, we describe an experimental evaluation of synthesized systems generated with K3 for a variety of common large-scale distributed data processing workloads, and compare their performance on a number of metrics with implementations on other state-of-the-art contemporary platforms.

5.1 K3 Evaluation Workflow

5.1.1 Compilation

Figure 5.1 depicts pictorially the sequence of work within the implementation of the K3 compiler. This largely follows our conceptual explanations of the preceding chapters; in this section, we provide a few particulars on the actual implementation.

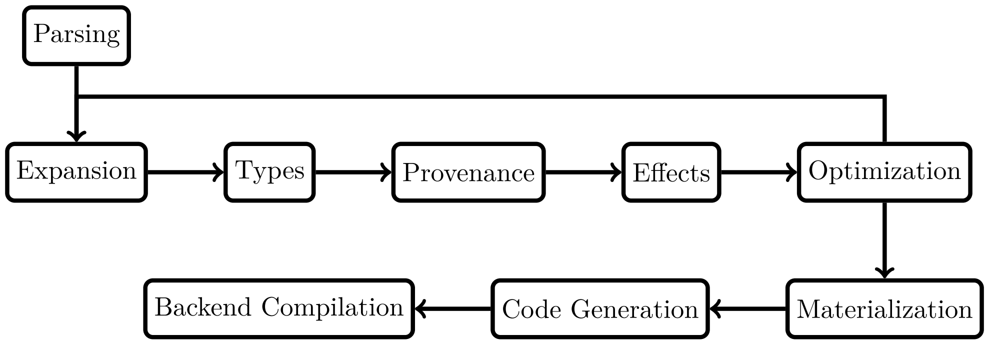


Figure 5.1: Architecture of the K3 Compiler

The core of the K3 compiler is implemented as a 40kLOC Haskell software artifact. The parser constructs an abstract syntax tree annotated with syntax information; each subsequent pass of the

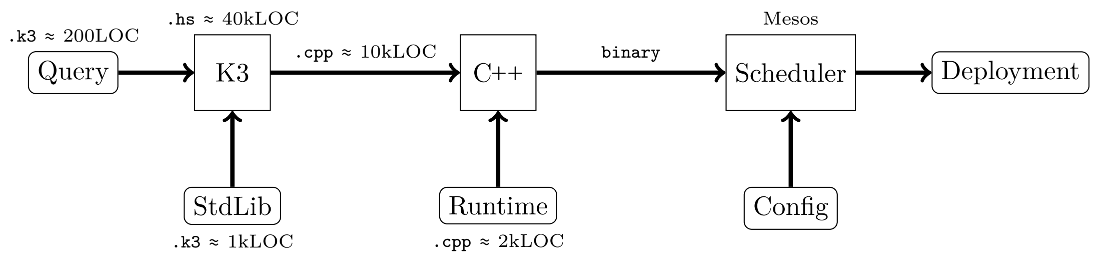


Figure 5.2: End-to-End compilation/deployment lifecycle of a K3 program

compiler either accumulates more information to be attached to the tree, or uses that information to perform tree transformations. As such transformations may invalidate previously attached program metadata, the compiler iteratively alternates between analysis and transformation until a fixed-point is reached.

The frontend of the compiler — metaprogram expansion, type, provenance and effect analyses, and source-level optimization form one such loop. Metaprograms may dispatch on the basis of static analysis information, and may in turn perform tree rewriting or introduce global declarations which necessitate re-analyzing the program.

The backend of the compiler — materialization analysis, code generation, and C++ compilation does not alter the static analysis artifacts of the program, and do not need to be iterated upon.

Our implementation heavily leverages parallelism to support the holistic static analysis of large and complex programs. Each of K3's static analyses are parallelizable at the per-declaration granularity — we make use of Haskell's built-in deterministic parallelism to compile a single program across multiple threads, with each thread being responsible for a one or more complete declarations.

For even larger programs, the K3 compiler supports *distributed compilation*, where the compilation workload may be split across multiple physical machines in a distributed network. The independence of static analysis across declarations allows for a compilation protocol with minimal-communication.

5.1.2 Deployment

Figure 5.2 shows how the K3 compiler fits within the larger deployment workflow of a K3 program.

Each query starts out as a physical plan implemented directly in K3; this choice reflects our ability to compare synthesized K3 implementations with those produced by state-of-the-art query optimizers found in other systems. Any optimizations or low-level techniques used in these systems — such as join-order optimization, vectorization, etc. — may be applied equally to K3 programs,

Query	Features	Varied Selectivity Parameter
1	Group-By	Upper-bound on ship date
3	3-Way Join, Group-By	Size of selected market segments
5	6-Way Join, Group-By	Size of selected regions
6	Aggregate	Lower-bound on ship date
18	3-Way Join, Correlated Sub-Query, Group-By	Lower-bound on subquery aggregate
22	Sub-Queries, Not Exists, Group-By	Size of valid phonebook

Table 5.1: Feature coverage of select TPC-H queries, and selectivity parameter varied.

as they are largely orthogonal to the techniques we evaluate here.

The writing of physical query plans in K3 is simplified greatly by the delimited compilation support explained in Section 2.4.2; this allows K3 programmers to efficiently operate at an appropriate level of abstraction.

Once compiled into a binary as described in the previous section, the synthesized implementation is shipped off to the distributed deployment environment. This environment is managed by a custom-implemented cluster scheduler based on Mesos [35]. This scheduler handles the initial handshake communication necessary to run the application; once started, network communication within the application is handled by the K3 runtime. In addition, dataset files are shipped independently out-of-band, pre-partitioned and compressed in line with standard practices.

5.2 Workloads and Datasets

5.2.1 TPC-H

The bulk of our evaluation uses the TPC-H Benchmark [76], a set of SQL queries and associated dataset schemata for evaluating the performance of relational analytics. TPC-H is an industry standard benchmark, and consists of 22 SQL queries in total. For our evaluation, we chose 6 of these queries, which appropriately cover a wide range of SQL feature complexity. The set of queries we chose, and their corresponding feature coverage is shown in Table 5.1.

The table also shows a *selectivtiy parameter*, which describes a dimension along which the query may be parameterized, in order to increase or decrease the amount of data allowed into the query's main operators. This is used to stress-test the memory benchmarks in Section 5.3.

A sample implementation of TPC-H Query #1 is shown in Figure 5.1.

This example showcases a number of the techniques presented in Chapter 2. The main operator of the query is a group-by aggregation, which is transformed into a distributed operator through the use of the `DistributedGroupBy` annotation. Most of the boilerplate in the arguments to the

```
trigger q1 : () = \_ -> (
  (ignore
    ((lineitem.filter   (\r -> r.l_shipdate <= 19980902))
              .group_by (\r -> (r.l_returnflag, r.l_linestatus))
                        accum_agg
                        init_agg)
  ) @DistributedGroupBy(
      lbl         = [# groupby],
      clear_expr  =
        [$ lineitem = empty q1_lineitem_r @Collection ],
      peer_next   = [$ (\x ->
        ((x.iterate (\kv -> results.insert (finalize_agg kv)));
          (() @:Result))) ],
      next        = [$ ()],
      merge       = [$ merge_agg],
      coordinator = [$ master],
      nodes       = [$ peers],
      masters     = [$ masters ],
      masters_map = [$ peer_masters ],
      profile     = [$ true] ))
```

Listing 5.1: The K3 kernel of TPC-H Query #1

distributed group-by annotation are deployment-specific details; these include `master`, an identifier corresponding to the coordinating node of the distributed operator, and `peers`, a collection of identifiers referring to the various workers in the distributed deployment.

The distributed group-by is sandwiched by initialization and finalization phases. The initialization is responsible for ensuring that all workers are available and ready for execution, including ingestion of the base data, and sparking the group-by phase with appropriate timer initialization. The finalization phase involves the collection of results from the various workers back to the master, and the coordination of worker termination and appropriate timer finalization and recording. These phases are common to all of our benchmark queries, and are therefore abstracted out to their own annotation.

The datasets for the TPC-H queries are derived from the dataset generator provided with the benchmark; for our experiments, we evaluated primarily with the 10G and 100G scale factors. These sizes correspond to the total size of the dataset including all tables; depending on the query itself, only a subset of tables are used.

5.2.2 AMPLab Big Data Benchmark

The AMPLab Big Data Benchmark [11] is an alternative relational benchmark, modeling a text-processing application over HTML data retrieved from an indexer. These queries stress-test simple scans, aggregations and joins. Our evaluation includes three out of the four provided queries.

The datasets used for this benchmark are provided along with the benchmark, and are approximately 120GB across two tables.

5.3 Contemporary Platforms

The landscape of state-of-the-art distributed large-scale data processing includes a variety of specialized platforms, each focused towards a specific type of workload. Platforms generally support a common functional-esque programming interface, subject to specific choices about data/storage layout, and task scheduling. For workloads not fitting a platform's chosen paradigm, some amount of encoding mismatch is present.

Of the systems surveyed below, those we included in our benchmark are marked with their deployment information. Two main reasons account for why some systems were excluded from the benchmarks: the difficulty of actually deploying the system from the software available, or the difficulty of taking measurements other than end-to-end timing, which does not allow for controlling for a large number of extraneous performance-sensitive variables.

5.3.1 Apache Spark

Apache Spark [4] is a general-purpose platform for large scale distributed data processing. Over time Spark has evolved from a main-memory focused implementation of the map-reduce paradigm [22] into a generalized substrate for a wide variety of data processing models, including SQL [6], stream processing [85], machine learning [56] and graph processing [83].

The foundation of the Spark programming model is the resilient distributed dataset (RDD) [86], a data abstraction representing an immutable collection represented either explicitly, or implicitly through the provenance of its computation.

Spark has experienced immense popularity in production data processing, due in part to its ability to interface with a wide variety of languages and existing libraries. In recent releases, the Spark project has pursued improvements in similar veins to those proposed in K3; in particular, low-level code generation and memory management are the focus of Project Tungsten [5]. Our benchmark includes multiple configurations of Spark to showcase these improvements; version 1.6 (before Tungsten), and version 2.0.1 (both with and without Tungsten).

5.3.2 Apache Impala

Apache Impala [42] is a parallel query engine on top of Hadoop, with the goal of supporting low-latency SQL workloads over Apache HDFS data stores, without the need for layout conversion or other memory transfer operations. Impala is the open source distribution of Google's F1 [69].

Our evaluation uses Impala version 2.0 used in concert with Cloudera distribution of Apache Hadoop (CDH5), storing input tables as internal relations.

5.3.3 Apache Flink

Apache Flink [16] is a streaming dataflow engine, which compiles queries down to a streaming dataflow graph, which is then optimized and mapped to a distributed deployment. Flink's streaming model is similar to K3's execution model at the macro-level; however, its message-passing paradigm operates closer to the relation-level, rather than K3's tuple-level.

Apache Flink also recognizes the dual goals of reducing garbage collection pressure and using more efficient data representation layouts in memory [16]; it approaches this by operating directly over binary serialized data representations, reducing memory usage and overheads due to object creation and collection. Flink also emphasizes object reuse to further reduce GC burden but relies on implementation discipline to ensure that correctness of programs is maintained in the presence of reused objects and in-place mutation.

5.3.4 HyPer DB

HyPer [60] is a query compiler focusing on low-level optimization by exploiting data locality. Queries are compiled directly to LLVM, bypassing intermediate language layers such as C++, and ensuring that the locality of data necessary to use optimizations such as vectorization instructions is available.

We do not include HyPer in our evaluation suite, as the software artifact is not freely available.

5.3.5 Tupleware

Tupleware [21] is similar to HyPer in its focus on low-level compilation through LLVM. However, Tupleware concentrates on the optimization of user-defined functions (UDFs), and their optimization along with the primary query dataflow.

The Tupleware compiler uses heuristics to introspect user-defined functions compiled to LLVM, and use those characteristics to inform the joint optimization and code generation process of the UDFs with the core dataflows.

5.3.6 LegoBase

Another query compilation framework is LegoBase [41], which uses staged metaprogramming through Scala's Lightweight Modular Staging [65] to individually compile each operator of a SQL query represented as a Scala AST construction. LegoBase's use of LMS corresponds closely to K3's use of staged metaprogramming to perform optimizations, including operator fusion.

5.4 Configurations

Our experimental deployment of K3 consisted of 128 distributed workers spread evenly across 8 physical nodes with 16 hyperthreads each. Physical memory totaled 72GB on each physical node, shared evenly between each node's workers. The datasets for each workload were pre-partitioned and disseminated among the various physical nodes' local disks.

5.5 Evaluation

5.5.1 End-to-End Comparisons

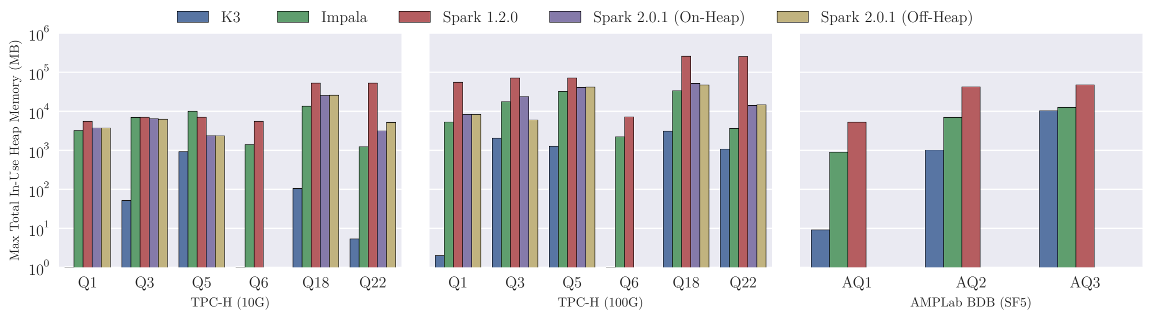


Figure 5.3: End-to-End heap memory usage comparison across all systems and workloads.

The **end-to-end heap memory usage** evaluation is shown in Figure 5.3. Heap memory usage was measured as the height of the cluster-wide peak usage during execution. K3 generally outperforms across the board, with gains ranging from 3.3x to 2638x.

K3's memory usage advantages may be correlated directly with query features and code patterns, emphasizing workload-specificity of materialization optimization. For example, both TPC-H queries Q1 and Q6 exhibit an extremely simple loop structure — a single filter over the largest relation (`lineitem`) — with Q1 performing a group-by aggregation, while Q6 performs a (non-group-by) aggregation. In both cases, K3 fuses all intermediate loops together and infers a materialization

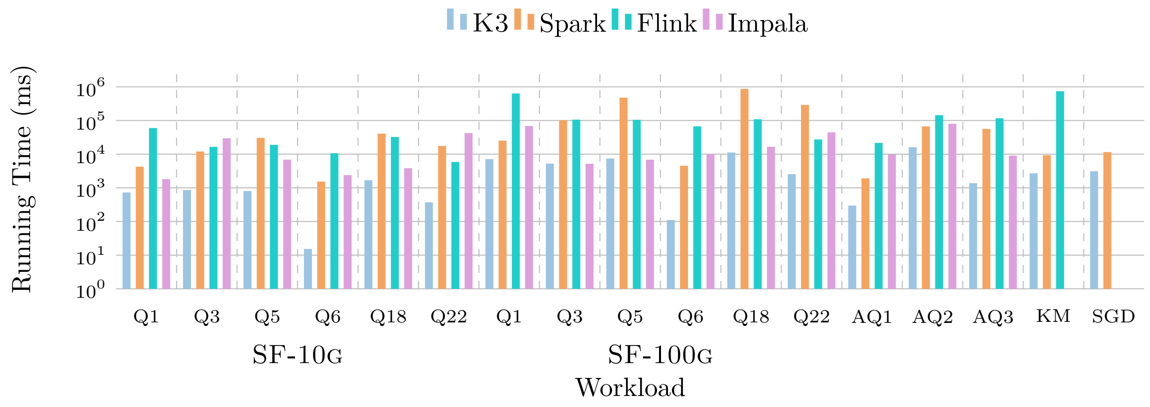


Figure 5.4: End-to-end running-time comparison of K3 against Flink, Spark (2.0.1) and Impala

scheme using almost entirely references for both loop and accumulator variables — each optimization eliminating the allocation of an entire class of intermediate data. Furthermore, the accumulator in Q6 is a single statically-sized integer, and is therefore inferred to be stack allocatable in K3. Apart from K3, only the newest version of Spark is able to take advantage of this property, in the simplest case of Q6.

On the remaining, more complex queries, materialization optimization provides a different kind of advantage — that of reducing allocation performed at the boundary between data-flow and runtime system. These queries involve significant communication, in the form of distributed joins and group-by operations. In K3, materialization is able to infer that the data to be sent is no longer needed by the data-flow, and transfers ownership of that data to the runtime system. Once sent out, the runtime system handles deallocation independently.

Spark shows modest to significant improvement between versions 1.2.0 and 2.0.1 depending on workload, attributable to work done on memory optimization through a unified memory-manager and whole-stage code generation. The on-heap variant corresponds to the traditional delegation of memory management responsibilities to the JVM garbage collector, while the off-heap variant uses manual heap allocation/deallocation through APIs such as `sun.misc.Unsafe`. The difference between the two variants in our experimentation was limited, although some queries (TPC-H Q3) show a reasonable advantage to off-heap memory.

An **end-to-end running-time** evaluation is shown in Figure 5.4. These results correspond closely with the memory experiments shown above; since the majority of relational workloads are not CPU-heavy, the predominant time-costs are memory operations, divided broadly between copy-costs and serialization-costs for communication. This is not true for the ML workloads however, where the intermediate data items (model parameters) are comparatively smaller, and consist of primitive types.

5.5.2 Isolation Variants

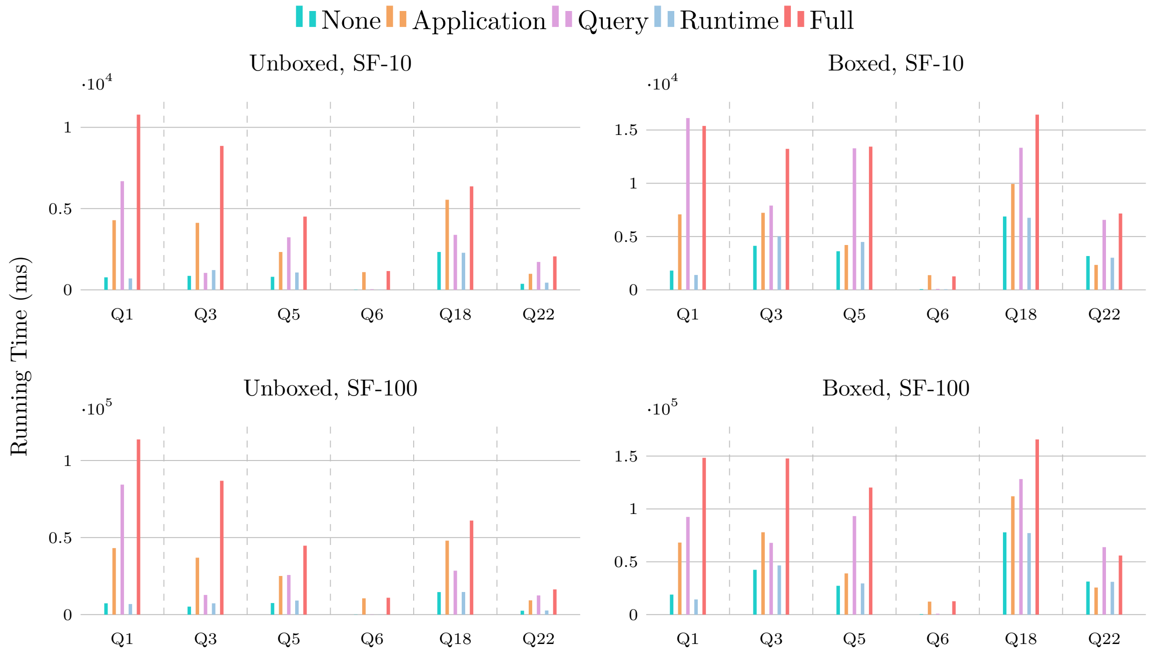


Figure 5.5: Comparison across isolation variants at default selectivity.

To further underscore this impact of memory operations on running time, we developed a number of *isolation variants* — modifications to the program and runtime system which model the existence of barriers between memory regions in existing data processing systems leading to various levels of isolation. In this part, we discuss the setup of the isolation variants, the properties of systems they model, and the impact of materialization optimization on each of them.

Application Isolation: This variant models the barrier between the primary data-flow of a query and the user-defined functions (UDFs) which perform the actual computation. The costs of transferring data between this barrier vary widely between systems; some systems require a full reallocation followed by representation changes both to and fro, while other systems only require the reallocation, allowing UDFs to operate over the same native data format. Regardless of the costs involved, most existing systems share a lack of safety at this barrier, no checks — static or dynamic — are available to verify that the memory management procedure is in concert with the semantics of the UDF.

In K3, we model application isolation by enforcing a manual data copy both into and out of the UDFs in the query.

Query Isolation: This variant models the barrier between operators in a dataflow — conventionally known as operator fusion — relating to the elimination of collection intermediates between stages of chained transformers. This technique is implemented in most languages, often in the form of operator *pipelining*, where each operator feeds its outputs row-at-a-time to the next operator, through a write buffer or shared memory location.

K3's approach to fusion is more aggressive, since our compilation process allows us to inspect the entire query pipeline and rewrite the entire sequence, performing materialization optimization over the rewritten operator. This reduces overhead from the tuple-level processing, as well as being able to handle more cases of individual operator fusion.

Runtime Isolation: This variant models the barrier between the dataflow and the underlying communication engine. The majority of time spent at this barrier is unavoidable in any system, since data to be sent over the network must be serialized in some form, requiring a representation change. However, joint consideration between the dataflow and the runtime itself allow K3 to perform a number of optimizations: moving data in and out of messages, using local shared memory to avoid network traffic on local messages, optimizing broadcast messages by using a two-tiered message tree and fusing distributed exchange operators with the user-defined functions which produced the collection to be distributed.

Each of these transformations gives K3's materialization analysis an additional opportunity to elide further value initializations, which brings down memory usage considerably, depending on the amount of data at the runtime boundary.

Stack Isolation: This variant models the capability of ahead-of-time compilers to allocate intermediates of known sizes to the stack, reducing — or eliminating — heap memory allocation costs, and pressure or reliance on a runtime garbage collector. K3's model of a stack-isolated variant involves manually boxing all composite objects in the program, outside of the runtime system and custom operator implementations which may be expected to be fully optimized. This variant is orthogonal to the others above, and is tested in addition to each above configuration.

None/Full Isolation: These meta-variants model a combination of the above — K3's default mode of operation is with no isolation, while a full isolation variant models a system unoptimized with respect to memory.

Discussion: Figure 5.5 shows K3's running time performance for each of the TPC-H queries across all the variants outlined above. The first key observation is that each query has a significantly different distribution of time across the different variants, attributable to workload characteristics.

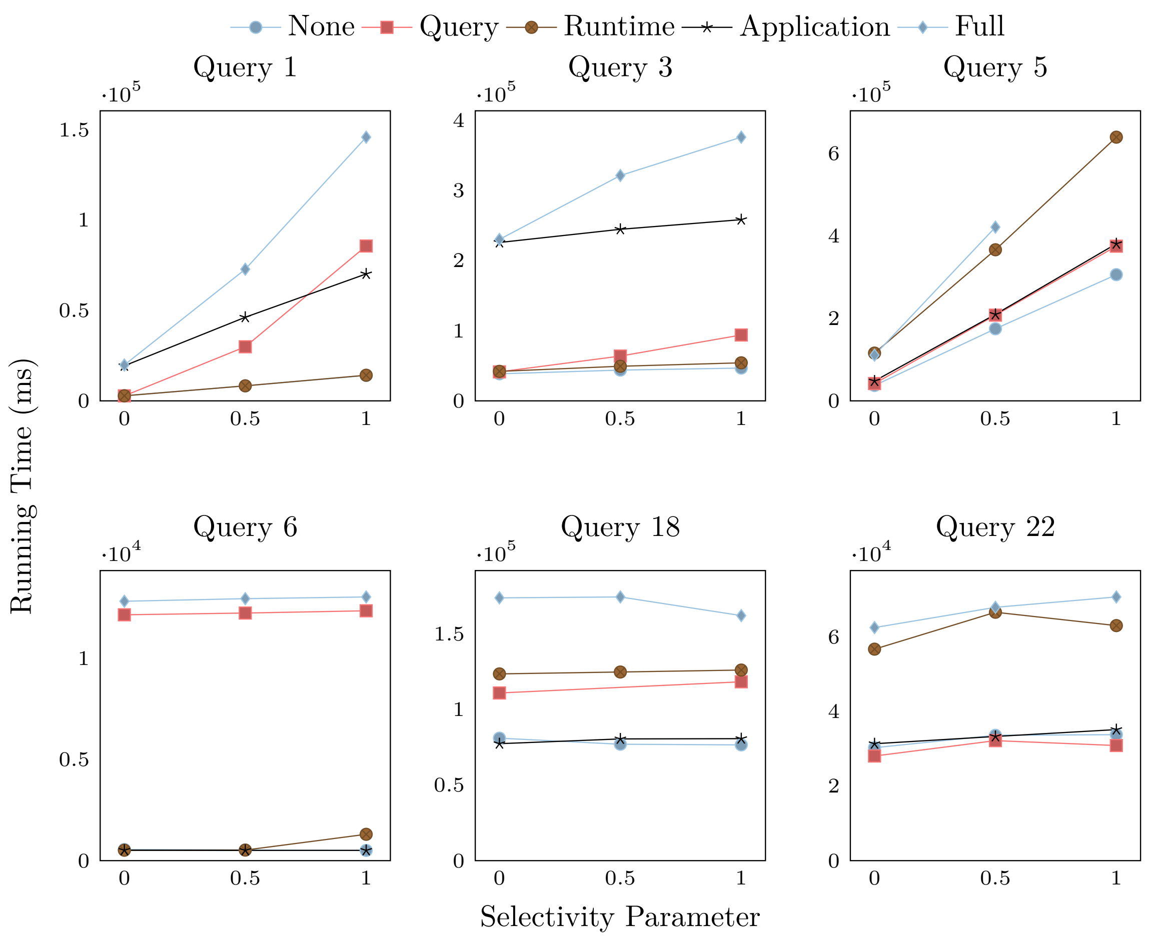


Figure 5.6: Comparison of boxed variants across selectivity thresholds.

For example, Q6 exhibits almost no slow-down under query and runtime isolation, since both local intermediates and communication intermediates can be materialized almost exclusively as references, keeping data allocation extremely low. Application isolation in contrast is expensive in Q6, as the input data is still being copied (element-wise) in this variant.

The second key observation is the 5x-10x advantage in K3's usage of ahead-of-time stack allocation — this benefit shows wherever a large number of small intermediates are used, as they have a high relative overhead of allocation/deallocation costs. This manifests in languages such as Java and others using the JVM, which will also incur additional overheads from garbage collection.

The benefits of optimizing repeated small memory operations is further showcased in Figure 5.6 — a modification of the workloads used in Figure 5.5 where the selectivities of the core data-flow scans in each query are varied to control the amount of data entering the query. For brevity, we show only the stack-isolated variants at the 100G scale factor, the other configurations behave similarly.

The same trends observed in the default selectivity case hold true but at magnified effects; where application or query isolation were dominant under default selectivities, they continue to have a high impact in absolute terms at each selectivity level. The differentiator is a greater relative impact of runtime isolation, whose effects were muted in the above experiments. Queries such as Q1, Q3 and Q5 show significantly worse runtime memory behavior due to the increased data admitted; Q5 failed to terminate due to insufficient memory under full isolation at the highest selectivity level.

5.5.3 Directly Observing Memory Load

To close the evaluation section of this thesis, we present an experiment which attempts to observe the memory impact of isolation levels more directly — through the metric of *memory load*.

For each object, we define its contribution to memory load as the product of its size (in bytes) and its lifetime (in seconds). In any but the simplest of toy examples, it becomes practically impossible to track the memory load of each and every object; in this evaluation, we track memory load in aggregate by approximately sampling the distribution of per-object memory load.

Figure 5.7 depicts the distribution of object memory loads for executions of TPC-H Query 18, under the full- and no-isolation variants described above. The x-axis represents the number of samples taken during query execution with the corresponding memory load on the y-axis.

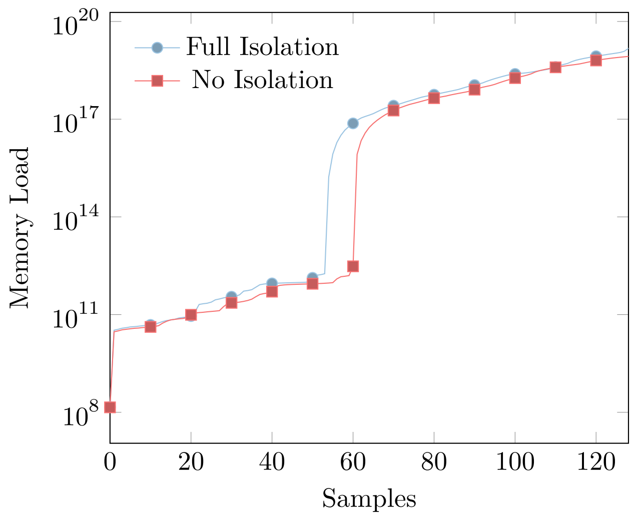


Figure 5.7: Lifetime distribution across boundary configurations for TPC-H Q18

In both variants, some number of objects at each end of the spectrum exhibit the same amount of memory load. The lower end of the spectrum constitutes small and short-lived objects such as primitive temporaries, while the higher end constitutes large and long-lived objects such as global collections.

The difference between the variants manifests in the middle of the spectrum, where more objects in the full isolation variant exhibit a higher memory load than in the no-isolation variant. This may broadly be attributed to the general reduction in average lifetimes of objects due to the removal of various isolation boundaries.

Figure 5.8 shows similar distributions of memory load at the same isolation variant (none) across a number of different TPC-H queries.

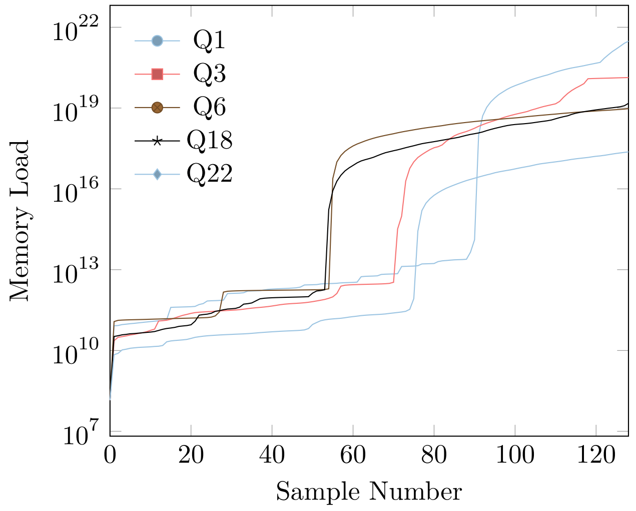


Figure 5.8: Memory Load Distribution of Objects across TPC-H Queries at Full Isolation

Two primary inferences may be drawn from this figure: first all of the queries exhibit the same general pattern of having approximately similar numbers of objects exhibiting memory loads at each end of the spectrum; this can be attributed to the work common to all queries — control flow, and the maintenance of global data structures.

Secondly, each query exhibits a transition point, the location of which is query-dependent. This would depend on the structure of the query itself, including the number and scope of iterations over collections, and the sizes and operations on intermediates.

We have not generally seen the use of direct lifetime and memory load as metric for the study of memory performance in systems, we believe it will prove a useful metric which merits further study.

Chapter 6

Formalizing Value Materialization

In Section 4.1, we discussed a static analysis pass which determined which of three different methods should be used to implement in C++, a K3 assignment operation. While the programs we generated using that technique were correct, there was no guarantee that the technique itself was correct in the general case. In particular, there is no guarantee that materialization constraint corresponding to a given assignment operation actually models the condition necessary to ensure safe materialization. Furthermore, it is not entirely obvious what a "safe" materialization is.

In this chapter, we try to rectify the situation by formalizing materialization analysis. Specifically, we boil K3 down to the bare minimum subset necessary to handle only assignment and materialization, and tackle characterization of the safe usage of move semantics in general.

6.1 Assignment Semantics

The *assignment semantics* of a language — what it means to perform the assignment of a right-hand-side expression to a left-hand-side named location — are an integral part of language design. They extend beyond just the literal assignment operator, to how a function's formal parameter is constructed from the actual parameter at each call-site, and how the corresponding return value is itself constructed when the function exits.

The choice of assignment semantics for a language informs both the behavior a programmer may expect out of an assignment operation (guarantees of data independence and isolation), as well its possible implementations (time and space performance characteristics). The most common choices for assignment semantics in contemporary languages are *copy* and *reference* semantics.

Languages differ in which types of assignments programmers may express; some languages give programmers access to both forms of assignment semantics (with one chosen as default), while others force a single assignment semantics for all programs.

With *deep* copy-semantics — the de-facto standard modern C++ — the source and target of an assignment are isolated copies of each other; a guarantee which may be relied upon. The lifetimes of each copy are likewise independent, and their memory may be deallocated independently at the end of each variable's lexical scope. The naïve implementation of copy assignment requires the allocation and construction of the copy, which may be arbitrarily expensive depending on the size of the object.

By contrast, with reference semantics — the default in Python and Java — the source and target of an assignment are both aliases to the same underlying object, and modifications on one reflect on the other. The lifetimes of the variables are tied to their lexical scope, whereas the lifetime of the underlying data may be arbitrary. Each must be deallocated either manually based on the programmer's own estimation of when it is safe to do so, or using a runtime garbage collector. However, the naïve implementation of a reference assignment is a constant-time operation, typically requiring only a fixed-size pointer copy.

These trade-offs are summarized in Table 6.1, with the addition of a third possible assignment method: the *move*.

Method	Cost	Equivalence?	Independence?
Copy	High	✓	✓
Reference	Constant	✓	✗
Move	Low	✗	✓

Table 6.1: Assignment Semantics Trade-Offs

In move semantics, ownership of memory is *transferred* from the right-hand-side to the left. This method establishes independence between the two sides, since resources are not shared. However, the two sides are no longer equivalent. A move is generally equivalent in cost to a *shallow copy*: cheaper in both time and space than a deep copy, but more expensive than a pointer switch.

Until recently, move semantics were implemented as a set of conventions encoded through copy and alias semantics. Newer efforts have established it as a first class method of assignment: in C++, through the use of *R-Value References*, and in Rust, by default. C++ focuses on the concept of a *temporary*: an object whose memory does not outlive the expression in which it was constructed, and whose ownership is available for transfer.

Outside of the local inference of the temporariness of values, the choice to use move semantics over copy semantics for any particular assignment resides with the programmer. This is done by forcing overload resolution, through the use of manual type annotations. However, this places a large burden on the programmer, who must now understand how ownership transfer interacts with the rest of the program, including the potential loss of the applicability of other, more impactful optimizations.

6.1.1 Compile-Time Move Inference

In this work, we consider an alternative approach: what if we treat the decision of which assignment semantics to use as a *per-assignment* optimization problem? This would be done automatically by a compiler, alongside other potentially more impactful optimizations. In particular, we propose starting with a by-default copy assignment semantics — which provides the greatest amount of static information to the compiler — and use the results of static analysis to *automatically relax* assignments where possible.

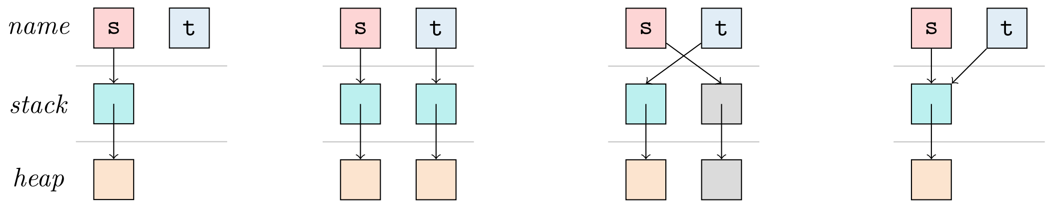
The goal of this approach is to enable use of optimizations such as reordering, deforestation, and parallelization using the guarantee of data independence throughout the program, under the assumption that such optimizations are more useful than the elision of any individual set of assignments. Once these optimizations have completed, move inference takes place to determine which of the remaining copies may be relaxed, or otherwise elided.

6.1.2 Our Contributions

System $\mathcal{M}$: A simple language for modeling the use of move semantics as an alternative to copy semantics. Assignments in System $\mathcal{M}$ are explicitly annotated by the programmer with a choice of a *bid method*, denoting the preferred method of carrying out the assignment. System $\mathcal{M}$ is designed to encompass the subset of languages such as C++ and Rust which deal with assignments, allowing the results of analyses of System $\mathcal{M}$ to apply to those languages in turn.

Trace Equivalence: A framework for reasoning about the observable equivalence between those System $\mathcal{M}$ programs which differ only in their choice of assignment semantics. As part of this framework, we present a conservative approximation to the equivalence relation between a program and its *canonical copy variant*, the version of itself which uses only copy semantics.

Iterative Last-Copy Relaxation: A straightforward optimization algorithm built on the definition of trace equivalence, which iterates over a program and selectively relaxes copies into moves where possible.



(a) Before (b) Copy (c) Move (d) Alias

Figure 6.1: Resource allocation, before and after assignment

6.2 A Technical Overview of Move Semantics

The idea behind move semantics — resource reuse through ownership transfer — has existed in the programmer's vernacular for decades. Academic research into the use of move semantics has tended to focus on static guarantees of correctness; mainstream language design efforts have focused on providing programmers with first-class tools to use move semantics as an alternative to copy semantics to improve performance. In the latter case, the burden of checking the correctness of implementations has generally rested with the programmers themselves.

For our purposes, we start with the view that copy semantics are the desiderata — and that moves are simply a more performant alternative to be used where safe to do so. This is the view espoused by modern C++, from which we primarily draw inspiration.

Move semantics were first proposed as an addition to C++ in [36], and subsequently incorporated into the 2011 revision of the language standard [14]. For our purposes, the proposal consists broadly of two parts: an inference framework for temporaries — which form the most likely candidates for moves — and a set of conventions on how moves might be implemented.

6.2.1 Temporary Inference and R-Value References

The use of move semantics in C++ is centered around the observation that the lifetimes of temporary objects — anonymous objects created by the compiler as the result of an expression — do not exceed the scope of the expression which created them. As a result, their resources may safely be reused, notably for the creation of other objects.

A common example of a temporary is the object returned by a call to a by-value return function: `{ T f(); T x; x = f() }`.

In this case, the object of type `T` returned by `f` is inferred as a temporary object, whose lifetime does not exceed its assignment to `x`. In some cases, this is optimized by the compiler directly as return-value optimization (RVO). In the general case however, the type of the return expression is inferred as an R-value reference (`T&&`), thereby invoking the R-value reference overload of the assignment operator for type `T` for its assignment to `x`. If no such overload exists (the type does not implement move semantics), reference collapsing rules [14, Section 8.3.2] apply and the corresponding copy assignment operator is invoked.

Alternatively, programmers may explicitly mark an object as moveable by manually typecasting it to an R-value reference using `std::move`, independent of its lifetime or temporary status. In essence, our research infers where such annotations are semantically safe.

6.2.2 The Implementation of a Move

The use of move operations in a program is predicated on the assumption that moves are in fact faster than a copy in any given situation. In C++, the exact implementation of a move is type-dependent; however, implementations typically follow the conventions established by the C++ standard library. The convention consists of a pre- and post-condition on the source of the move, as follows:

Pre-Condition: The resources held by the object to be moved from are forfeit, free to be consumed or transferred to any other object. "Resources" may refer to any data which is expensive or even impossible to copy, such as large heap-allocated data structures, file handles or network sockets.

Post-Condition: Following the completion of the move operation, the source object must be left in a *valid, but unspecified* state. An object is valid but unspecified if it will faithfully support any operation on it which places no precondition on its state. Notably for our purposes, the object must be reinitializable, allowing the reuse of any non-consumed resources.

One implementation of this convention is the `swap/delete` idiom, which performs a shallow copy of any non-moveable resource (primitive data types, etc.) and *swaps* handles to moveable resources (pointers to heap-allocated data blocks, file handles, etc.).

6.2.3 Choosing between Assignment Methods

Move semantics provide an alternative approach to performing the same operation that copies do — the construction of a new object, from an old one. However, they accomplish this goal in different ways.

Figure 6.1 shows a conceptual representation of the allocation of stack and heap memory resources before and after assignment between two variables under different assignment methods.

Before assignment (6.1a), the source variable `s` refers to a piece of data allocated on the stack, which itself may contain pointers to data allocated on the heap. The target `t` of the assignment may or may not hold its own memory.

After a copy assignment (6.1b), the source's memory resources are duplicated under the target, both on the stack, and on the heap. In the case of a move (6.1c), the target takes sole ownership of the memory previously held under the source, which in turn is invalidated in an implementation-specific manner. Finally, performing an assignment by alias (6.1d) results in both source and target sharing ownership of the same resources.

We can evaluate the differences between each of these assignment methods on the basis of multiple criteria: the speed of performing the assignment; the amount of memory necessary to perform the assignment; and the strength or weakness of isolation guarantees between source and target afterwards.

Assignment Speed: Performing a *deep* copy of an object takes time proportional to the size of the object at the time, which can be arbitrarily large for complex data structures such as collections. In contrast, a move is analogous to a *shallow* copy; taking time proportional to the size of the *type* of the object, which is much smaller. Finally, performing an aliasing assignment is typically a constant-time operation, independent of the size of the object.

Memory Usage: Clearly, maintaining a complete copy of an object requires just as much memory as the original; the allocation of this memory may take place in one chunk, or in multiple smaller chunks. In typical implementations, performing an assignment by move requires only an additional stack allocation to hold the handles to the source's heap data. Aliasing assignments generally require no additional allocation beyond the initial pointer.

Isolation Guarantees: Copy assignments guarantee that both source and target are equivalent and independent. Move assignments guarantee the latter — independence — but not equivalence, since the source object is invalidated. Aliasing makes the other trade-off — equivalence, in exchange for independence.

Even if a language adopts a by-default copy semantics, it is not necessarily the case that any *particular* assignment requires the isolation guarantees provided by a copy. There exists therefore a potential for optimization to relax assignments from the default copy method to more efficient implementations, if the isolation guarantees provided are sufficient for the context in which the assignment is carried out.

		Language	
p	$::=$	$\circ \mid s \parallel p$	*Programs*
s	$::=$	$n = e \mid n? \mid \hookleftarrow$	*Statements*
n	$::=$	$\texttt{id} \mid \texttt{id} \,.\, n$	*Names*
e	$::=$	$e_a \mid e_b \mid e_c$	*Expressions*
e_a	$::=$	$n \oplus e_b$	*Applications*
e_b	$::=$	$b(n)$	*Bids*
e_c	$::=$	$\Lambda \bar{C} \,.\, v$	*Captures/Literals*
$\bar{C}$	$::=$	$\{\texttt{id} = e_b\}$	*Capture Spec*
b	$::=$	$\mathbb{C} \mid \mathbb{M}$	*Bid Types*
v	$::=$	$\phi \mid v_a \mid \mu\texttt{id} \,.\, \lambda\texttt{id} \,.\, p \rightarrow e_b$	*Values*
		Abstract State	
σ	$::=$	$\langle S, I, M \rangle$	*Stores*
S	$::=$	$\circ \mid F \parallel S$	*Stacks*
F	$::=$	$\langle N, N \rangle$	*Frames*
I	$::=$	$\{\delta \mapsto \langle \alpha, N \rangle\}$	*Identities*
M	$::=$	$\{\alpha \mapsto v\}$	*Memory*
N	$::=$	$\{\texttt{id} \mapsto \delta\}$	*Namespace*
C	$::=$	$\langle \sigma, p \rangle$	*Configuration*
δ, α	$::=$	$\texttt{id}$	*Identifiers*

Figure 6.2: System $\mathcal{M}$ — Syntax

6.3 System $\mathcal{M}$: Syntax and Semantics

In this section, we present the syntax and semantics of System $\mathcal{M}$, our core calculus for analyzing the use of assignment methods — referred to here as *bid* methods — in a program.

The surface syntax of System $\mathcal{M}$ is presented in the first part of Figure 6.2. System $\mathcal{M}$ is a quasi-functional/imperative language with features from both, loosely abstracting the features of C++ relevant to move semantics. System $\mathcal{M}$ programs consist of a sequence of statements, which are of three types: assignments, observations, and returns.

Return statements ($\hookleftarrow$) are used for bookkeeping, denoting program points where stack frames are to be unwound. These are inserted into the statement stream by the evaluator, and do not occur in user programs.

Observations of the form $n?$ represent a program point where the value currently referred to by the name n is externally inspected. Examples of observation points include printing a value to the screen, writing it to disk or serializing it for network communication. Regardless of the reason, an observation signifies that a valid value is required at that program point.

Assignment statements $n = e$ form the bulk of the language, and represent the act of making some resource denoted by e available under the name n. Assignments center around the notion of a *bid*

Constant Assignment

$$\frac{n \bullet\!\!\multimap \delta \in \sigma \overleftarrow{\cup} \{n \bullet\!\!\multimap \mathbf{Fresh}_\delta\} \\ \delta \mapsto \langle \alpha, \circ \rangle \in \sigma \overleftarrow{\cup} \{\delta \mapsto \langle \mathbf{Fresh}_\alpha, \circ \rangle\} \qquad \sigma' = \sigma \overleftarrow{\cup} \{n \bullet\!\!\multimap \delta\} \overrightarrow{\cup} \{\delta \mapsto \langle \alpha, \circ \rangle, \alpha \mapsto v\}}{\langle \sigma, n = \Lambda p_b \,.\, v \parallel p \rangle \to \langle \sigma', [n \,.\, n' = e_b \mid n' = e_b \in p_b] \parallel p \rangle}$$

Copy Assignment

$$\frac{n_x \bullet\!\!\multimap \delta_x \in \sigma \overleftarrow{\cup} \{n_x \bullet\!\!\multimap \mathbf{Fresh}_\delta\} \qquad \delta_x \mapsto \langle \alpha_x, N_x \rangle \in \sigma \overleftarrow{\cup} \{\delta_x \mapsto \langle \mathbf{Fresh}_\alpha, \circ \rangle\} \qquad n_y \bullet\!\!\multimap \delta_y \in \sigma \\ \delta_y \mapsto \langle \alpha_y, N_y \rangle \in \sigma \qquad \alpha_y \mapsto v_y \in \sigma \qquad \sigma' = \sigma \overleftarrow{\cup} \{n_x \bullet\!\!\multimap \delta_x, \delta_x \mapsto \langle \alpha_x, N_x \rangle\} \overrightarrow{\cup} \{\alpha_x \mapsto v_y\}}{\langle \sigma, n_x = \mathbb{C}(n_y) \parallel p \rangle \to \langle \sigma', [n_x \,.\, n = \mathbb{C}(n_y \,.\, n) \mid n \in N_y] \parallel p \rangle}$$

Move Assignment

$$\frac{n_x \bullet\!\!\multimap \delta_x \in \sigma \overleftarrow{\cup} \{n_x \bullet\!\!\multimap \mathbf{Fresh}_\delta\} \qquad \delta_x \mapsto \langle \alpha_x, N_x \rangle \in \sigma \overleftarrow{\cup} \{\delta_x \mapsto \langle \mathbf{Fresh}_\alpha, \circ \rangle\} \qquad n_y \bullet\!\!\multimap \delta_y \in \sigma \\ \delta_y \mapsto \langle \alpha_y, N_y \rangle \in \sigma \qquad \alpha_y \mapsto v_y \in \sigma \qquad \sigma' = \sigma \overleftarrow{\cup} \{n_x \bullet\!\!\multimap \delta_x, \delta_x \mapsto \langle \alpha_x, N_x \rangle\} \overrightarrow{\cup} \{\alpha_x \mapsto v_y, \alpha_y \mapsto \phi\}}{\langle \sigma, n_x = \mathbb{M}(n_y) \parallel p \rangle \to \langle \sigma', [n_x \,.\, n = \mathbb{M}(n_y \,.\, n) \mid n \in N_y] \parallel p \rangle}$$

Function Call

$$\frac{n_f \bullet\!\!\multimap \delta_f \in \sigma \qquad \delta_f \mapsto \langle \alpha_f, N \rangle \in \sigma \qquad \alpha_f \mapsto \mu s \,.\, \lambda x \,.\, p' \to e' \in \sigma \qquad \sigma' = \langle \circ, N \cup \{s \mapsto \alpha_f\} \rangle \parallel \sigma}{\langle \sigma, n = n_f \oplus e_b \parallel p \rangle \to \langle \sigma', n_x = e_b \parallel p' \parallel n = e' \parallel \hookleftarrow \parallel p \rangle}$$

Function Return

$$\frac{F \parallel S' = S \qquad I' = I \restriction_\delta S \qquad M' = M \restriction_\alpha I'}{\langle \langle S, I, M \rangle, \hookleftarrow \parallel p \rangle \to \langle \langle S', I', M' \rangle, p \rangle}$$

Observation

$$\frac{n \bullet\!\!\multimap \delta \in \sigma \qquad \delta \mapsto \langle \alpha, N \rangle \in \sigma \qquad \alpha \mapsto v \in \sigma}{\langle \sigma, n? \parallel p \rangle \to \langle \sigma, [n \,.\, n'? \mid n' \in N] \parallel p \rangle}$$

Figure 6.3: System $\mathcal{M}$ — Semantics

method, representing the method by which a given variable is willing to give up its resources for an assignment. System $\mathcal{M}$ currently supports two bid methods — Copy and Move — the incorporation of an Alias bid method is discussed in Section 6.6.

Right-hand-side expressions may have several forms: variable names, explicitly annotated with the desired bid method (e_b), function application with explicitly annotated argument-passing bid method (e_a), or a capture specified literal expression (e_c). Capture specifications are sequences of bid-annotated variable assignments, specifying how captured variables are constructed from their source counterparts. Well-formed programs do not capture for non-function literals; and in that case, they do not capture the same variable more than once.

Literals include null values (ϕ) appearing in the source after a move operation, atoms (v_a), and function values — consisting of self name, formal argument name, body and return expression.

6.3.1 Abstract Machine

The semantics of System $\mathcal{M}$ are presented in small-step operational form over an abstract machine; the state maintained by this machine is described in the latter half of Figure 6.2.

The goal of the abstract machine is to model the distinction between *variable scope* and *value lifetime* in a contemporary stack/heap-based execution environment. While the duration for which a variable may be accessed is tied to the scope in which it is declared, its value may be moved around between variables, extending its lifetime. This distinction is captured by the use of three environments: the *stack*, the *identity pool* and *memory*.

Stack: The stack contains a sequence of frames in LIFO order, with each frame corresponding to a function call. Each frame contains two namespaces: a *local* and *closure* namespace.

The local namespace contains the names of all local variables declared during the function's body; these names are marked for deallocation at the end of the function's execution. Each subsequent call to the same function acquires a new, empty local namespace.

The closure namespace contains the names of all variables captured in the closure of the current function; these names are deallocated at the end of the function's own lifetime. This mimics the lifetimes and accessibilities of member variables within a method of an object in C++. Each name in a namespace maps to an identity, present in the identity pool.

Identity Pool: The identity pool is a mapping from an identity to a pair of memory address and a dependent namespace, and serves several purposes.

The first is to permit names on the stack to serve as indirections; two names may act as aliases of each other if they point to the same identity. The second purpose is to capture the dependent lifetime relationship between variables; a mapping $n_d \mapsto \delta_d$ is present in the dependent namespace of an identity δ if the lifetime of δ_d is tied to that of δ. In this case, any resources held under δ_d are deallocated when δ itself is deallocated.

This occurs with the closure of a function; the dependent namespace N_f of an identity δ_f representing a function comprises of the names of each of the variables captured in its closure, mapped to their own corresponding identities. Each of these closure-captured variables is deallocated when the function is deallocated.

Memory: The address component α of each identity is an indirection into the memory segment M, and holds the local or *shallow* value of a variable. The nature of this value depends on the type of the variable; System $\mathcal{M}$ recognizes three kinds of values: null values ϕ, atoms v_a, and functions themselves.

$$\frac{\texttt{id} \mapsto \delta \in N_l}{\texttt{id} \multimap \delta \in \langle\langle N_l, N_c\rangle \mid\mid S, I, M\rangle}$$

$$\frac{\texttt{id} \mapsto \delta \in N_c}{\texttt{id} \multimap \delta \in \langle\langle N_l, N_c\rangle \mid\mid S, I, M\rangle}$$

$$\frac{\texttt{id} \multimap \delta \in \langle S, I, M\rangle}{\texttt{id} \multimap \delta \in \langle\langle N_l, N_c\rangle \mid\mid S, I, M\rangle}$$

$$\frac{n \multimap \delta' \in \sigma \qquad \delta' \mapsto \langle \alpha, N_d\rangle \in I \qquad \texttt{id} \mapsto \delta \in N_d}{n\,.\,\texttt{id} \multimap \delta \in \langle S, I, M\rangle}$$

Figure 6.4: Name Resolution Semantics

Name Resolution

This breakdown in the abstract state necessitates a more complex name resolution procedure: the procedure relating variable names and identities. We use the *resolution operator* $n \multimap \delta \in \sigma$ to denote that in the context of σ, the name n is associated with the identity δ. The definition of $\multimap$ is given in Figure 6.4

Broadly, the behavior of $\multimap$ corresponds to the name resolution procedure used during compilation. Identifiers are resolved in the stack, starting at the current frame and proceeding upwards. Qualified names are resolved by recursively resolving the prefix, then looking up the suffix in the resolved identity's dependent namespace.

6.3.2 Semantics

The small-step operational semantics of System $\mathcal{M}$ are presented in Figure 6.3. Most of the semantic complexity is in the manipulation of the various abstract store subcomponents; we use a number of notational shorthands to help with this exposition.

The $\overleftarrow{\cup}$ and $\overrightarrow{\cup}$ operators represent left- and right-biased unions of associative maps, these are used to express the idea of "declaration on first use". The map into which the right-hand-side is merged is type-directed: mappings of the form $\alpha \mapsto v$ are installed into M; $\delta \mapsto \langle \alpha, N\rangle$ into I; and $n \mapsto \delta$ into the appropriate namespace of S, according to the name resolution semantics above.

The restriction operators $\restriction_\delta$ and $\restriction_\alpha$ denote the expiration of variables at the end of their scope; $I \restriction_\delta S$ constructs the set of all identities in the pool reachable by resolving some name on the stack, while $M \restriction_\alpha I$ performs a similar role in projecting the subset of memory reachable from the identity pool.

Since we focus purely on bid methods in this work, we only consider programs which are otherwise *identical.* We call this relation *bid equivalence*, and define two programs to be bid equivalent if they are identical up to bid methods.

Of all the programs bid equivalent to a given program p, there exists one distinguished member — the *canonical copy variant*, denoted $\mathbb{C}(p)$. The canonical copy variant of p is the program which is identical to p, but in which all of its bids are by copy. In a copy-semantic language, this represents the "original intent" of the programmer, the behavior which all subsequently optimized programs must be semantically equivalent to.

6.3.3 Traces, and Trace Equivalence

An appropriate definition of semantic equivalence between two System $\mathcal{M}$ programs must allow variables in one program the freedom to diverge in their values temporarily from those of their counterparts in the other, for as long as those values are not observed. Such an observation may occur explicitly as described above, or implicitly as the source of an assignment or application. We reify these observations as a list of values called a *program trace*: the notation $\langle \sigma, p \rangle \Rightarrow \langle T, \sigma' \rangle$ denotes that there exists a finite number of steps from an initial configuration $\langle \sigma, p \rangle$, to a final configuration $\langle \sigma', \circ \rangle$, accumulating a trace T in the process. The details of the trace accumulation procedure are omitted for brevity.

Two configurations are *trace equivalent* if they produce identical traces, regardless of their respective final stores. This relation is denoted by $\equiv_{\mathrm{T}}$ and is defined as follows:

$$\begin{aligned}\langle \sigma_1, p_1 \rangle \equiv_{\mathrm{T}} \langle \sigma_2, p_2 \rangle \triangleq \exists \sigma_1', \sigma_2', T. \langle \sigma_1, p_1 \rangle &\Rightarrow \langle T, \sigma_1' \rangle \\ \wedge \langle \sigma_2, p_2 \rangle &\Rightarrow \langle T, \sigma_2' \rangle\end{aligned}$$

Trace equivalence applies to any pair of programs — bid equivalent or otherwise. However for our purpose, we are only concerned with a subset of pairs — an arbitrary program p, and its canonical copy variant $\mathbb{C}(p)$. This allows us to construct a tractable approximation to trace equivalence, which in the general case is undecidable.

6.4 Approximating Trace Equivalence

In this section, we develop a static analysis framework which conservatively approximates the trace equivalence of a program with its corresponding canonical-copy variant. This allows us to prove the soundness of bid-equivalent optimizations — transformations which only alter the bid methods in a program — by using the transitivity of the trace equivalence relation.

6.4.1 The Supply and Demand of Canonical Values

The key idea behind our analysis framework is the *canonical value* — the value that a variable would have at a given point in time, in the canonical copy variant of the program in question. We wish to allow variables' values to diverge from the canonical values at various points in an optimized program, as long as those canonical values are reinstated before they are observed.

Any assignment operation causes the target to get its canonical value; in the case of a move assignment however, the source of the assignment is invalidated, "consuming" its own canonical value. We can then formulate the notions of *supply*, *demand*, and *consumption* of a program with respect to canonical values — each a set of variable names — as follows:

Supply: The supply of a program represents the set of variables whose canonical values are guaranteed to reach the end of the program.

Demand: The demand of a program represents the set of variables whose canonical values *must* reach the beginning of the program, due to their observation during its execution.

Consumption: The consumption of a program represents the set of variables whose canonical values are guaranteed *not* to reach the end of the program, due to their consumption by move assignments or explicit deinitialization.

The supply, demand and consumption sets of a program are themselves interrelated; e.g. if a variable is moved from and subsequently reinitialized, it is still supplied by the program segment which includes both statements. Alternatively, if a variable is initialized and later observed, it is not demanded in net by the program containing both statements. The equational definitions for the construction of these sets are given in Figure 6.5.

The supply of a program constitutes a necessary condition on the set of stores under which it may be safely executed; in contrast, the demand of a program forms a sufficient condition on the set of stores which may result from any such execution. Taken together, they comprise an interface

$$
\begin{aligned}
\mathcal{S}(\circ) &\triangleq \phi \\
\mathcal{S}(p \,||\, s) &\triangleq \mathcal{S}(p) \setminus \mathcal{C}(s) \cup \mathcal{S}(s) \\
\mathcal{S}(n?) &\triangleq \phi \\
\mathcal{S}(n = e) &\triangleq \{n_y\} \\
\mathcal{D}(p \,||\, s) &\triangleq \mathcal{D}(p) \cup (\mathcal{D}(s) \setminus \mathcal{S}(p)) \\
\mathcal{D}(n?) &\triangleq \{n\} \\
\mathcal{D}(n_y = b(n_x)) &\triangleq \{n_x\} \\
\mathcal{D}(n_y = n_f \oplus b(n_x)) &\triangleq \{n_f, n_x\} \\
\mathcal{D}(n_y = \Lambda \bar{C}.\, \mu f.\, \lambda n_a.\, p \to e_b) &\triangleq \mathrm{dom}(\bar{C}) \\
\mathcal{C}(p \,||\, s) &\triangleq \mathcal{C}(p) \setminus \mathcal{S}(s) \cup \mathcal{C}(s) \\
\mathcal{C}(n_y = \mathbb{M}(n_x)) &\triangleq \{n_x\}
\end{aligned}
$$

Figure 6.5: Supply, Demand and Consumption

specification we refer to as a *market.* We define a market m as a pair of sets — written as $\mathcal{S}(m)$ and $\mathcal{D}(m)$ — which denote the set of variables whose canonical values will be provided to, or demanded of respectively, any program which is evaluated in the context of that market.

We can now refine our previous definition of trace equivalence to be with respect to a specific market; quantifying over the set of all program stores which have bound canonical values for at least the set of variables for which the market promises to supply canonical values; producing stores which themselves provide canonical values for at least the set of variables whose canonical values the market demands. This notion of *market-compliant trace equivalence* is denoted by $\equiv_T^m$ and defined as:

$$
\begin{aligned}
p_1 \equiv_T^m p_2 \triangleq \forall \sigma_1, \sigma_2.\, \sigma_1 \cong_{S(m)} \sigma_2 \implies \\
\langle p_1, \sigma_1 \rangle \equiv_T^m \langle p_2, \sigma_2 \rangle \wedge \sigma_1^{'} \cong_{D(m)} \sigma_2^{'}
\end{aligned}
$$

The above definition is expressed in terms of a separate relation $\sigma_1 \cong_N \sigma_2$, specifying the equivalence of two stores on the basis of the names in N. This is necessary due to the presence of higher-order functions; we must also be able constrain the behavior of functions already present in the initial store, as well as any functions created over the course of the program. This relation — *market-compliant store equivalence* — is denoted by $\cong_n$; $\sigma_1 \cong_N \sigma_2$ is defined as

$$
\forall n \in N.\, \exists v_1, v_2.\, n \multimapdotbothA v_1 \in \sigma_1 \wedge n \multimapdotbothA v_2 \in \sigma_2 \wedge v_1 \cong v_2
$$

where

$$v_1 \cong v_2 \triangleq \begin{cases} p_1 \equiv_T^m p_2 & v_1 = \Lambda\bar{C}.\, \lambda n_x.\, p_1 \to b_1(n_r) \\ & \wedge v_2 = \Lambda\bar{C}.\, \lambda n_x.\, p_2 \to b_2(n_r) \\ & \wedge \mathcal{S}(m) = \mathrm{dom}(\bar{C}) \cup \{n_x\} \\ & \wedge \mathcal{D}(m) = \mathrm{dom}(\bar{C}) \cup \{n_r\} \\ v_1 = v_2 & \text{otherwise} \end{cases}$$

When considering a sequence of statements executing within a single market, at any given point in the sequence, it is possible that the canonical value for a variable may be supplied either initially from the market, or from a previous statement in the sequence. Correspondingly, a variable's canonical value may be demanded either by a statement after the current program point, or by the market itself. This gives rise to the dual notions of *net-supply* and *net-demand*, which combines the above computations of supply and demand of programs, with their market equivalents. The equational definitions of these sets are similar to those above.

$$\mathcal{NS}(p, m) \triangleq \mathcal{S}(m) \setminus \mathcal{C}(p) \cup \mathcal{S}(p)$$
$$\mathcal{ND}(p, m) \triangleq \mathcal{D}(p) \cup (\mathcal{D}(m) \setminus \mathcal{S}(p))$$

We now formally state a property that may be intuitively expected of any "well-behaved" System $\mathcal{M}$ program: *partition-safety*, denoted $\mathcal{PS}(p, m)$.

$$\begin{aligned} \mathcal{PS}(p, m) \triangleq & \forall p_a, p_b.\, p = p_a \mid\mid p_b \\ & \implies \mathcal{NS}(p_a, m) \supseteq \mathcal{ND}(p_b, m) \end{aligned}$$

A program is partition-safe within a market if, at any point within the program, the set of variables whose canonical values are guaranteed to reach that program point contains the set of variables whose canonical values are required to reach that point. Partition safety represents a form of internal consistency of a program; it ensures that there exists no subprogram which independently creates a variable, invalidates it, and subsequently demands it, all hidden from the market at large.

While the equivalence of canonical values from the market ensure the equivalence of closures already defined before the start of a program, we must also ensure that closures defined *during* the program are partition safe. This must be guaranteed ahead-of-time; we encode this in a *well-*

formedness condition $\mathcal{WF}(p)$:

$$\mathcal{WF}(p) \triangleq \forall (n_y = \Lambda\bar{C}.\,\mu s.\,\lambda n_x.\,p' \rightarrow e_b) \in p.\,\mathcal{PS}(p', m)$$
$$\text{where } m = \langle \mathrm{dom}(\bar{C}) \cup \{n_x\}, \mathrm{dom}(\bar{C}) \cup \mathcal{D}(e_b)\rangle$$

We now state the central claim of this work: any well-formed program which is partition safe within a given market, is trace-equivalent to its canonical-copy variant within that market.

Lemma 1.

$$\forall p, m.\,\mathcal{WF}(p) \wedge \mathcal{PS}(p, m) \implies p \equiv_T^m \mathbb{C}(p)$$

We provide the sketch of a proof by induction over the structure of an arbitrary, p, for an arbitrary m.

Proof (empty): *If $p = \circ$, then $\mathbb{C}(p) = \circ$; correspondingly, both $\langle \sigma, p\rangle \Rightarrow \langle \circ, \sigma\rangle$ and $\langle \sigma, \mathbb{C}(p)\rangle \Rightarrow \langle \circ, \sigma\rangle$.* ▌

Since by induction we have $\mathcal{PS}(\circ, m)$, it follows that

$$\begin{aligned}
\mathcal{PS}(\circ, m) &\implies \mathcal{NS}(\circ, m) \supseteq \mathcal{ND}(\circ, m)\\
&\implies \mathcal{S}(m) \supseteq \mathcal{D}(m)\\
&\implies (\forall \sigma_1, \sigma_2.\,\sigma_1 \cong_{\mathcal{S}(m)} \sigma_2 \implies \sigma_1 \cong_{\mathcal{D}(m)} \sigma_2)\\
&\implies p \equiv_T^m \mathbb{C}(p)
\end{aligned}$$

Proof (non-empty): *For any non-empty program $p = p' \,\|\, s$, we case analyze s, and show that trace equivalence of p' and $\mathbb{C}(p')$ under m, in addition to the partition-safety of p (in particular, the partition consisting of p' and s) is sufficient to prove trace equivalence of p and $\mathbb{C}(p)$.*

One such case is when s is a move assignment, namely $n_y = \mathbb{M}(n_x)$. For $m' = \langle \mathcal{S}(m), \mathcal{D}(m) \setminus \mathcal{S}(s)\rangle$, if $\mathcal{PS}(p', m)$ (which can be shown by algebraic set manipulation), then $p \equiv_T^{m'} \mathbb{C}(p)$ follows, which entails $p \equiv_T^m \mathbb{C}(p)$, since $\mathcal{S}(m') = \mathcal{S}(m)$ and $\mathcal{D}(m') \subseteq \mathcal{D}(m)$.

The above proof only applies directly to a program and its canonical-copy variant; we can extend this to any pair of bid-equivalent programs with a straightforward use of the transitivity of trace equivalence.

Theorem 1.

$$\forall p_1, p_2, m. \mathcal{WF}(p_1) \wedge \mathcal{WF}(p_2) \wedge \mathcal{PS}(p_1, m) \wedge \mathcal{PS}(p_2, m) \implies p_1 \equiv_T^m p_2$$

Proof *By transitivity of trace equivalence.*

From this formal machinery, it is clear that the burden now rests with our ability to find an appropriate market under which a given snippet of the program may be analyzed. In the following section, we demonstrate several possible approaches we might use to find such a market, and the optimization algorithms that arise from those approaches.

6.5 Bid-Equivalent Optimization

Theorem 1 gives us an efficient test to determine if two programs differing only in their choice of bid methods still behave equivalently to their canonical-copy variant, under assumptions of required incoming and outgoing canonical values. The burden therefore shifts to determining what these assumptions — the initial market — can be for a real-world program.

6.5.1 The Null and Ideal Markets

By definition, an appropriate market for a whole program executed in its entirety is the *null market*: i.e. the market m_ϕ such that $\mathcal{S}(m_\phi) = \mathcal{D}(m_\phi) = \phi$.

A straightforward — if naïve — optimization algorithm arising from this observation is, given a program p, to test the partition-safety under m_ϕ of each of the $2^{\#b}$ programs bid-equivalent to p, where $\#b$ is the number of bid methods in p.

At the other end of the granularity spectrum, for any sub-program p' of a larger program $p = p_a \,||\, p' \,||\, p_b$, we can infer a market m' which p' may be realistically be expected to execute in as $\mathcal{S}(m') = \mathcal{NS}(p_a, m_\phi)$ and $\mathcal{D}(m') = \mathcal{ND}(p_b, m_\phi)$. This formulation shifts the computational burden from the test of partition-safety to the initial construction of m' for the sub-program p' under consideration.

6.5.2 The Closure Market

The proof structure of Lemma 1 however provides us with an alternative construction; one that compromises between accuracy, granularity and upfront cost — the closure market.

The set of closure-captured variables for a function, along with the function's formal parameter and its own name, form a safe lower bound on the set of variables whose canonical values are supplied to a function's body. In turn, the same set of closure captured variables, along with any variable required for the function's return expression and the name of the function itself form a safe upper bound on those variables whose canonical values are demanded of the function's body. This closure market: $m_{\bar{C}}$, defined by $\mathcal{S}(m_{\bar{C}}) = \text{dom}(\bar{C}) \cup \{f, x\}$ and $\mathcal{D}(m_{\bar{C}}) = \text{dom}(\bar{C}) \cup \{f, r\}$ is an appropriate market for the body of a function defined as $\Lambda\bar{C} \,.\, \mu f \,.\, \lambda x \,.\, p \to b(r)$.

6.5.3 Iterative Last-Copy Relaxation

Algorithm 3 Iterative Last-Copy Relaxation

```
procedure ITERATIVE-LCR($p_a, p_b, m$)
    while $p_a = p'_a \,||\, s$ do
        $s' \leftarrow$ RELAX($s, \mathcal{NS}(p'_a, m), \mathcal{ND}(p_b, m)$)
        $p_a \leftarrow p'_a$
        $p_b \leftarrow s' \,||\, p_b$
    end while
end procedure

procedure RELAX($p, \mathcal{NS}(m), \mathcal{ND}(m)$)
    if $s = [n_y = \mathbb{C}(n_x) \wedge n_x \notin \mathcal{ND}(m)]$ then
        return $[n_y = \mathbb{M}(n_x)]$.
    else if ... then
        ...
    end if
end procedure
```

Given an appropriate market using one of the above techniques, we can formulate the *iterative last-copy relaxation* (ILCR) algorithm, shown in Figure 3. ILCR iterates over the program and considers each statement for relaxation from copy to move.

The task of relaxation is delegated to the RELAX subprocedure, which inspects the statement and uses a decision structure similar to the case analysis of Lemma 1 to determine if the variable whose canonical value would be consumed by a relaxation is present in the net demand of the remainder of the program.

ILCR relies on the difference between the net-supply of a program, and the corresponding net-demand of the market; if a variable is not demanded by the current market, its value is safe to be consumed at an earlier point in the program.

The efficiency of ILCR relies on the efficiency of the repeated recomputations of $\mathcal{NS}(p'_a m)$ and $\mathcal{ND}(p_b, m)$ at each step of the iteration. However, this may be trivially incrementalized.

6.6 Extending System $\mathcal{M}$ to Alias Analysis

The main focus of future work on System $\mathcal{M}$ is an extension of the analysis to a limited form of static alias analysis. In this section, we outline what such an extension might look like.

6.6.1 The Alias Bid Method

Introducing aliases to System $\mathcal{M}$ is done through a new bid method, $\mathbb{R}$. An assignment of the form $n_y = \mathbb{R}(n_x)$ results in a store where n_y and n_x are both handles to the same memory resources — those previously held by n_x. This is modeled in the abstract machine by allowing multiple names to map to the same identity in any given namespace. In its simplest form, the reference assignment rule would be:

$$\textsc{Reference Assignment}\quad \frac{n_x \mathrel{\bullet\!\!-\!\!\circ} \delta_x \in \sigma \overleftarrow{\cup}\{n_x \mathrel{\bullet\!\!-\!\!\circ} \mathbf{Fresh}_\delta\} \qquad \sigma' = \sigma \overrightarrow{\cup}\{n_y \mathrel{\bullet\!\!-\!\!\circ} \delta_x\}}{\langle \sigma, n_y = \mathbb{R}(n_x) \mathbin{||} p\rangle \rightarrow \langle \sigma', p\rangle}$$

A C++-style model prevents the *reseating* of a reference; once created, a reference is indistinguishable from its target. To prevent dangling references on deallocation, the abstract machine must be further extended to understand ownership; each entry in a namespace is augmented to the form $n \mapsto \langle \delta, \mathbf{2}\rangle$, where $\mathbf{2}$ is a bit representing whether the handle n is the *owning handle*, whose scope defines the lifetime of the resources held under δ.

6.6.2 Conservative Alias Approximation

Aliasing may be incorporated into our market-based formal framework by recognizing the effect of an aliasing assignment on canonical values: performing the assignment $n_y = \mathbb{R}(n_x)$ supplies the canonical value for n_y, but *consumes* the canonical values of any variable which is an alias to n_y at that point. This derives from the observation that in the canonical copy variant of the program, the assignment to n_y would not reflect on any other variable; therefore the visibility of the assignment to n_y on any other variable represents a consumption of canonical values, similar to that of the source of a move.

Determining which variables' canonical values are consumed at any aliasing assignment requires a reified alias graph. Efficient and accurate construction of this graph is a topic of ongoing research

6.7 Applications and Future Work

We now turn to the question of how we can realize the benefits of System $\mathcal{M}$ in practice. System $\mathcal{M}$ is expressive enough to model the variable assignment of most languages; as such, its analyses may be *overlaid* on top of any host language which can encode by-default copy semantics, and determine optimal assignment methods.

6.7.1 Optimizing DSL Embeddings

One application of this idea is with *embedded domain specific languages* (eDSLs); e.g. functional query languages in large-scale data processing frameworks including Spark [4] and Flink [16]. The semantics of these query languages are generally purely functional, making them amenable to a wide range of typically functional optimizations such as deforestation [79]. However, the nature of their embedding requires them to interact with the semantics of the host language — in particular, its assignment semantics — which make those potentially more impactful optimizations difficult or impossible to apply.

With the benefit of System $\mathcal{M}$'s analysis, any code in the host language interacting with the DSL may be assumed to be copy-semantic, allowing functional optimizations to proceed. Any copy assignments remaining may be selectively relaxed, achieving the best of both worlds in copy elision and optimization.

6.7.2 Partial Garbage Collection

The current System $\mathcal{M}$ cost model assumes that the memory management scheme obtained from static lifetime information is a superior alternative to runtime garbage collection; however, this might not always be the case. In some cases, garbage collection overhead may be justifiable in circumstances where the *expected* lifetime of a value may be far lower than the best-effort static estimate.

In System $\mathcal{M}$, the decision to use a garbage collector may be made on a per-variable basis, depending on the compiler's confidence in the accuracy of its lifetime estimates. This results in a best-of-both-worlds scenario; garbage collection is only used for those variables which need the flexibility, reducing latency.

6.7.3 Ask/Bid Polymorphism

The determination of the method by which an assignment is performed is currently the sole responsibility of the right-hand-side of the assignment; there is however, a scenario in which the left-hand-side exerts a preference. Consider an in-place collection insertion function, which inserts a provided argument directly into a collection of indeterminate lifetime. Such a function *requires* that it have ownership over its formal argument, to be able to transfer it to the collection itself. This gives rise to two possible implementations — one in which the argument is passed by alias, forcing a copy within the function; and one where an extra copy is not required, since the assignment of the formal argument was done using some method which guaranteed it exclusive ownership.

System $\mathcal{M}$ functions are currently not able to express this level of polymorphism — where a function may *ask* for a form of assignment, as a counterpart to a *bid*. Ask-polymorphism is relevant in models of System $\mathcal{M}$ which permit sharing without ownership transfer, such as with aliasing assignments.

In Rust, such a situation is marked as a compilation error, to be resolved by the programmer. In C++, ask-polymorphism is embodied by the notion of an *implicit conversion*, where an argument passed by reference to a function expecting a value is silently copied.

In an *ask-polymorphic* variant of System $\mathcal{M}$, a function may be polymorphic in the ask of its formal parameter, potentially leading to multiple variants of the same function, one in which the function demands ownership of its arguments' resources, and another which operates without such ownership, compensating accordingly. This also models the more subtle feature of *perfect-forwarding* in C++, where the calling convention of a function is chosen from a set of possible template instantiations, according to the types of arguments at each call-site.

The main use-case of ask-polymorphic analysis is with the use of compound *data-constructors*; the constructor of e.g. a tuple would necessarily take ownership of the resources of its arguments, or copy them if ownership may not be transferred. On the other hand, parts of a tuple may be moved out of, without consuming the canonical values of other parts of the tuple.

Chapter 7

Related Work

The work presented in this thesis ties together techniques common in programming language design and compilation, and applies them to problems common in large-scale data analytics and system design.

7.1 Specialization for Scale

The idea of specialization — adapting an implementation to the specifics of a given application — has been the focus of a great deal of recent and ongoing research. Approaches have varied depending on exactly *which* specifics the implementation attempts to specialize.

The most straightforward approach to specialization is the synthesis of custom implementations specific to the application workload, manifesting most commonly through custom compilation and code generation.

MonetDB/X100 [12] passes vectors of tuples between operators, and uses vectorized processing, to avoid the overhead of calling iterators on each tuple. HIQUE [43] generates C++ code by instantiating code templates with minimal functional calls for each operator in a physical plan. The HyPer [60] compiler generates LLVM code in order to achieve faster compilation times than two stage compilers. Additionally, HyPer's generated code maximizes the amount of work done in each loop by placing loop boundaries at mandatory materialization points instead of single operators. Other work [59] has shown that query compilation techniques are effective for processing language integrated queries in a managed runtime, specifically LINQ queries over `C#` collections.

Legobase [41] makes use of the Scala LMS framework [65] and database-style optimizations to translate high-level SQL to efficient C code in a single-core model. Projects such as Graal and

Truffle [73] perform partial evaluation and type-based specialization at the AST level.

These approaches predominantly target a specific aspect of the application for compilation, as opposed to K3's approach of providing a general-purpose *platform* for specialization. As a result, the above-mentioned approaches have a hard ceiling in terms of the specialization they can exploit; K3's specialization framework does not exhibit this theoretical limitation.

7.2 Memory Management for Data Analytics

A great deal of work has been done on understanding and improving the memory characteristics and performance of large-scale data processing systems, as experiments show [62] that CPU/memory, and not disk or network are the current predominant performance bottlenecks.

Among the major commercial data processing frameworks, **Apache Spark** has made significant improvements to their memory management strategy with the implementation of a unified memory manager. This approach further blurs the line between regions of memory management responsibilities, allowing a more seamless allocation of resources between query execution and data caching layers.

In addition, the first generation of Spark's Tungsten execution engine [63] focuses on memory performance by using off-heap memory allocation in Java (through unsafe APIs) to avoid garbage collector overheads, as well as synthesizing more efficient in-memory data representations without the burden of JVM object overheads.

Apache Flink [15] also recognizes the dual goals of reducing garbage collection pressure and using more efficient data representation layouts in memory [3]; it approaches this by operating directly over binary serialized data representations, reducing memory usage and overheads due to object creation and collection. Flink also emphasizes object reuse to further reduce GC burden but relies on implementation discipline to ensure that correctness of programs is maintained in the presence of reused objects and in-place mutation.

Many of the techniques used in these systems manifest as emergent patterns in K3; by tackling the problem at a fine granularity, K3's materialization analysis combines the efficient management of individual object resources into a whole-program strategy, which exhibit many of the desirable properties discussed in this section.

The authors of **Emma** [2] argue that the functional style of Hadoop and such lacks expressivity for common distributed operations, and that systems such as Spark and Flink need to break with the fully functional style to provide that functionality. Emma is an attempt to bring more functionality to the domain of such distributed data flow systems by embedding more powerful abstractions inside the host language.

7.3 Language Techniques for Memory Management

Techniques used at the language level for memory management revolve primarily around two aspects: *safety* and *performance*.

Memory Safety: A great deal of work has been done on the use of type systems to verify memory safety; in particular, the use of substructural type systems such as *linear types* [80], *ownership types* [18] and *region types* [75].

The single-use property enforced by a linear type system corresponds closely to our notion of a moved-from variable; however, it does not capture the idea that a variable once moved from, may be reinitialized and subsequently reused. This difference is reflected in the implementation of move semantics in Rust [53] and C++ [14]: the former disallows the reuse of a moved-from variable, while the latter permits it. Implementations of linear type systems vary in their effectiveness, either due to inefficiency [82] or complexity to the user. The RustBelt [37] project focuses on formally verifying Rust's safety guarantees, while Hask-LL [10] aims to provide an opt-in linear typing layer to Haskell [51].

Region typing/inference [75, 74] makes the trade-off of permitting inaccurate determinations of object lifetimes in exchange for more efficient batched allocation and deallocation; move semantics accomplishes a similar effect by allowing values to be moved out of their originating scope. The scope to which the escaped value is associated with subsequently is nevertheless known at compile time without annotation, keeping the benefits of cheap deallocation without needing the overhead of extended lifetimes. Cyclone [31] provides a memory-safe dialect of C based on linear regions.

Memory Performance: A primary goal of improving memory performance is to reduce a program's dependence on runtime garbage collection. The overheads of garbage collection are significant [34], with even contemporary garbage collector proposals aiming for a worst case 15% throughput penalty [46].

The idea of *compile-time garbage collection* [9, 23] has existed for as long as garbage collection itself, and focuses on the use of static analysis to perform some of the work otherwise relegated to the runtime GC; this approach was popular with languages such as Lisp and Prolog [81] before rapidly improving hardware made the problem less pressing to deal with. Static memory optimization techniques have regained prominence, particularly in languages with a highly regular structure, such as Mercury [54], as well as the query languages supported by a wide variety [4, 16] of highly parallelizable, large-scale data distributed data processing systems.

Chapter 8

Conclusions

Trends in hardware architectures, the emergence of rich and varied computation models, and the availability of domain-specific information have created the perfect storm of performance optimization: the sheer number of factors which could be accounted for is typically beyond the extent of any single programmer. This situation is particularly exemplified in the context of large-scale data analytics, whose application dataflows exhibit quite regular structures, making them amenable to a high degree of specialization. One aspect of performance on which the impact of such specialization is singularly effective is that of memory management, due to the position of main-memory as the nexus of a wide variety of software and hardware components.

In our work we present K3 a language and associated compilation toolchain which facilitates the synthesis of dataflow implementations for large-scale data analytics applications. The work presented in this thesis focuses on the use of static analysis to synthesize application-specific memory management strategies, consisting of tailored placement of memory-management primitives such as allocations deallocations, and copy/move/alias operations. Our implementation of materialization analysis builds on top of well-known type-, effect- and provenance-analyses to infer a relaxation from a known copy semantics to a more efficient hybrid semantics, subject to context-specific safety conditions.

Experimental evaluation of implementations synthesized with K3 for a number of workloads provides several key insights: that memory management operations contribute significantly to overall system performance in both time and space compared to contemporary implementations; and that a holistic approach to memory management at compile-time succeeds in reducing superfluous memory operations, without the need for a dynamic garbage collector.

Finally, we present a formal framework for the verification of soundness of materialization analysis in its full generality, and describe a provably sound construction for a version of materialization analysis with only copy/move semantics. This lays the foundation for their use as a component of a broader language independent system compilation framework, as well as opening up potential hybrid static/dynamic approaches to memory management.

References

[1] Martín Abadi et al. “TensorFlow: A System for Large-Scale Machine Learning.” In: *OSDI*. Vol. 16. 2016, pp. 265–283.

[2] Alexander Alexandrov et al. “Implicit Parallelism through Deep Language Embedding”. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM. 2015, pp. 47–61.

[3] *Apache Flink: Juggling with Bits and Bytes*. `https://flink.apache.org/news/2015/05/11/Juggling-with-Bits-and-Bytes.html`.

[4] Apache Spark. *Apache Spark: Lightning-Fast Cluster Computing*. `http://spark.apache.org`. 2016.

[5] *Apache Spark as a Compiler: Joining a Billion Rows per Second on a Laptop*. `https://databricks.com/blog/2016/05/23/apache-spark-as-a-compiler-joining-a-billion-rows-per-second-on-a-laptop.html`.

[6] Michael Armbrust et al. “Spark Sql: Relational Data Processing in Spark”. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM. 2015, pp. 1383–1394.

[7] Ahsan Javed Awan et al. “How Data Volume affects Spark-Based Data Analytics on a Scale-Up Server”. In: *Workshop on Big Data Benchmarks, Performance Optimization, and Emerging Hardware*. Springer. 2015, pp. 81–92.

[8] Ahsan Javed Awan et al. “Performance Characterization of In-Memory Data Analytics on a Modern Cloud Server”. In: *Big Data and Cloud Computing (BDCloud), 2015 IEEE Fifth International Conference on*. IEEE. 2015, pp. 1–8.

[9] Jeffrey M Barth. “Shifting Garbage Collection Overhead to Compile Time”. In: *Communications of the ACM* 20.7 (1977), pp. 513–518.

[10] Jean-Philippe Bernardy et al. “Linear Haskell: Practical Linearity in a Higher-Order Polymorphic Language”. In: *Proc ACM Symposium on Principles of Programming Languages*. Vol. 2. ACM, July 2017. URL: https://www.microsoft.com/en-us/research/publication/retrofitting-linear-types/.

[11] *Big Data Benchmark*. http://amplab.cs.berkeley.edu/benchmark/. AMPLab, Berkeley. 2014.

[12] Peter A Boncz, Marcin Zukowski, and Niels Nes. “MonetDB/X100: Hyper-Pipelining Query Execution.” In: *CIDR*. Vol. 5. 2005, pp. 225–237.

[13] Peter Buneman et al. “Principles of Programming With Complex Objects and Collection Types”. In: *Theor. Comput. Sci.* 149.1 (1995), pp. 3–48.

[14] C++ Standards Committee. *ISO/IEC 14882: 2011, Standard for Programming Language C++*. Tech. rep. International Standards Organization, 2011. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3690.pdf.

[15] Paris Carbone et al. “Apache Flink: Stream and Batch Processing in a Single Engine”. In: *Data Engineering* (2015), p. 28.

[16] Paris Carbone et al. “Apache Flink: Stream and Batch Processing in a Single Engine”. In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36.4 (2015).

[17] Hassan Chafi et al. “A Domain-Specific Approach to Heterogeneous Parallelism”. In: *ACM SIGPLAN Notices* 46.8 (2011), pp. 35–46.

[18] David G Clarke, John M Potter, and James Noble. “Ownership Types for Flexible Alias Protection”. In: *ACM SIGPLAN Notices*. Vol. 33. ACM. 1998, pp. 48–64.

[19] Edgar F Codd. “A Relational Model of Data for Large Shared Data Banks”. In: *Communications of the ACM* 13.6 (1970), pp. 377–387.

[20] Alain Colmerauer et al. *Un Systeme de Communication Homme-Machine en Francais*. Tech. rep. Technical report, Groupe de Intelligence Artificielle Universite de Aix-Marseille II, 1973.

[21] Andrew Crotty et al. “An Architecture for Compiling UDF-Centric Workflows”. In: *VLDB* 8.12 (2015), pp. 1466–1477.

[22] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”. In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[23] L Peter Deutsch and Daniel G Bobrow. “An Efficient, Incremental, Automatic Garbage Collector”. In: *Communications of the ACM* 19.9 (1976), pp. 522–526.

[24] Jennie Duggan et al. “The BigDawg Polystore System”. In: *ACM Sigmod Record* 44.2 (2015), pp. 11–16.

[25] Jeff Epstein, Andrew P Black, and Simon Peyton-Jones. “Towards Haskell in the Cloud”. In: *ACM SIGPLAN Notices*. Vol. 46. 12. ACM. 2011, pp. 118–129.

[26] *Erlang Programming Language*. URL: https://www.erlang.org/.

[27] Kayvon Fatahalian et al. “Sequoia: Programming the Memory Hierarchy”. In: *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM. 2006, p. 83.

[28] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. “Classes and Mixins”. In: *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1998, pp. 171–183.

[29] David Garlan, Robert Allen, and John Ockerbloom. “Architectural Mismatch: Why Reuse is so Hard”. In: *IEEE software* 12.6 (1995), pp. 17–26.

[30] David Garlan, Robert Allen, and John Ockerbloom. “Architectural Mismatch: Why Reuse is Still so Hard”. In: *IEEE software* 26.4 (2009).

[31] Dan Grossman et al. “Region-Based Memory Management in Cyclone”. In: *ACM Sigplan Notices* 37.5 (2002), pp. 282–293.

[32] Apache Hadoop. *Hadoop*. 2009.

[33] Joseph M. Hellerstein. *The Declarative Imperative: Experiences and Conjectures in Distributed Logic*. Tech. rep. UCB/EECS-2010-90. EECS Department, University of California, Berkeley, June 2010. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-90.html.

[34] Matthew Hertz and Emery D Berger. “Quantifying the Performance of Garbage Collection vs. Explicit Memory Management”. In: *ACM SIGPLAN Notices*. Vol. 40. ACM. 2005, pp. 313–326.

[35] Benjamin Hindman et al. “Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center.” In: *NSDI*. Vol. 11. 2011. 2011, pp. 22–22.

[36] Howard E Hinnant, Peter Dimov, and Dave Abrahams. “A Proposal to add Move Semantics Support to the C++ Language”. In: *ISO/IEC JTC1/SC22/WG21 Document N* 1377 (2002), pp. 02–0035.

[37] Ralf Jung et al. “RustBelt: Securing the foundations of the Rust programming language”. In: *Proc. ACM Program. Lang. 2, POPL, Article* (2018).

[38] Alfons Kemper and Thomas Neumann. "HyPer: A Hybrid OLTP/OLAP Main Memory Database System based on Virtual Memory Snapshots". In: *ICDE*. IEEE. 2011.

[39] Gregor Kiczales et al. "Aspect-Oriented Programming". In: *European Conference on Object-Oriented Programming*. Springer. 1997, pp. 220–242.

[40] Oleg Kiselyov and Chung-chieh Shan. "Interpreting Types as Abstract Values". In: *Lecture notes from the Formosan Summer School on Logic, Language, and Computation* (2008).

[41] Yannis Klonatos et al. "Building Efficient Query Engines in a High-Level Language". In: *VLDB* 7.10 (2014), pp. 853–864.

[42] Marcel Kornacker and Justin Erickson. "Cloudera Impala: Real Time Queries in Apache Hadoop, For Real". In: *http://blog.cloudera.com/blog/2012/10/cloudera-impala-real-time-queries-in-apache-hadoop-for-real* (2012).

[43] Konstantinos Krikellas, Stratis Viglas, and Marcelo Cintra. "Generating Code for Holistic Query Evaluation". In: *ICDE*. 2010. DOI: 10.1109/ICDE.2010.5447892. URL: http://dx.doi.org/10.1109/ICDE.2010.5447892.

[44] William Landi. "Undecidability of Static Analysis". In: *ACM Letters on Programming Languages and Systems (LOPLAS)* 1.4 (1992), pp. 323–337.

[45] Xavier Leroy et al. "The OCaml System Release 4.02: Documentation and User's Manual". PhD thesis. Inria, 2014.

[46] Per Lidén and Stefan Karlsson. *The Z Garbage Collector*. http://openjdk.java.net/projects/zgc/. 2018.

[47] Jimmy Lin. "MapReduce is Good Enough? If All You Have is A Hammer, Throw Away Everything That's Not a Nail!" In: *Big Data* 1.1 (2013), pp. 28–37.

[48] Yucheng Low et al. "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud". In: *Proceedings of the VLDB Endowment* 5.8 (2012), pp. 716–727.

[49] John M Lucassen and David K Gifford. "Polymorphic Effect Systems". In: *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1988, pp. 47–57.

[50] Grzegorz Malewicz et al. "Pregel: A System for Large-Scale Graph Processing". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM. 2010, pp. 135–146.

[51] Simon Marlow et al. "Haskell 2010 Language Report". In: *http://www.haskell.org/* (2010).

[52] Ian Mason and Carolyn Talcott. "Equivalence in Functional Languages with Effects". In: *Journal of functional programming* 1.3 (1991), pp. 287–327.

[53] Nicholas D Matsakis and Felix S Klock II. "The Rust Language". In: *ACM SIGAda Ada Letters*. Vol. 34. ACM. 2014, pp. 103–104.

[54] Nancy Mazur. "Compile-Time Garbage Collection for the Declarative Language Mercury". PhD. Katholieke Universiteit Leuven, 2004.

[55] John McCarthy. "Recursive Functions of Symbolic Expressions and their Computation By Machine, Part I". In: *Communications of the ACM* 3.4 (1960), pp. 184–195.

[56] Xiangrui Meng et al. "MLLib: Machine Learning in Apache Spark". In: *The Journal of Machine Learning Research* 17.1 (2016), pp. 1235–1241.

[57] Gordon Moore. "Cramming More Components onto Integrated Circuits". In: *Electronics* 38.8 (1965).

[58] Derek G Murray et al. "Naiad: A Timely Dataflow System". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM. 2013, pp. 439–455.

[59] Fabian Nagel, Gavin Bierman, and Stratis D Viglas. "Code Generation for Efficient Query Processing in Managed Runtimes". In: *PVLDB* 7.12 (2014).

[60] Thomas Neumann. "Efficiently Compiling Efficient Query Plans for Modern Hardware". In: *PVLDB* 4.9 (2011), pp. 539–550.

[61] Martin Odersky et al. "The Scala Programming Language". In: *http://www.scala-lang.org* (2008).

[62] Kay Ousterhout et al. "Making Sense of Performance in Data Analytics Frameworks". In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. 2015, pp. 293–307.

[63] *Project Tungsten: Bringing Spark Closer to Bare Metal*. https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html. Accessed: 2015-11-19.

[64] *Python Programming Language*. URL: http://www.python.org/.

[65] Tiark Rompf and Martin Odersky. "Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs". In: *Communications of the ACM* 55.6 (2012), pp. 121–130.

[66] *Ruby Programming Language*. URL: http://www.ruby-lang.org/.

[67] Vivek Sarkar. *Why the End-Game for Moore's Law will be driven by a Compiler Renaissance?* Keynote Address. Compiler Construction, 2017. URL: https://conf.researchr.org/event/CC-2017/cc-2017-keynote-why-the-end-game-for-moore-s-law-will-be-driven-by-a-compiler-renaissance-.

[68] Srinath Shankar et al. "Query Optimization in Microsoft SQL Server PDW". In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM. 2012, pp. 767–776.

[69] Jeff Shute et al. "F1: The Fault-Tolerant Distributed RDBMS Supporting Google's Ad Business". In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM. 2012, pp. 777–778.

[70] Zoltan Somogyi, Fergus J Henderson, and Thomas Charles Conway. "Mercury, An Efficient Purely Declarative Logic Programming Language". In: *Australian Computer Science Communications* 17 (1995), pp. 499–512.

[71] Michael Stonebraker et al. "SciDB: A Database Management System for Applications with Complex Analytics". In: *Computing in Science & Engineering* 15.3 (2013), pp. 54–62.

[72] Arvind K Sujeeth et al. "Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages". In: *ACM Transactions on Embedded Computing Systems (TECS)* 13.4s (2014), p. 134.

[73] T. Wurthinger et al. "One VM to Rule Them All". In: *Onward!* 2013.

[74] Mads Tofte and Lars Birkedal. "A Region Inference Algorithm". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 20.4 (1998), pp. 724–767.

[75] Mads Tofte and Jean-Pierre Talpin. "Region-Based Memory Management". In: *Information and computation* 132.2 (1997), pp. 109–176.

[76] *TPC-H Benchmark Specification*. http://www.tpc.org/tpch/. Transaction Processing Performance Council. 2008.

[77] Madhavi Valluri and Lizy K John. "Is Compiling for Performance—Compiling for Power?" In: *Interaction between compilers and computer architectures*. Springer, 2001, pp. 101–115.

[78] Guido Van Rossum et al. “Python Programming Language.” In: *USENIX Annual Technical Conference*. Vol. 41. 2007.

[79] Philip Wadler. “Deforestation: Transforming Programs to Eliminate Trees”. In: *European Symposium on Programming*. Springer. 1988, pp. 344–358.

[80] Philip Wadler. “Linear Types can Change The World”. In: *IFIP TC*. Vol. 2. 1990, pp. 347–359.

[81] Philip Wadler. “Listlessness is Better than Laziness: Lazy Evaluation and Garbage Collection at Compile-Time”. In: *Proceedings of the 1984 ACM Symposium on LISP and functional programming*. ACM. 1984, pp. 45–52.

[82] David Wakeling and Colin Runciman. “Linearity and Laziness”. In: *Conference on Functional Programming Languages and Computer Architecture*. Springer. 1991, pp. 215–240.

[83] Reynold S Xin et al. “Graphx: A Resilient Distributed Graph System on Spark”. In: *First International Workshop on Graph Data Management Experiences and Systems*. ACM. 2013, p. 2.

[84] Zhihong Xu, Martin Hirzel, and Gregg Rothermel. “Semantic Characterization of MapReduce Workloads”. In: *Workload Characterization (IISWC), 2013 IEEE International Symposium on*. IEEE. 2013, pp. 87–97.

[85] Matei Zaharia et al. “Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters.” In: *HotCloud* 12 (2012), pp. 10–10.

[86] Matei Zaharia et al. “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing”. In: *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association. 2012, pp. 2–2.

[87] Matei Zaharia et al. “Spark: Cluster Computing with Working Sets”. In: *Proceedings of the 2nd USENIX conference on Hot Topics in Cloud Computing*. Vol. 10. 2010, p. 10.

Biography

P.C. Shyamshankar was born in 1988 in Ithaca, NY.

Shyam did his undergraduate studies in Chennai India, majoring in Computer Science and Engineering, with a focus on Algorithms and Distributed Data Analytics.

In 2011, Shyam joined the doctoral program at Johns Hopkins University with Dr. Yanif Ahmad, working at the intersection of Databases, Programming Languages and Compilers research.

During this time, he has served as a teaching assistant for a variety of courses, including Database System Implementation, Declarative Methods and Randomized Algorithms. In addition, he has undertaken industrial research at Facebook Inc, and at Galois Inc.