Functional Compilation and Functional Program Analysis

By
Ashraful Islam
Supervisor: Dr. Thomas Gilray

Reported to
Dr. Da Yan
Department of Computer Science
The University of Alabama at Birmingham

Semester: Fall 2023
November 29, 2023

Contents

1 Abstract **3**

2 Introduction **3**

3 Functional Programming **4**

3.1 Lambda Calculus 5
3.2 Fundamental Concepts 6
3.2.1 First-class Functions 6
3.2.2 Higher-order Functions 6
3.2.3 Pure Functions 7
3.2.4 Strict and Lazy Evaluation 7

4 Functional Program Analysis **8**

4.1 Dynamic Analysis 8
4.2 Static Analysis 8
4.3 Control Flow Analysis 9
4.3.1 0-CFA 10
4.3.2 k-CFA 10

5 Language and Compiler Design **10**

5.1 The Brouhaha Language 10
5.2 Compiler Design 11
5.2.1 Desugaring 12
5.2.2 Alphatization 13
5.2.3 ANF Conversion 14
5.2.4 CPS Conversion 16
5.2.5 Closure Conversion 17
5.2.6 Code Generation 17

6 Abstract Interpretation **21**

6.1 Limitation of Concrete Interpretation 22
6.2 Abstract Interpretation and Galois Connection 22
6.3 Challenges and Limitations 23
6.4 Abstracting Abstract Machines (AAM) 23
6.4.1 CEK Machine 23
6.4.2 CESK Machine 24
6.4.3 CESK* Machine 25
6.5 Environment Analysis 25
6.6 Abstract Counting 26

7 Implementation Approaches For Reasoning Systems **26**

7.1 Interprocedural Program Analysis (IPA) 27
7.2 SAT 27
7.3 Datalog 28
7.3.1 Soufflé 29
7.3.2 Flix 29
7.3.3 Datafun 29
7.3.4 IncA 30
7.3.5 Ascent 30
7.3.6 Slog 31
7.3.7 Slog-Based Analysis 31

8 Future Research Direction **32**

9 Conclusion **32**

1 Abstract

Functional programming (FP) focuses on mathematical functions and immutable data structures, setting it apart from imperative programming which often depends on mutability and side effects. While imperative programming frequently use state changes, FP prioritizes encapsulation and composability. Its inherently expressive and maintainable nature has driven FP's principles into not only pure functional languages but also multi-paradigm languages such as Java and `C++`. The analysis of functional programs on the other hand employs techniques such as type inference, control-flow analysis, and program transformation. These methodologies allow for a comprehensive understanding and reasoning about program behaviors. This analytical aspect has its roots in lambda calculus which forms the theoretical backbone of functional programming. Abstract interpretation, a notable static analysis technique, approximates a program's behavior, and abstract machines—like CEK and CESK—provide the means to implement these techniques. The utilization of such analyses benefits functional compilers immensely. As they transform a high-level program into an efficient low-level code through multiple phases, insights from techniques like abstract interpretation and continuation-passing style empower them to detect bugs and seize optimization opportunities more effectively. As a result, functional compilation and functional program analysis have grown into pivotal areas of research in computer science, especially in the areas of parallelism, modularity, and language design.

2 Introduction

Functional programming (FP) is a programming paradigm that focuses on mathematical functions and immutable data structures for writing programs unlike traditional imperative programming, which revolves around making repeated effects to the program. While FP promotes encapsulation and composability, imperative programming often leads to complicated invariants and is susceptible to bugs due to the extensive use of assignment operation [1]. Over the years, FP has made a significant impact on both academia and industry due to its expressive and maintainable nature [2, 3, 4, 5, 6]. As a result, it has been widely adopted not just in purely functional languages like Haskell, Standard ML (SML), Scheme, Clojure, Miranda, and Scala, but also in multi-paradigm languages such as Java and `C++`, which now includes lambdas, maps, and other functional features [7, 8, 9, 10, 11, 12, 13, 14]. FP has also become a critical research field in computer science, especially in the areas of parallelism, modularity, and language design [15, 16, 17].

Functional program analysis applies mathematical logic and methods such as type inference [18, 19], control-flow analysis [20, 21, 22, 23, 24], and program transformation [25, 26] to reason about programs and improve their efficiency and correctness. However, these methods are influenced by earlier work and have their roots in lambda calculus, which is a formal system for expressing computation based on only the three capacities of function abstraction, function application, and variable reference [27]. Since its inception, λ-calculus has formed the theoretical backbone of functional programming, and its principles have been extensively studied [25]. Hence, functional compilers and their associated optimization approaches rely heavily on λ-calculus to perform their tasks efficiently [26]. For example, the Glasgow Haskell Compiler is widely adopted in academia and industry, featuring advanced optimization techniques and a rich ecosystem of libraries [28, 29].

Functional compilers typically undergo several passes to transform a high-level input program into efficient low-level code that can be executed on various platforms. For example, one of the initial simplifications performed by a functional compiler involves eliminating complex language features, such as core macros, by replacing them with semantically equivalent simpler forms. This pass is commonly known as *desugaring*. The next pass is *Alphatization*, which transforms a program to ensure that every variable-binding has a distinct name for a single binding point. Then *Administrative Normal Form (ANF)* conversion administratively binds each sub-expression to a unique identifier and enforces an explicit order of evaluation to simplify the continuation structure [30]. The *continuation-passing-style (CPS)* conversion is the next pass, which imposes a constraint on call sites to always be in the tail position, meaning that functions never return in the conventional sense; instead, a continuation is explicitly passed forward to be invoked on the return point [31]. CPS is a source-to-source translation that implements the stack, and it is also possible to reconstruct a program back to its direct-style form with no loss of efficiency or analysis results [32, 33]. After these passes, the program is ready for the *closure-conversion* pass, which removes free variables by requiring an

explicit environment structure [34, 35]. The final pass is often *code generation*, where the program is typically compiled down to `C` or `C++`, and linked with libraries for the target platform [34].

Abstract interpretation is a static analysis technique that approximates program behavior over abstract domains. This technique has been extensively used in functional programming analyses and serves applications such as proving program properties, bug detection, compiler optimization, and so on [36, 37, 38, 39]. One approach to implementing this technique is to use abstract machines and interpreters, such as CE, CEK, CESK, and CESK* [39, 40, 41, 42, 43, 44]. These simplified models of computation and interpreters provide the foundation for analyzing programs and a basis for further optimizations.

This survey paper provides a comprehensive overview of functional compilation and functional program analysis covering essential topics such as λ-calculus, compiler construction, and abstract interpretation. It also summarizes the latest developments in the field and discusses the significance of abstract machines and interpreters [42, 43, 44]. As part of our ongoing research, we are working on a prototype compiler and using SLOG[45] for performing analyses such as abstract counting [46] on it. By doing so, we are trying to address novel research problems, evaluate the compiler's performance, and figure out opportunities for optimization.

The remainder of the survey is structured as follows. In Section 3 we provide a brief introduction to functional programming, lambda calculus, and FP concepts. Section 4 discusses functional program analysis and some of its methodologies. Then, in Section 5, we define a functional language and demonstrate how our prototype functional compiler works for that language. Moving forward, we cover abstract interpretation and abstract machines in Section 6. Then, Section **??** covers modern Datalog languages and how we are performing program analysis using a Datalog-like language—Slog. Section 8 outlines future research directions and finally, Section 9 concludes the survey.

3 Functional Programming

Functional programming (FP) is both a paradigm and an approach that has deeply impacted the history and development of computer science. Its roots can be traced back to the foundational works in the λ-calculus introduced by Alonzo Church [27], which established the groundwork for understanding computation through function application. The fundamental operation in a functional program is the application of functions to arguments, where the main program takes input and returns the output as a result. Much like mathematical functions, typically the main program or function is composed of multiple layers of functions, each defined in terms of others, until reaching the foundational language primitives [3].

One of the defining characteristics of FP is its focus on immutability and the use of first-class functions. Unlike imperative programming, where computation largely centers around mutable states and the sequences in which commands are executed, functional programming emphasizes the declarative description of what should be computed, rather than how it should be computed [4].

```
# Python Code
>> def sum(lst):
     total = 0
     for num in lst:
         total += num
     return total

>> sum([1, 2, 3])
# output: 6
```

```
; Racket Code
>> (define (sum lst)
       (cond [(null? lst) 0]
             [else
                 (+ (car lst)
                    (sum (cdr lst))
                       )]))

>> (sum (list 1 2 3))
; output: 6
```

Figure 1: Comparison between imperative and functional programming

For example, a program written in imperative style often computes a value step-by-step, where we exactly describe *how* to perform a computation as shown in the left side of Figure 1. We begin the computation process by assigning *total=0*, then accessing each item from the list, while repeatedly updating the *total* variable, and finally returning the *summation* of the list. In contrast programs written in functional style as shown on the right side of Figure 1, we perform the computation in a declarative way—analogous to how we define mathematical functions—by describing *what* the sum of a list is in terms of its structure, rather than detailing *how* to compute it step-by-step. Here, we check if the list is *null?*, and if so the *sum* is 0. Otherwise, the *sum* is the first element of the list added to the *sum* of the remaining elements. This shift in focus offers numerous benefits, including increased expressiveness, easier reasoning about program behavior, and natural facilities for parallelism and concurrency.

During the latter part of the 20th century, functional programming made its mark with languages like Scheme, introduced by Sussman and Steele [9], as well as Miranda [11] and Haskell [7]. Each of these languages supported unique features of the FP paradigm and significantly impacted both academia and the real-world software domain. Notably, Haskell—which is a purely functional language—contributed to a great extent to evolving FP concepts such as lazy evaluation, type classes, and monads [28, 29].

Yet, functional programming is not just a way to write code; it is deeply rooted in the foundational theories of computer science. The ties among FP, program analysis, and compiler optimization techniques, notably abstract interpretation and continuation-passing style highlight how FP combines conceptual theory with actual practice [36, 37, 47, 33]. Researchers such as Landin [40], Felleisen, and Friedman [42, 43] further illustrate these intricate interactions in their pivotal works.

In recent years, functional programming principles have not remained confined to purely functional languages. Hybrid languages, such as Scala [12] and Clojure [10], have emerged, fusing functional paradigms with object-oriented and imperative paradigms. Some of the other well-known functional languages include ML, OCaml, Lisp, Erlang, `F#`, and Racket [8, 48, 49, 50, 51, 52, 53, 54]. Now, most of the multi-paradigm languages, including Java, `C++`, Python, `C#`, Ruby, Go, PHP, Kotlin, Rust, Perl, and JavaScript have also embraced functional concepts [13, 14, 55, 56, 57, 58, 59, 60, 61, 62, 63], signaling a broader industry acceptance and appreciation of the benefits that FP can offer.

Over time, FP has evolved, leading to the development of numerous influential programming languages, theories, and methodologies, and remained at the forefront of both academic research and industry application. By providing tools and methodologies that allow for rigorous analysis, elegant solutions, and scalable applications, FP servers as a crucial area of study for anyone interested in the future of computation.

In this section, we first provide a brief introduction to lambda calculus (Section 3.1), and then we talk about some of the fundamental concepts of functional programming (Section 3.2).

3.1 Lambda Calculus

The heart of any functional programming language are the three essential forms of the *lambda calculus*: lambda abstraction (defining a function), application (invoking a function), and variable reference. In fact, the rest of Scheme (or any other programming language) can be compiled down into the language consisting of just these three forms: defining unary (single input) functions that bind a variable when invoked, invoking functions on a single argument expression, and referencing a variable. Compiling down to the pure lambda calculus is called *Church compiling/encoding*, after the creator of the lambda calculus, Alonzo Church.

While there are many different lambda calculi—systems for calculating (calculi) using functions (lambdas)—*the lambda calculus* generally refers to a specific classic system: the untyped, three-form lambda calculus with unary functions. We can define a grammar for its expressions/terms like so:

$$
\begin{aligned}
e, t \in \mathsf{E} ::=\ & \texttt{(}\lambda\ \texttt{(}x\texttt{)}\ e_b\texttt{)} && \lambda x.\, e_b \\
\mid\ & \texttt{(}e_f\ \ e_a\texttt{)} && e_f\ e_b \\
\mid\ & x && x \\
x, y \in \mathsf{Var} =\ & \langle\text{program identifiers}\rangle
\end{aligned}
$$

It is also common to see a notation without (required) parentheses and with a dot before the body of lambdas (alternatively shown on right)—λ is highest precedence.

Lambda calculus is found within many programming languages; it is the heart of functional programming languages such as Racket, Haskell, OCaml, but is also found within multi-paradigm languages permitting first-class functions such as Python, Ruby, Javascript, and even Java, and `C++`. For instance:

Python: `id = lambda x:  x`

Java: `Function<Object, Object> id = x -> x;`

Javascript: `const id = x => x;`

With an application form, we may apply an identity function on a value, which will yield that value unchanged. Let's take the identity function $(\lambda\ (x)\ x)$ and apply it to the number 5: $((\lambda\ (x)\ x)\ 5)$. This expression may be textually simplified using an evaluation-step relation ($\rightarrow$): $((\lambda\ (x)\ x)\ 5) \rightarrow 5$.

$$((\lambda\ (x)\ x)\ (\lambda\ (y)\ y)) \rightarrow (\lambda\ (y)\ y)$$

In pure lambda calculus, we do not have the ability to represent a constant like 5, but could still apply the identity function on itself as shown above.

3.2 Fundamental Concepts

This sub-section introduces several programming concepts that are prevalent in functional programming but are seldom encountered in other paradigms.

3.2.1 First-class Functions

In computer science, the term *functions as first-class citizens* was first used in the mid-1960s [64]. The idea is that, in a language, if we can pass functions to other functions as arguments, can have functions that return other functions as their results, and allow functions to be assigned to variables or stored within various data structures, then the language supports first-class functions. With this feature, functions are not merely procedural constructs but are recognized as first-class entities—numbers, strings, etc.—in the language.

Taking a step further, Scheme—a functional programming language (a dialect of Lisp)—introduced proper support for lexically scoped first-class functions through *closures* [64]. A *closures* encapsulates a function together with an environment of bound variables, ensuring that the function has access to the variables it is supposed to, even when it is executed in a different scope. This handling of *free-variables* has profound implications for function behavior and memory management.

3.2.2 Higher-order Functions

Higher-order functions are those that can both accept other functions as parameters and return functions as results. A language with first-class functions often implies the availability of higher-order functions in the language.

For example, the *make-adder* function in the left side of Figure 2 is an example of a first-class function because it is treated like any other variable. It returns a *lambda* function that takes a parameter *y* and adds it to *x*. We assign this returned function to the variable *add1*, illustrating that functions can be assigned to variables and thus are *first-class citizens* in the language. Finally, when we make a call to the *add1* function with argument *2*, to produce the final output as *3*.

On the other hand, in the right side of Figure 2, we show how higher-order functions work by using *map* and *filter*. First, we define a list containing numbers *1* to *5* and assign it to the variable *lst*. The *map* function takes the list and applies the *add1* function to each of the numbers to increment them by *1* and finally returns the output. Whereas, the *filter* function takes the output of *map* applied on *lst* as input and applies the predicate *even?* to each of the items and finally returns the output.

```
; First-class function example
>> (define (make-adder x)
       (lambda (y) (+ x y)))

>> (define add1 (make-adder 1))

>> (add1 2)
; output: 3
```

```
; Higher-order function example
>> (define lst (list 1 2 3 4 5))

>> (map add1 lst)
; output: '(2 3 4 5 6)

>> (filter even? (map add1 lst))
; output: '(2 4 6)
```

Figure 2: First-class and Higher-order functions (implemented in Racket)

Functional languages extensively use higher-order functions such as map, filter, apply, etc., to provide a level of abstraction and flexibility that is not possible with just first-class functions, allowing programmers to write more generalized and reusable code.

3.2.3 Pure Functions

A function is said to be pure if it does not have any side effects and always yields the same result for a given input making it have no discernible impact on the broader program execution. It can also be said that we can always replace a pure function with its result without changing the outcome of the program i.e., an expression may safely be replaced by its value [65]. This property ensures predictability and reproducibility, echoing the characteristics of mathematical functions and strengthening the theoretical foundations of functional programming. The simplicity and clarity introduced by pure functions facilitate easier program analysis, driving more accurate and efficient evaluations of program behavior. Functional programming advocates the use of pure functions and the absence of side effects has several benefits including opportunities for compiler optimizations, and faster code transformation.

3.2.4 Strict and Lazy Evaluation

Based on the order of evaluation of expressions, a functional programming language can be categorized as either strict/eager or lazy/delayed. For example, SML is a strict language, while Haskell adopted a lazy evaluation strategy. In literature, they are often referred to by different names, for instance, *call-by-value (CBV)* means strict evaluation, whereas *call-by-need (CBN)* means lazy evaluation. The evaluation order differs by how they treat arguments to functions. In strict evaluation, a function's arguments are reduced first before applying the function itself, while in lazy evaluation, a function gets called with unevaluated arguments, and they are only reduced if the computation requires them in order to continue. One of the benefits of this delayed evaluation is that, once they are reduced, this reduced value is then cached to avoid any recomputation (also known as memorization [66]). Both of the evaluation order has their benefit over the other one, strict languages are easier to reason in terms of asymptotic complexity, while for lazy languages reasoning about programs are not usually straightforward [15]. Figure 3 shows how these evaluation strategies typically work in a functional language (in this case lambda calculus). Here, β-reduction simulates function application (i.e., invocation, instantiation).

$$
\begin{aligned}
& ((\lambda\ (y)\ y)\ ((\lambda\ (z)\ z)\ w)) \\
\to_\beta\ & ((\lambda\ (y)\ y)\ w) \\
\to_\beta\ & w
\end{aligned}
$$

$$
\begin{aligned}
& ((\lambda\ (y)\ y)\ ((\lambda\ (z)\ z)\ w)) \\
\to_\beta\ & ((\lambda\ (z)\ z)\ w) \\
\to_\beta\ & w
\end{aligned}
$$

Figure 3: A side-by-side comparison of evaluation strategies: Strict (left), Lazy (right)

4 Functional Program Analysis

The analysis of functional programs played a vital role in terms of enriching our comprehension and refinement of functional programming paradigms over the years. It helps us in examining and interpreting functional code, ensuring its accuracy and efficiency. This meticulous examination highlights FP's capacity for delivering concise, readable, and maintainable code, bolstered by its underlying mathematical principles. The trajectory of functional program analysis began in 1969 with the first systematic evaluation of LISP programs [67]. The initial phase focused on the intricate structures and behaviors inherent in Lisp-like data constructs [68, 69]. This foundation spurred a series of advancements, with studies and techniques evolving and expanding in complexity and depth. The ongoing evolution underscores the sustained importance of functional program analysis in understanding and optimizing program behaviors from its inception to contemporary applications.

In this section, first, we briefly cover two program analysis approaches in Section 4.1 and Section 4.2. Then, in Section 4.3 control-flow analysis is discussed.

4.1 Dynamic Analysis

Dynamic analysis is a program analysis technique that examines program properties during the actual execution of a program [70]. It is facilitated through program instrumentation—a process of inserting additional statements into a program to create traces. This process adeptly evaluates a program's runtime behaviors, capturing and recording specific execution events that are encapsulated within a dynamic state, also referred to as profiling or tracing [71, 72, 73]. These recorded events encompass a broad spectrum including the execution of lines of code, basic blocks, control edges, and routines [74]. The effectiveness of dynamic analysis hinges on the efficiency of the instrumentation and profiling infrastructures [75] and these systems must be adept at collecting comprehensive runtime information without compromising the program's operational efficiency. This information then becomes instrumental for enhancing the program's memory layout to improve locality, identifying dominant program segments for optimization, and facilitating debugging among other applications [76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87]. Over the years, various dynamic analysis tools have been established to perform these functions effectively, including but not limited to Valgrind, Google Address sanitizer, Daikon, Javana, Purify, DynaMetrics, and Caffeine [77, 88, 83, 81, 80, 87, 89]. However, a notable limitation looms over dynamic analysis results because of its specificity to the set of inputs used during testing. This constraint makes the analysis results not universally applicable to future program executions or reliable for applications requiring precise input, such as semantics-preserving code transformations [90]. It is worth noting that while dynamic analysis excels in bug detection and offers a granular understanding of program behavior, it cannot be used to prove program properties [91], unlike static analysis.

4.2 Static Analysis

Static analysis is a program analysis technique that, unlike dynamic analysis, reasons about the behavior of programs without running them. Its root can be traced back to the 1970s when it was primarily used for compiler optimizations [92]. Since then this technique has been studied extensively by computer scientists and applied in real-world software by engineers or developers [93, 90]. Nowadays, it is integral in various stages of program development, including identifying potential errors early in the development process, verification, optimization, refactoring, and maintenance [94, 95]. It is especially crucial for certifying critical software and enhancing the quality of general-purpose applications. A static program analyzer is a program itself that reason about the behavior of other programs [96]. Over the years for most of the multi-paradigm languages, a number of analyzers or tools based on static analysis techniques have been developed. Some of the most prominent tools are Lint, FindBugs, SpotBugs, CryptoGuard, PMD, mygcc, Synopsys, Clang Static Analyzer, PyLint, Pyflakes, and Frosted, CrySL [97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108].

Although every possible behavior of a program cannot be entirely predicted due to the undecidability issues stated by Turing [109] and Rice [110], static program analysis, with appropriate approximations, can inspect all conceivable executions of the program [96]. Analyzing a program's source text, static analysis aims to discern and predict the program's behavior. It strives to offer guarantees about the program's operation, balancing the need for precision with the necessity for efficiency. This balance is crucial for producing

results that are actionable and reliable without being overly resource-intensive or time-consuming [96, 111]. Compared to dynamic analysis, static analysis has the advantage of not affecting the program's performance during execution and provides an opportunity to identify and rectify issues before the program is put into use [95]. Dynamic analysis, while offering real-time insights, can impact performance and does not necessarily lead to immediate remediation of identified issues.

4.3 Control Flow Analysis

The term control-flow analysis (CFA) was first used for lambda calculus in 1981 [20, 21] and since then CFA of functional programs has been an active area of research for the past four decades. It helps us to approximate which values a variable might take on during a program's execution allowing us to understand a program's behavior. In a program without higher-order functions, the operator of a function call is explicitly discernible from the program's text, given that it is represented by a lexically visible identifier. Consequently, the called function is accessible at compile time, allowing an analysis to directly correlate with the program's control flow [39].

For example, notice the simple Python function (in Figure 4) named *add* that takes in two arguments and returns their sum. When we call the *add* function with the numbers 4 and 5, it is quite clear from the code that the *add* function will be executed. The function being called, which we refer to as the *operator*, is directly visible in the code. In this kind of scenario, where the control flow is evident at compile time, is relatively straightforward to analyze.

```
# Python Code
>> def add(x, y):
     return x + y

>> (add 4 5)
# output: 6
```

```
; Racket Code
>> (define (add x y)
      (+ x y))

>> (define (apply fname x y)
             (fname x y))

>> (apply add 4 5)
; output: 6
```

Figure 4: A side-by-side comparison of programs: with and without Higher-Order functions

In contrast, in programs that use higher-order functions, identifying the operator of a function call from the program's text can be more complex: the operator can emerge as an outcome of a specific computation, making it inaccessible until the program is run. This complexity necessitates the implementation of control-flow analysis to estimate during compile-time the functions that might potentially be invoked during runtime.

For example (see Figure 4), we have two functions: *apply* and *add*. The *apply* function takes another function as its first argument and then two numbers. It then calls the passed function with the two numbers as its arguments. So, when we call *apply* with *add* and the numbers 4 and 5, the *add* function is executed, and we get the final sum as a result. Notice the operator of the function call, in this case, is the *apply* function. But indirectly, it is also the *add* function because *apply* calls the *add* function in its body. This dynamic nature means that the actual function being executed might not be determined until runtime and this is what makes the control flow less obvious during compile time. In essence, this analysis aids in predicting the potential pathways the program might follow during its execution, providing insights into the dynamic behaviors exhibited in the presence of higher-order functions.

Control flow analysis has been studied extensively over the years and evolved significantly in terms of advancements in application techniques, proof of soundness, and methodological refinements [22, 23, 24, 46, 112, 113, 114, 115, 116].

4.3.1 0-CFA

0-CFA is a monovariant analysis or context-insensitive analysis developed by Shivers' for Scheme [23, 24]. Sometimes it is also referred to as inclusion-based [117] or subset-based analysis [118]. During the analysis, operationally it neglects coordination between bindings and merges all bindings of the same variable, trading precision for efficiency. Despite this, 0-CFA is surprisingly effective in practice [119] and runs in cubic time and variants are used in optimizing compilers like Bigloo [120] and MLton [121]. Operationally, in 0-CFA there is just one abstract address per variable that exists in a program. For example, consider a program that has a variable x, and it was bounded in 10 places. All 10 references to the variable x will be merged and mapped to one abstract address in the analysis without keeping track of any context/scope for the variable x.

The original 0-CFA of Shivers is actually flow-sensitive, which computes an abstract environment per expression rather than maintaining a global one [39, 122]. Other contemporary researchers worked on a flow-insensitive approach that went through a lot of refinement in subsequent years [113, 114, 115, 116, 123]. However, flow-sensitivity alone does not significantly improve precision in purely functional languages, but combining it with other abstractions can [124].

4.3.2 k-CFA

Unlike, 0-CFA, k-CFA is a polyvariant analysis meaning that it enhances the the precision of the analysis by incorporating context sensitivity, which considers the calling context of functions. It works by approximating the dynamic calling contexts of a program up to a certain fixed length, to analyze the flow of values to expressions and variables more accurately than the context-insensitive 0-CFA. In 1991 in his PhD thesis Shivers, proposed 1-CFA and hinted towards a generalized implementation of k-CFA [24]. The complexity of k-CFA is significant; for instance, it has been proven to be complete for EXPTIME [125], implying that for any fixed k, the analysis can simulate an exponential time Turing machine. Later on, in order to improve k-CFA's efficiency without compromising precision, polynomial-time variants have been introduced, which offer consistent context distinction across the analysis [126]. Furthermore, for object-oriented languages, where the distinction between objects and closures enables more streamlined analyses, a context-sensitive, polynomial-time analysis (m-CFA) closely resembling the precision of traditional k-CFA was developed [127].

Operationally, an analysis based on k-CFA would use an address allocation scheme to represent abstract bindings and these address then becomes the key parameter for tuning the analysis. However, there is a fundamental limitation in how k-CFA works i.e., no matter how much context we add to our analysis by increasing the k parameter. It cannot be determined with a 100% guarantee that given two abstract environments they are equal or not. It is possible that environments are abstractly equal but their concrete values may not be equal in all the cases. Might and Shivers however addressed this limitation with environment analysis [46] making it the most precise context-sensitive analysis in the family of CFA. We briefly discuss environment analysis in Section 6.5.

5 Language and Compiler Design

To illustrate the concepts of a typical functional compiler, we do not require a language with a very large set of features. Therefore, in this section, we formally define a functional language *Brouhaha*—which is a minimal Racket-like language (Section 5.1), and then we demonstrate how our prototype functional compiler works for the *Brouhaha* language (Section 5.2).

5.1 The Brouhaha Language

The *Brouhaha* language only has top-level *defines*, and we may define it using a context-free grammar, see Figure 5. Top-level functions can handle a varying number of parameters, similar to functions in the Racket language. The function body is composed of expressions, which can be as simple as a variable reference or as complex as a series of expressions, and operations, including literals, primitive operations, and conditional statements. Each function in Brouhaha, whether defined at the top level or as a *lambda* expression, can take multiple parameters, echoing the flexibility found in Racket's variadic functions.

The *quote* construct allows the direct inclusion of atomic values such as *numbers, boolean, strings*, and *symbols*—which do not require any evaluation. For operations, *prim* applies built-in functions to multiple arguments, whereas *apply-prim* is used for single-argument cases. The language handles decision-making through *if, and, or*, and *not* constructs while other forms like *let*, *let**, and *apply* in Brouhaha also functions just like Racket.

```
⟨program⟩ ::= ⟨def⟩*
    ⟨def⟩ ::= (define (⟨var⟩ ⟨var⟩*) ⟨exp⟩)
    ⟨exp⟩ ::= ⟨var⟩
            | (quote ⟨datum⟩)
            | (prim ⟨op⟩ ⟨exp⟩*)
            | (apply-prim ⟨op⟩ ⟨exp⟩)
            | (lambda (⟨var⟩*) ⟨exp⟩)
            | (lambda ⟨var⟩ ⟨exp⟩)
            | (let ((⟨var⟩ ⟨exp⟩)*) ⟨exp⟩)
            | (let* ((⟨var⟩ ⟨exp⟩)*) ⟨exp⟩)
            | (if ⟨exp⟩ ⟨exp⟩ ⟨exp⟩)
            | (and ⟨exp⟩*)
            | (or ⟨exp⟩*)
            | (not ⟨exp⟩)
            | (apply ⟨exp⟩ ⟨exp⟩)
            | (⟨exp⟩ ⟨exp⟩*)
    ⟨var⟩ ::= ⟨program identifiers⟩
  ⟨datum⟩ ::= ⟨integer⟩ | ⟨float⟩ | ⟨boolean⟩ | ⟨string⟩ | ⟨symbol⟩
     ⟨op⟩ ::= ⟨set of built-in primitives⟩
```

Figure 5: The Brouhaha language IR

5.2 Compiler Design

The *Brouhaha* compiler takes a source *brouhaha* file and undergoes a series of passes, where the output from one pass serves as the input for the next pass. In the end, the compiler generates a `C++` file that can be compiled to produce the same output as the *brouhaha* source file does. To illustrate this process, we will trace the transformation of the following (see Figure 6) factorial function (implemented in Racket) through each pass.

```
(define (fact n)
    (let* ([zero 0]
           [one 1])
        (if (= zero n)
            one
            (* n (fact (- n one))))))
```

Figure 6: The factorial function

In this section, we will discuss the following compiler passes: Desugaring, Alphatization, Administrative

Normal Form (ANF) Conversion, Continuation-Passing Style (CPS) Conversion, Closure Conversion, and Code Generation. Figure 7 depicts the overall structure of the compiler.

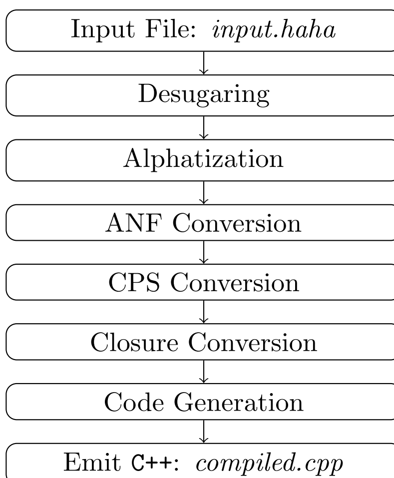
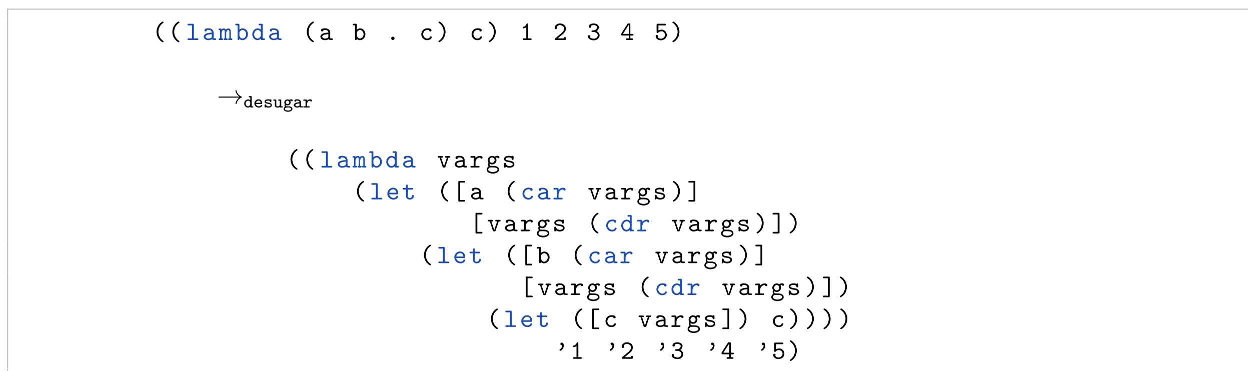


Figure 7: The compiler structure

5.2.1 Desugaring

Brouhaha is a minimal Racket-like language, but it supports a variety of complex language features and in this pass, the compiler removes these features with semantically equivalent simpler forms. Although these features provide a programmer more flexibility to write the same code in different ways, it makes the compilation process harder. So, removing the syntactic sugar ensures that the remaining compiler passes can only focus on simpler or core language forms.

Brouhaha has two types of functions: top-level defines and lambdas. Each function can be of the following three distinct categories: fully variadic functions, fixed arity functions, and functions with improper lists that allow optional arguments. To support the latter in our language, we desugar them into fully variadic functions, see Figure 8.



```
((lambda (a b . c) c) 1 2 3 4 5)

    →desugar

        ((lambda vargs
            (let ([a (car vargs)]
                  [vargs (cdr vargs)])
              (let ([b (car vargs)]
                    [vargs (cdr vargs)])
                (let ([c vargs]) c))))
          '1 '2 '3 '4 '5)
```

Figure 8: Function with improper list to fully variadic function

We desugar *let** in terms of nested *lets* and other features such as *and, or, not* are expressed in terms of *if* form. Whereas *datums* are desugared into *(quote datum)*. So, after this pass, our language only contains its core features and we may define it using a context-free grammar, see Figure 9.

Example Figure 10 shows the transformation of the factorial function following the desugaring pass. It also shows how *let** form is expressed in terms of nested *lets*.

$$
\begin{aligned}
\langle\text{exp}\rangle ::= &\ \langle\text{var}\rangle \\
&|\ (\text{quote}\ \langle\text{datum}\rangle) \\
&|\ (\text{prim}\ \langle\text{op}\rangle\ \langle\text{exp}\rangle^*) \\
&|\ (\text{apply-prim}\ \langle\text{op}\rangle\ \langle\text{exp}\rangle) \\
&|\ (\text{lambda}\ (\langle\text{var}\rangle^*)\ \langle\text{exp}\rangle) \\
&|\ (\text{lambda}\ \langle\text{var}\rangle\ \langle\text{exp}\rangle) \\
&|\ (\text{let}\ ((\langle\text{var}\rangle\ \langle\text{exp}\rangle)^*)\ \langle\text{exp}\rangle) \\
&|\ (\text{if}\ \langle\text{exp}\rangle\ \langle\text{exp}\rangle\ \langle\text{exp}\rangle) \\
&|\ (\text{apply}\ \langle\text{exp}\rangle\ \langle\text{exp}\rangle) \\
&|\ (\langle\text{exp}\rangle\ \langle\text{exp}\rangle^*)
\end{aligned}
$$

Figure 9: Desugar IR

```
(define (fact n)
    (let ((zero '0))
        (let ((one '1))
            (if (= zero n)
                one
                (* n (fact (- n one)))))))
```

Figure 10: Factorial function after Desugaring

5.2.2 Alphatization

Alphatization, also known as α-conversion, transforms a program to ensure that every variable binding has a distinct name for a single binding point. We use alphatization as a compiler pass to address two fundamental issues: first, to eliminate the issue of variable conflation during program analysis, and second, to simplify subsequent compiler passes.

Variable conflation can happen during program analysis, especially in monovariant analysis such as 0-CFA when separate variables defined in different contexts are incorrectly assigned the same address. Alphatization addresses this issue by creating a unique correspondence between each variable name and its defining site. Consider the following example (Figure 11) where we rename the binding of variable a in both inner and outer *let* expressions.

```
(let ([a 5])
    (let ([a 10]) a))

    →α

        (let ([a1222 '5])
            (let ([a1223 '10]) a1223))
```

Figure 11: Variable renaming example

This pass also establishes an invariant, a consistency that we rely upon in the later stages of the compiler, which helps us avoid the possibility of unexpected program behavior and complicated debugging. Note that the grammar of our language remains unchanged after this pass.

Example Following this pass, the factorial function becomes as below, see Figure 12.

```
(define (fact n)
    (let ([zero13132 '0])
        (let ([one13133 '1])
            (if (= zero13132 n)
                one13133
                (* n (fact (- n one13133)))))))
```

Figure 12: Factorial function after Alphatization

5.2.3 ANF Conversion

Within the ANF framework [47], sub-expressions are restricted to appear exclusively on the right-hand side of a *let-binding*. By administratively binding each sub-expression to a unique identifier, it enforces an explicit order of evaluation. As a result, ANF simplifies the continuation structure, leaving only one type of continuation, namely the *let-continuation*. This simplification not only facilitates later compiler passes but is convenient for constructing interpreters and program analysis.

ANF organizes expressions into two categories: atomic expressions (ae) and complex expressions (ce). Atomic expressions are comprised of lambdas, datums, or variable references. We can evaluate them immediately owing to their inherent properties of guaranteed termination and non-occurrence of side effects. In contrast, all non-atomic expressions are classified as complex expressions within our language structure.

This transformation process is in fact manually CPS encoded, granting flexibility in terms of where the code is inserted. It simplifies expression by lifting sub-expressions out and binding them explicitly with a *let*. This implies that *let* becomes the singular mechanism allowing the execution of an arbitrary amount of work, binding the return value to a variable, and then continuing to perform another arbitrary amount of work. The following example (Figure 13) illustrates this mechanism effectively.

```
(let ([a 5]
      [b 10])
    (let ([c 15]) c))

    →anf

        (let ([a1224 '5])
            (let ([b1225 '10])
                (let ([c1226 '15]) c1226)))
```

Figure 13: Explicit let binding example

After this pass, only *if* and *let* forms contain more than one true sub-expression. *if* form contains an atomic expression in the guard position and two complex expressions in the tail position, while *let* forms extend the stack. Other forms such as *primitive operations* and *function applications*, only contain atomic sub-expressions. Notably, this transformation also ensures that *prim* and *apply-prim* forms are bound within a *let*, given that they do not extend the continuation. Following this pass, the context-free grammar of our language evolves as below (see Figure 14):

Example The simplification of the factorial function shows, how this pass lifts out sub-expressions and explicitly binds them with a *let*. Also notice the guard branch of *if* now contains an atomic expression, See Figure 15.

$$
\begin{aligned}
\langle \text{def} \rangle &::= (\text{define } (\langle \text{var} \rangle\ \langle \text{var} \rangle^*)\ \langle \text{ce} \rangle) \\
\langle \text{ce} \rangle &::= \langle \text{ae} \rangle \\
&\mid (\text{let } (((\langle \text{var} \rangle\ \langle \text{ae} \rangle))\ \langle \text{ce} \rangle) \\
&\mid (\text{let } (((\langle \text{var} \rangle\ \langle \text{ce} \rangle))\ \langle \text{ce} \rangle) \\
&\mid (\text{let } (((\langle \text{var} \rangle\ (\text{prim } \langle \text{op} \rangle\ \langle \text{ae} \rangle^*)))\ \langle \text{ce} \rangle) \\
&\mid (\text{let } (((\langle \text{var} \rangle\ (\text{apply-prim } \langle \text{op} \rangle\ \langle \text{ae} \rangle)))\ \langle \text{ce} \rangle) \\
&\mid (\text{if } \langle \text{ae} \rangle\ \langle \text{ce} \rangle\ \langle \text{ce} \rangle) \\
&\mid (\text{apply } \langle \text{ae} \rangle\ \langle \text{ae} \rangle) \\
&\mid (\langle \text{ae} \rangle\ \langle \text{ae} \rangle^*) \\
\langle \text{ae} \rangle &::= \langle \text{var} \rangle \\
&\mid (\text{quote } \langle \text{datum} \rangle) \\
&\mid (\text{lambda } \langle \text{var} \rangle\ \langle \text{ce} \rangle) \\
&\mid (\text{lambda } (\langle \text{var} \rangle^*)\ \langle \text{ce} \rangle)
\end{aligned}
$$

Figure 14: ANF IR

```
(define (fact n)
    (let ([zero13132 '0])
        (let ([one13133 '1])
            (let ([a13232 (= zero13132 n)])
                (if a13232
                    one13133
                    (let ([a13233 (- n one13133)])
                        (let ([a13234 (fact a13233)])
                            (* n a13234))))))))
```

Figure 15: Factorial function after ANF conversion

5.2.4 CPS Conversion

CPS imposes a constraint on call sites to always be in the tail position, meaning that functions never return in the conventional sense, and instead, a continuation is explicitly passed forward to be invoked on the return point [31]. CPS is a powerful transformation phase for compiler optimization and program analysis, and it is also possible to reconstruct a program back to its direct-style form with no loss of efficiency or analysis results [32, 33].

For example, consider the arithmetic expression `(* (+ 5 10) 20)`. Here, `(+ 5 10)` is evaluated first, but the result is initially unknown. So, imagine the expression with a "placeholder" (`[ ]`) which will be filled by that result, creating `(* [ ] 20)` as the *future* of the computation, or the *continuation* of the expression. Once `(+ 5 10)` is evaluated to `15`, we can invoke the continuation—`(* 15 20)`—by substituting `15` back into the placeholder. Which we can then be evaluated to `300`. At this point, there is no further computation to be done; the continuation is *empty*, and we have our final result. The following Racket implementation demonstrates the above example effectively in Figure 16.

```
;takes (lambda (x) x) as the continuation
(define (multiplication cont sum num)
    (cont (* sum num)))

;takes multiplication as the continuation
(define (addition cont a b)
    (cont (lambda (x) x) (+ a b) 20))

;makes the initial call
(addition multiplication 5 10) ;⟶ 300
```

Figure 16: A CPS example

CPS is a source-to-source translation that implements the stack meaning that for *Brouhaha* this pass implements all the continuations in terms of lambdas, but no lambda ever returns. Rather, every lambda at its return point calls another continuation lambda to handle the return. Notice that CPS further simplifies our grammar by binding all the expressions on the right side of a *let*, except for *if* and *function applications*. *if* form now contains a variable reference or symbol in the guard position and two complex expressions in the tail position, while *function applications* only have symbols as sub-expressions. Thus, following this conversion, the context-free grammar of our languages changes as below (see Figure 17).

$$
\begin{aligned}
\langle \text{ce} \rangle ::= {} & (\text{let } (((\langle \text{var} \rangle\ (\text{quote } \langle \text{datum} \rangle)))\ \langle \text{ce} \rangle) \\
| {} & (\text{let } (((\langle \text{var} \rangle\ (\text{prim } \langle \text{op} \rangle\ \langle \text{var} \rangle^*)))\ \langle \text{ce} \rangle) \\
| {} & (\text{let } (((\langle \text{var} \rangle\ (\text{apply-prim } \langle \text{op} \rangle\ \langle \text{var} \rangle)))\ \langle \text{ce} \rangle) \\
| {} & (\text{let } (((\langle \text{var} \rangle\ (\text{lambda } \langle \text{var} \rangle\ \langle \text{ce} \rangle)))\ \langle \text{ce} \rangle) \\
| {} & (\text{let } (((\langle \text{var} \rangle\ (\text{lambda } (\langle \text{var} \rangle^*)\ \langle \text{ce} \rangle)))\ \langle \text{ce} \rangle) \\
| {} & (\text{if } \langle \text{var} \rangle\ \langle \text{ce} \rangle\ \langle \text{ce} \rangle) \\
| {} & (\text{apply } \langle \text{var} \rangle\ \langle \text{var} \rangle) \\
| {} & (\langle \text{var} \rangle\ \langle \text{var} \rangle^*)
\end{aligned}
$$

Figure 17: CPS IR

Example Note the procedure *kont13350* in Figure 18, is the continuation passed by the external caller of the function *fact*. Similarly, operators $-$ and $*$ are also taking an extra continuation argument which they will invoke at their return point. Hence, the factorial function becomes.

```
(define (fact kont13350 n)
  (let ([zero13132 '0])
    (let ([one13133 '1])
      (let ([f13353
        (lambda (a13232)
          (if a13232
              (kont13350 one13133)
              (let ([f13352
                (lambda
                  (a13233)
                    (let ([f13351
                      (lambda
                        (a13234)
                          (* kont13350 n a13234))])
                      (fact f13351 a13233)))])
                (- f13352 n one13133))))])
        (= f13353 zero13132 n)))))
```

Figure 18: Factorial function after CPS conversion

5.2.5 Closure Conversion

Closure conversion is a program transformation technique that turns functions with free variables into a structure called a closure—which consists of the function code and an environment capturing these variables [34, 35, 128]. This process abstracts the function's code to include an additional parameter for the environment, replacing free variables with references to this environment. The resulting closure stays inactive until the function is actually invoked with arguments [129].

For closure conversion, there are two major approaches: top-down (linked closures) and bottom-up (flat closures). For *Brouhaha* we adopted the latter. Operationally, we compute free variables as we loop through the CPS-converted code. Whenever we find a lambda, we compute its set of free variables, however, if we find any lambda in the body of the lambda itself, then we compute its set of free variables first. After that, we generate the relevant code to allocate the lambda's environment, which effectively removes the free variables with *env-ref*s. So, overall, this pass eliminates all lambda abstractions and substitutes them with *make-closure* and *env-ref* forms. This transformation actually converts the higher-order program, into a list of first-order procedures whereas *untagged-application* and *apply* form become *clo-app* and *clo-apply*, respectively. Hence the context-free grammar of our languages changes as below (see Figure 19).

Example Figure 20 shows the factorial function after the closure conversion phase.

5.2.6 Code Generation

The final pass is often *code generation*, where the program is typically compiled down to `C` or `C++`, and linked with libraries for the target platform [34]. In *Brouhaha*, after closure conversion, the program is composed exclusively of top-level procedures. The code generation phase takes this closure-converted code and translates it into `C++`. In the runtime environment of the generated `C++` code, functions do not directly accept arguments in the conventional sense. Instead, the arguments are placed into a *buffer* by the calling function prior to reaching the call site. The *callee* function then retrieves the argument count from the buffer's initial position and subsequently accesses each argument. The encoding scheme within the `C++` code ensures that every variable is of the *void** type, with the least significant three bits of the pointer tagged to signify the *datatype*

$$
\begin{aligned}
\langle\text{program}\rangle &::= ((\text{proc } (\langle\text{var}\rangle\ \langle\text{var}\rangle^*)\ \langle\text{ce}\rangle)^*) \\
\langle\text{ce}\rangle &::= (\text{let } (((\langle\text{var}\rangle\ (\text{quote } \langle\text{datum}\rangle)))\ \langle\text{ce}\rangle) \\
&\mid (\text{let } (((\langle\text{var}\rangle\ (\text{prim } \langle\text{op}\rangle\ \langle\text{var}\rangle^*)))\ \langle\text{ce}\rangle) \\
&\mid (\text{let } (((\langle\text{var}\rangle\ (\text{apply-prim } \langle\text{op}\rangle\ \langle\text{var}\rangle)))\ \langle\text{ce}\rangle) \\
&\mid (\text{let } (((\langle\text{var}\rangle\ (\text{make-closure } \langle\text{var}\rangle\ \langle\text{var}\rangle^*))\ \langle\text{ce}\rangle) \\
&\mid (\text{let } (((\langle\text{var}\rangle\ (\text{env-ref } \langle\text{var}\rangle\ \langle\text{integer}\rangle))\ \langle\text{ce}\rangle) \\
&\mid (\text{if } \langle\text{var}\rangle\ \langle\text{ce}\rangle\ \langle\text{ce}\rangle) \\
&\mid (\text{clo-apply } \langle\text{var}\rangle\ \langle\text{var}\rangle) \\
&\mid (\text{clo-app } \langle\text{var}\rangle\ \langle\text{var}\rangle^*)
\end{aligned}
$$

Figure 19: Closure Converted IR

```
(proc
   (lam2362 env2363 a2344)
   (let ((kont2353 (env-ref env2363 3)))
     (let ((* (env-ref env2363 2)))
       (let ((n (env-ref env2363 1)))
        (clo-app * kont2353 n a2344)))))
(proc
   (lam2364 env2365 a2343)
   (let ((kont2353 (env-ref env2365 4)))
     (let ((* (env-ref env2365 3)))
       (let ((fact (env-ref env2365 2)))
         (let ((n (env-ref env2365 1)))
           (let ((f2354 (make-closure lam2362 n * kont2353)))
             (clo-app fact f2354 a2343)))))))
(proc
   (lam2366 env2367 a2342)
   (let ((* (env-ref env2367 6)))
     (let ((fact (env-ref env2367 5)))
       (let ((n (env-ref env2367 4)))
         (let ((- (env-ref env2367 3)))
           (let ((one2337 (env-ref env2367 2)))
             (let ((kont2353 (env-ref env2367 1)))
               (if a2342
                 (clo-app kont2353 one2337)
                 (let ((f2355 (make-closure lam2364 n fact * kont2353)))
                   (clo-app - f2355 n one2337)))))))))
(proc
   (fact _2370 kont2353 n)
   (let ((zero2336 '0))
     (let ((one2337 '1))
       (let ((f2356 (make-closure lam2366 kont2353 one2337 - n fact *)))
         (clo-app = f2356 zero2336 n)))))
```

Figure 20: Factorial function after closure conversion

it represents. When accessing a variable, the pointer undergoes *decoding* to confirm its expected type, and only then is the decoded pointer utilized to carry out the necessary operations. Furthermore, to accommodate large integers and floating-point numbers, the program leverages the GNU Multi Precision (GMP) library. The GMP types, MPZ for integers, and MPF for floats are integrated to support arithmetic operations on numbers unrestricted by size and limited only by available memory.

In *Brouhaha*, memory management for the C++ code is managed by Boehm's Garbage Collector, a conservative approach to garbage collection. Memory allocation is performed using the GC_MALLOC function, and the *new(GC)* operator is employed in place of the standard C++ *new* operator to ensure the garbage collector manages the allocated memory. String handling is facilitated by the *std::string* class from the C++ standard library, with language-specific string functions mapped directly to their *std::string* counterparts. Finally, *hash* functions within the language are supported by a functional variant of the Hash Array Mapped Trie (HAMT), providing efficient and effective hash operations.

Example The generated C++ code for the factorial function is given below. Note that, around 4000 lines of built-in *Brouhaha* code is omitted for simplicity.

```
#include <stdio.h>
#include <string.h>
#include "gmp_func.h"
#include "../../prelude.hpp"

//....

void *lam8714_fptr()
{
    numArgs = reinterpret_cast<long>(arg_buffer[0]);
    void *env8715 = arg_buffer[1];
    void *a8441 = arg_buffer[2];
    void *kont8540 = (decode_clo(env8715))[3];
    void *_u42 = (decode_clo(env8715))[2];
    void *n = (decode_clo(env8715))[1];

    arg_buffer[1] = reinterpret_cast<void *>(_u42);
    arg_buffer[2] = kont8540;
    arg_buffer[3] = n;
    arg_buffer[4] = a8441;
    arg_buffer[0] = reinterpret_cast<void *>(4);
    auto function_ptr = reinterpret_cast<void (*)()>((decode_clo(_u42))
        [0]);
    function_ptr();
    return nullptr;
}
void *lam8714 = encode_clo(alloc_clo(lam8714_fptr, 0));
void *lam8716_fptr()
{
    numArgs = reinterpret_cast<long>(arg_buffer[0]);
    void *env8717 = arg_buffer[1];
    void *a8440 = arg_buffer[2];
    void *kont8540 = (decode_clo(env8717))[4];
    void *_u42 = (decode_clo(env8717))[3];
    void *fact = (decode_clo(env8717))[2];
    void *n = (decode_clo(env8717))[1];

    void **clo8856 = alloc_clo(lam8714_fptr, 3);
```

```
    clo8856[1] = n;
    clo8856[2] = _u42;
    clo8856[3] = kont8540;
    void *f8541 = encode_clo(clo8856);

    arg_buffer[1] = reinterpret_cast<void *>(fact);
    arg_buffer[2] = f8541;
    arg_buffer[3] = a8440;
    arg_buffer[0] = reinterpret_cast<void *>(3);
    auto function_ptr = reinterpret_cast<void (*)()>((decode_clo(fact))
       [0]);
    function_ptr();
    return nullptr;
}
void *lam8716 = encode_clo(alloc_clo(lam8716_fptr, 0));
void *lam8718_fptr()
{
    numArgs = reinterpret_cast<long>(arg_buffer[0]);
    void *env8719 = arg_buffer[1];
    void *a8439 = arg_buffer[2];
    void *kont8540 = (decode_clo(env8719))[6];
    void *one = (decode_clo(env8719))[5];
    void *_u42 = (decode_clo(env8719))[4];
    void *fact = (decode_clo(env8719))[3];
    void *n = (decode_clo(env8719))[2];
    void *_u45 = (decode_clo(env8719))[1];

    bool if_guard8857 = is_true(a8439);
    if (if_guard8857)
    {
        arg_buffer[1] = reinterpret_cast<void *>(kont8540);
        arg_buffer[2] = one;
        arg_buffer[0] = reinterpret_cast<void *>(2);
        auto function_ptr = reinterpret_cast<void (*)()>((decode_clo(
           kont8540))[0]);

        function_ptr();
        return nullptr;
    }
    else{
        void **clo8859 = alloc_clo(lam8716_fptr, 4);

        clo8859[1] = n;
        clo8859[2] = fact;
        clo8859[3] = _u42;
        clo8859[4] = kont8540;
        void *f8542 = encode_clo(clo8859);

        arg_buffer[2] = apply_prim__u45_2(n, one);
        arg_buffer[1] = reinterpret_cast<void *>(f8542);
        arg_buffer[0] = reinterpret_cast<void *>(2);
        auto function_ptr = reinterpret_cast<void (*)()>((decode_clo(
           f8542))[0]);
```

```
        function_ptr();
        return nullptr;
    }
}
void *lam8718 = encode_clo(alloc_clo(lam8718_fptr, 0));
void *fact_fptr()
{
    numArgs = reinterpret_cast<long>(arg_buffer[0]);
    void *_8722 = arg_buffer[1];
    void *kont8540 = arg_buffer[2];
    void *n = arg_buffer[3];
    mpz_t *mpzvar8860 = (mpz_t *)(GC_MALLOC(sizeof(mpz_t)));
    mpz_init_set_str(*mpzvar8860, "0", 10);
    void *zero = encode_mpz(mpzvar8860);
    mpz_t *mpzvar8861 = (mpz_t *)(GC_MALLOC(sizeof(mpz_t)));
    mpz_init_set_str(*mpzvar8861, "1", 10);
    void *one = encode_mpz(mpzvar8861);

    void **clo8863 = alloc_clo(lam8718_fptr, 6);

    clo8863[1] = _u45;
    clo8863[2] = n;
    void *fact = encode_clo(alloc_clo(fact_fptr, 0));

    clo8863[3] = fact;
    clo8863[4] = _u42;
    clo8863[5] = one;
    clo8863[6] = kont8540;
    void *f8543 = encode_clo(clo8863);

    arg_buffer[1] = reinterpret_cast<void *>(_u61);
    arg_buffer[2] = f8543;
    arg_buffer[3] = zero;
    arg_buffer[4] = n;
    arg_buffer[0] = reinterpret_cast<void *>(4);
    auto function_ptr = reinterpret_cast<void (*)()>((decode_clo(_u61))
        [0]);
    function_ptr();
    return nullptr;
}
void *fact = encode_clo(alloc_clo(fact_fptr, 0));

//...
```

6 Abstract Interpretation

Abstract interpretation is a static analysis technique that approximates program behavior over abstract domains. This technique has been extensively used in functional programming analyses and serves applications such as proving program properties, bug detection, compiler optimization, and so on [36, 37, 38, 39]. One approach to implementing this technique is to use abstract machines and interpreters, such as CE, CEK, CESK, and CESK* [39, 40, 41, 42, 43, 44]. These simplified models of computation and interpreters provide the foundation for analyzing programs and a basis for further optimizations.

In this section, we first cover why we need abstract interpretation instead of concrete interpretation (Section 6.1), and its challenge and limitation (Section 6.3). Then in Section 6.4 we talk about abstracting abstract machines. Finally, in Section 6.5 and Section 6.6 environment analysis and abstract counting are discussed.

6.1 Limitation of Concrete Interpretation

One may wonder why we even need abstract interpretation over concrete interpretation in program analysis. The answer lies in the inherent limitations of the latter when faced with intricate programs that may include loops, conditional branches, and function calls. Attempting to trace every possible program behavior in such complex scenarios becomes an impractical endeavor.

```
(define (function num)
            (function (+ num 1)))
```

Consider this simple recursive function above, invoking it with *(function -2)* initiates an infinite sequence: *-2, -1, 0, 1, 2, 3...*∞. This function continuously increments its input without an endpoint because it lacks a terminating condition—a base case. The problem here is, that while we aim for sound reasoning about our programs, the sheer complexity and infinite paths, as seen in this example, make concrete reasoning nearly impossible, even for a small piece of code like this. Therefore, to analyze our programs thoroughly and reliably, we must transcend beyond the concrete to embrace abstraction. Abstract interpretation offers us a way to generalize program behavior without getting tangled in the details of every conceivable path, especially those that stretch into infinity.

6.2 Abstract Interpretation and Galois Connection

In the study of abstract interpretation, the Galois Connection stands as a fundamental concept that sheds light on the relationship between concrete and abstract domains. This relationship is characterized by two mathematical structures known as complete lattices (is a poset such that $\langle C; \sqsubseteq \bot \top \sqcup \sqcap \rangle$). The connection comprises two crucial functions: the abstraction function (α) and the concretization function (γ). The abstraction function maps a detailed concrete state to its abstract representation, distilling complex systems into essential generalizations. Conversely, the concretization function maps an abstract representation back to its detailed concrete counterpart, enabling the examination of specific details as needed. The Galois Connection's significance lies in its ability to simplify the analysis of complex systems without compromising the integrity of the representation. It offers a tractable approach to understanding systems by abstracting away the specifics while maintaining a sound and precise correlation between the abstract and the concrete domain. This dual functionality ensures that abstract interpretations are not just simplifications but are closely tied to the concrete elements they represent.

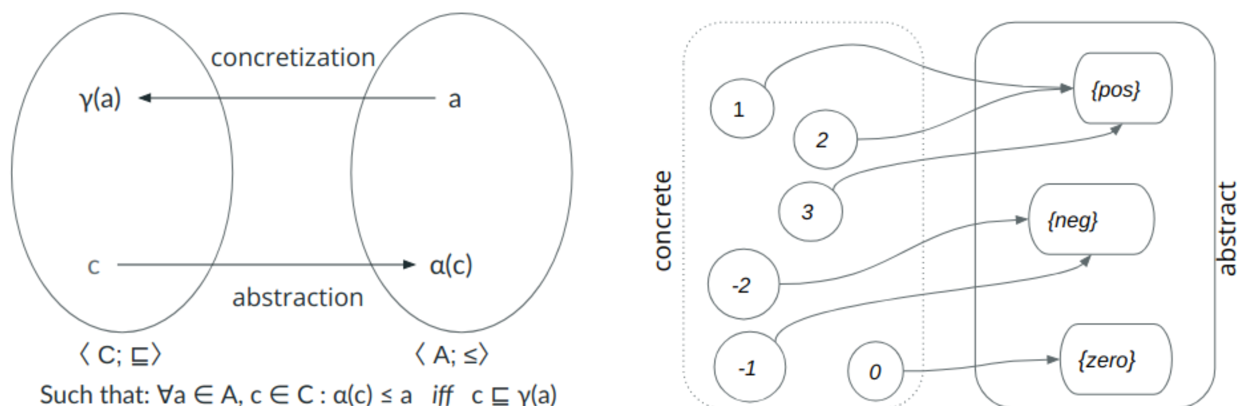


Figure 21: Galois Connection in Abstract Interpretation

One of the core challenges of computer science is the problem of incompatibility. There are certain problems

and program behaviors that we simply cannot determine with certainty, like, no matter how powerful our computers are. Abstract interpretation provides us with a way to navigate around this challenge. It does so by approximating the meaning of programs rather than precisely calculating the exact answers. Operationally, abstract interpretation works with abstract values and each abstract element represents a set of concrete elements. For instance, rather than tracking every individual integer that a variable might take, the analysis may group them into categories such as positive, negative, or zero (as shown in Figure 21). These categories, or abstract elements, effectively represent an infinite set of possible concrete values, thereby making the analysis of complex program behaviors manageable. In essence, the Galois Connection is the backbone of abstract interpretation, ensuring that our abstractions are both meaningful and reliable.

6.3 Challenges and Limitations

While abstract interpretation is a powerful technique, it does come with its own set of challenges and limitations. The choice of an abstract domain is crucial because if we choose a domain that is too coarse, we might lose important details about the program's behavior. On the other hand, if our domain is too fine-grained, we might end up with an analysis that is too complex or time-consuming. During an analysis performed, it is typically the case that an infinite, concrete space is compressed into some finite, abstract space and it is inevitable, then, that some elements of the abstract domain represent multiple elements of the concrete space (as shown in Figure 21). It is this overlapping in the abstract domain that leads to imprecision in reasoning. For example, in Figure 21, we are representing all the positive integers with a single abstract value *pos*, so, we are losing the distinction between numbers 1, 2, and 3. So, striking the right balance between precision and efficiency is one of the fundamental trade-offs of abstract interpretation. If we aim for high precision, the analysis might become too slow or computationally intensive to be practical. Conversely, if we prioritize efficiency by using very coarse abstractions, we might overlook important details. Hence, finding the sweet spot often requires expertise and iterative refinement.

6.4 Abstracting Abstract Machines (AAM)

Over the years abstract machines and interpreters such as CE, CEK, and CSEK [39, 40, 41, 42, 43] have been extensively studied. They are first-order state transition system that represents the core of a real language implementation [44] and are used to evaluate programs and determine their behavior. They follow a set of rules to transition between different states based on the expressions and evaluation contexts in the program. On the other hand, the goal of AAM is to craft a methodology that directly facilitates abstract interpretations of abstract machines, through a process that converts an existing machine description into a variant that computes a finite approximation of its behavior.

6.4.1 CEK Machine

The CEK [42] machine provides an operational model for evaluating lambda calculus expressions. It represents the state of computation as a triple (as shown below, taken from [44]) consisting of a *control string* (the expression being evaluated), an *environment* mapping variables to closures, and a *continuation* that captures the rest of the computation or what to do next. The machine transitions by evaluating the expression in the control string according to the standard λ-calculus reduction rules, using the environment to look up variable values and the continuation to push contexts as needed.

$$\begin{aligned}
\varsigma \in \Sigma \quad &= \mathit{Exp} \times \mathit{Env} \times \mathit{Kont} \\
v \in \mathit{Val} \quad &::= (\lambda x \,.\, e) \\
\rho \in \mathit{Env} \quad &= \mathit{Var} \to_{\text{fin}} \mathit{Val} \times \mathit{Env} \\
\kappa \in \mathit{Kont} &::= \mathbf{mt} \mid \mathbf{ar}(e, \rho, \kappa) \mid \mathbf{fn}(v, \rho, \kappa).
\end{aligned}$$

For example, looking up a variable returns its value from the environment, applying a function substitutes the argument for the parameter in the function's body after creating a closure for the function value and environment, and pushing an argument or function onto the continuation models building up the context. The *ar* and *fn* frames are part of the continuation states which represent the evaluation context for expressions.

The *ar(e ρ' κ)* is a continuation state that occurs during the argument evaluation phase. When an expression is being evaluated and it reaches a point where the argument of a function needs to be evaluated, *ar* captures the state of the machine at that point—this includes the expression for the argument e, the current environment *ρ'*, and the continuation κ to proceed after the argument is evaluated. Similarly, *fn(v ρ κ)* is a continuation state during function application. When a function is ready to be applied, *fn* captures the state of the machine where v is the value that is the result of evaluating the function's body, ρ is the environment that should be used for the evaluation, and κ represents the continuation for the rest of the computation after the function application. Whereas *mt* means the evaluation context is *empty.*

The initial state for evaluating a closed expression simply includes the expression itself, an empty environment, and an empty continuation. The meaning of a program is defined as the set of all reachable machine states starting from the initial configuration. By modeling substitution and evaluation contexts, the CEK machine provides an operational interpretation of lambda calculus evaluation.

$\varsigma \longmapsto_{CEK} \varsigma'$	
$\langle x, \rho, \kappa\rangle$	$\langle v, \rho', \kappa\rangle$ where $\rho(x) = (v, \rho')$
$\langle (e_0 e_1), \rho, \kappa\rangle$	$\langle e_0, \rho, \mathbf{ar}(e_1, \rho, \kappa)\rangle$
$\langle v, \rho, \mathbf{ar}(e, \rho', \kappa)\rangle$	$\langle e, \rho', \mathbf{fn}(v, \rho, \kappa)\rangle$
$\langle v, \rho, \mathbf{fn}((\lambda x . e), \rho', \kappa)\rangle$	$\langle e, \rho'[x \mapsto (v, \rho)], \kappa\rangle$

Figure 22: The CEK machine [44].

In semantic-based program analysis, the intensional properties of the machine are of interest, which refers to the set of all possible states the machine could reach during the evaluation of a program. However, due to the halting problem, it is not feasible to determine all reachable states for every program. AAM addresses this challenge by constructing an approximation of the CEK machine. This approximation is achieved by defining an abstract state transition relation and an abstraction map, which aims to provide a computable and sound method to infer the program's behavior without executing it in its entirety. However, this method encounters a problem because environments and continuations in the CEK machine are recursive, leading to an abstract state space that is potentially infinite. To manage such an infinite state space, a widening operator is required, but finding a suitable widening operator for this scenario is challenging. One way to tackle the recursive structures involves introducing a level of indirection—using explicitly allocated addresses to manage recursion within the machine's state space. This strategy effectively decouples the program's recursion from the state-space recursion and the CESK machine is one step toward addressing this issue which removes recursion from the environment component of the CEK machine.

6.4.2 CESK Machine

The CESK [43] machine is an extension of the CEK machine. Similar to the CEK machine, it represents the state (as shown below, taken from [44]) using a control string, environment, and continuation. Additionally, it introduces a store component that holds variable bindings, eliminating the mutual recursion between environments and closures. The environment maps variables to addresses rather than directly to values, and the store then maps these addresses to storable values. This modification allows the machine to look up a variable's value through its address, providing a more flexible and dynamic handling of variable bindings.

$$
\begin{aligned}
\varsigma \in \Sigma &= \mathit{Exp} \times \mathit{Env} \times \mathit{Store} \times \mathit{Kont} \\
\rho \in \mathit{Env} &= \mathit{Var} \rightarrow_{\text{fin}} \mathit{Addr} \\
\sigma \in \mathit{Store} &= \mathit{Addr} \rightarrow_{\text{fin}} \mathit{Storable} \\
s \in \mathit{Storable} &= \mathit{Val} \times \mathit{Env} \\
a, b, c \in \mathit{Addr} & \quad \text{an infinite set.}
\end{aligned}
$$

The initial state for evaluating a closed expression simply includes the expression itself, an empty environment, an empty store, and an empty continuation. The machine evaluates the control string using the environment

and continuation as in the CEK machine. However, instead of mapping variables directly to values, the environment maps variables to addresses in the store. Looking up a variable fetches the closure (value, environment pair) from the store location corresponding to the variable's address. When binding a variable, it allocates a fresh address and stores the closure there.

$\varsigma \longmapsto_{CESK} \varsigma'$	
$\langle x, \rho, \sigma, \kappa \rangle$	$\langle v, \rho', \sigma, \kappa \rangle$ where $\sigma(\rho(x)) = (v, \rho')$
$\langle (e_0 e_1), \rho, \sigma, \kappa \rangle$	$\langle e_0, \rho, \sigma, \mathbf{ar}(e_1, \rho, \kappa) \rangle$
$\langle v, \rho, \sigma, \mathbf{ar}(e, \rho', \kappa) \rangle$	$\langle e, \rho', \sigma, \mathbf{fn}(v, \rho, \kappa) \rangle$
$\langle v, \rho, \sigma, \mathbf{fn}((\lambda x . e), \rho', \kappa) \rangle$	$\langle e, \rho'[x \mapsto a], \sigma[a \mapsto (v, \rho)], \kappa \rangle$ where $a \notin \mathtt{dom}(\sigma)$

Figure 23: The CESK machine [44].

The key advantage of the CESK machine is that environments and closures are no longer mutually recursive. However, continuations remain recursively structured. Simply abstracting continuations as unordered sets would compromise return-flow analysis. AAM addresses this issue by turning the CESK machine into a CESK* machine, which redirects the recursive structure through the store, similar to environments.

6.4.3 CESK* Machine

The CESK* [44] machine is a variant of the CESK machine that uses store-allocated continuations, a pivotal tool in the static analysis of higher-order languages derived from λ-calculus. It eliminates the remaining recursive structure in the CESK machine associated with continuations. The adaptability of Abstract CESK* lies in its capacity to let analysts adjust its precision and polyvariance by altering the *tick* and *alloc* functions integral to the analysis [130]. In the CESK* machine, continuations are no longer represented directly. Instead, the state (as shown below, taken from [44]) contains a pointer to the current continuation allocated in the store. The store component maps addresses to storable values consisting of closures or continuations.

$$\begin{array}{lcl} \varsigma \in \Sigma & = & Exp \times Env \times Store \times Addr \\ s \in Storable & = & Val \times Env + Kont \\ \kappa \in Kont & ::= & \mathbf{mt} \mid \mathbf{ar}(e, \rho, a) \mid \mathbf{fn}(v, \rho, a). \end{array}$$

Looking up a variable or applying a function proceeds similarly to the CESK machine, but now indirectly through the store when accessing or updating continuations. For example, applying a function allocates a *fn* continuation in the store and updates the pointer. This refactoring serves the same purpose as the CESK store—providing indirection to eliminate recursion. In this case, the mutual recursion between continuations and evaluation contexts is removed. With both environments and continuations allocated in the store, the CESK* machine enables abstract interpretation by bounding just the store. Approximating the potentially unbounded store contents yields a finite-state abstract machine that soundly approximates the original CESK semantics. The initial machine configuration pairs the expression with an *empty* environment, a *store* containing just the *mt* continuation, and a *pointer* to that continuation.

6.5 Environment Analysis

Environment analysis [46, 131] is based on k-CFA and a response to K-CFA's limitation we highlighted in Section 4.3.2. It allows a compiler to reason about the equivalence of environments, i.e., name-to-value mappings, that arise during a program's execution. Whereas, a binding—the atomic unit of the environment—is an individual mapping from one name to one value. Focusing on bindings, the key aspect of environment analysis is discerning when the equivalence of two abstract bindings implies the equivalence of the corresponding concrete bindings. Consider the below example again, if we invoke this with *(function 0)*, then the longer this code runs, the more environment it creates (e.g., $[x \to 0], [x \to 1] \ldots [x \to 19279] \ldots \infty$).

$\varsigma \longmapsto_{CESK_t^\star} \varsigma'$, where $\kappa = \sigma(a), b = alloc(\varsigma), u = tick(\varsigma)$	
$\langle x, \rho, \sigma, a, t \rangle$	$\langle v, \rho', \sigma, a, u \rangle$ where $(v, \rho') = \sigma(\rho(x))$
$\langle (e_0 e_1), \rho, \sigma, a, t \rangle$	$\langle e_0, \rho, \sigma[b \mapsto \mathbf{ar}(e_1, \rho, a)], b, u \rangle$
$\langle v, \rho, \sigma, a, t \rangle$	
if $\kappa = \mathbf{ar}(e, \rho, c)$	$\langle e, \rho, \sigma[b \mapsto \mathbf{fn}(v, \rho, c)], b, u \rangle$
if $\kappa = \mathbf{fn}((\lambda x . e), \rho', c)$	$\langle e, \rho'[x \mapsto b], \sigma[b \mapsto (v, \rho)], c, u \rangle$

Figure 24: The time-stamped CESK⋆ machine [44].

```
(define (function num)
                (function (+ num 1)))
```

Environment analysis tries to evaluate the correlations between these different environments and cautiously determines which bindings in any pair of environments are guaranteed to be identical. Operationally, given a pair of environments ρ_1 and ρ_2 that assign values to variables, the aim of environment analysis is to estimate the collection of variables for which ρ_1 and ρ_2 have matching assignments.

$$\{v : \rho_1(v) = \rho_2(v)\}$$

The technique environment analysis uses to prove this property is called *Abstract Counting*, which we briefly cover next.

6.6 Abstract Counting

Abstract counting [46] is a mechanism for performing environment analysis through abstract interpretation techniques such as k-CFA. While performing the analysis it tracks the allocation frequency of an abstract resource. When the count stands at one, it indicates that, for the time being, the abstract resource correlates with a single concrete resource. This correlation facilitates the process of *environment analysis* and broadens the scope of optimizations that the compiler can apply, not just in quantity but in variety as well. The main objective is to identify singleton sets based on the analogy that if two such sets, A and B, are equivalent and each contains only one element, then they are identical in both the abstract and concrete domains. An analysis builds a map (Galois Connection) between abstract and concrete addresses and abstract counting is an approximation of this map. The process of analysis, therefore, is the modeling of program properties, and abstract counting serves as a model for the properties of the analysis itself. It encapsulates an abstraction of the number of concrete values that an abstract value might represent. Operationally, the allocation scheme for abstract addresses plays a pivotal role in this process. When multiple concrete addresses are mapped to a singular abstract address, the count corresponding to that abstract address is incremented. A newly allocated abstract address is initially treated as though it were concrete, assuming a one-to-one correspondence until it is reallocated.

7 Implementation Approaches For Reasoning Systems

Reasoning systems are designed to simulate the human ability to reason or make inferences based on given information or assumptions. By processing and analyzing a set of rules and data, these systems draw logical conclusions, a capability crucial in domains where complex and rule-intensive information is prevalent. They mechanize different forms of reasoning, particularly deductive reasoning as found in mathematics and formal logic, including propositional and predicate logic [132]. This functionality allows reasoning systems to handle tasks ranging from simple classification to intricate problem-solving. The primary function of these systems is to uncover new insights from a knowledge base (KB), determining what additional knowledge logically follows from existing information and thus enhancing decision-making capabilities through logical inference [133].

Implementation approaches vary depending on the type of logic, knowledge representation, inference method, and search strategy used. Some of the common and prominent approaches are Rule-based reasoning, Logic-based reasoning, Probabilistic reasoning, and Non-monotonic reasoning. Each approach has advantages and disadvantages, and the best approach depends on the characteristics and requirements of the domain and the problem to be solved.

In this section, we will briefly review some of the key approaches in reasoning systems, such as Interprocedural Program Analysis (IPA), Satisfiability Problem (SAT), and Datalog. Notably, our examination will predominantly focus on Datalog, as it offers pertinent insights and methodologies that are particularly relevant to the scope of our survey and research directions.

7.1 Interprocedural Program Analysis (IPA)

The interprocedural analysis focuses on evaluating a program encompassing multiple procedures, with a particular emphasis on understanding how information is exchanged and flows between these procedures [134]. IPA can be performed using various program analysis techniques, such as data-flow analysis, control-flow analysis, abstract interpretation, may-alias analysis, etc [135, 136, 137, 138, 139, 140, 141]. A compiler then uses this analysis result to understand the behavior of all procedures in a program and applies this knowledge to optimize individual procedures [142]. On the other hand, intraprocedural analysis is a mechanism that targets optimization within each function of a compilation unit, relying solely on the information available for that specific function and compilation unit [143]. At compile time, optimization is often done by employing a dual approach, incorporating both intraprocedural analysis and interprocedural analysis. While intraprocedural analysis focuses on optimizing individual functions within their scope, interprocedural analysis broadens this scope, performing optimizations across the entire program. This comprehensive approach leverages techniques including but not limited to inlining, program partitioning, dead-code elimination, and constant propagation [143, 144]. These optimizations are geared towards enhancing performance by reducing overhead, improving data locality, and streamlining the program's flow.

Despite the advantages, performing IPA is a complex task as it requires the compiler to understand not only the code it is currently compiling but also the effects of every procedure throughout the entire program, including those compiled at different times [145]. Not all applications equally benefit from IPA optimizations, and the extent of performance gains varies [143]. Applications with numerous functions, multiple compilation units, and those where functions are not in the same compilation unit as their callers are more likely to see improvements from IPA. Additionally, applications with fewer input and output operations generally exhibit more noticeable performance enhancements. However, the degree of performance improvement is contingent on the application type, and in some instances, the use of interprocedural analysis might even lead to performance degradation [143]. Therefore, debugging and thoroughly assessing programs before applying IPA is crucial to ensure that the optimizations align well with the program's requirements and characteristics.

7.2 SAT

Satisfiability, commonly known as SAT, is a principle in logic that deals with determining whether a logical formula or a set of formulas, can be deemed satisfiable. In simpler terms, it means checking if there is a way to assign truth values to the variables in these formulas so that the entire formula holds true [146]. Algorithms or tools designed to perform this task are known as SAT solvers, and they play a pivotal role in figuring out the satisfiability status of given logical statements. SAT solvers have found applications in various fields including computer science, computer engineering, graph theory, logic, and operations research [146]. Their utility shines in addressing complex issues that can be framed as SAT problems. SAT solvers offer a universal framework for problem-solving, converting various problem types into SAT problems. This adaptability is strengthened by the extensive mathematical tools and techniques developed for logic and proof theory. The study of SAT algorithms has been an active area of research for over six decades and some of the prominent algorithms are DPLL, MiniSat, Max-SAT, Chaff, CryptoMiniSAT, and ProbSAT, GRASP [147, 148, 149, 150, 151, 152, 153].

Nonetheless, SAT solvers are not without their challenges. A notable drawback is the exponential worst-case complexity associated with SAT problems making some SAT problem instances practically unsolvable with

current computational means. Moreover, SAT solvers may falter when dealing with problems that involve real-world constraints or intricate dependencies. Their effectiveness is also contingent on the accuracy and completeness of how a problem is represented logically, which can be a demanding task in some scenarios. Thus, while SAT solvers are formidable tools capable of addressing a wide array of problems, their efficiency is inherently dependent on the specific nature and complexity of the problem at hand.

7.3 Datalog

Datalog is a logic programming-based database query language [154]. A Datalog program combines a set of specific predicates, known as the extensional database (EDB), with a series of Horn clauses, forming the intensional database (IDB) [155]. In the context of program analysis, Datalog simplifies the implementation of analysis techniques in an intuitive way [156, 157]. Analysis techniques, including data-flow, control-flow, and pointer analysis, inherently involve recursion, making Datalog, with its strong recursive capabilities, an ideal choice for performing such tasks[158]. For instance, data-flow analysis can be expressed in a few lines of code written in Datalog, but performing the same analysis can take hundreds to thousands of lines in a traditional language [156].

Consider a datalog example (Figure 25), where we define a directed graph containing nodes $n1$ to $n5$, then we will *query* to find out all the nodes that are reachable from a particular node. The graph is constructed using edge/2 facts—edge is the name of the predicate and 2 indicates that the predicate takes two arguments—representing directed edges between nodes ($n1$ to $n5$). The graph also includes a loop from $n4$ back to $n1$. The reachable/2 predicate is defined with two rules: The first states that a node Y is reachable from another node X if there is a direct edge from X to Y. The second, recursive rule states that Y is reachable from X if there exists an intermediate node Z such that Z is reachable from X, and there is a direct edge from Z to Y. Now if we query which nodes are reachable from $n1$, the Datalog engine uses these rules to recursively explore the graph, concluding that $n2, n3$, and $n4$ are reachable from n1, while n5 remains unreachable because there is no edge leading to $n5$ from any of $n1, n2, n3$, or $n4$.

% Facts $edge(n1, n2)$. $edge(n2, n3)$. $edge(n3, n4)$. $edge(n4, n1)$. $edge(n5, n4)$.	% Rules $reachable(X, Y)$:- $edge(X, Y)$. $reachable(X, Y)$:- $edge(X, Z), reachable(Z, Y)$. % Query ?- $reachable(n1, Y)$.

Figure 25: A datalog example: EDB (left), IDB (right), and Query (right-bottom)

In the 1980s and early 1990s, Datalog attained considerable attention within the database community, primarily due to its potential in various systems applications. Despite this initial interest, the language eventually entered a period of inactivity, partly because it did not find immediate, widespread practical use [159]. However, Datalog has recently experienced a revival, now playing a critical role in numerous contemporary application areas. This resurgence is especially noticeable in fields such as data integration, networking, security, cloud computing, and program analysis [45, 160, 161, 162, 163, 164]. A key factor in this renewed interest is Datalog's ability to serve as a high-level language for efficiently querying both graphs and relational structures. Its capabilities in executing recursive queries and maintaining views incrementally are noteworthy, as they leverage the relational model's strength for structured formal reasoning and analysis. Datalog's adaptability enables customization to meet diverse application-specific needs. Its core is often enhanced and modified across various fields, reflecting its evolving role and growing importance in today's computational landscape. This versatility cements Datalog's status as a crucial tool in data processing and analysis [158].

The remaining part of this section aims to delve into the contemporary landscape of Datalog and its modern

adaptations, particularly focusing on Soufflé [165], Slog [45], Flix [166], Datafun [167], IncA [168], and Ascent [169]. It also aims to provide a holistic view of the capabilities and limitations of each system, offering insights into their suitability for various computational tasks and environments. Finally, Section 7.3.7 shows how our prototype compiler is getting informed by the slog-based analysis in order to perform various optimizations.

7.3.1 Soufflé

Soufflé [165] is a state-of-the-art open-source Datalog system focused on scalability and used primarily for program analysis tasks. It achieves performance through the compilation of Datalog to parallel `C++` code [170]. It utilizes domain-specific optimizations including a novel polynomial-time algorithm inspired by Dilworth's theorem to construct an optimal minimal set of B-tree indexes that cover all necessary access patterns. It further optimizes the compiled program through template metaprogramming techniques that specialize data structures and algorithms at compile-time. To handle concurrent operations, Soufflé implements specialized lock-free concurrent data structures rather than relying on generic libraries. A key data structure is a variant of concurrent B-trees employing fine-grained optimistic read/write locking to separate read and write paths during semi-naïve evaluation. The paper [165] provides an overview of Soufflé's architecture and staged compilation process translating Datalog to a relational algebra machine, then to specialized `C++`. However, it does not detail the specifics of Soufflé's semi-naïve evaluation implementation. A case study shows Soufflé scaling to analyze the large OpenJDK codebase orders of magnitude faster than existing Datalog engines, demonstrating its viability for large-scale program analysis. One of Soufflé's strengths is the configurability and ability to rapidly prototype custom program analyzers that approach hand-optimized tools' performance levels. Although Soufflé's performance is impressive with a low thread count, it faces challenges in scaling efficiently due to internal locking mechanisms and its reliance on coarse-grained parallelism [45, 171].

7.3.2 Flix

Flix [166] is a declarative language for specifying and solving least fixed point problems, particularly static program analyses. It takes inspiration from Datalog and extends it with support for lattices and monotone functions. In Flix, users can define lattices like constant propagation, intervals, etc to represent abstract domains. They can also specify monotone functions over these lattices to implement transfer functions. This allows Flix to express a broader range of static analyses than pure Datalog while retaining Datalog's familiar rule-based syntax. The semantics of Flix builds on Datalog by associating predicates with lattices, extending the Herbrand universe with lattice elements, and defining lattice operations like joins. This provides a clean model-theoretic semantics that captures the meaning of Flix programs. Flix ensures programs have a unique minimal model that can be computed using semi-naïve evaluation. The key capabilities of Flix include (a) support for user-defined lattices and monotone functions, (b) integration of a pure functional language to specify lattices and functions, and (c) compatibility of analyses like in Datalog.

Compared to other Datalog systems, Flix allows a wider range of static analyses to be expressed and implemented effectively. The use of lattices avoids the need to encode them in relations which can be inefficient. Flix programs also interoperate well with existing Java code, unlike many Datalog solvers. Recently [172] Flix added the support for embedding logic programs as first-class values in a functional programming language. However, Flix does have certain limitations. It does not provide support for negation and non-monotonic reasoning. Additionally, it lacks a strong emphasis on efficient compilation [45], and it does not allow expressions in argument positions or enable pattern matching within rules [169].

7.3.3 Datafun

Datafun [167] is a typed functional programming language that allows programming in a style similar to Datalog, while also supporting higher-order functions. It tracks monotonicity in its type system, which allows taking the least fixed points of monotone functions in a safe and terminating manner. Datafun employs a top-down evaluation strategy rooted in the λ-calculus and its two major capabilities beyond standard Datalog are Higher-order functions and Expressivity. Datafun allows defining higher-order monotone and non-monotone functions, and using these functions in fixed point computations. This makes it easy to abstract common patterns like generic transitive closure. Datalog is limited to first-order predicate definitions while

Datafun can define arbitrary monotone functions on posets and semilattices. Datalog is limited to predicates on finite sets of ground terms. Datafun's approach lets programs compute with the data they operate on. The key idea in Datafun is to track monotonicity information in the type system. By distinguishing monotone and non-monotone functions and requiring fixed point bodies to be monotone, the language guarantees termination. The denotational semantics models types as posets and uses adjunctions between posets, sets, and semilattices. Recent research has explored its semi-naïve evaluation in connection with the incremental λ-calculus [173, 174].

While Datafun advances Datalog capabilities, there are still limitations. It does not yet have optimization techniques like the *magic sets* algorithm that make Datalog implementations efficient. In future implementations, the authors also aim to add support for the general recursion to the language.

7.3.4 IncA

IncA [168] is a domain-specific language (DSL) language. It allows developers to define incremental program analyses that efficiently update results as code changes. It works by representing computations as graph patterns on the abstract syntax tree (AST) of the analyzed program. Its compiler translates user-defined analyses into interconnected graph patterns and turns regular pattern functions into graph patterns. Whereas the runtime system then incrementally maintains the analysis results using incremental graph pattern-matching algorithms when code changes occur.

Compared to directly using graph patterns, IncA aims to make incremental program analysis more accessible by providing pattern functions as an abstraction. Pattern functions take a single input, operate in a linear fashion similar to forward or backward analyses, and hide the complexities of sets and graph operations. To optimize performance, the IncA compiler analyzes the pattern functions to determine which AST nodes are relevant for the analysis. This allows pruning irrelevant change notifications and reducing caching during incremental reevaluation. The authors demonstrate the capabilities of IncA by implementing control flow analysis, points-to analysis, well-formedness checks for C programs, and FindBugs checks for Java. Measurements show that incremental analyses provide significant speedups over reanalyzing from scratch after changes. Contrary to the general incremental computation systems like i3QL or Adapton, IncA focuses specifically on incremental program analysis rather than arbitrary computations. The domain-specific assumptions allow additional optimizations like static analysis of the pattern functions. In the future, the authors plan to extend IncA to support additional kinds of program analyses by generating incremental runtime data representations.

7.3.5 Ascent

Ascent [169] is a logic programming language that extends Datalog, embedded in Rust via procedural macros. It allows writing declarative rules that can perform deductive inference and compute fixed points over lattices. Ascent rules can seamlessly call Rust functions and vice versa, enabling integration of logic programming with application code. The key capabilities of Ascent include support for user-defined types and pattern matching, allowing logic programs to directly operate over complex data. It utilizes Rust's trait system to enable fixed point computations over non-powerset lattices, such as computing shortest paths in graphs. Common aggregators including but not limited to min, max, sum, etc. are provided in the library and can be user-extended. Ascent performs optimal index selection and semi-naïve evaluation for efficient deduction. Ultimately the rules compile to high-performance Rust code.

Compared to Datalog systems like Souffle and Flix, Ascent aims for tighter integration with a host language to avoid serialization costs. It matches Souffle's performance on benchmarks while requiring far less code than Datafrog [175] (A lightweight Datalog engine in Rust). By compiling to native code, Ascent achieves orders of magnitude speedups over Flix. The authors evaluate Ascent by reimplementing Polonius, the Rust borrow checker, in half the lines of code. Ascent matches its performance, demonstrating viability for large real-world analyses. Experiments on shortest path computation and graph mining show Ascent is highly performant compared to Flix and competitive with Soufflé. However, Ascent lacks capabilities for parallel evaluation, proving termination by tracking monotonicity and combining programs functionally like later Flix versions.

7.3.6 Slog

Slog [45] is a deductive logic programming language that extends Datalog to support structured, recursive facts and higher-order relations. Slog's key innovation is allowing subfacts—facts nested within other facts—to be first-class citizens just like top-level facts. This means subfacts can trigger rules, be recursively nested, and be referenced as values by other facts. To implement this, Slog interns every structurally unique fact and subfact, assigning it a unique ID. Rules can then match on subfacts and generate new (sub)facts in response. This simple extension enables Slog to naturally represent structured data like abstract syntax trees as nested facts. It also allows higher-order relations through defunctionalization. For example, a rule can look up a value for variable x in a global environment relation, providing the enclosing environment fact. Slog compiles to parallel relational algebra, distributing computation across facts and subfacts uniformly.

Compared to other datalog-like systems such as Soufflé and RadLog, Slog demonstrates better scaling and performance in benchmarks. By treating subfacts as first-class, Slog avoids representing trees of facts as flat relations like Souffle does with abstract data types. This allows direct access and indexing of structured values instead of materializing large unwieldy intermediate relations. The MPI-based parallel backend also shows better scaling than Spark-based systems like RadLog [176]. While inspired by Datalog, Slog is a fully featured language for data-parallel structured deduction. Slog's implementation contains a compiler, runtime, and REPL totaling over 20K lines of code and its application includes but is not limited to more direct implementations of abstract machines (CEK, Krivine's, CESK), rich program analyses (k-CFA, m-CFA), and type systems.

7.3.7 Slog-Based Analysis

The *Brouhaha* is a whole-program compiler, and its performance significantly improves by the analysis that we perform using Slog. Slog is a deductive logic programming language and it allows us to write abstract machines in a very intuitive way. For *Brouhaha* we adopted the CESK* [44] abstract machine to perform whole-program analysis and the goal of this analysis is to provide us with sound and useful information that can inform the compilation process.

We sent the output of Alphatizations pass to *Slog*, where we turn the s-expressions into recognizable facts within the Slog framework. This conversion is achieved by uniquely tagging each Brouhaha form to correspond to a specific *fact* in Slog. The code that facilitates this conversion process is handled in the file named *emit-slog.rkt*. To generate the control-flow graph, the rules are structured within an *evaluate*, *return*, and *apply* framework to bring clarity and order to the CFA's architecture. During the analysis, any control expression initially enters an evaluate phase, marked by an *eval* tag, along with its associated environment and continuation. This phase is designed to yield a value, which is then stored along with its address for subsequent processing in the return phase, indicated by a *ret* tag.

The return state resumes the evaluation based on the directions specified by the continuation, which is typically modified during the *evaluate* phase. When a function or lambda application is directed to a return state, the process transitions to the apply state. Apply state handles both variadic and fixed parameter scenarios, ensuring that parameters and arguments are bound correctly. Finally, the results of the analysis are encapsulated in *answer* tagged facts, which include the addresses and corresponding values of the outputs.

Upon finishing the analysis, the compiler writes all the generated facts into a text file named *fact.txt* and continues the compilation process. Finally, in the code generation phase, whenever we are at a call site, the compiler introspects on the facts from the *fact.txt* file, to perform various kinds of optimization. For example, one of the optimizations that the compiler does is, before emitting `C++`, it looks up the facts to determine if the function being called is one of the built-in *Brouahaha* functions with a specific number of arguments or not. If this is the case, instead of emitting `C++` using the global *arg-buffer* array, which is often slower, it directly makes a call to the built-in *Brouhaha* function. By doing so, the compiler avoids the usage of the *arg-buffer* and ensures that any `C++` compiler then can inline the function call to reduce the compilation time.

8 Future Research Direction

Functional program analysis is a vast research field and analysis techniques such as control-flow analysis, and program transformation have been active research areas for many decades. In this survey, we specifically focused on functional compilers and analysis techniques such as abstract interpretation. As part of our ongoing research, we have demonstrated how our prototype compiler works and how we are using Slog to perform analyses to address novel research problems and optimize the compiler's performance.

In the future, we aim to delve into making our analysis more precise and performing many more flow-directed optimizations to the compiler. For example, turning our current monovariant analysis (0-CFA) into a polyvariant analysis (k-CFA) would certainly allow us to make better decisions in terms of call-site optimization. We also aim to address novel research problems such as abstract counting for dalalog-like languages. One of the other optimizations that we are especially interested in is implementing super-beta inlining mentioned in Mights' dissertation [131].

9 Conclusion

This survey paper has provided a comprehensive overview of functional compilation and functional program analysis covering essential topics such as λ-calculus, compiler construction, and abstract interpretation. It also covered the latest developments in the field and discussed the significance of abstract machines and interpreters. In Section 3 we provided a brief introduction to functional programming, lambda calculus, and FP concepts. Then in Section 4 we discussed functional program analysis and some of its methodologies. In Section 5, we introduced a functional language and demonstrated how our prototype functional compiler works for that language. Moving forward, in Section 6, we covered abstract interpretation and abstract machines. Then, Section **??** talked about modern Datalog languages, and compared them in terms of performance and how we are performing program analysis using a Datalog-like language—Slog. Finally, Section 8 then outlined our future research direction.

References

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, Cambridge, MA, 2 edition, 1996.

[2] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv.*, 21(3):359–411, sep 1989.

[3] John Hughes. Why functional programming matters. *The computer journal*, 32(2):98–107, 1989.

[4] Zhenjiang Hu, John Hughes, and Meng Wang. How functional programming mattered. *National Science Review*, 2(3):349–370, 07 2015.

[5] Patrick Thomson, Rob Rix, Nicolas Wu, and Tom Schrijvers. Fusing industry and academia at github (experience report). *Proc. ACM Program. Lang.*, 6(ICFP), aug 2022.

[6] Abdullah Khanfor and Ye Yang. An overview of practical impacts of functional programming. *2017 24th Asia-Pacific Software Engineering Conference Workshops (APSECW)*, pages 50–54, 2017.

[7] Paul Hudak, Simon L. Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph H. Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Richard B. Kieburtz, Rishiyur S. Nikhil, Will Partain, and John Peterson. Report on the programming language haskell: a non-strict, purely functional language version 1.2. *ACM SIGPLAN Notices*, 27:1–, 1992.

[8] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[9] Gerald Sussman and Guy Steele. Scheme: A interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation*, 11:405–439, 12 1998.

[10] Rich Hickey. The clojure programming language. In *Proceedings of the 2008 Symposium on Dynamic Languages*, DLS '08, New York, NY, USA, 2008. Association for Computing Machinery.

[11] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Proc. of a Conference on Functional Programming Languages and Computer Architecture*, page 1–16, Berlin, Heidelberg, 1985. Springer-Verlag.

[12] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sean McDirmid, Stéphane Micheloud, Nikolay L Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the scala programming language second edition. 2006.

[13] Pierre-Yves Saumont. *Functional Programming in Java*. Manning Publications, 2017.

[14] Ivan Čukić. *Functional Programming in C++*. Manning Publications, 2018.

[15] Richard Bird. *Introduction to Functional Programming*. Prentice Hall, London, 1998.

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

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

[18] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[19] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, page 207–212, New York, NY, USA, 1982. Association for Computing Machinery.

[20] Neil D. Jones. Flow analysis of lambda expressions (preliminary version). In *Proceedings of the 8th Colloquium on Automata, Languages and Programming*, page 114–128, Berlin, Heidelberg, 1981. Springer-Verlag.

[21] Neil D. Jones. Flow analysis of lambda expressions. In Shimon Even and Oded Kariv, editors, *Automata, Languages and Programming*, pages 114–128, Berlin, Heidelberg, 1981. Springer Berlin Heidelberg.

[22] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer Publishing Company, Incorporated, 2010.

[23] O. Shivers. Control flow analysis in scheme. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, page 164–174, New York, NY, USA, 1988. Association for Computing Machinery.

[24] Olin Grigsby Shivers. *Control-flow analysis of higher-order languages or taming lambda.* Carnegie Mellon University, 1991.

[25] Alexander Augusteijn. Functional programming, program transformations and compiler construction. 1993.

[26] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science).* Prentice-Hall, Inc., USA, 1987.

[27] ALONZO CHURCH. *The Calculi of Lambda Conversion. (AM-6).* Princeton University Press, 1941.

[28] SL Peyton Jones, K Hammond, WD Partain, PL Wadler, CV Hall, and Simon Peyton Jones. The glasgow haskell compiler: a technical overview. In *Proceedings of Joint Framework for Information Technology Technical Conference, Keele*, pages 249–257. DTI/SERC, March 1993.

[29] Simon Marlow and Simon L. Peyton Jones. The glasgow haskell compiler. 2012.

[30] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. *SIGPLAN Not.*, 28(6):237–247, jun 1993.

[31] G.D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(2):125–159, 1975.

[32] Andrew W Appel. *Compiling with continuations.* Cambridge university press, 2007.

[33] Andrew Kennedy. Compiling with continuations, continued. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, page 177–190, New York, NY, USA, 2007. Association for Computing Machinery.

[34] Christian Queinnec. *Lisp in Small Pieces.* Cambridge University Press, 1996.

[35] A. W. Appel and T. Jim. Continuation-passing, closure-passing style. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, page 293–302, New York, NY, USA, 1989. Association for Computing Machinery.

[36] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.

[37] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. pages 238–252, 01 1977.

[38] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '79, page 269–282, New York, NY, USA, 1979. Association for Computing Machinery.

[39] Jan Midtgaard. Control-flow analysis of functional programs. *ACM Computing Surveys - CSUR*, 44:1–33, 06 2012.

[40] Peter J. Landin. The mechanical evaluation of expressions. *Comput. J.*, 6:308–320, 1964.

[41] Jan Midtgaard and Thomas Jensen. A calculational approach to control-flow analysis by abstract interpretation. In María Alpuente and Germán Vidal, editors, *Static Analysis*, pages 347–362, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[42] Matthias Felleisen and Daniel P. Friedman. Control operators, the secd-machine, and the λ-calculus. In *Formal Description of Programming Concepts*, 1987.

[43] Matthias Felleisen and Daniel P. Friedman. A calculus for assignments in higher-order languages. In *ACM-SIGACT Symposium on Principles of Programming Languages*, 1987.

[44] David Van Horn and Matthew Might. Abstracting abstract machines. *SIGPLAN Not.*, 45(9):51–62, sep 2010.

[45] Thomas Gilray, Arash Sahebolamri, Sidharth Kumar, and Kristopher Micinski. Higher-order, data-parallel structured deduction, 2022.

[46] Matthew Might and Olin Shivers. Improving flow analyses via Γcfa: Abstract garbage collection and counting. *SIGPLAN Not.*, 41(9):13–25, sep 2006.

[47] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, page 237–247, New York, NY, USA, 1993. Association for Computing Machinery.

[48] Yaron Minsky, Anil Madhavapeddy, and Jason Hickey. *Real World OCaml: Functional programming for the masses.* " O'Reilly Media, Inc.", 2013.

[49] Yaron Minsky. Ocaml for the masses. *Communications of the ACM*, 54(11):53–58, 2011.

[50] GL Steele. Common lisp: The language. bedford, 1984.

[51] Peter Seibel. *Practical common lisp.* Apress, 2006.

[52] Joe Armstrong. Programming erlang: software for a concurrent world. *Programming Erlang*, pages 1–548, 2013.

[53] Michael R Hansen and Hans Rischel. *Functional programming using F.* Cambridge University Press, 2013.

[54] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. How to design programs: An introduction to programming and computing. *Education Review*, Jun. 2015.

[55] David Mertz. *Text Processing with Python.* Addison-Wesley Longman Publishing Co., Inc., USA, 2003.

[56] E. Buonanno. *Functional Programming in C#: How to write better C# code.* Manning, 2017.

[57] J. Leo. *The Well-Grounded Rubyist.* Manning, 2019.

[58] Lex Sheehan. *Learning Functional Programming in Go: Change the way you approach your applications using functional programming in Go.* Packt Publishing Ltd, 2017.

[59] Rob Aley. *Pro Functional PHP Programming.* Springer, 2017.

[60] Marco Vermeulen, Rúnar Bjarnason, and Paul Chiusano. *Functional Programming in Kotlin.* Simon and Schuster, 2021.

[61] Steve Klabnik and Carol Nichols. *The Rust programming language.* No Starch Press, 2023.

[62] L. Rosenfeld and A.B. Downey. *Think Perl 6: How to Think Like a Computer Scientist.* O'Reilly Media, 2017.

[63] L. Atencio. *Functional Programming in JavaScript: How to improve your JavaScript programs using functional techniques.* Manning, 2016.

[64] Rod Burstall. Christopher strachey—understanding programming languages. *Higher Order Symbol. Comput.*, 13(1–2):51–55, apr 2000.

[65] Lawrence C. Paulson. *ML for the Working Programmer (2nd Ed.)*. Cambridge University Press, USA, 1996.

[66] Donald Michie. "memo" functions and machine learning. *Nature*, 218:19–22, 1968.

[67] John C Reynolds. Automatic computation of data set definitions. 1969.

[68] Neil D Jones and Steven S Muchnick. Flow analysis and optimization of lisp-like structures. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 244–256, 1979.

[69] Neil D Jones and Steven S Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 66–74, 1982.

[70] Thoms Ball. The concept of dynamic analysis. *ACM SIGSOFT Software Engineering Notes*, 24(6):216–234, 1999.

[71] James R Larus and Thomas Ball. Rewriting executable files to measure program behavior. *Software: Practice and Experience*, 24(2):197–218, 1994.

[72] Michael D Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27, 2003.

[73] Thomas Ball and James R Larus. Efficient path profiling. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*, pages 46–57. IEEE, 1996.

[74] Anjana Gosain and Ganga Sharma. A survey of dynamic program analysis techniques and tools. In Suresh Chandra Satapathy, Bhabendra Narayan Biswal, Siba K. Udgata, and J.K. Mandal, editors, *Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA) 2014*, pages 113–122, Cham, 2015. Springer International Publishing.

[75] Markus Mock. Dynamic analysis from the bottom up. In *WODA 2003 ICSE Workshop on Dynamic Analysis*, page 13, 2003.

[76] Wim De Pauw, Erik Jensen, Nick Mitchell, Gary Sevitsky, John Vlissides, and Jeaha Yang. Visualizing the execution of java programs. In *Software Visualization: International Seminar Dagstuhl Castle, Germany, May 20–25, 2001 Revised Papers*, pages 151–162. Springer, 2002.

[77] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, jun 2007.

[78] Parasoft Inc. Automating c/c++ runtime error detection with parasoft insure++. Technical report, 2006. White Paper.

[79] David J Pearce, Matthew Webster, Robert Berry, and Paul HJ Kelly. Profiling with aspectj. *Software: Practice and Experience*, 37(7):747–777, 2007.

[80] Reed Hastings. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the USENIX Winter'92 Conference*, pages 125–136, 1992.

[81] Jonas Maebe, Dries Buytaert, Lieven Eeckhout, and Koen De Bosschere. Javana: A system for building customized java program analysis tools. *ACM SIGPLAN Notices*, 41(10):153–168, 2006.

[82] Alex Skaletsky, Tevi Devor, Nadav Chachmon, Robert Cohn, Kim Hazelwood, Vladimir Vladimirov, and Moshe Bach. Dynamic program analysis of microsoft windows applications. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pages 2–12. IEEE, 2010.

[83] Michael D. Ernst. *Dynamically Discovering Likely Program Invariants*. Ph.D., University of Washington Department of Computer Science and Engineering, Seattle, Washington, August 2000.

[84] Sudheendra Hangal and Monica S Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th international conference on Software engineering*, pages 291–301, 2002.

[85] Eric Bodden and Klaus Havelund. Aspect-oriented race detection in java. *IEEE Transactions on Software Engineering*, 36(4):509–527, 2010.

[86] Bruno Dufour, Laurie Hendren, and Clark Verbrugge. * j: a tool for dynamic analysis of java programs. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 306–307, 2003.

[87] Paramvir Singh. Design and validation of dynamic metrics for object-oriented software systems. 2009.

[88] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, page 28, USA, 2012. USENIX Association.

[89] Yann-Gaël Guéhéneuc, Rémi Douence, and Narendra Jussien. No java without caffeine: A tool for dynamic analysis of java programs. In *Proceedings of the 17th IEEE International Conference on Automated Software Engineering*, ASE '02, page 117, USA, 2002. IEEE Computer Society.

[90] Ivo Vieira Gomes, Pedro Morgado, Tiago Gomes, and Rodrigo M. L. M. Moreira. An overview on the static code analysis approach in software development. 2009.

[91] Thomas Reps, Thomas Ball, Manuvir Das, and James Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Proceedings of the 6th European SOFTWARE ENGINEERING conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 432–449, 1997.

[92] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Syst. J.*, 15(3):182–211, sep 1976.

[93] Bilal Ilyas and Islam Elkhalifa. Static code analysis: A systematic literature review and an industrial survey. 2016.

[94] Aybuke Aurum, Håkan Petersson, and Claes Wohlin. State-of-the-art: software inspections after 25 years. *Software Testing, Verification and Reliability*, 12(3):133–154, 2002.

[95] Xavier Rival and Kwangkeun Yi. *Introduction to static analysis: an abstract interpretation perspective.* Mit Press, 2020.

[96] Anders Møller and Michael I Schwartzbach. Static program analysis. *Notes. Feb*, 2012.

[97] S. C. Johnson and Murray Hill. Lint, a c program checker. 1978.

[98] David Hovemeyer and William Pugh. Finding more null pointer bugs, but not too many. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '07, page 9–14, New York, NY, USA, 2007. Association for Computing Machinery.

[99] Spotbugs. `https://spotbugs.github.io/`.

[100] Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng (Daphne) Yao. Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized java projects. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 2455–2472, New York, NY, USA, 2019. Association for Computing Machinery.

[101] Pmd. `https://pmd.github.io/`.

[102] mygcc. `http://mygcc.free.fr/`.

[103] Synopsys. `https://scan.coverity.com/`.

[104] Clang static analyzer. `https://clang-analyzer.llvm.org/`.

[105] Pylint. `https://pylint.readthedocs.io/en/stable/`.

[106] pyflakes. `https://pypi.org/project/pyflakes/`.

[107] frosted. `https://pypi.org/project/frosted/`.

[108] Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs. In Todd Millstein, editor, *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*, volume 109 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:27, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[109] Alan Mathison Turing et al. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.

[110] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical society*, 74(2):358–366, 1953.

[111] Thomas Gilray. *Introspective Polyvariance for Control-Flow Analyses*. PhD thesis, The University of Utah, 2017.

[112] Guillermo J Rozas. Liar, an algol-like compiler for scheme. sb thesis, 1984.

[113] Peter Sestoft. Replacing function parameters by global variables. master's thesis, diku. *Computer Science Department, University of Copenhagen, Copenhagen, Denmark*, 1988.

[114] Peter Sestoft. Replacing function parameters by global variables. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, page 39–53, New York, NY, USA, 1989. Association for Computing Machinery.

[115] Anders Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 17(1):3–34, 1991.

[116] Jens Palsberg. Global program analysis in constraint form. In Sophie Tison, editor, *Trees in Algebra and Programming — CAAP'94*, pages 276–290, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[117] Anders Bondorf and Jesper Jørgensen. Efficient analyses for realistic off-line partial evaluation. *Journal of Functional Programming*, 3(3):315–346, 1993.

[118] Jens Palsberg. Equality-based flow analysis versus recursive types. *ACM Trans. Program. Lang. Syst.*, 20(6):1251–1264, nov 1998.

[119] Neil D. Jones and Nils Andersen. Flow analysis of lazy higher-order functional programs. *Theoretical Computer Science*, 375(1):120–136, 2007. Festschrift for John C. Reynolds's 70th birthday.

[120] Manuel Serrano. Control flow analysis: A functional languages compilation paradigm. In *Proceedings of the 1995 ACM Symposium on Applied Computing*, SAC '95, page 118–122, New York, NY, USA, 1995. Association for Computing Machinery.

[121] Henry Cejtin, Suresh Jagannathan, and Stephen Weeks. Flow-directed closure conversion for typed languages. In *European Symposium on Programming*, 2000.

[122] Christian Mossin. *Flow analysis of typed higher-order programs*. PhD thesis, University of Copenhagen, 1997.

[123] Jens Palsberg. Closure analysis in constraint form. *ACM Trans. Program. Lang. Syst.*, 17(1):47–62, jan 1995.

[124] J. Michael Ashley and R. Kent Dybvig. A practical and flexible flow analysis for higher-order languages. *ACM Trans. Program. Lang. Syst.*, 20(4):845–868, jul 1998.

[125] David Van Horn and Harry G. Mairson. Deciding kcfa is complete for exptime. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, page 275–282, New York, NY, USA, 2008. Association for Computing Machinery.

[126] Suresh Jagannathan and Stephen Weeks. A unified treatment of flow analysis in higher-order languages. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, page 393–407, New York, NY, USA, 1995. Association for Computing Machinery.

[127] Matthew Might, Yannis Smaragdakis, and David Van Horn. Resolving and exploiting the k-cfa paradox: Illuminating functional vs. object-oriented program analysis. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, page 305–315, New York, NY, USA, 2010. Association for Computing Machinery.

[128] John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11:363–397, 1972.

[129] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, page 271–283, New York, NY, USA, 1996. Association for Computing Machinery.

[130] Thomas Gilray, Michael D. Adams, and Matthew Might. Allocation characterizes polyvariance: A unified methodology for polyvariant control-flow analysis. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, page 407–420, New York, NY, USA, 2016. Association for Computing Machinery.

[131] Matthew Might. *Environment analysis of higher-order languages.* Georgia Institute of Technology, 2007.

[132] S.J. Russell, S. Russell, and P. Norvig. *Artificial Intelligence: A Modern Approach.* Pearson series in artificial intelligence. Pearson, 2020.

[133] R. Brachman and H. Levesque. *Knowledge Representation and Reasoning.* The Morgan Kaufmann Series in Artificial Intelligence. Elsevier Science, 2004.

[134] Jonathan Aldrich. Lecture notes: Interprocedural analysis. `https://www.cs.cmu.edu/~aldrich/courses/15-819O-13sp/resources/interprocedural.pdf`, 2013. Course note for the course *Program Analysis*.

[135] Neil D. Jones and Steven S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *ACM-SIGACT Symposium on Principles of Programming Languages*, 1982.

[136] Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, page 230–241, New York, NY, USA, 1994. Association for Computing Machinery.

[137] François Bourdoncle. Interprocedural abstract interpretation of block structured languages with nested procedures, aliasing and recursivity. In *International Workshop on Programming Language Implementation and Logic Programming*, pages 307–323. Springer, 1990.

[138] Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. *SIGPLAN Not.*, 29(6):230–241, jun 1994.

[139] Flemming Nielson and Hanne Riis Nielson. Interprocedural control flow analysis. In S. Doaitse Swierstra, editor, *Programming Languages and Systems*, pages 20–39, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[140] Micha Sharir, Amir Pnueli, et al. *Two approaches to interprocedural data flow analysis.* New York University. Courant Institute of Mathematical Sciences, 1978.

[141] Markus Müller-Olm and Helmut Seidl. Precise interprocedural analysis through linear algebra. *ACM SIGPLAN Notices*, 39(1):330–341, 2004.

[142] Keith D. Cooper and Linda Torczon. Chapter 9 - data-flow analysis. In Keith D. Cooper and Linda Torczon, editors, *Engineering a Compiler (Second Edition)*, pages 475–538. Morgan Kaufmann, Boston, second edition edition, 2012.

[143] IBM. Official documentation on: Interprocedural analysis (ipa). `https://www.ibm.com/docs/en/i/7.5?topic=techniques-interprocedural-analysis-ipa`, 2023.

[144] David Callahan, Keith D Cooper, Ken Kennedy, and Linda Torczon. Interprocedural constant propagation. *ACM SIGPLAN Notices*, 21(7):152–161, 1986.

[145] Keith D Cooper, Ken Kennedy, and Linda Torczon. The impact of interprocedural analysis and optimization in the rn programming environment. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(4):491–523, 1986.

[146] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.

[147] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, jul 1962.

[148] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, pages 502–518, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[149] Josep Argelich and Felip Manyà. Exact max-sat solvers for over-constrained problems. *J. Heuristics*, 12:375–392, 09 2006.

[150] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, page 530–535, New York, NY, USA, 2001. Association for Computing Machinery.

[151] Wu Kehui, Wang Tao, Zhao Xinjie, and Liu Huiying. Cryptominisat solver based algebraic side-channel attack on present. In *2011 First International Conference on Instrumentation, Measurement, Computer, Communication and Control*, pages 561–565, 2011.

[152] Adrian Balint and Uwe Schöning. Choosing probability distributions for stochastic local search and the role of make versus break. In *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing*, SAT'12, page 16–29, Berlin, Heidelberg, 2012. Springer-Verlag.

[153] Joao Marques-Silva and Karem A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, 48:506–521, 1999.

[154] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Trans. on Knowl. and Data Eng.*, 1(1):146–166, mar 1989.

[155] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[156] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using datalog with binary decision diagrams for program analysis. In Kwangkeun Yi, editor, *Programming Languages and Systems*, pages 97–118, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[157] Oege De Moor, Georg Gottlob, Tim Furche, and Andrew Sellers. *Datalog Reloaded: First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers*, volume 6702. Springer, 2012.

[158] Shan Shan Huang, Todd Jeffrey Green, and Boon Thau Loo. Datalog and emerging applications: An interactive tutorial. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, page 1213–1216, New York, NY, USA, 2011. Association for Computing Machinery.

[159] J.M. Hellerstein and M. Stonebraker. *Readings in Database Systems.* Mit Press. MIT Press, 2005.

[160] Maurizio Lenzerini. Data integration: A theoretical perspective. In *Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, page 233–246, New York, NY, USA, 2002. Association for Computing Machinery.

[161] Boon Thau Loo, Joseph M. Hellerstein, Ion Stoica, and Raghu Ramakrishnan. Declarative routing: extensible routing with declarative queries. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 2005.

[162] Trevor Jim. Sd3: a trust management system with certified evaluation. *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*, pages 106–115, 2001.

[163] Peter Alvaro, Tyson Condie, Neil Conway, Khaled Elmeleegy, Joseph M. Hellerstein, and Russell Sears. Boom analytics: Exploring data-centric, declarative programming for the cloud. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, page 223–236, New York, NY, USA, 2010. Association for Computing Machinery.

[164] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, page 243–262, New York, NY, USA, 2009. Association for Computing Machinery.

[165] Herbert Jordan, Bernhard Scholz, and Pavle Subotic. Soufflé: On synthesis of program analyzers. In *International Conference on Computer Aided Verification*, 2016.

[166] Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. From datalog to flix: A declarative language for fixed points on lattices. *SIGPLAN Not.*, 51(6):194–208, jun 2016.

[167] Michael Arntzenius and Neelakantan R. Krishnaswami. Datafun: A functional datalog. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, page 214–227, New York, NY, USA, 2016. Association for Computing Machinery.

[168] Tamás Szabó, Sebastian Erdweg, and Markus Völter. Inca: A dsl for the definition of incremental program analyses. *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 320–331, 2016.

[169] Arash Sahebolamri, Thomas Gilray, and Kristopher Micinski. Seamless deductive inference via macros. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, CC 2022, page 77–88, New York, NY, USA, 2022. Association for Computing Machinery.

[170] Bas Ketsman and Paraschos Koutris. Modern datalog engines. *Found. Trends Databases*, 12(1):1–68, jun 2022.

[171] Thomas Gilray, Sidharth Kumar, and Kristopher Micinski. Compiling data-parallel datalog. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, CC 2021, page 23–35, New York, NY, USA, 2021. Association for Computing Machinery.

[172] Magnus Madsen and Ondřej Lhoták. Fixpoints for the masses: Programming with first-class datalog constraints. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020.

[173] Yufei Cai, Paolo G. Giarrusso, Tillmann Rendel, and Klaus Ostermann. A theory of changes for higher-order languages: Incrementalizing λ-calculi by static differentiation. *SIGPLAN Not.*, 49(6):145–155, jun 2014.

[174] Paolo G. Giarrusso, Yann Régis-Gianas, and Philipp Schuster. Incremental λ-calculus in cache-transfer style. In Luís Caires, editor, *Programming Languages and Systems*, pages 553–580, Cham, 2019. Springer International Publishing.

[175] Frank McSherry and The Rust Developers. Rust-lang/datafrog: A lightweight datalog engine in rust. `https://github.com/rust-lang/datafrog`, 2021.

[176] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. Big data analytics with datalog queries on spark. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1135–1149, New York, NY, USA, 2016. Association for Computing Machinery.