UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CIÊNCIAS DA COMPUTAÇÃO

Gabriel Baiocchi de Sant'Anna

Make UNCOL cool again

Florianópolis
2022

Gabriel Baiocchi de Sant'Anna

Make UNCOL cool again

Trabalho de Conclusão de Curso de Graduação em Ciências da Computação do Centro Tecnológico da Universidade Federal de Santa Catarina como requisito para obtenção do título de Bacharel em Ciências da Computação.
Orientador: Prof. Erwan Jahier, Dr.
Coorientadora: Prof[a]. Jerusa Marchi, Dra.

Florianópolis
2022

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Sant'Anna, Gabriel Baiocchi de
Make UNCOL cool again / Gabriel Baiocchi de Sant'Anna ; orientador, Erwan Jahier, coorientadora, Jerusa Marchi, 2022.
120 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Ciências da Computação, Florianópolis, 2022.

Inclui referências.

1. Ciências da Computação. 2. Linguagens de Programação. 3. Compiladores. 4. Representação Intermediária. 5. Portabilidade. I. Jahier, Erwan. II. Marchi, Jerusa. III. Universidade Federal de Santa Catarina. Graduação em Ciências da Computação. IV. Título.

Gabriel Baiocchi de Sant'Anna

Make UNCOL cool again

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do Título de "Bacharel em Ciências da Computação" e aprovado em sua forma final pelo curso de Graduação em Ciências da Computação.

Florianópolis, 31 de Julho de 2022.

Prof. Jean Everson Martina, Dr.
Coordenador do Curso

Banca Examinadora:

Prof. Erwan Jahier, Dr.
Orientador
CNRS - Verimag - Université Grenoble Alpes

Prof[a]. Jerusa Marchi, Dra.
Coorientadora
Universidade Federal de Santa Catarina

Albert Cohen, Dr.
Avaliador
Google

Prof. Pascal Raymond, Dr.
Avaliador
CNRS - Verimag - Université Grenoble Alpes

ACKNOWLEDGEMENTS

First and foremost I wish to thank my family, especially my parents, Helio and Mara, for the careful development and continuous maintenance of my self over the years. Whatever it is that I may achieve, it is only be possible because of what they have provided me with.

Thanks are due to my advisors, Erwan Jahier and Jerusa Marchi, for lending me their time, supporting my ideas for the thesis and reviewing drafts of this document. I must also mention Luiz Henrique Cancellier and other members of UFSC's Embedded Computing Lab., who taught me the 'Science' in Computer Science.

I would like to thank my friends and colleagues (too many to cite by name) for all of our interesting conversations and for the eventual healthy distraction from work. Lastly, I will mention a few other people who have contributed to this work in some way. Conversation with Stavros Macrakis provided some historical perspective on portable program representations, notably their trade-offs from a business perspective. I would also like to acknowledge Alexandra Elbakyan, for her indirect assistance with the literature review. Finally, the Programming Language Design community's subreddit and Discord have been useful channels for discussing ideas and finding learning resources.

RESUMO

Criar e manter um compilador otimizador requer grandes esforços de desenvolvimento. Ao mesmo tempo, o número de compiladores necessários para traduzir cada linguagem de alto nível para várias arquiteturas de hardware cresce de forma multiplicativa. Uma possível abordagem para resolver esse problema envolve adotar uma representação intermediária comum, também conhecida como *Universal Computer Oriented Language (UNCOL)*. Embora tal solução tenha sido proposta pela primeira vez em 1958, a tecnologia de compiladores e a teoria de linguagens de programação muito evoluíram desde então. No contexto destes avanços, o presente trabalho tem como objetivo reavaliar a ideia de uma representação intermediária universal; começando por um estudo da literatura e do estado da arte de linguagens de programação, de forma a elicitar requisitos e identificar princípios de criação para uma UNCOL moderna. Esses requisitos são a base de uma análise de algumas das representações intermediárias utilizadas em compiladores existentes. Além disso, os princípios extraídos da revisão sistemática motivaram a criação de uma nova representação intermediária, que combina técnicas de compilação originárias de diversas fontes na literatura. A nova representação intermediária é descrita em múltiplos aspectos, incluindo uma definição formal da sua estrutura em multigrafo e uma correspondência informal de sua semântica em termos de um modelo computacional já existente. Por fim, este trabalho descreve alguns algoritmos de análise e transformação de código customizados para a nova representação. Entre estes consta uma formulação de eliminação de código morto (uma otimização global) através de um algoritmo de coleção de lixo.

Palavras-chave: Linguagens de Programação. Compiladores. Representação Intermediária. Portabilidade.

ABSTRACT

Developing and maintaining an optimizing compiler requires great amounts of effort. At the same time, the number of compilers needed in order to translate many high-level languages to every other target architecture grows multiplicatively. One possible approach to solve this problem is the adoption of a shared intermediate representation, also known as Universal Computer Oriented Language (UNCOL). While the UNCOL solution was first proposed in 1958, there have been many developments in compiler technology and programming language theory since then. This work aims to re-evaluate the idea of a universal intermediate representation in light of these advances, beginning by surveying the programming language literature and state of the art in order to identify requirements and design principles for a modern version of UNCOL. Then, these requirements are used to analyze program representations in existing compiler infrastructures. Furthermore, the set of principles extracted from the systematic review has motivated the design of a new intermediate representation, which combines compilation techniques from various sources in the literature. Multiple aspects of the new intermediate representation are described, encompassing a formal definition of its multigraph structure, an informal explanation of its semantics in terms of the join calculus, and a few custom optimization algorithms, including a formulation of global Dead Code Elimination as a garbage collection process.

Keywords: Programming Languages. Compilers. Intermediate Representation. Portability.

LIST OF FIGURES

LIST OF ABBREVIATIONS

3AC Three-Address Code

ANDF Architecture Neutral Distribution Format

ANF Administrative Normal Form

AST Abstract Syntax Tree

CFG Control Flow Graph

CIL Common Intermediate Language

CLI Common Language Infrastructure

CPS Continuation-Passing Style

DAG Directed Acyclic Graph

DFG Data Flow Graph

EM Encoding Machine

HSA Heterogeneous System Architecture

IL Intermediate Language

IR Intermediate Representation

JIT Just-In-Time

JVM Java Virtual Machine

PDG Program Dependence Graph

PDW Program Dependence Web

RTL Register Transfer Language

RVSDG Regionalized VSDG

SSA Static Single Assignment

SSC Strongly Connected Component

SUIF Stanford University Intermediate Format

TCO Tail Call Optimization

UNCOL Universal Computer Oriented Language

VDG Value Dependence Graph

VM Virtual Machine

VSDG Value State Dependence Graph

Wasm WebAssembly

[0] **Author's note:** Acronyms are commonplace in technical writing. Works on programming languages and compilers, in particular, use abbreviations abundantly. However, it is not rare to have an abbreviation change its meaning over time (e.g., GCC), or even become a name by itself (a notable example being LLVM). Adding to the confusion, technology-related names and trademarks are often registered in capital letters (e.g., UNIX), even if they do not form an abbreviation. Thus, I refrain from providing the full extension of acronyms unless that definition is relevant to the understanding of the text. Abbreviations not listed here should be treated as proper names. An index is provided to aid readers in locating contextual information about relevant terms.

CONTENTS

1 INTRODUCTION

> An interpreter raises the machine to the level of the user program; a compiler lowers the user program to the level of the machine language. We can regard the Scheme language (or any programming language) as a coherent family of abstractions erected on the machine language. (ABELSON; SUSSMAN, 1996, ch. 5)

This is a thesis about Intermediate Representations (IRs), which are used in the context of programming languages and compilers. More specifically, the thesis pivots around the concept of UNCOL: a Universal Computer Oriented Language.

The first "high-level" programming languages were designed in the 1940s, for the earliest fully-operational digital computers (GILOI, 1997). However, these programming languages could not be used until the advent of the first compilers, in the early 1950s (BACKUS, 1978). Already by 1958, UNCOL was proposed as a solution to an apparent problem in compiler development; it is an old idea which deserves to be re-evaluated in the present day.

Before expanding upon UNCOL, it is necessary to lay down the surrounding context. The current chapter connects fundamental notions of machines, computers, programming languages, compilers and IRs. It introduces some of the concepts used throughout the thesis and describes the organization of the rest of this document.

1.1 MACHINES AND LANGUAGES

Automatic machines or *automata* are devices, physical or otherwise, which operate through predefined actions (TURING, 1937). The set of possible actions, properties of these actions and the rules which govern the machine form a model of computation (SAVAGE, 1997). Any particular pattern of rules that controls a machine is called procedure or program (ABELSON; SUSSMAN, 1996, ch. 1). Among procedures, those which induce a machine to produce the result of a precise mathematical function are called algorithms (PERLIS, 1996). Church's Law[1] states that, for every effectively calculable function, there exists an algorithm to compute its values (CHURCH, 1936). Finally, universal computing machines – whose existence was shown by Turing (1937) – are those capable of executing any algorithm, and therefore of computing all effectively calculable functions.

A formal language is a set of strings, each composed of certain symbols of an alphabet according to specific formation rules (HOPCROFT; MOTWANI; ULLMAN, 2006). The connection between languages and machines can be seen by relating each language to a function mapping arbitrary strings to Boolean values, such that its output indicates whether or not each string belongs to the language. Then, classes of languages can be distinguished based on which machines are able to compute the aforementioned function; it is said that those machines

[1] More commonly referred to as Church's Thesis, Church-Turing Thesis or Church-Turing Conjecture. The term 'law' is used here instead of the alternatives, following the argument of Harper (2016, ch. 21).

(or, their model of computation) have enough "language-recognizing power" for that language (HOPCROFT; MOTWANI; ULLMAN, 2006). Notice that this notion of power is associated with machines and their models of computation, not with the language being recognized.

Another relation between machines and languages is that encodings of machine programs can be described by (formal) languages. Thus, when a language is used to encode programs for certain machines, it can be called a machine language, but also a programming language or a computer language. Other related concepts are those of high-level language and low-level language. Although widely used in programming circles and in the literature, these terms are not well-defined.

According to Franz (1994, ch. 2), programming languages correspond to abstract ("relatively complex") universal machines, and machine languages characterize specific realizations of universal machines as digital computers. The same meaning of "machine language" is implied by the term's usage in Abelson & Sussman (1996) and Strong et al. (1958a). While Perlis (1982) indicates that "a programming language is low level when its programs require attention to the irrelevant", Chisnall (2018) prefers to associate "low-level" with "closer-to-metal" (that is, close to the language of a physical machine). Chow (2013) seems to agree on that definition of low-level languages. He also indicates that high-level languages should be easy for humans (as opposed to machines) to use and understand. Sammet & Hemmendinger (2003) and Click (1995) associate "high-level" and "low-level" to programming languages and machine languages, respectively. Throughout this thesis, a programming language is a language designed for humans to use for the purpose of programming abstract universal computing machines under the rule of a certain model of computation. Meanwhile, machine languages are those which control the execution of programs on real machines. The former are considered high-level; the latter, low-level. Computer language intentionally remains an ambiguous term and the unqualified 'language' refers to some formal (as opposed to a natural) language.

At first glance, it would seem reasonable to extend the notion of power associated with a machine to the computer language used to program it. However, all programming languages (under the definition given above, in terms of universal machines) are Turing-complete, that is, they expose models of computation equivalent to a Turing machine, in the sense that they can compute exactly the same set of functions. By this equivalence, universal machines are capable of emulating all other machines, and different universal machines can emulate each other (HOPCROFT; MOTWANI; ULLMAN, 2006). This has motivated Felleisen (1991) to formalize the concept of a programming language's expressive power (or, its expressiveness). His theory of expressiveness chiefly relies on a program equivalence relation (also called operational equivalence) and equivalence-preserving transformations between programs. In other words, it assumes the existence of compilers.

1.2 OPTIMIZING COMPILERS

Compilers are translators. They transform programs encoded in a source language into equivalent programs encoded in a target language (AHO et al., 2006, p. 1). While the last section hinted at how they may also be used from a theoretical perspective, Click (1995) very succinctly describes why compilers are useful in practice:

> Computers are machines; they follow their instructions precisely. Writing precisely correct instructions is hard. People are fallible; programmers have difficulty understanding large programs; and computers require a tremendous number of very limited instructions. Therefore, programmers write programs in high-level languages, languages more easily understood by a human. Computers do not understand high-level languages. They require compilers to translate high-level languages down to the very low-level machine language they do understand. (CLICK, 1995, p. 1)

Thus, compilers are tools which enable the separation of high-level programming languages from low-level machine languages by providing a bridge between them. The process of translating a high-level program into a lower-level representation is also known as lowering.

Compilation must preserve the meaning of programs being translated (LAWRENCE, 2007, p. 12). In order to do so, a compiler has to understand all possible programs written in the source language (as long as they are valid[2]) and must know how to precisely encode their semantics in the target language. The semantics of a program are determined by the model of computation exposed by the computer language it is encoded in. Fortunately, most source-target pairings of computer languages allow compilers to choose between many possible ways to carry out a translation (CLICK, 1995, p. 1). In this context, an optimizing compiler performs semantics-preserving transformations, not only to re-encode a program in another language, but also to do so while attempting to optimize its usage of machine resources (CLICK, 1995, p. 1).

However, reasoning about programs is not trivial. Indeed, when dealing with universal models of computation, some properties are impossible to prove in the general case (CHURCH, 1936; TURING, 1937) and a compiler must make conservative decisions, reducing its potential to apply optimizations (SHIVERS, 1988). Hence, compilers employ many techniques to facilitate reasoning about programs, including dividing the process into multiple phases and using different program representations between each separate phase (AHO et al., 2006, p. 5).

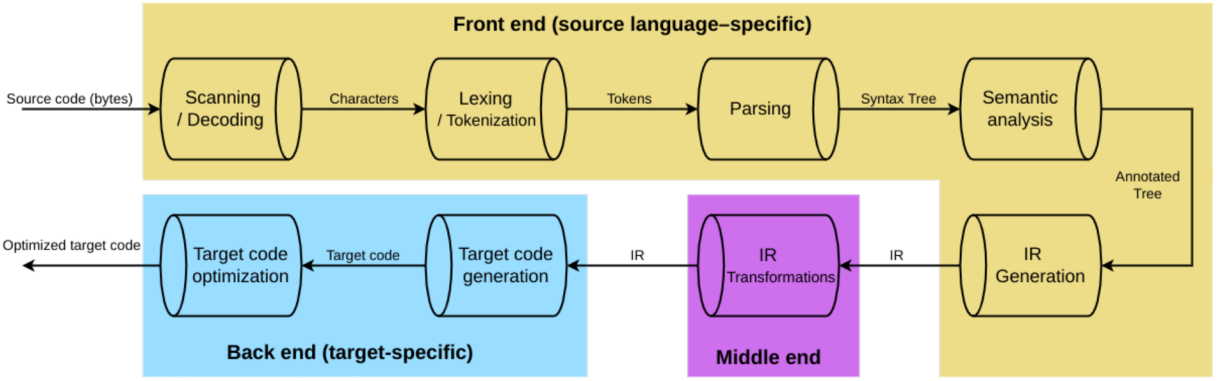
1.3 PROGRAM REPRESENTATIONS

Often, and for a variety of reasons, compilers are structured as a pipeline consisting of multiple stages, each with a distinct purpose (AHO et al., 2006, p. 5). Figure 1 displays the

[2] One does not usually expect a compiler to assign meaning to an invalid program, especially when it can detect this is the case and report the reason to the programmer. In C, for example, when a program is erroneous, but this fact is not (or cannot be) signaled by the compiler, program behavior is “undefined” (ISO/IEC 9899, 2017).

logical structure of a typical optimizing compiler. In some cases, the separation may be only conceptual: some compilers "take shortcuts", mixing some phases together and skipping others entirely (NYSTROM, 2021, ch. 2). Still, a program usually takes multiple forms during the compilation process, some of which can be seen in Figure 1. Any such representation between input source and generated code might be called an Intermediate Representation (IR).

Figure 1 – Logical structure of a typical optimizing compiler.



Source: Gabriel B. Sant'Anna, 2022.

The more conventional definition of IR refers specifically to the representations used between a compiler's front end, responsible for checking and respecting the rules of the source language, and its back end, which handles the idiosyncrasies of the target encoding (AHO et al., 2006, p. 357). Although this is not shown in Figure 1, it is not uncommon for optimizing compilers to use multiple IRs (AHO et al., 2006, p. 358). In fact, IRs play a major role in modern compiler infrastructures, warranting the rise of the middle end as one of the most important parts of an optimizing compiler (CHOW, 2013). Furthermore, one might want to distinguish IRs, which usually only exist as transient data structures in memory (FRANZ, 1994, ch. 2), from Intermediate Languages (ILs), which represent a persistent external encoding of some IR (ZHAO; SARKAR, 2011a). More often than not, IR and IL are used as interchangeable terms.

1.4 THESIS OUTLINE

This chapter introduced concepts related to the subject of programming languages and compilers; definitions provided here will be used in the rest of the document. Still, one of the main goals of this work is to shed a modern light on the concept of UNCOL, which will be described in Chapter 2, along with justification for the research topic.

Chapter 3 constitutes the bulk of the text, presenting a survey on compiler IRs. The literature review is used to uncover requirements and design principles for portable IRs, which are then used to analyze existing some compilers. Those are the main contributions of the thesis.

Additionally, Chapter 4 describes a new IR design, which gathers ideas from the previous chapter's research. Finally, Chapter 5 summarizes what has been achieved as part of this thesis and outlines possible future work directions.

2 COMMUNICATION WITH CHANGING MACHINES

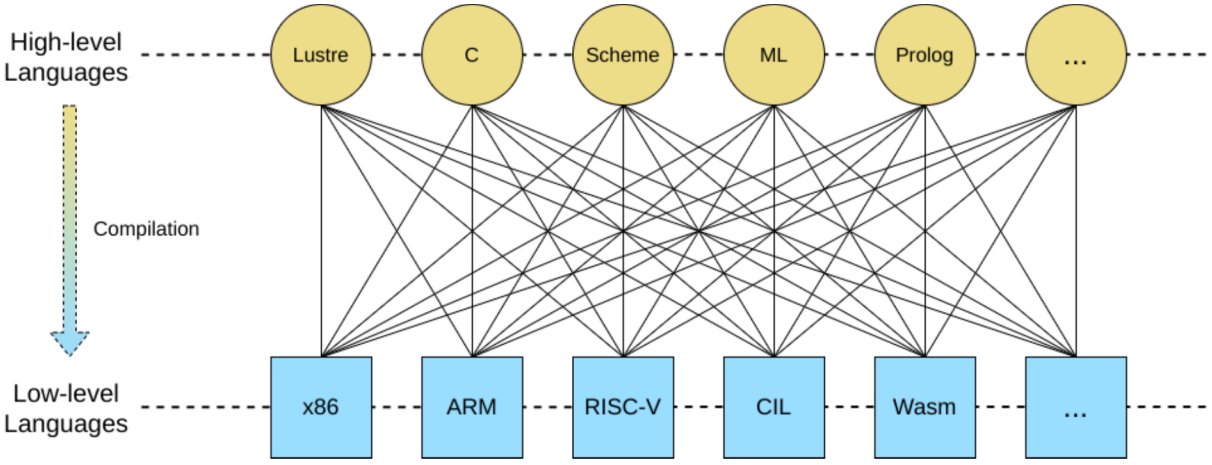
> A standard for a universal IR that enables target-independent program binary distribution and is usable internally by all compilers may sound idealistic, but it is a good cause that holds promises for the entire computing industry. (CHOW, 2013)

2.1 THE COMPILER CONSTRUCTION PROBLEM

Chapter 1 described computer languages as the means by which humans can communicate with machines. The communication is generally considered more effective when programs are written in a high-level programming language (CLICK, 1995, p. 1), and programs more efficient when an optimizing compiler performs the translation to machine language (JOHNSON, 2004, p. 16). However, developing (and maintaining) an optimizing compiler is, and has always been, far from trivial (STEEL, 1961a; OLIVA; NORDIN; JONES, 1997; CHOW, 2013; SUSUNGI; TADONKI, 2021)[3].

The cost of producing optimizing compilers is exacerbated by the variety of programming languages and machines. If programs written in N distinct high-level languages are to be efficiently executed on machines varying across M different architectures, the number of compilers needed would be $N \times M$. This non-linear relation has been dubbed "the compiler construction problem" (MACRAKIS, 1993b) and is illustrated in Figure 2.

Figure 2 – The compiler construction problem.



Source: Gabriel B. Sant'Anna, 2022.

Back in 1958, the compiler construction problem was foretold by Strong et al. (1958a). They observed the number of distinct types of computers increasing by the dozens every year (STEEL, 1961a); Moore's Law had not yet been proclaimed (PATTERSON; HENNESSY, 2013, p. 11), but the rapid change in computer architecture was already apparent. At the same

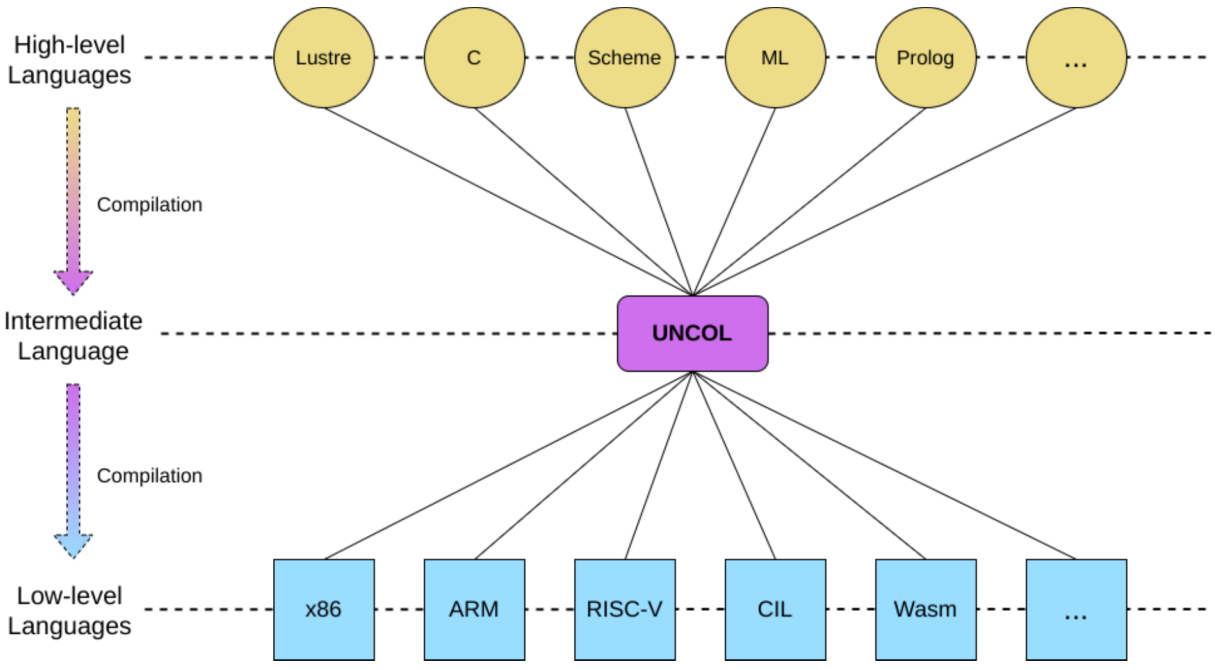
[3] Optimizing compilers are not the only way to implement a programming language. Interpreters, for instance, provide a simpler path which almost anyone can follow (NYSTROM, 2021, ch. 1).

time, high-level languages were becoming increasingly popular and numerous (STEEL, 1961a). In this setting, Strong et al. (1958a) were one of the first to acknowledge the challenges of "programming communication with changing machines", as they called it. They also realized that any acceptable resolution of this problem would need to embrace the heterogeneity of programming languages and machines, instead of rejecting this diversity (STEEL, 1961b).

2.2 UNCOL AND OTHER SOLUTIONS

Strong et al. (1958a) proposed a solution to the compiler construction problem: the Universal Computer Oriented Language (UNCOL). It consists in adding a common intermediate layer to compilers; each high-level source language would be translated to UNCOL, which in turn would be encoded into the target machine language (STRONG et al., 1958a). Figure 3 demonstrates this process. With this approach, a programming language implementor need only care about the language-specific parts of the compilation process, up to UNCOL generation. Symmetrically, a hardware vendor could provide a single machine-specific back end and immediately have existing software available on their new platform. In summary, the adoption of UNCOL would mean that only $N + M$ optimizing compilers would be required for the efficient execution of any high-level language across all architectures, to which Steel (1961a) adds: "when M and N are greater than two it is game, set and match to UNCOL". It should be noted that the term 'UNCOL' is identified with the idea of a universal compiler IR acting as a "linguistic switchbox" (FRANZ, 1994); it does not refer to any particular implementation.

Figure 3 – The UNCOL solution.



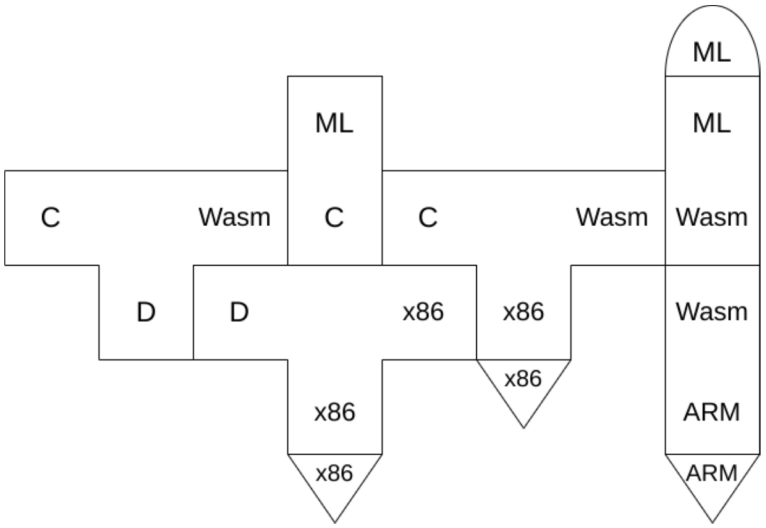
Source: Gabriel B. Sant'Anna, 2022.

The three-level (or, two-step) compilation process was not particularly original; as

Strong et al. (1958a) put it, "it might not be difficult to prove that 'this was well-known to Babbage' ". Indeed, universal machine codes were being discussed as long ago as 1951 (FORRESTER, 1951) and two-step translation in order to communicate with computers had already been noted by Ross (1956) a couple of years earlier. Still, Strong et al. (1958a) were the first to frame this in the context of high-level programming languages and rapidly changing machines. In fact, the UNCOL report seems to have been (to the best of the author's knowledge) the first published source of the present definitions of high-level and low-level languages: the former are described as "the highest level" (as in Figure 3), "furthest from the machine"; the latter as machine languages, "closest to the bits in the machine hardware" (STRONG et al., 1958a).

This first work on UNCOL was influential in other ways. For instance, a second part of the report (STRONG et al., 1958b) described bootstrapping techniques which, according to Aho et al. (2006, p. 425), are still in use. As a matter of fact, the schematics introduced by Strong et al. (1958b) were later reshaped by Bratman (1961) and the resulting "T-diagrams" are still used to illustrate and teach complex methods of computer language implementation (MOGENSEN, 2010, ch. 13) – see Figure 4 for an example[4].

Figure 4 – Example T-diagram, using Mongensen's notation.



Source: Gabriel B. Sant'Anna, 2022.

Unfortunately, no proposal for an UNCOL has been met with universal agreement, and the concept seemed too ambitious for its time (TANENBAUM et al., 1983; MACRAKIS,

[4] In Figure 4, the leftmost 'T' represents a C compiler which targets WebAssembly (Wasm) and is implemented in D. The D-to-x86 compiler (the bottom 'T') is an x86 binary, so it runs on the x86 machine represented by the triangle at the bottom. After compilation from D to x86, the C-to-Wasm compiler (now seen as the 'T' on the right) also executes on an x86 machine. Its input, a C program, happens to implement an ML interpreter, represented by the leftmost vertical rectangle. The result, as seen on the other side of the C-to-Wasm compiler, is an ML interpreter encoded in Wasm. It can be used to run an ML application, represented by the half circle on top of it. However, a user may wish to run the application on their mobile device, which uses ARM hardware (the rightmost triangle) and does not natively recognize Wasm code. In that case, the rectangle in the lower right corner of Figure 4 completes the "puzzle" by emulating the Wasm abstract machine on the user's ARM device. Hopefully, this example suffices to demonstrate the compactness of T-diagram illustrations.

1993b; FRANZ, 1994; LATTNER, 2002). Kegley (1977) goes as far as saying that the idea "died a natural death", albeit Franz (1994) suggests its "spirit" was kept alive. This seems like an accurate depiction since, years later, Oliva, Nordin & Jones (1997) indicate that the "ghost of UNCOL" still haunts compiler writers.

Multiple factors have contributed to UNCOL's downfall. One of the first problems arises from the fact that not all high-level programs are entirely machine-independent (BAGLEY, 1962). According to Franz (1994, ch. 6), software portability is one of the most elusive concepts in Computer Science, yet much of the value proposition of UNCOL implicitly relied on it. Moreover, the compiler construction problem was becoming less relevant as the computer market stabilized: by 2000, over 90% of all workstations, personal computers and servers were based on the x86 architecture (LIEDTKE et al., 1998). Finally, it could even be said that UNCOL was an idea too advanced for its time:

> A striking feature of the UNCOL papers is the number of innovations that would have been needed to make it work. Since there was no standard character code, UNCOL would have had to define one. [...] Bootstrapping was a novel technique. [...] Indeed, the very notion of an intermediate language for compilers [...] was rather novel. At the same time, programming language technology was in its infancy. Concepts that we now take for granted were novel or even non-existent [...] (MACRAKIS, 1993b)

Indeed, the second half of the UNCOL report was fully dedicated to bootstrapping (STRONG et al., 1958b) and a good part of Steel's later articles described character encodings[5] and defended the theoretical existence of a universal computer language (STEEL, 1961a; STEEL, 1961b).

UNCOL was a proposed solution to the compiler construction problem, but not the only one. Steel (1961b) noted two alternatives, listed below. As shall be explained later, these have some overlap and do not exclude the UNCOL approach.

1. Making compiler development easier. This solution does not actually handle the $N \times M$ problem, but aims to reduce the cost of each front end and back end by improving methods of compiler writing. A plausible approach involves reusing compiler components, in a shared (usually open-source) infrastructure. LLVM is a popular framework which follows this principle (LATTNER; ADVE, 2004).

2. Developing a compiler-compiler[6]. Such a program would generate compilers based on descriptions of source languages and target machines. Steel (1961b) considered this to be the "more dignified" solution to the compiler construction problem, and hoped that effort spent on UNCOL would later help develop compiler-compilers.

[5] One must keep in mind that UNCOL predates character encoding standards such as Unicode (UNICODE, 1991) and Latin-1 (ECMA-94, 1985). Even ASCII had not been defined yet (GORN; BEMER; GREEN, 1963).

[6] Although the first compiler-compiler was conceived and developed by Brooker et al. (1963) in the 1950s, Steel (1961b) may have been the first to publish a description of (along with some motivation for) the idea.

Full compiler-compilers – as defined by Steel (1961b), or Futamura (1999) – have not seen much practical usage in the implementation of programming languages (WÜRTHINGER et al., 2017). On the other hand, restricting automatic programming to specific parts of the compilation pipeline has been more successful: parser generators have become standard compiler technology (AHO et al., 2006, p. 287) and table-driven back end descriptors are not unusual in modern compiler infrastructures (GOLDBERG, 2017, p. 28). Thus, it seems that, over time, the first alternative to UNCOL has been favored, since it encourages the development of reusable compiler parts and may also profit from advances in compiler-compiler technology.

2.3 A NEW GOLDEN AGE

In theory, UNCOL can be developed, and this is not up for debate (STEEL, 1961b). By Church's Law, the mere existence of an algorithm is enough to guarantee that one can produce a corresponding program for every universal machine (FRANZ, 1994, ch. 2). What remains are practical considerations such as UNCOL's impact on the quality and development cost of compilers and, arguably more importantly, on the efficiency of generated code.

Coincidentally, many properties originally attributed to UNCOL are expected of modern IRs and ILs: being general enough to represent many high-level languages, translate into multiple target architectures, represent the essential parts of any algorithm and having features designed to facilitate its generation and manipulation by compilers – as opposed to human programmers (CHOW, 2013). Furthermore, even the solutions noted as alternatives to UNCOL would benefit from its implementation. Compiler-compilers often require common ILs – more importantly, they rely on a well-known model of computation (LEE; PLEBAN, 1987) – to express the effects of machine operations (SCHMIDT; VÖLLER, 1984); the code generation problem then resolves around inverting that mapping (CATTELL, 1980). Additionally, the shared IR is a central component of modern compiler infrastructures like LLVM, which use it to communicate program information across separate modules (CHOW, 2013). Finally, any improvement in methods of compiler writing will also be absorbed by an UNCOL-based approach (STEEL, 1961b), so it becomes clear that the alternatives listed in Section 2.2 are not actually mutually exclusive. This is to say that UNCOL is still a viable solution to the compiler construction problem. In fact, although a universal IR may sound idealistic, Chow (2013) argues that the time is ripe to pursue the idea, which "holds promises for the entire computing industry".

Truthfully, much of the computing industry has changed since the inception of UNCOL, back in 1958. The field of programming languages – theory, design, implementation and application – has been in active research since its origins, motivating advances in compiler technology from the late 1950s to the present (MACRAKIS, 1993b). It is now common practice to develop portable software for abstract (as opposed to real) machines, which are usually defined in a written specification and may exist only as Virtual Machines (VMs) – machine emulation programs implemented on top of the actual computer (CALVERT, 2015, p. 11). Examples of

such abstract machines include the Java Virtual Machine (JVM), the execution system for the Common Language Infrastructure (CLI) and WebAssembly. Notice that there is no significant difference between an abstract machine, a model of computation (recall the definition from Section 1.1) and the specification of a VM. Hence, it often makes sense to complect an abstract machine with the IL used to program it (GANAPATHI; FISCHER; HENNESSY, 1982). One might then reinterpret the quest for UNCOL as the search for an abstract machine or model of computation which, along with a standard encoding, is suitable for tackling the compiler construction problem.

Meanwhile, the problem which UNCOL was set out to solve has never entirely disappeared. Evidence of this is the fact that a great number of programming language implementors choose to generate C as a portable compiler target, typically as a shortcut to avoid implementing back ends for multiple machines (MACRAKIS, 1993b; OLIVA; NORDIN; JONES, 1997; CHOW, 2013). Even if it did once disappear, the compiler construction problem seems to be about to resurge: Hennessy & Patterson (2019) predict a "Cambrian explosion" of domain-specific computer architectures in the next decade. They point out the need for domain-specific programming languages and related compiler technology. This was also evident to Lattner et al. (2021), who developed MLIR as a compiler infrastructure better suited to build domain-specific compilers in the upcoming "golden age" of computer architecture, hardware accelerators, programming languages and optimizing compilers (LATTNER, 2021).

In summary: the compiler construction problem is about to become more relevant than ever, UNCOL appears to be a viable solution, and, this time around, practitioners in the field have over 60 years of accumulated design experience and technological advances in programming language theory and compiler technology. This motivates taking another look at the compiler construction problem from an UNCOL perspective.

2.4 CHAPTER SUMMARY

This chapter defined the compiler construction problem and described possible solutions to it, UNCOL in particular. Through a retrospective overview on the history of UNCOL, its influence on programming languages and compilers has been unearthed. Finally, the motivations for the thesis were given and justified.

The first specific goal of this work is to argumentatively put forward UNCOL as a viable solution to the compiler construction problem, which is what has been done in this chapter. In addition, the thesis aims to identify requirements and design principles for modern iterations on UNCOL; and to use said requirements to review existing compiler IRs, looking for suitable UNCOL candidates. The two latter points are expanded upon in the following chapters.

3 A SURVEY ON INTERMEDIATE REPRESENTATIONS

Syntax without representation is tyranny. (SUSSMAN, 2011)[7]

This chapter presents a survey on portable compiler IRs (and their abstract machines). The UNCOL theme motivates a relatively wide (for a Bachelor's thesis) literature review scope, which has resulted in an extensive and diverse set of references. Thus, for the sake of brevity, basic familiarity with programming languages and compilers is assumed, although important concepts related to IRs will be explained throughout the chapter.

An overview on the topic of compiler IRs can be found in the ACM Queue article written by Chow (2013). Directly related works begin with Stanier & Watson (2013), with a first survey on IRs used in compilers for imperative languages. Belwal & TSB (2015) later published their research on IRs for heterogeneous multi-core computing. More recently, Susungi & Tadonki (2021) reviewed IRs designed to represent "explicitly parallel" programs. These previous surveys narrowed their scope to IRs used and designed for certain purposes; this one is no exception. UNCOL is a universal IR. Hence, special attention is given to portable IRs, that is, to abstract program representations which may be able to accommodate multiple high-level languages and target architectures.

Each of the three related works classifies IRs differently. Stanier & Watson (2013) use three aspects: structure, which can be linear or graph-based; dependency information, including data flow and control flow; and program content, whether programs can be fully or only partially encoded in the IR. The classification of Belwal & TSB (2015) is simpler: IRs can be "syntactic" (represented as trees), graph-based or a mix of the two previous options. Susungi & Tadonki (2021), on the other hand, consider four IR categories: ILs (recall the distinction described in Section 1.3), graphs, Static Single Assignment (SSA) form and polyhedral representations.

Meanwhile, this survey groups IRs solely by their structure: Sections 3.1, 3.2 and 3.3 respectively describe linear, tree-structured and graph-based IRs. Despite being separately classified, these kinds of IRs are frequently used together in modern compilers, albeit at different stages of the translation pipeline (refer to Figure 1). Stanier & Watson (2013) identify a tendency towards: (a) parse trees and Abstract Syntax Trees (ASTs) in the front end; (b) graph-based IRs to represent whole functions; and (c) linear IRs (usually put in SSA form during optimizations) used for local "low-level" transformations. Additionally, they notice the trend of using linear ILs as a human-readable format when parts of the compilers themselves are being developed and debugged.

Section 3.4 distills the requirements and design principles, identified during the literature review, which could be applied to a modern version of UNCOL. Some existing compiler infrastructures and their IRs are then reviewed in light of these requirements. Those are the main contributions of this survey, but one may consider yet another to be its consideration of functional and algebraic IRs, since these are generally not covered by related works.

[7] Pronounced by Gerald J. Sussman at the 2011 Strange Loop Conference, Language Panel; St. Louis, USA.

3.1 LINEAR REPRESENTATIONS

Linear IRs represent programs as sequences of instructions for an abstract machine. Imagining an IL in this format is particularly straightforward when the abstract machine resembles a simple stored-program computer. Thus, most ILs, including the first UNCOL ever proposed (CONWAY, 1958), follow this pattern (FRANZ, 1994, ch. 2). It is also the preferred IR type of VMs, since a baseline implementation can operate by interpreting an individual instruction, moving to the next one in the sequence and repeating this process in a loop, essentially emulating a sequential computer. This is why linear IRs are said to be "low-level" (LATTNER; ADVE, 2004). When interpreted by a VM, linear ILs are called bytecode formats, since each instruction can often be encoded in a single byte (NYSTROM, 2021, ch. 14). They are also known as P-code (portable or pseudo code), in praise of the VM-based implementation of the Pascal programming language (WIRTH, 1993).

Despite their simplicity, linear IRs are not without disadvantages. Haddon & Waite (1978) illustrate the challenge of choosing a minimal set of portable instructions. Their example shows how a "low-level" IL may introduce multiple levels of inefficiency during compilation:

> For some time, we included in the definition of Janus two integer division operators: `DIVN` (Result truncated towards Negative infinity) and `DIVZ` (Result truncated towards Zero). We reasoned that both were necessary, as it is expensive to synthesize one from the other. The argument went that if a high-level language specified one of these operations for its integer division, e.g. `DIVN`, and Janus contained only `DIVZ`, then a compiler would have to generate Janus code to simulate the effect of `DIVN`. But a given target machine may in fact have only a `DIVN` instruction. But the translator seeing only the `DIVZ` in the Janus text must generate code using the `DIVN` of the target machine to simulate the effect of the `DIVZ`. (HADDON; WAITE, 1978)

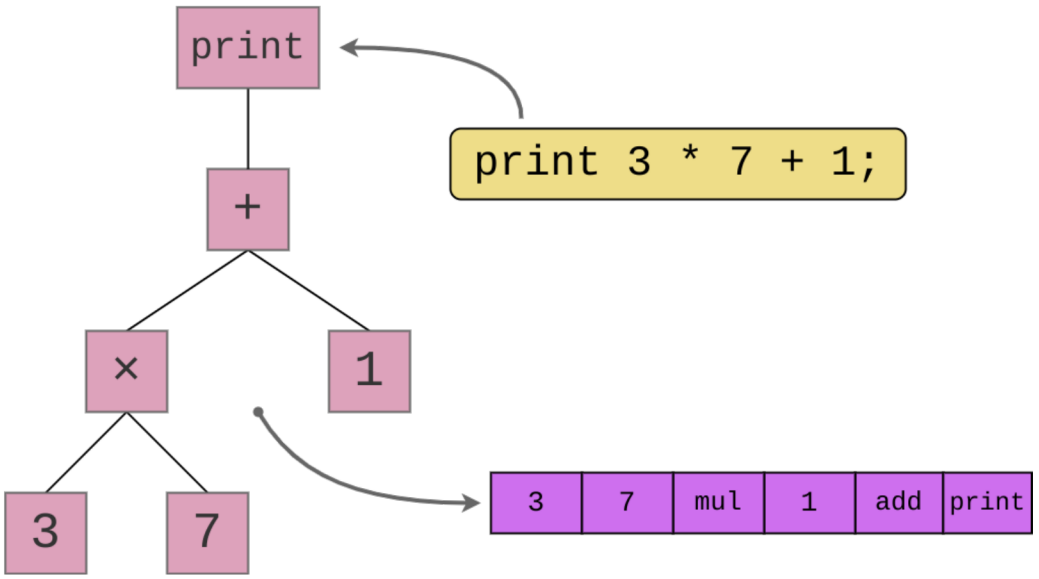
Recovering the original intent of high-level code after it has been lowered to a linear IR is not trivial, especially after multiple rounds of program transformations (LOGOZZO; FAHNDRICH, 2008). Furthermore, this "primitive mismatch" problem (which also extends to data types, number of available registers and properties of memory operations, among other aspects) is unavoidable: unless the IR is to expand uncontrollably, there will always be programming language operations and machine instructions without a one-to-one mapping in the IR (GANAPATHI; FISCHER; HENNESSY, 1982).

Similar to instruction formats of real computers, linear compiler ILs have a simple, regular structure. Also like the instructions of real machines, the biggest difference between them lies in what the instructions themselves do: what sort of data they operate on, how they access memory and what effects are caused by their execution (STANIER; WATSON, 2013). Despite the great variety of possible instructions, linear compiler IRs can be categorized as either stack-based or register-based (IERUSALIMSCHY; FIGUEIREDO; CELES, 2005).

3.1.1 Stack-based Bytecode

Another way to conceive linear IRs is by attempting to serialize syntax trees into a more densely packed format. For example, a simple compiler for arithmetic expressions – such as the one described by Aho et al. (2006, ch. 2) – can traverse ASTs in post-order and emit bytecode at each visited node; the resulting program would be ready to be executed by a stack-based VM – like the one described by Nystrom (2021). Figure 5 illustrates how this process can be applied to a short program, which is parsed into an AST and serialized as a sequence of stack-based bytecode instructions. Stanier & Watson (2013) would designate such an IL as an extension of postfix Polish (or, reverse Polish) notation. These IRs are very compact, but become difficult to optimize (STANIER; WATSON, 2013).

Figure 5 – Translating arithmetic to stack-based bytecode.



Source: Gabriel B. Sant'Anna, 2022.

Koopman (1989) presented a stack computer taxonomy which can be adapted to stack-based compiler IRs. One dimension of the design space is the number of distinct stacks available to the machine: single-stack machines are simpler and VM implementations can be faster, but having multiple stacks may help prevent certain classes of security vulnerabilities by separating data and control flow manipulation (HAAS et al., 2017). The other significant aspect is the number of operands an instruction may have: while a 0-operand "pure stack" architecture is almost always manipulating the topmost stack elements, 1-operand machines – also known as single-address or "stack/accumulator" architectures – designate an additional machine register or memory address as either input or output to each instruction (KOOPMAN, 1989, ch. 2).

The following is a non-exhaustive list of stack-based compiler IRs:

- **The SECD machine.** Landin (1964) designed an abstract machine to evaluate λ-calculus terms. Its 0-operand IR controlled four distinct memory devices, two of which were

stacks. Later, the SECD machine was used as a central component of the LispKit compiler (HENDERSON, 1980) and there was an effort to implement it directly on hardware (GRAHAM, 1989).

- **Janus.** Janus was an IL which shared most of UNCOL's goals. Coleman, Poole & Waite (1974) describe the system as a family of abstract machines with a common basic structure (a 1-operand stack-based architecture) and a universal IL. They achieved portability through the abstraction of high-level operations as macros, which are called in the IL and expanded in a target-specific manner when lowering to machine language (NEWEY; POOLE; WAITE, 1972). This technique was rediscovered some years later, in the "Universal Compiling System" proposed by Gyllstrom et al. (1979), although they do not describe their IR in much detail. Janus was used in the implementation of a Pascal compiler and in the design of an Algol 68 compiler (HADDON; WAITE, 1978).

- **U-code.** A variant of P-code designed by Perkins & Sites (1979) to enable compiler optimizations across a wide variety of machines and languages. The 1-operand stack-based IL was used to compile both Pascal and Fortran programs and had some degree of backwards compatibility with P-code–based compilers (PERKINS; SITES, 1979). U-code had dedicated metadata instructions for the purposes of debugging, guiding optimizations and stating program requirements (e.g., defining memory alignment restrictions).

- **The Amsterdam Compiler Kit IL.** The Amsterdam Compiler Kit was a collection of compiler modules created by Tanenbaum et al. (1983), based on the UNCOL idea. Their main IR, Encoding Machine (EM) language, is linear, but the tool kit also included graph-based global optimizations. As its name suggests, the stack-based IL was used mainly as an encoding mechanism, where the back end maintained a simulated stack and generated code with the aid of a target-specific "driving table", which described how to translate certain stack configurations into machine language. Lacking an entry in the driving table, the compiler would inject IL instructions into the program with the purpose of coercing the stack into a known configuration. Tanenbaum et al. (1983) describe this method as being analogous to a chess-playing artificial intelligence program, since it searches through many possible sequences of "moves". There were also meta-linguistic declarations, not unlike those previously suggested by Steel (1961a), which helped guide the "chess-playing" procedure. The Amsterdam Compiler Kit, originally a commercial product, was used for many years as the native compiler toolchain in Tanembaum's Minix operating system (GIVEN, 2005), with support for C, Plain, Pascal, Algol 68 and more than a dozen different hardware architectures. According to (TANENBAUM et al., 1983), it showed that "the old UNCOL idea is a sound way to produce compilers". The toolkit has since been released under an open-source license, but has not received much attention.

- **The Categorical Abstract Machine.** The Categorical Abstract Machine was created as a mathematically well-founded method of functional programming language imple-

mentation (COUSINEAU; CURIEN; MAUNY, 1987). The core stack-based architecture was formally described by Cousineau, Curien & Mauny (1987), who claimed the small changes proposed over the SECD machine enabled much simpler correctness proofs. It was used in an ML implementation which eventually originated the OCaml programming language (MACQUEEN; HARPER; REPPY, 2020).

- **JVM bytecode.** Many qualities associated with the Java programming language, such as machine-independence and memory safety, stem from its VM (LINDHOLM et al., 2015, ch. 1). The JVM is a portable stack-based and object-oriented architecture specifically designed for Java. Nonetheless, compilers for other programming languages (Kotlin, Scala and Clojure, to cite a few) frequently target JVM bytecode in order to profit from the mature platform. Since most JVM implementations employ Just-In-Time (JIT) compilers, which translate bytecode to machine language at run-time, JVM bytecode can be considered an IL. Still, compilers often convert the stack-based IR into a format better suited for transformations before lowering it to machine code (LATTNER; ADVE, 2004).

- **Microsoft's Common Intermediate Language (CIL).** An Ecma International standard defines the CLI (ECMA-335, 2012), which includes a common type system, a stack-based virtual execution system and an accompanying IL, the CIL. The machine-independent infrastructure was built with maximum portability in mind; in order to integrate multiple programming languages in the same platform, it was "designed to be large enough that it is properly expressive yet small enough that all languages can reasonably accommodate it" (ECMA-335, 2012, p. 14). One way the CLI attempts to deliver practical language interoperability is by providing abstract operations (such as a "virtual calling convention") and extensible IL metadata that let compiler front ends communicate optimization-relevant information all the way to the execution engine, which can then tailor code generation decisions to the user's machine. In general, such on-the-fly code generation methods aim to counteract the efficiency losses which tend to arise when using machine-independent IRs (FRANZ, 1994, ch. 7). The standard also defines a subset of the CLI which is verifiable with respect to memory safety (ECMA-335, 2012, p. 14).

- **WebAssembly.** WebAssembly is a VM-bytecode pair for the Web platform. Despite currently being in a minimum-viable-product state, it was collaboratively designed by engineers from major browser vendors (HAAS et al., 2017) and therefore boasts wide availability on consumer devices. The stack-based architecture is explicitly noted by Haas et al. (2017) as merely a mechanism to easily achieve a formalization of the abstract machine and provide a compact program representation. Hence, many design decisions – notably, prohibiting irreducible control flow – aim to enable quick conversion into IRs better suited for program transformations and compiler optimizations, such as a register-based SSA (HAAS et al., 2017).

3.1.2 Register-based Three-Address Codes

Register-based compiler IRs are often synonymous with Three-Address Codes (3ACs), although alternative architectures exist (and are mentioned below). In a typical 3AC, each instruction performs a single operation, having at most two inputs and one output (AHO et al., 2006, ch. 6). An example is shown in Figure 6, which displays the small program from Figure 5 encoded in a 3AC IL.

Figure 6 – A simple 3AC program, using LLVM-like syntax.

```
%t1 = i32 1
%t3 = i32 3
%t7 = i32 7
%tm = mul %t3, %t7
%ta = add %tm, %t1
%_ = call @print(%ta)
```

Source: Gabriel B. Sant'Anna, 2022.

In most register-based IRs, the compiler assigns a named register to all intermediate results. This has the advantage of facilitating the reuse of previously computed values: one simply needs to refer to the name associated with it, as opposed to having to perform a series of memory and stack manipulation operations (KOOPMAN, 1989, ch. 3). A disadvantage lies in the fact that the compiler *needs* to allocate a register for every single intermediate value in the program: if the abstract machine has a limited number of registers, this becomes a non-trivial problem (AHO et al., 2006, p. 505). According to Aho et al. (2006, p. 363), the use of names in 3ACs allows programs to be transformed and rearranged much more easily than would be the case with stack-based bytecodes, since the latter requires the insertion of instructions in a very specific order to achieve the effects described by the original high-level program.

There are many ways for a compiler to implement 3ACs. The most straightforward representation consists of quadruples: sequences of four-field records containing the instruction code, (up to) two argument registers and a named result (AHO et al., 2006, p. 366). Triples are a close alternative, where the result's "name" is implicitly defined by the position of the instruction in the containing sequence. The triple scheme requires less space and lets the compiler use numbers (which are faster indexes and easier to generate) instead of arbitrary names (STANIER; WATSON, 2013). Since triples associate instruction indexes with their produced results, reordering operations requires the compiler to update all uses of the result of a relocated triple. Indirect triples offer a solution to that problem by uncoupling program order from result handles, such that a reference to a triple is kept stable even when the program is being transformed (AHO et al., 2006, p. 368). Figure 7 shows a single program (the same one from Figure 6) represented as quadruples, triples and indirect triples. In the last representation, the program was reordered so as to load the constant `1` after the `mul` instruction.

Figure 7 – Quadruples (left), triples (center) and indirect triples (right).

```
PROGRAM:                          PROGRAM:                          0: <const, i32, 1>
  <const, i32, 1,   %t1>            [0: const, i32,  1 ]            1: <const, i32, 3>
  <const, i32, 3,   %t3>            [1: const, i32,  3 ]            2: <const, i32, 7>
  <const, i32, 7,   %t7>            [2: const, i32,  7 ]            3: <mul,   (1), (2)>
  <mul,   %t3, %t7, %tm>            [3: mul,   (1), (2)]            4: <add,   (3), (0)>
  <add,   %tm, %t1, %ta>            [4: add,   (3), (0)]            5: <print, (4), _>
  <print, %ta, _,   _>              [5: print, (4),  _ ]          PROG': [ 1; 2; 3; 0; 4; 5 ]
```

Source: Gabriel B. Sant'Anna, 2022.

Register-based compiler IRs include, but are not limited to:

- **Conway's UNCOL.** In the first proposed UNCOL design, Conway (1958) partially describes a register-based IR. He believed that minimalistic abstract machines, with fewer registers and instructions, would provide greater opportunity for code optimization. Thus, his IR follows a single-address architecture, where operations specify only their output locations, and operands are always read from predefined "argument-storage" registers.

- **Steel's UNCOL.** As one of the authors of the UNCOL report, Steel (1961a) provides further detail on how a universal IR could be envisioned. Like Conway, Steel points towards a single-address machine. He indicates the need to describe, for every item of data expressed in the IL, properties such as type, numerical range, precision and storage allocation, arguing that (at least some of) these could be achieved with a mechanism to define the lexical and syntactical structure of language elements[8]. Steel also suggests that some instructions should aid compiler optimizations through meta-linguistic declarations such as "the next 20 statements form a loop". Finally, a striking feature of Steel's UNCOL was that commands could have different behavior based on the data types being operated on. This type-based instruction overloading, not unlike the one used in the CIL, results in a smaller IR: a single addition command, for example, would work with both integers and floating-point numbers. Like Conway's UNCOL, this IL was never implemented.

- **IBM's PL.8 compiler.** The PL.8 project was a compiler which could accept multiple source languages (Pascal and a variant of PL/I) and produce optimized code for several different machines, notably by the means of a common IR "closely matching the computational semantics of the target machines" (AUSLANDER; HOPKINS, 1982). Its IR was a 3AC with an unlimited number of virtual registers. This is a notable feature of the IR's design because it may have prompted the invention of SSA form, especially since Rosen, Wegman & Zadeck (1988) were familiar with IBM's PL.8 infrastructure.

[8] Classic formal language techniques were relatively novel at the time (MACRAKIS, 1993b), so Steel (1961a) proposed using first-order logic – and not grammars or regular expressions – to describe syntax.

- **The Stanford University Intermediate Format (SUIF).** SUIF was a compiler framework heavily influenced by the concept of a universal IR. It is divided into a "kernel", which consists of IR generation and manipulation procedures, and a "toolkit" with front end and back end modules using the IR as a common interface (WILSON et al., 1994). SUIF's IR is described as a "mixed-level" program representation, being for the most part a 3AC, but also including tree-like constructs for loops, conditionals and array accesses (WILSON et al., 1994). The infrastructure was released for free as an attempt to solve the compiler construction problem in research. It succeeded to some degree: there were multiple research projects based on SUIF, with front ends implemented for C, Fortran, Java and C++ (SUIF, 2005). However, its place in research has since been occupied by LLVM (FARVARDIN; REPPY, 2020).

- **GCC's Register Transfer Language (RTL).** GCC is another well known compiler collection and infrastructure, notable for being used in the Linux kernel (FSF, 2022). According to Lattner (2021), GCC's open-source nature created a turning point in the compiler industry around the 1990s (back in a time when most compilers where commercial products, all mutually incompatible), enabling collaboration from academia and industry, reducing language community fragmentation and to a great extent achieving the $N + M$ scaling of multiple front ends and back ends. GCC officially supports C, C++, Objective-C, Fortran, Ada, Go and D, for countless back ends (FSF, 2022). Still, modern compilers tend to gravitate towards the LLVM infrastructure, mainly due to its modular architecture and better-specified (even if slightly unstable) IL. GCC has multiple IRs (MERRILL, 2003), the oldest one being RTL, a 3AC with an infinite number of virtual registers. Unlike LLVM IR, RTL's SSA form is optional (FSF, 2022, ch. 14).

- **The LLVM Project (and derivatives).** LLVM is a modern compiler framework originally designed by Lattner (2002) to tackle the challenges originated by recently developed programming practices, namely the support for dynamic extensions to long-running programs and multi-language software development. A solution is envisioned as the combination of capabilities previously considered to be mutually exclusive: while classic compilers provide ahead-of-time code optimization and a language-agnostic runtime model, VM-based approaches such as the JVM or the CLI can implement profile-guided JIT transformations by preserving program information until run-time (LATTNER; ADVE, 2004). Providing all of these features is LLVM's goal, which, according to Lattner & Adve (2004), is achieved through its innovative IR design and the open-source compiler infrastructure built around it. LLVM's IR can be described as a register-based 3AC (although the implementation places the linear IR inside Control Flow Graph (CFG) nodes) in SSA form, extended with "high-level" type information. The LLVM infrastructure has been successfully applied in academia and in the industry, with countless programming languages implemented on it due to its maturity and ability to emit optimized code for many target architectures (LLVM, 2022). Although earlier versions of LLVM were quite

minimalistic, with only 31 opcodes (LATTNER; ADVE, 2004), it has been in a constant state of change and expansion: the current version (LLVM 13) of the IL reference documentation (LLVM, 2022) can fill over 300 printed pages. This instability and constantly increasing complexity has led some groups to create similar IRs, isolated on purpose from the main LLVM infrastructure. Examples include Khronos' SPIR-V (KHRONOS, 2020), which can be described as a fork of LLVM IR specialized for parallel computation and graphics processing, and Cranelift, which has been considered as a faster (in terms of compile time) alternative to LLVM for debug builds (LARABEL, 2020).

- **The Join Calculus Abstract Machine.** Calvert (2015) proposes using the join calculus as the foundation for a parallel abstract machine and LLVM-like linear compiler IR. He argues that the resulting design could be used as a universal IR, especially since, among other members of the process calculi family, the join calculus seems to be particularly suitable for the efficient implementation of programming languages (FESSANT; MARANGET, 1998; FOURNET; GONTHIER, 2000).

- **Heterogeneous System Architecture (HSA) IL .** HSA IL is described as a low-level virtual instruction set for the architectures supported by the HSA Foundation (HSA, 2018). According to Chow (2013), although the register-based IL's extensive specification could be a step forward in what concerns universal IRs, the complexity of the HSA programming model makes it unlikely that compilers will use it as anything other than a target for HSA-compatible devices. For instance, a limited number of registers (HSA, 2018) implies that most program transformations may need to invoke a register allocator.

3.2 TREE-STRUCTURED REPRESENTATIONS

Trees are probably the single most common compiler data structure. Although mainly utilized to represent the syntactic structure of high-level languages, tree-based IRs are also used for macro expansion, type checking and some program transformations (AHO et al., 2006, ch. 6). A simple high-level program represented as an AST has been shown in Figure 5.

3.2.1 Generic Tree Encodings

Trees are generally considered language-dependent compiler IRs, since each node in the tree normally corresponds to a specific programming language construct (AHO et al., 2006, sec. 2.5.1). Still, a few works propose language-independent tree-structured ILs:

- **Architecture Neutral Distribution Format (ANDF).** ANDF has been described as an "architecture- and language-neutral distribution format resembling a compiler intermediate language", aiming to succeed where UNCOL proposals had previously failed (MACRAKIS, 1993b). ANDF was essentially a compact binary format used to encode

program trees in a standard manner. Defined by the Open Software Foundation, the system took a very pragmatic position on the problem of portability: "ANDF is not a tool for making non-portable software into portable software [an unsolvable problem], but a tool for distributing portable software" (MACRAKIS, 1993b). The major mechanism used to achieve practical language and machine independence was a macro system reminiscent of Lisp's syntax macros. In ANDF, static conditional branches, identifier references, constant literals, type layout and even other macros could be identified and stored separately from the main program. This allowed the system to retain portability and preserve the potential for both language- and machine-specific optimizations, since such delayed definitions could be "linked in" and expanded only when appropriate, up until the moment the program was loaded (MACRAKIS, 1993c). ANDF was only ever implemented in UNIX systems, and for the C programming language (MACRAKIS, 1993a). The implementation, documented as TenDRA, was later released as an open-source compiler framework (TenDRA, 2022). Stavros Macrakis (personal communication, 2021) suggests ANDF was not widely adopted mainly because of: (a) lack of interest from UNIX vendors; and (b) the compiler construction problem becoming less relevant as the computer market was dominated by specific (software and hardware) systems.

- **Semantic Dictionary Encoding.** Proposed by Franz (1994), "semantic dictionaries" can encode general ASTs in a compact tabular form. It was used as Oberon's module interchange format, also known as "slim binaries". It achieved portability by distributing high-level source programs, while the "fat binary" alternative was to package multiple machine-specific objects in the same executable file (FRANZ; KISTLER, 1997).

- **The ROSE compiler.** ROSE is an open compiler infrastructure for the development of program analysis tools and source-to-source transformations. Given its goals, a generic tree-based IR is a natural fit. The framework has been used to develop tools and research projects for C, C++, Fortran, Java, Python and PHP, among others (LLNL, 2022).

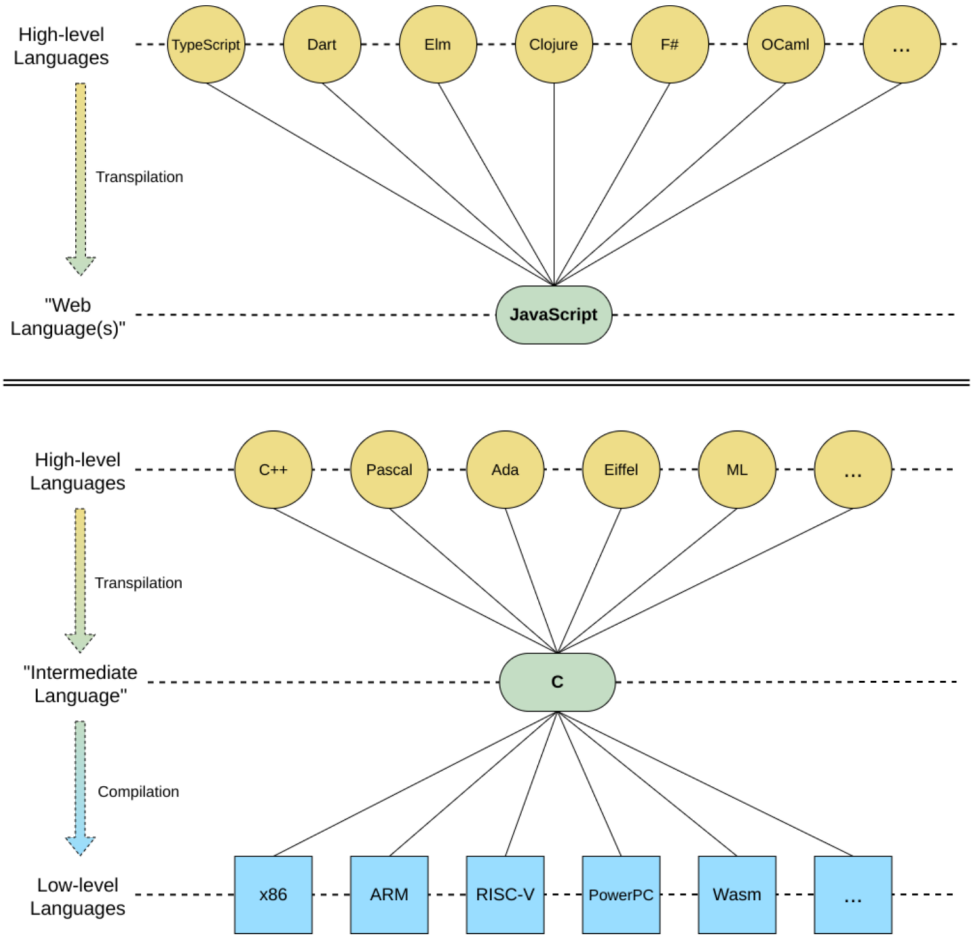
3.2.2 Programming Languages as Compiler ILs

Some high-level programming languages have been used as compiler ILs. Instead of generating machine code (or VM bytecode) directly, compilers can emit a program encoded in another high-level language[9] and then call one of its existing back ends (or use an existing VM implementation). Typically, this is done to avoid re-implementing architecture-specific transformations and code generation (OLIVA; NORDIN; JONES, 1997), but also provides a way to reuse developer tools (debuggers, for instance) in a newly-created programming language. This approach is hereby grouped with other tree-structured IRs, since trees are the standard computer-oriented representation of high-level languages.

[9] Compilers translating one high-level language into another are also called "transpilers" (POUNTAIN, 1989).

Although not the only programming language to have been repurposed as a compiler target, C is the most notorious. It has been used in countless implementations of other high-level languages, including C++, Ada, Cedar, Eiffel, Modula-3, Pascal, Standard ML (MACRAKIS, 1993b), Scheme and Haskell (OLIVA; NORDIN; JONES, 1997). Alan Kay even considers C to have become a "*de facto* UNCOL" (FELDMAN, 2004), since the standard is portable and compilers are available on most platforms. Another programming language being targeted by various compilers is JavaScript. According to Haas et al. (2017), this is mainly due to "historical accident", since JavaScript was (until the very recent advent of WebAssembly) the only computer language with standardized support in the Web platform. Figure 8 illustrates the use of C and JavaScript as compiler targets.

Figure 8 – JavaScript's monopoly and C's ubiquity.



Source: Gabriel B. Sant'Anna, 2022.

High-level programming languages are designed to be written and understood by humans. This leads to a set of design decisions which make them undesirable as compiler IRs (CHOW, 2013): text encodings are far less compact – and also much harder to parse and transform – than binary formats (HAAS et al., 2017); some high-level languages simply lack the mechanisms to perform lower-level operations needed by other languages (JavaScript, for example, does not have integers); and the semantics of a "target" programming language are more

often than not incompatible with those of other "source" programming languages. This last point has been noted as the main reason not to use C as a compiler IL.

Oliva, Nordin & Jones (1997) argue that C is not suitable to represent functional languages, mainly because C compilers do not ensure Tail Call Optimization (TCO) (BAKER, 1995), lack precise garbage collection and cannot encode – neither discover by themselves – some of the program properties guaranteed by functional languages (such as data immutability or pointer aliasing restrictions) which would otherwise enable more optimizations during code generation. Furthermore, Macrakis (1993b) considers C to be under-specified: aspects which would be relevant to an optimizing compiler, such as supported integer ranges or the behavior of out-of-bounds array accesses, are either implementation-defined or not defined at all. Of course, Turing-completeness means that none of this prevents a transpiler from generating portable C programs which emulate the abstract machine exposed by other languages. This, however, would introduce unacceptable performance degradation when the models of computation do not match (HADDON; WAITE, 1978). According to Macrakis (1993b), emulating another high-level language requires C generators to commit to a concrete implementation of every single construct in the source language; whichever strategy is chosen will almost certainly not be appropriate for all target architectures and cannot be adapted by the C compiler being used as a back end, since it knows nothing about the higher-level source language.

In this category of compiler ILs, a final candidate is Lisp:

> A carefully chosen small dialect of LISP would be a good UNCOL, that is, a good intermediate compilation language. [...] Up to now, the UNCOL idea has failed; the usual problem is that proposed UNCOLs are not sufficiently general. I suspect that this is because they tend to be too low-level, too much like machine language. I believe that `LAMBDA` expressions, combining the most primitive control operator (`GOTO`) with the most primitive environment operator (renaming) put LISP at a low enough level to make a good UNCOL, yet at a high enough level to be completely machine independent. (STEELE, 1976)

Steele (1978) implemented this idea in his Scheme compiler, Rabbit, a seminal work presenting the first use of Continuation-Passing Style (CPS) in a compiler. CPS is a representation in which control points are explicitly named, manipulated and invoked (STEELE, 1976). Steele's proposal relies on Scheme's guaranteed TCO and its strong proximity to the lexical-scoped applicative-order λ-calculus. These qualities, combined with Lisp macros and an imperative assignment operator, allow for a straightforward implementation of most programming language constructs; thus, Scheme was its own IL (STEELE, 1976).

The Rabbit compiler operated through source-to-source transformations, lowering high-level functional Scheme code into an imperative-style CPS IL, which happened to be a subset of Scheme itself. Further research on the topic of compiling with continuations adopted the same source-to-source explanation of CPS (FLANAGAN et al., 1993; KENNEDY, 2007; MAURER

et al., 2017; CONG et al., 2019). Although Steele's work was primarily research-oriented, Adams et al. (1986) used its design as a base for their optimizing Scheme compiler, Orbit. This later work proved that CPS transformations could be an effective tool in optimizing compilers, since Orbit was used to produce code competitive with C, Pascal, Modula-3 and Lisp systems of the time (ADAMS et al., 1986).

At last, it should be noted that Lisp's syntax appears to be more suitable for a compiler IL than that of most other programming languages, since it trivially maps to ASTs[10]. Regardless, Macrakis (1993b) points out that Scheme as a compiler IL shares some flaws with other high-level language options, notably due to Lisp systems' reliance on garbage collection and lack of concrete data definition facilities – both properties shared by the untyped λ-calculus.

3.2.3 Algebraic ILs

Another class of representations whose use in compilers has been proposed consists of "algebraic" ILs. These are formal languages from the lineage of term rewriting systems (e.g., mathematical logic), which have become foundational for programming language and type theory (PIERCE, 2002, ch. 5). While there have been operational machine-like approaches (such as the SECD machine, mentioned in Section 3.1) to defining these models of computation, the usual method is more syntax-directed, with ASTs used to express programs, as well as their execution (HARPER, 2016, sec. 5.5). Manipulating code through an algebraic representation has the advantage of enabling the application of well-known mathematical proof methods to programming languages (PIERCE, 2002). The λ-calculus is one such algebraic language; its syntax is displayed in Figure 9. It could also be considered a family of languages (or calculi), due to its many different extensions and typed variants (PIERCE, 2002).

Figure 9 – The λ-calculus (left) and the π-calculus (right).

$M, N ::=$	terms
x	variable
$\lambda x.M$	abstraction
$M\ N$	application

$P, Q ::=$	processes
$\mathbf{0}$	null process
$\nu x.P$	new local channel x
$x\langle y\rangle.P$	receive message y on x
$\bar{x}\langle y\rangle.P$	send message y on x
$\tau.P$	silent action
$P \mid Q$	parallel composition
$P + Q$	choice operator
$!P$	replication

Source: Gabriel B. Sant'Anna, 2022.

Milner (1993) has claimed that, although λ-functions are "an essential ingredient" of sequential program semantics, handling concurrency requires a different framework. Albeit

[10] As explained by McCarthy (1978), this is by design. The infamous S-expression notation, which ended up becoming Lisp's standard programming syntax, was originally created as a computer-oriented representation.

notoriously hard for humans to reason about, concurrent programs are an important part of modern computing (PESCHANSKI, 2011). This had already been predicted in the 1950s:

> Everyone looks with dread at the possible computers of the next decade, which will be simultaneously executing multiple asynchronous stored programs. There is little reason to expect a reversal of this trend. (STRONG et al., 1958a)

The need for mechanisms to formally reason about concurrent processes has fostered the family of process algebras (or, process calculi), notably including Milner's π-calculus (also shown in Figure 9) and its many variations (PALAMIDESSI, 1997). According to Fournet & Gonthier (2000), calling these formal systems calculi implies in the possibility of using them for equational reasoning (i.e., to calculate). This requires a notion of "equality", an operational equivalence relation between programs, and many such relations have been proposed for process calculi (FOURNET; GONTHIER, 2000). These are particularly interesting to compiler writers, who must ensure that transformations and optimizations preserve program equivalences.

Algebraic ILs are usually associated with compilers of functional programming languages, possibly due to their strong relation with the λ-calculus. Examples include:

- **π-threads.** Peschanski (2011) proposes a variation of the π-calculus as a parallel abstract machine model and compiler IL. He provides some safety proof outlines in the resulting model, but no benchmarks are shown.

- **Haskell Core.** Haskell's main IL, Core, is based on System Fω, a typed variant of the λ-calculus (DOWNEN et al., 2016). The compiler maintains this functional IL in Administrative Normal Form (ANF), an alternative to CPS which admits many of the same transformations (MAURER et al., 2017).

- **Sequent Core.** Under the Curry-Howard correspondence, the classic λ-calculus is equivalent to Gentzen's natural deduction logic (WADLER, 2015). However, natural deduction was defined along with another reduction system, the sequent calculus (GENTZEN, 1964). Downen et al. (2016) describe Sequent Core as an experimental Haskell IL based on sequent calculi. The result, as Cong et al. (2019) observe, comes closer to CPS form than to Core's traditional ANF, enabling more optimizations (DOWNEN et al., 2016).

3.3 GRAPH-BASED REPRESENTATIONS

Directed graphs (or, just graphs) are one of the most pervasive structures in Discrete Mathematics and Computer Science (CORMEN et al., 2009, ch. VI). A graph can be defined as a pair (N,E), where N is a set of nodes (or, vertices) and $E \subseteq (N \times N)$ a set of edges (alternatively, arcs), consisting in ordered pairs of nodes. A path in a graph is any sequence of nodes $[v_0,...,v_n]$ such that every consecutive pair (v_i,v_{i+1}) is an edge in E. When all paths beginning at node v_0 and eventually reaching node v_n pass through an intermediate node v_d, it is said that v_d dominates v_n with respect to v_0. Furthermore, any two nodes (a,b) are said to be strongly connected if there is a path (however long) from a to b; when such a path can only be formed by adding reverse edges to the graph (that is, adding edges (v,u) for any subset of (u,v) already present in E), a and b are said to be weakly connected; otherwise they are disconnected. By extension, an entire graph can be called strongly (or weakly) connected when all of its nodes are strongly (resp. weakly) connected to all other nodes in the graph. A Strongly Connected Component (SSC) in a graph $G = (N,E)$ is any subgraph $G' = (N' \subseteq N,\ E' \subseteq E)$ which is itself a strongly connected graph. Finally, if a directed graph contains at least one node strongly connected to itself through a non-trivial (i.e., non-empty) path, the graph is said to be cyclic, as this implies the existence of a path starting at that node and eventually circling back to it; otherwise, it can be called a Directed Acyclic Graph (DAG). Offner (2013) details the applications of graphs in optimizing compilers.

Graphs are very general mathematical structures: nodes in a graph may be associated to objects of any kind, and edges can be used to represent arbitrary relations between such objects. In the context of compilers, graphs have been used to represent various relations between parts of programs[11], such as "flows to" (control flows from a subroutine to the next), "depends on" (an instruction's result depends on its arguments), "calls" (e.g., procedure `foo` may call procedure `bar`), "happens before" (one memory operation happens before the other), etc (STANIER; WATSON, 2013; SHIVERS, 1988; ADVE; BOEHM, 2010). Each distinct set of relations being expressed gives rise to a different graph-based IR.

3.3.1 Control Flow Graphs

Control Flow Graphs (CFGs) are classic IRs used in optimizing compilers to partition procedures into nodes and express control flow between them (AHO et al., 2006, ch. 525). Nodes in a CFG are "basic blocks", which, according to Aho et al. (2006, p. 525), consist in linear instruction sequences with a single entry and no exit points until the very last instruction. Then, graph edges are used to link basic blocks to every other node the program could branch off to by the end of that instruction sequence. This leads to the classification of compiler optimizations as either local (to a basic block) or global (but still within the same procedure) (AHO et al., 2006, ch. 533). Figures 10 and 11 show a C program and an equivalent CFG.

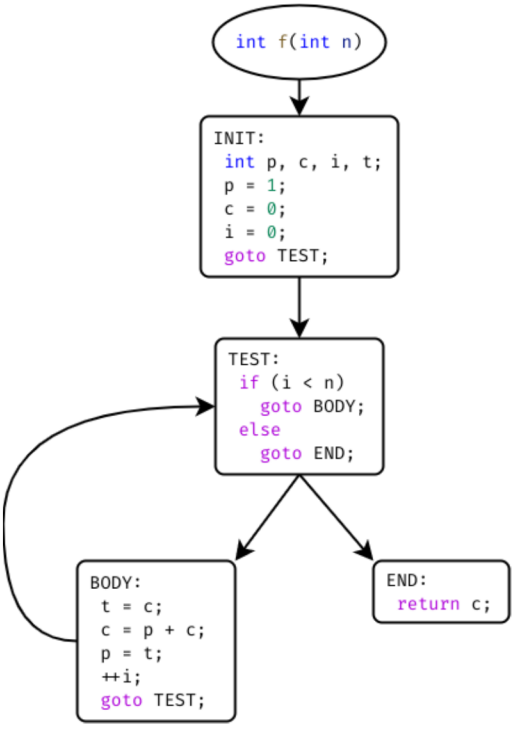
[11] In fact, trees are also a limited kind of graph. Still, this survey treats them as separate IR structures.

Figure 10 – C program with an input-controlled loop.

```
1  int f(int n)
2  {
3      int p = 1;
4      int c = 0;
5
6      for (int i = 0; i < n; ++i) {
7          int t = c;
8          c = p + c;
9          p = t;
10     }
11
12     return c;
13 }
```

Source: Gabriel B. Sant'Anna, 2022.

Figure 11 – A CFG, equivalent to Figure 10.



Source: Gabriel B. Sant'Anna, 2022.

As exemplified in Figure 11, cyclic CFGs can represent the branching behavior of programs with loops. However, not all loops are created equal. Computer languages with `goto` or equivalent constructs may be used to write programs displaying irreducible control flow (STANIER; WATSON, 2012). Irreducible programs can be identified as such when (a) their CFG contains a cycle and (b) there is no node in the cycle which dominates all others (in the cycle, with respect to the procedure's entry point). More informally, irreducible control flow is caused by "jumping inside a loop" (UNGER; MUELLER, 2002). However, even in programming languages without `goto`, this property may "naturally" arise as a result of program transformations and compiler optimizations (STANIER; WATSON, 2012).

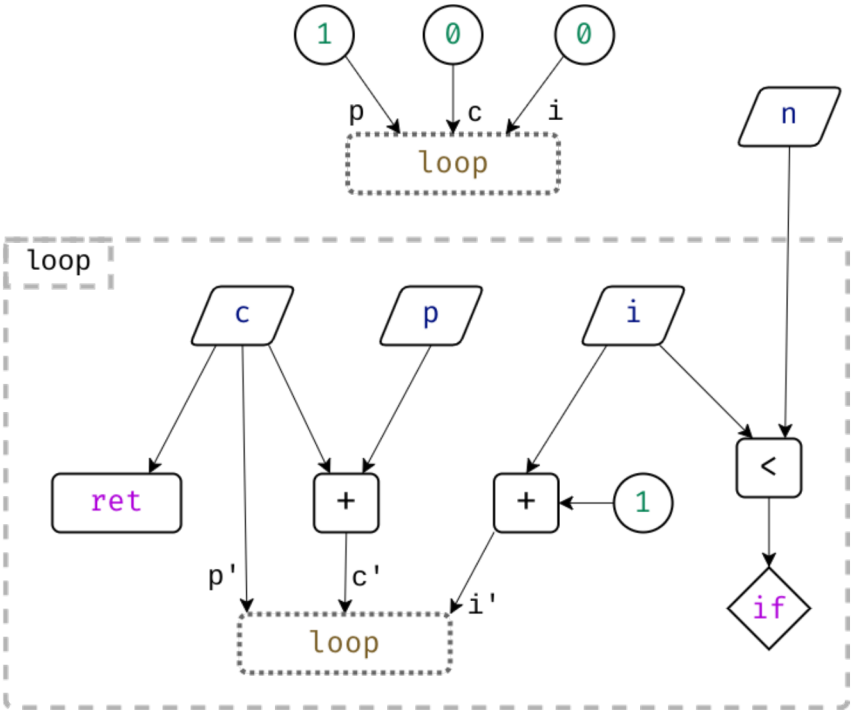
Irreducibility is often considered a problem because many compiler algorithms either do not work or have different asymptotic complexity in its presence (STANIER; WATSON, 2012). For this reason, some IRs (e.g., WebAssembly) make irreducible control flow impossible to represent. Therefore, compilers which lower optimized programs into an IR (or programming language, in JavaScript's case) which cannot represent irreducible control flow have to implement non-trivial workarounds (UNGER; MUELLER, 2002). Although standard techniques to do so exist in the compiler literature (AHO et al., 2006, sec. 9.7.6), they either cause an exponential size blowup at the IR level (CARTER; FERRANTE; THOMBORSON, 2003) or require complex analyses to modify the program, adding guard predicates and state variables (ZAKAI, 2011; BAHMANN et al., 2015). Lastly, although irreducible flow can be detected and eliminated within individual procedures, doing so across function calls is not possible in general (SHIVERS, 1988).

3.3.2 Data Flow Graphs

CFGs are chiefly concerned with control flow, the "shape" of a program. They do not directly represent data flow information, especially not across basic blocks. Hence, Data Flow Graphs (DFGs) are the traditional approach to representing relations between definitions and uses of program values – also known as "def-use" chains (STANIER; WATSON, 2013). Figure 12 shows a DFG derived from the previously discussed C program.

Since data dependencies are generally expected (and often required) to be acyclic, so are DFGs. In most computer languages, this raises the problem of having to express many different values associated with the same variable name. For instance, imperative assignments such as `c = p + c` (as seen in Figure 10, line 8) would apparently require a cyclic DFG, since data stored in a variable may depend on its own previous definition. Fortunately, there are techniques (e.g., SSA form) which can be used to "break" apparent data flow cycles.

Figure 12 – Acyclic DFG for the looping program of Figure 10.



Source: Gabriel B. Sant'Anna, 2022.

Most programs cannot be represented by a DFG alone. This becomes clear when comparing Figure 12 with the original program: the illustrated DFG cannot represent the relation between the conditional form (indicated by the node labeled as "`if`") and its mutually-exclusive branches (one which returns from the function and the other which continues its internal computation), nor does it clearly indicate that nodes `p, c, i` correspond to inputs of the `loop` node (whose body is outlined within Figure 12 for the sake of clarity). Therefore, the DFG does not substitute the CFG, but rather enriches it with information that is useful for many analyses and transformations (AHO et al., 2006, ch. 9).

3.3.3 Hybrid Dependence Graphs

Unfortunately, having both a CFG and a DFG forces compilers to keep the separate structures consistent with each other across program transformations, which can be costly (STANIER; WATSON, 2013). Given the importance of def-use – and its converse, "use-def" – information for compiler optimizations, this has motivated the creation of alternative IRs and techniques to combine control and data flow information in a single IR. One such approach is what Lawrence (2007, p. 16) refers to as an "SSA CFG", that is, a CFG which respects SSA form in the contents of its basic blocks. According to Stanier & Watson (2013), maintaining SSA form within compiler IRs has become standard practice in research and in the industry.

Static Single Assignment (SSA) is a technique invented by Rosen, Wegman & Zadeck (1988) to enable the efficient representation of use-def and def-use relations. It should be noted that SSA, by itself, is not an IR, but a program property. Therefore, it can be achieved in any representation, including linear 3ACs, graphs-based IRs and even high-level programming languages (STANIER; WATSON, 2013). For example, Figure 13 displays the same program from Figures 10 and 11, now encoded in LLVM IR (left) and as a Scheme program (right). These programs are equivalent and respect SSA form, as explained below.

Figure 13 – Imperative ϕs versus functional blocks.

```
define i32 @f(i32 %n)
{
INIT:
    br label %TEST

TEST:
    %p1 = phi i32 [ 1, %INIT ], [ %c1, %BODY ]
    %c1 = phi i32 [ 0, %INIT ], [ %c2, %BODY ]
    %i1 = phi i32 [ 0, %INIT ], [ %i2, %BODY ]
    %b0 = icmp slt i32 %i1, %n
    br i1 %b0, label %BODY, label %END

BODY:
    %c2 = add i32 %p1, %c1
    %i2 = add i32 %i1, 1
    br label %TEST

END:
    ret i32 %c1
}
```

```
(define (f n ret)

  (define (loop p c i)
    (define (BODY)
      (let ((p2 c)
            (c2 (+ p c))
            (i2 (+ i 1)))
        (loop p2 c2 i2)))

    (define (END)
      (ret c))

    (begin ; TEST
      (if (< i n)
          (BODY)
          (END))))

  (begin ; INIT
    (loop 1 0 0)))
```

Source: Gabriel B. Sant'Anna, 2022.

As the acronym suggests, a program in SSA form assigns all of its "variables" precisely once, creating a local mapping from names to program values. Furthermore, no variable may

be used without being defined first (in a CFG, this can be rephrased as "every basic block which uses a value is dominated by the basic block which defines it"). These properties make it trivial to find the definition of a value when, later in the program, it is being used in some operation (LATTNER, 2002, p. 16). In order to handle different mutable assignments to the same variable, SSA uses a special ϕ operator. ϕ nodes (when represented in a graph-based IR) or ϕ functions (in a register-based IR) use control flow information to resolve the correct value of variables within their basic blocks (STANIER; WATSON, 2013). Intuitively, ϕ nodes extend basic blocks with formal arguments, much like parameters in a procedure. In fact, some compilers (e.g., MLIR) achieve SSA form through such "extended basic blocks", instead of the equivalent ϕ nodes (LATTNER et al., 2021). Figure 13 (left) displays LLVM syntax for ϕ functions: each operand to a `phi` pairs a value and a block label, and its result depends on which block preceded the current one during program execution.

Programs expressed in a "purely functional" manner – with no mutable variable assignments – are in SSA form by definition. This similarity has been observed by Kelsey (1995), who proved that SSA corresponds to a restricted version of CPS by describing conversion methods between both representations. Appel (1998) states that SSA *is* functional programming, rediscovered by developers of imperative language compilers. To illustrate this point, Figure 13 displays two equivalent SSA-form programs; while the LLVM versions requires ϕ instructions, the Scheme one uses pure functions as extended basic blocks – notice that it displays the same number of "block arguments" (three) as there are `phi` instructions in the LLVM IR program. This Scheme program was manually produced as an intermediate step when building the acyclic DFG shown in Figure 12. Basing the DFG on the LLVM version instead would have resulted in an equivalent, yet cyclic, graph, with a ϕ node at every cycle, clearly marking a "temporal dependency", that is, the influence of control information in the data flow graph.

Maintaining SSA form within a CFG is not the only way to combine control and data flow information in a single IR. Within this group of hybrid graph-based IRs, it is also possible to identify a chronological succession of so-called "dependence graph" designs, each one aiming to improve upon its predecessors:

- **The Program Dependence Graph (PDG)** – Originally developed by Ferrante, Ottenstein & Warren (1987), the PDG is a directed multigraph[12] combining a CFG and a data *dependency* graph. The latter differs from a data *flow* graph because edges follow the use-def direction, instead of def-use. The PDG is a useful representation for many compiler optimizations (LAWRENCE, 2007; STANIER; WATSON, 2013). However, maintaining its invariants across program transformations has a high cost in comparison to the alternatives, and it complicates other analyses due to aliasing and side-effect problems (JOHNSON, 2004, sec. 3.1). Stanier & Watson (2013) note that they were not aware of

[12] Multigraphs extend graphs with the ability to identify multiple different edges with the same nodes at their extremities. This is particularly useful for dependency graph IRs, which may attribute different meanings to two edges leaving a certain node, even if both connect to a common node on the other side.

any compiler primarily using the PDG, despite it being the most cited IR in their survey. A more detailed critique of the PDG is provided in the work of Johnson (2004, sec. 3.1).

- **The Program Dependence Web (PDW)** – Ottenstein, Ballance & MacCabe (1990) designed the PDW by augmenting the PDG with "gated SSA form", which (unlike classic SSA) is a representation that admits direct program interpretation. Unfortunately, the PDW puts an even higher burden on compilers, since its construction requires five passes over the PDG and it cannot directly represent irreducible control flow (JOHNSON, 2004; STANIER; WATSON, 2013; REISSMANN et al., 2020).

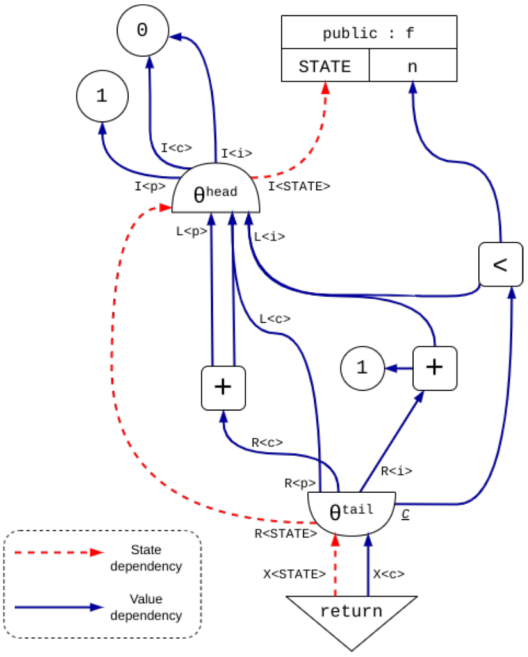
- **The Value Dependence Graph (VDG)** – According to Weise et al. (1994), "dependence graph" IRs can be viewed as different approaches to removing the limitations of a CFG. They propose the VDG as a next step in this direction, aiming to represent programs only with dependency information and a few special operators. The IR was deemed unsuitable for non-experimental use because it may fail to preserve program termination behavior (JOHNSON, 2004; STANIER; WATSON, 2013; REISSMANN et al., 2020).

- **The Value State Dependence Graph (VSDG)** – In order to address the VDG's termination problem, Johnson (2004) augmented it with state dependency edges. The resulting IR is similar to the PDG in the sense that it combines control and data flow information in a single graph, but without the ordering restrictions of a CFG. Like the VDG, the VSDG is implicitly in SSA form, which eliminates certain problems found in the PDG (JOHNSON, 2004, sec. 3.1). According to Lawrence (2007, sec. 2.2), while SSA CFGs correspond to functional programs under an *eager* evaluation strategy, VSDGs correspond to functional programs under a *lazy* evaluation strategy. While this aids some optimizations, it makes it hard to efficiently preserve the semantics of effectful computations, which are not functional in nature (REISSMANN et al., 2020). Figure 14 displays a VSDG.

- **The Regionalized Value State Dependence Graph (RVSDG)** – Lawrence (2007) argues that the VSDG's problem when representing effectful computations can be solved by augmenting it with the concept of graph "regions". The RVSDG is a multigraph with a hierarchical structure; some of its nodes introduce regions, and regions can contain entire subgraphs. This also leads to an IR which can capture more abstract notions directly in its structure. For example, θ nodes in an RVSDG explicitly indicate that the subgraph within their region constitutes the body of a loop (REISSMANN et al., 2020). Meanwhile, detecting the same loop in a CFG would require a graph traversal. Another key aspect of the RVSDG is that it provides interprocedural constructs in the IR itself: it can "represent a program as a unified data structure where a def-use dependency of one function on another is modeled the same way as the def-use dependency of scalar quantities" (REISSMANN et al., 2020). Despite having been initially described by Lawrence (2007), it seems that the RVSDG was not implemented and empirically evaluated until the work

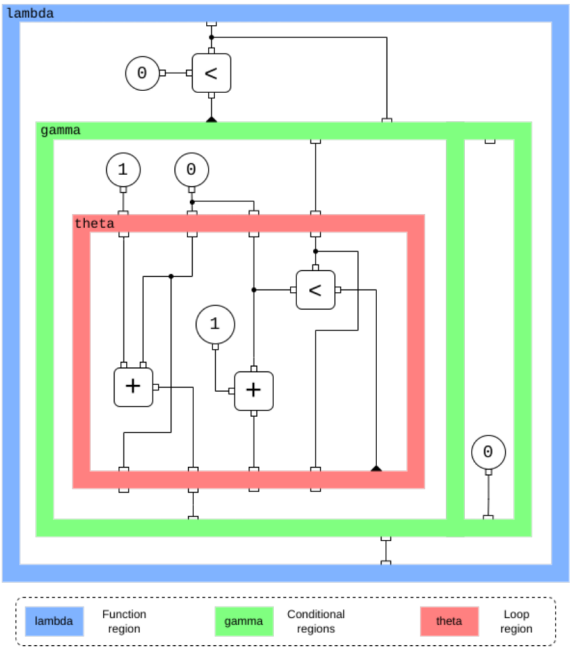
of Reissmann et al. (2020) on the jlm compiler. Figure 15 illustrates an example RVSDG, which is equivalent to the program previously shown in Figures 10–14.

Figure 14 – VSDG equivalent to Figures 10–13.



Source: Gabriel B. Sant'Anna, 2022.

Figure 15 – An RVSDG, equivalent to Figure 14.



Source: Gabriel B. Sant'Anna, 2022.

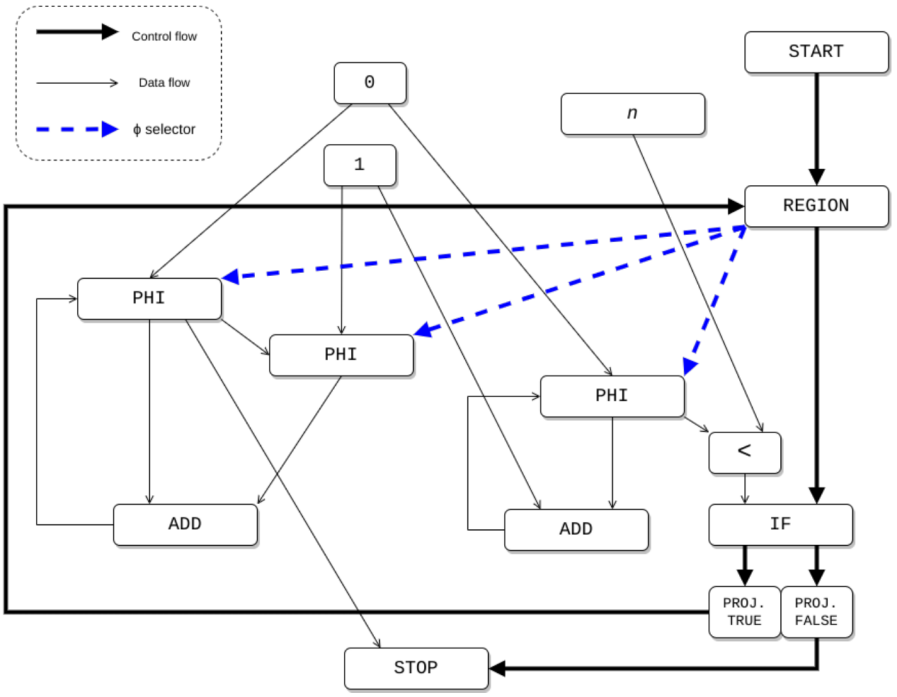
Separate from the *-DG line of research (from PDG to RVSDG) is Click's IR (CLICK; PALECZNY, 1995). According to Johnson (2004, sec. 2.1.5), it can be seen as a variation of the PDG which combines a control subgraph and a data subgraph, connecting them at `PHI` nodes (which use control information to resolve data flow) or `IF` nodes (which use program values to define control flow). Unlike the PDG, it is always in SSA form. Click's IR is also known as "the sea of nodes" (CLICK, 1995, p. 113), since it maps each operation to a node without putting it in any specific "place" in the graph; nodes are "floating around", ordered only by their dependency relations (the same could also be said of the DFG and the VSDG). This is in contrast to the CFG, which enforces a total order of instructions within basic blocks.

In order to model effectful computations, Click's IR treats memory as a special kind of value which is produced by load and store operations (CLICK; PALECZNY, 1995). It also defines its own notion of a "region", which allows it to represent the same control flow information that could be found in a CFG (CLICK; PALECZNY, 1995). Unlike in the RVSDG, however, regions in Click's IR correspond to simple nodes (albeit with additional edges linking them to the aforementioned `PHI` nodes); this avoids the additional structural restrictions of an RVSDG, which requires all of its regions to nest hierarchically and hence cannot represent irreducible control without performing restructuring transformations (BAHMANN et al., 2015).

Click's sea of nodes is an SSA graph which can be built and optimized while a program is being parsed (CLICK, 1995, ch. 7). This makes it particularly desirable in JIT compilers such as the Java HotSpot Server Compiler (PALECZNY; VICK; CLICK, 2001), libFirm (BRAUN;

BUCHWALD; ZWINKAU, 2011) and the Graal OpenJDK project (DUBOSCQ et al., 2013), all of which use adaptations of Click's IR. Braun, Buchwald & Zwinkau (2011) explain how the sea of nodes design leads to "implicit" and "inherent" transformations; in other words, the mere act of representing a program in the IR can lead to optimizations such as Copy Propagation, Dead Code Elimination and Common Subexpression Elimination. Figure 16 shows the previously discussed C program represented in Click's IR.

Figure 16 – Click's sea of nodes.



Source: Gabriel B. Sant'Anna, 2022.

An idea similar to the sea of nodes can be found in Thorin, an IR which aims to effectively represent and optimize both imperative and functional programs, exploiting the correspondence between SSA and CPS to model control flow in an uniform manner (LEISSA; KÖSTER; HACK, 2015). Unlike other functional IRs, Thorin encodes CPS in a graph instead of the more traditional tree-based approach. According to Leißa, Köster & Hack (2015), this avoids the compile-time and implementation overhead of resolving name conflicts during program transformations. In fact, the sea-of-nodes-like design completely eliminates the need to explicitly encode names and scope nesting; the former are substituted by use-def edges in the SSA graph, and the latter can be found by analyzing the sets of nodes linked by these edges (LEISSA; KÖSTER; HACK, 2015).

The influence of Click's IR is also noted in MLIR (LATTNER et al., 2021), a framework designed for the construction of domain-specific compilers. MLIR can be viewed as an extremely extensible SSA-form IR, designed to represent and optimize programs at multiple levels of abstraction. It has a generic textual encoding, but can be grouped with other graph-based IRs due to its ability to assign custom semantics to "regions" of basic blocks – the most common one being the semantics of an SSA CFG (LATTNER et al., 2021).

3.4 FINDING UNCOL

3.4.1 Design Principles and Requirements

Throughout history, many compiler IRs were designed with the goal (or simply ended up being used for the purpose) of accommodating as many programming languages as possible, while targeting multiple machine architectures. This survey lists, albeit not exhaustively, various IRs claiming to be a suitable universal program representation. Evidently, the UNCOL idea has been very much alive for the past 60 years, even if the term is not always explicitly mentioned in IR proposals, or if it is merely used to refer to a previous realization of the concept (LATTNER; ADVE, 2004). From Steel (1961a) to Lattner et al. (2021), compiler writers sometimes state the requirements and design principles applied in the creation of an IR. This section summarizes those which could be applied to a modern version of UNCOL.

Some requirements are immediately clear. A universal IR must be independent of any specific machine (CHOW, 2013), although one must remember that "portable" program representations do not cause software to become portable, but only serve as a medium to represent programs which, by themselves, may or may not be machine-independent (MACRAKIS, 1993b). In the context of optimizing compilers, a universal IR should permit the efficient implementation of various programming language constructs on a great variety of machines, even if this may be a non-trivial problem by itself – irrespective of the IR (KESSENICH, 2015). Designing for these goals is not easy:

> An intermediate language is a gate, through which an algorithm must pass to reach the machine. We found that our major design problem was to determine the 'width' of the gate: How much information about the source program must be carried in the intermediate code to permit efficient implementation on a variety of hardware. (COLEMAN; POOLE; WAITE, 1974)

According to Bagley (1962), the principle should be that translations using the IR as an intermediate step must preserve the core invariant aspects of source programs: the "essential algorithm", such that no unnecessary restrictions (e.g., a strict ordering of operations which could otherwise be executed in parallel) are imposed upon the program; the "data forms", that is, information about the nature of program data (e.g., types and their relations, units of measure, range and precision); and "data formats", which indicate the physical layout of program data (e.g., byte order in a communication protocol). Steel (1961a) had previously stated that UNCOL would need both "imperative" and "declarative" sentences, presumably in order to represent these very same core aspects. Still, expressing essential program information in the IR is not very helpful if this cannot be effectively used during code generation (FRANZ, 1994). Thus, retaining this information for as long as possible (up until load-time, or even run-time) has become an increasingly important concern for compilers aiming to perform both machine-

independent and machine-specific optimizations (LATTNER; ADVE, 2004). As summarized by Lattner et al. (2021), “premature lowering is the root of all evil”.

A major concern of universal compiler IRs is efficiency (BAGLEY, 1962), both in the manipulation of a program’s representation and in the generated code. A key step to achieving the first point is remembering that compiler IRs are computer-oriented representations. Unlike programming languages, which are restricted by human-related (and often subjective) requirements such as code readability (CHOW, 2013), IRs can be explicitly designed to support program analysis, transformation and optimization in the most efficient way possible (BRAUN; BUCHWALD; ZWINKAU, 2011; STANIER; WATSON, 2013). In fact, some use cases require the exact opposite of human readability, worrying instead about the prevention of reverse-engineering through program obfuscation (MACRAKIS, 1993a; CHOW, 2013). Even then, this must not make it impossible to provide translation traceability, which is precious for debugging and compiler diagnostic purposes (LATTNER et al., 2021).

As for the efficiency of generated code, it requires compilers to reason about programs so as to optimize them more effectively. According to Click & Paleczny (1995), this calls for the IR to have its own model of execution, which should also define the properties of concurrent programs – as proven necessary by Boehm (2005) and later discussed by Zhao & Sarkar (2011b) and Khaldi et al. (2013). Fournet & Gonthier (2000, p. 20) add that algebraic approaches have their benefits for this purpose, noting many different operational equivalence relations which could be used to mathematically ensure that some transformations preserve program semantics.

Felleisen (1991) explains how a formal treatment of expressiveness (based on semantics-preserving transformations) can indicate the “core” of a programming language – a concept which had been previously discussed by Steele & Sussman (1976) in the context of Lisp and Scheme. Although no existing work (as far as the author is aware) has applied this theory to the design of a compiler IR (maybe with the exception of Scheme, which has been used as an IL), the importance of having a small language core has been noted more than once. Haddon & Waite (1978) and Lattner et al. (2021) highlight the value of maintaining a sharp Occam’s razor to cut off redundant IR constructs. Furthermore, having a simple language core does not exclude the straightforward expression of various complex operations in the IR, if only it can be made extensible (HADDON; WAITE, 1978; MACRAKIS, 1993c). Extensibility also helps preserve “high-level” information until as late as possible (ECMA-335, 2012), which enables more optimizations, as discussed above.

Finally, if any compiler IR really is to become universal and widely adopted in the computing industry, it needs to be specified in an open and royalty-free standard. Otherwise, the benefits of a common IL would never be fully realized (CHOW, 2013). One such benefit is ease of distribution. In some cases, this will also raise the need for IL validation and program safety verification, which usually implies the existence of a formal specification for the model of computation exposed by the IR (HAAS et al., 2017).

The following list summarizes the UNCOL design principles discussed above:

- **Expressivity** – The IR must be able to represent, up to some level of detail, the essential properties of any program. Software which is not portable is also of interest, even if only to indicate where and how it uses machine-specific constructs and assumptions, or when compiling it to a compatible architecture (something which ought to be supported).
- **Efficiency** – The IL format should be fast to decode and validate. At the same time, the IR itself must not introduce unreasonable costs to code transformations, nor impose additional restrictions (unnecessary from a logical standpoint) to its programs.
- **Progressivity** – An optimizing compiler should prefer to retain, rather than recover, information about a program. Premature lowering decreases optimization potential, so the IR must be able to preserve "high-level" structure for as long as it is desirable to do so.
- **Parsimony** – The IR must not expand uncontrollably to accommodate new languages or machines. Instead, it ought to contain a well-defined core which is simple, yet expressive.
- **Extensibility** – The IL should be amenable to extensions, both from users and to its own specification. These must be clearly signaled (possibly at different levels of program granularity) for the sake of compatibility. This principle complements the previous one.

Other than design principles, concrete requirements have also been identified in the survey, based on previous works. These can be used to review existing compiler IRs, or guide the design of new UNCOLs. Briefly, a suitable universal compiler IL must have:

1. A fully machine-independent design, or, at the very least, one which encompasses entire classes of machine architectures (e.g., stored-program, byte-oriented computers).
2. Independence from any specific high-level programming language.
3. An efficient and transparent model of computation, which does not impose additional costs (such as a garbage-collected runtime) to programs which do not need them.
4. A structure optimized for program transformations and compiler optimizations.
5. At least one actively-developed compiler making use of it, which is what enables the IL to be evaluated and experimented with.
6. A free standard, specifying its semantics and possible exchange formats.

*. Optionally: mechanisms for source traceability and obfuscation.

Since source traceability and obfuscation are diametrically opposed to each other, they are marked as "optional". This indicates that the IL should, at the very least, not make it infeasible to provide such mechanisms when they are demanded.

3.4.2 IR Review and Comparison

Many compiler infrastructures, old and new, have been analyzed as part of this survey. Table 1 displays the outcome of reviewing, in light of the design principles and requirements which were previously described, the IRs used in some of these compilers. Each requirement is referenced by its respective number in the enumeration above, with markings indicating whether an IR fulfills it (✓), does not fulfill it (✗), if the result is unclear (?) or if that particular requirement is not applicable (–) to the IR in question.

It can be seen that all requirements have been met at some point, although not necessarily by the same IR. For example, LLVM IR has been used to efficiently compile many programming languages, but it lacks a stable specification and exposes some architecture-specific constructs. The CIL, on the other hand, is formally specified but requires a managed runtime. Other IRs, like SPIR-V, have not yet seen much use outside of a specific problem domain, so it is hard to evaluate how well they would accommodate various high-level languages.

Table 1 – Comparison of compiler IRs.

Name	Structure	Additional labels	Req. 1	Req. 2	Req. 3	Req. 4	Req. 5	Req. 6
ANDF	Tree	Generic encoding	–	?	–	✗	✓	✓
C (as an IL)	Tree	High-level language	✓	✗	✓	✗	✓	✓
CIL	Linear	Stack machine	✓	?	✗	✗	✓	✓
Cranelift IR	Linear	Register-based, SSA	✗	?	✓	✓	✓	✗
EM IL	Linear	Stack machine	?	?	✓	✗	✓	✗
Firm	Graph	Sea of nodes	?	?	✓	✓	✓	✗
GCC RTL	Linear	Register-based	?	?	✓	✓	✓	✗
HSA IL	Linear	Register-based	✗	?	?	✗	✓	✓
Janus	Linear	Stack machine	?	?	✓	✗	✗	✗
jlm IR	Graph	RVSDG	?	?	✓	✓	✓	✗
JVM bytecode	Linear	Stack machine	✓	✗	✗	✗	✓	✓
LLVM IR	Linear	Register-based, SSA	✗	✓	✓	✓	✓	✗
MLIR	Graph	SSA-form graph	?	?	✓	✓	✓	✗
ROSE	Tree	Generic encoding	–	✓	–	?	✓	✗
Scheme (as an IL)	Tree	High-level language	✓	✗	✗	✗	✓	✓
Semantic Dictionaries	Tree	Generic encoding	–	✓	–	✗	✗	✗
SPIR-V	Linear	Register-based, SSA	?	?	✓	✓	✓	✓
SUIF	Linear	Register-based	?	?	?	?	✗	✗
Thorin	Graph	CPS graph	?	?	✓	✓	✓	✗
WebAssembly	Linear	Stack machine	✓	?	✓	✗	✓	✓

Source: Gabriel B. Sant'Anna, 2022.

Requirements 5 and 6 are the most straightforward to verify. For nearly all IRs in Table 1, there is at least one compiler in active development (with publicly-available changes in the last five years) making use of it. On the other hand, most lack a standard IL specification.

Meanwhile, markings for requirements 3 and 4 are derived from the survey. While stack-based IRs are more compact than the alternatives and, like few other IRs, can be directly executed by a VM, compiler optimizations are usually performed over a different program representation. Furthermore, tree-structured IRs do not seem to be particularly well-suited for data-flow analyses, since these require def-use chains; this information is more efficiently maintained in SSA form, which appears to be rare in tree-structured IRs.

It is unclear how to evaluate requirements 1 and 2 in practical terms. Machine- and language-independence are often stated as IR design goals, yet there is no known method to verify or measure these qualities. This appears to be an open problem, which is briefly discussed as a future line of work in Section 5.2.

Still, there are certain litmus tests which could be applied to an IR in order to argue that it lacks a fully machine- or language-independent design. One such criterion is the presence of architecture-specific instructions and data types in the IR – e.g., x86-specific constructs in LLVM (2022). It also seems reasonable to state that a programming language being used as a compiler IL cannot be language-independent; and similarly for the bytecode format of a VM designed for a specific programming language (as is the case of the JVM). Meanwhile, although generic tree encodings can represent the syntax of most programming languages, they do not directly specify an execution model, so requirements 1 and 3 are not applicable.

Finally, instead of filling the remainder of Table 1 with inconclusive markings, certain IRs are hereby considered language- and/or machine-independent in case they have been repeatedly used as such in research or in the industry. This means, for example, that in spite of the various arguments against using C as a portable compiler IL (OLIVA; NORDIN; JONES, 1997; MACRAKIS, 1993b) or using LLVM IR to represent functional language constructs (LEISSA; KÖSTER; HACK, 2015), the practical advantages often outweigh the disadvantages.

4 DESIGNING A NEW COMPILER IR

> The language designer should be familiar with many alternative features designed by others [...] One thing he should not do is to include untried ideas of his own. His task is consolidation, not innovation. (HOARE, 1989)

The goal of this thesis is to identify design principles and requirements which could be applied to modern iterations on the UNCOL idea. The survey presented in Chapter 3, as well as the literature review leading up to it, can be seen as a means to that end. While the author's initial hypothesis was that one or more suitable UNCOL candidates would be found as part of the research, that does not seem to be the case. Still, this is not to say that UNCOL would not be viable in the present day. In fact, Subsection 3.4.2 shows that the evolution of compiler technology over the last 60 years provides most of the needed tools and techniques to fulfill the requirements of a practical universal compiler IR.

According to Lattner et al. (2021), IR design is an "art", not completely understood even by practitioners in the field of programming languages and compilers. It follows that it would be hard to judge the design principles previously listed without seeing them being applied. Therefore, this chapter presents a new compiler IR, which is summarized here and detailed in the next few sections. Abiding by the words of Hoare (1989), the proposal does not include any especially innovative ideas, aiming instead for the consolidation of existing techniques under a unified architecture. It must be emphasized that this should not be interpreted as an endeavor to develop the Platonic ideal of an UNCOL. Rather, it is an attempt to show that the requirements and principles identified in this work are sound, and explore one possible path of the design space.

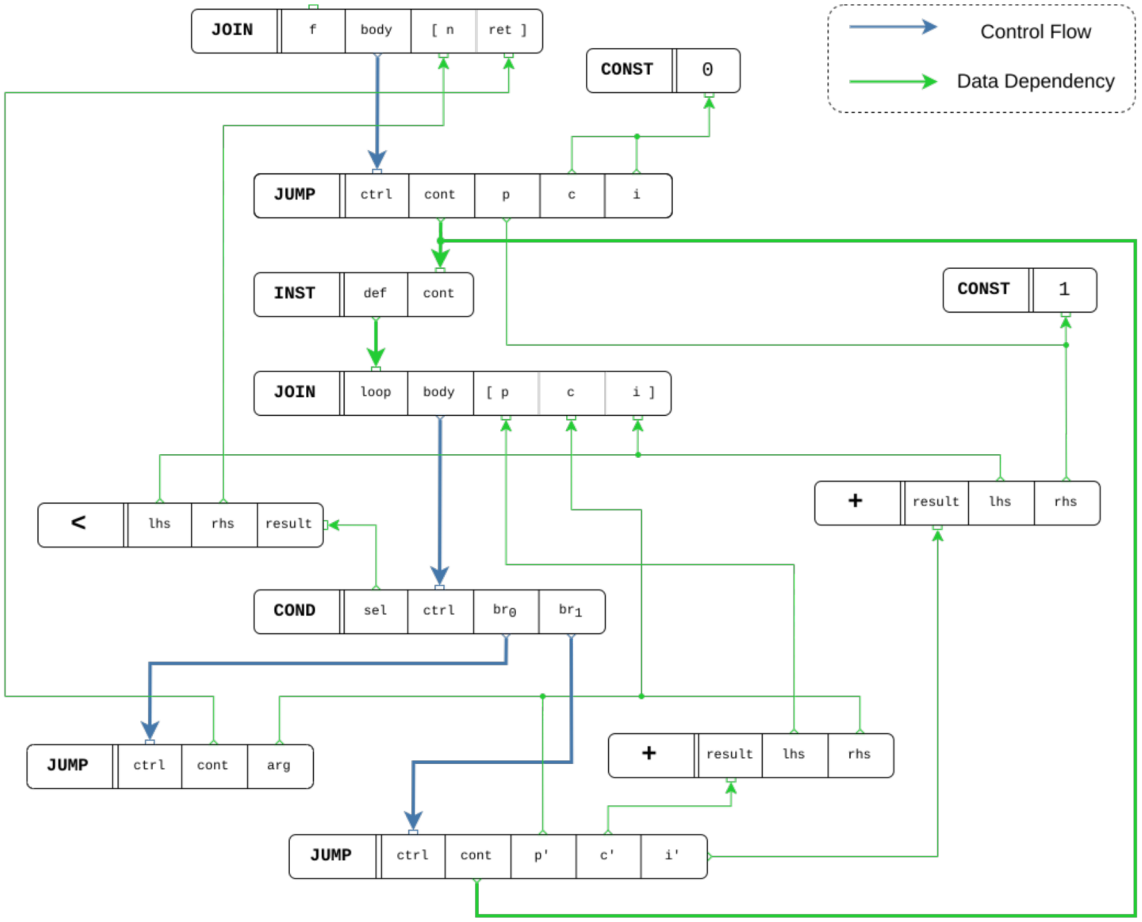
The new IR is structured as a graph, building mainly upon Click's sea of nodes and Thorin; the first provides a good foundation for an SSA graph – as seen in Section 3.3 – and the second extends it with support for the functional paradigm through CPS. In order to accommodate explicit concurrency, the new IR tentatively adopts C's memory consistency model and a message-passing mechanism from the join calculus; the former is all that is needed to encode typical multi-threaded algorithms (ADVE; BOEHM, 2010) and the latter can be seen as a concurrent extension to CPS (FOURNET; GONTHIER, 2000, sec. 1.3). The IR also defines a structural type system and type-level operations which are not unlike those found in LLVM (LATTNER, 2002, sec. 3.3) or SPIR-V (KESSENICH, 2015, sec. 2.8). Additionally, abstraction at the IR level is achieved through a macro system with a focus on portability (inspired by ANDF and Janus). The combination of these last two features leads to a mechanism which approximates parametric polymorphism – as seen, for example, in Standard ML (MACQUEEN; HARPER; REPPY, 2020). Finally, macros allow the IR to represent full programs, like the RVSDG. It does so by implementing an idea from the Standard ML of New Jersey system, in which compilation units are modeled as compile-time "functions".

4.1 INFORMAL DESCRIPTION

4.1.1 The CPS Soup

The new IR combines control flow and data dependence information in a single graph. It is founded upon Click's sea of nodes design, being inherently in SSA form and eliminating the concept of basic blocks altogether by making each operation an individual node. A first sample is shown in Figure 17, which encodes the program used as a running example in Section 3.3.

Figure 17 – A functional program encoded in the new IR.



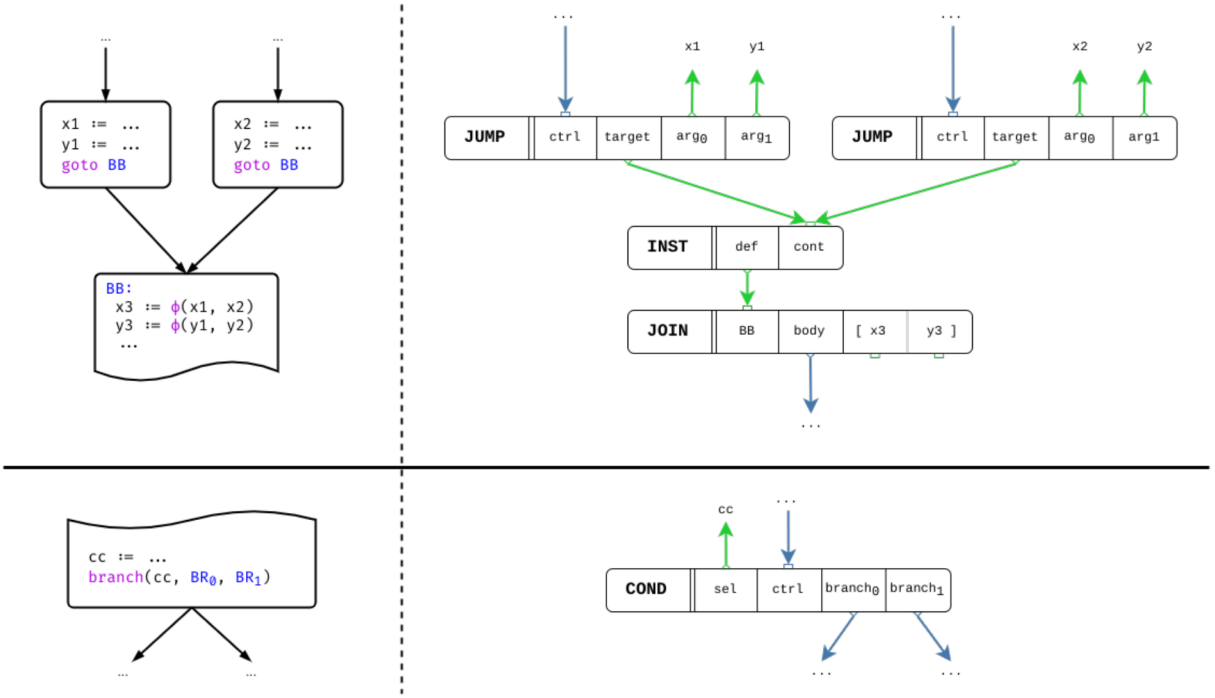
Source: Gabriel B. Sant'Anna, 2022.

In the figure and throughout this chapter, each node is tagged with a fully-capitalized label in bold, indicating its kind. After the tag, nodes have a sequence of slots, which can either be an outgoing edge slot (drawn as an arrow leaving the node) or an incoming edge slot (which can be used as a target for any number of outgoing edges coming from other nodes). There are also different kinds of edge slots, which are indicated by their color[13]: control flow edges are colored in blue and data dependency edges in green. The former direct the overall execution of the program and the latter specify what data operands are used by each node.

[13] With apologies to the colorblind reader.

`COND` nodes direct control flow information based on a data selector, similarly to Click's `IF` nodes. On the other hand, whereas Click's IR uses both "region" and ϕ nodes as a way to communicate control flow information to the data subgraph, the new IR adopts the equivalent notion of extended basic blocks, which are represented by `JOIN` nodes. As in Thorin, this leads to an uniform treatment of procedure and basic blocks, a key aspect of its use of CPS. It is also reflected in `JUMP` nodes, which transfer control (as well as a sequence of arguments) to the continuation identified by their first data dependency slot (labeled as "`cont`" in Figure 17). An analogy can be drawn between this representation and classic SSA form: while a `JOIN` node defines a basic block and a sequence of ϕ functions (corresponding to its arguments), `JUMP` nodes are a mechanism to resolve these ϕs (i.e., to provide values for each argument) before control flow continues at the basic block (or procedure) which is being targeted. Figure 18 illustrates how a typical SSA CFG representation (left) with ϕ instructions (top) and conditional branches (bottom) could be translated into the new IR (right).

Figure 18 – Merging control flow and data dependencies.



Source: Gabriel B. Sant'Anna, 2022.

Continuations are modeled as data dependencies, like any other program value. In Figure 17, one can be found in the "`ret`" parameter of the topmost `JOIN` node; it corresponds to the procedure's return address. Figure 18 also shows a continuation being directly constructed: `INST` nodes produce a continuation when provided with the definition (exposed in the very first slot) of a `JOIN` node. In this context, `JOIN` nodes can be viewed as inert subprograms, of which active copies can be created by `INST` nodes; such "live" instances represent the possible continuations of a program. To highlight the fact that using a continuation changes the control flow of a program, dependencies on `JOIN` node definitions and instantiated continuations have

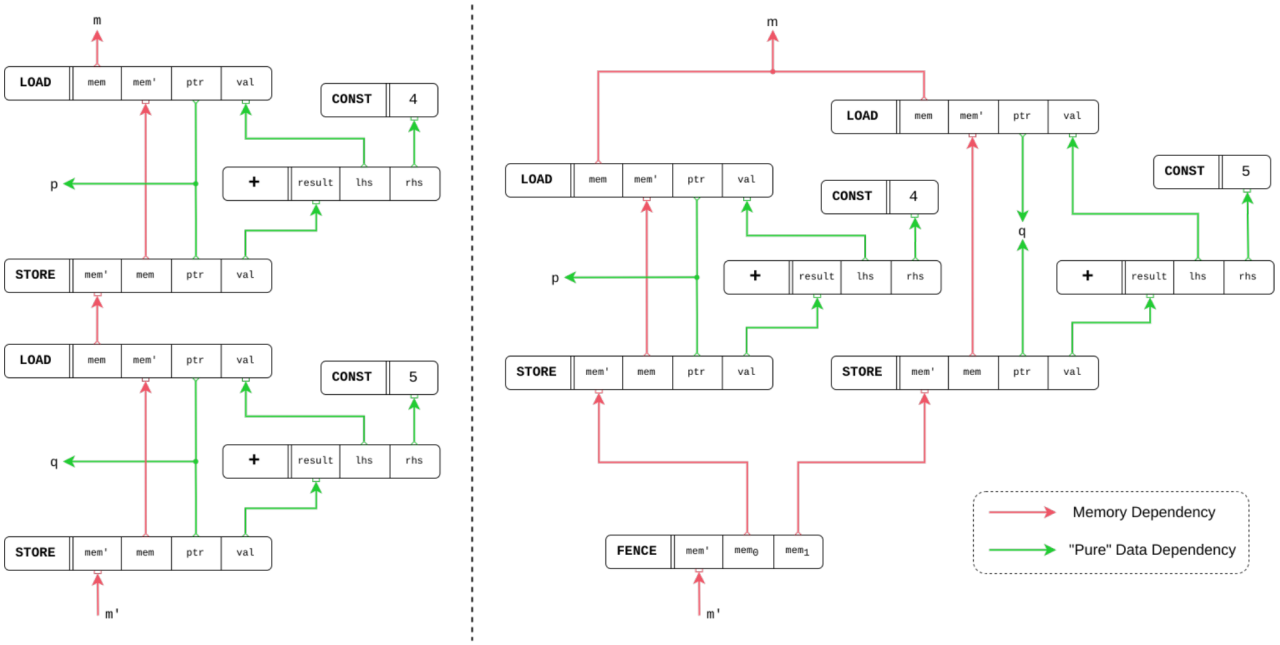
been drawn with thicker arrows in Figure 17. When one observes only these arrows, as well as control flow edges, it is possible to identify an overall shape similar to the CFG of Figure 11.

Another similarity to Thorin is that scope nesting becomes implicit in the graph. For example, the `JOIN` node whose definition slot is labeled "`loop`" in Figure 17 stands for a basic block within the "`f`" procedure, a containment relation which can be inferred solely from control flow and data dependency edges, as explained in Section 4.2. The combination of CPS and implicit scope nesting is also known as "CPS soup" (WINGO, 2016), matching the sea-of-nodes notion of subprograms "floating around" in the IR.

4.1.2 State and Concurrency

Control flow, data dependence information and a few operations suffice to encode functional programs (LAWRENCE, 2007). Effectful computations, however, require different mechanisms. In order to accommodate these, the new IR borrows the idea of "state dependencies" from Lawrence (2007) and "I/O states" from Click & Paleczny (1995); the first are used to represent memory operations and their relations, which include the second in the special case of memory-mapped input and output. Hence, the new IR represents memory through a special kind of data dependency (colored in red), whose main purpose is to inform the compiler of additional scheduling constraints.

Figure 19 – Effectful operations under different scheduling constraints.



Source: Gabriel B. Sant'Anna, 2022.

These constraints apply to memory-related operations (e.g., `LOAD` and `STORE`) and the memory dependency edges which they use and produce. An example is given in Figure 19, which displays two possible encodings of the C program snippet `{ *p += 4; *q += 5; }`.

While the graph on the left enforces a strict ordering of all memory operations (which could be required in the case of incomplete aliasing information, or if the pointers were marked with C's `volatile` qualifier), the one on the right allows the compiler to reorder independent loads and stores. Finally, a `FENCE` node is used in the second encoding to merge the two symbolic memory dependencies. This has the effect of ordering future memory operations which depend on `m'` against the stores to both variables (even if said stores are not ordered among themselves).

State dependency edges allow the IR to represent the ordering constraints of effectful computations. Optimizing compilers, however, may still reorder (and, in some cases, add or remove) memory operations as long as the resulting program has the same semantics as the original. Unfortunately, the set of optimizations which a compiler can validly apply to effectful computations shrinks considerably under the possibility that they could execute in parallel (and while sharing memory) with other effectful subprograms. This is the crux of the argument of Boehm (2005), who explains why "threads cannot be implemented as a library". In essence: assuming that any code could be part of a concurrent execution would lead to significant performance losses, while ignoring that possibility leads to invalid transformations when certain subprograms are indeed executed in a multi-threaded setting. Accordingly, Zhao & Sarkar (2011a) indicate that compiler ILs should directly expose a concurrent model of computation.

In this context, the new IR adopts the "release consistency" memory model proposed by Gharachorloo et al. (1990). This is the same memory model specified for the C programming language (ADVE; BOEHM, 2010) and, much like C may be considered the *lingua franca* of high-level programming languages (FELDMAN, 2004), its memory consistency model has also become a *de facto* standard for both software and hardware architectures:

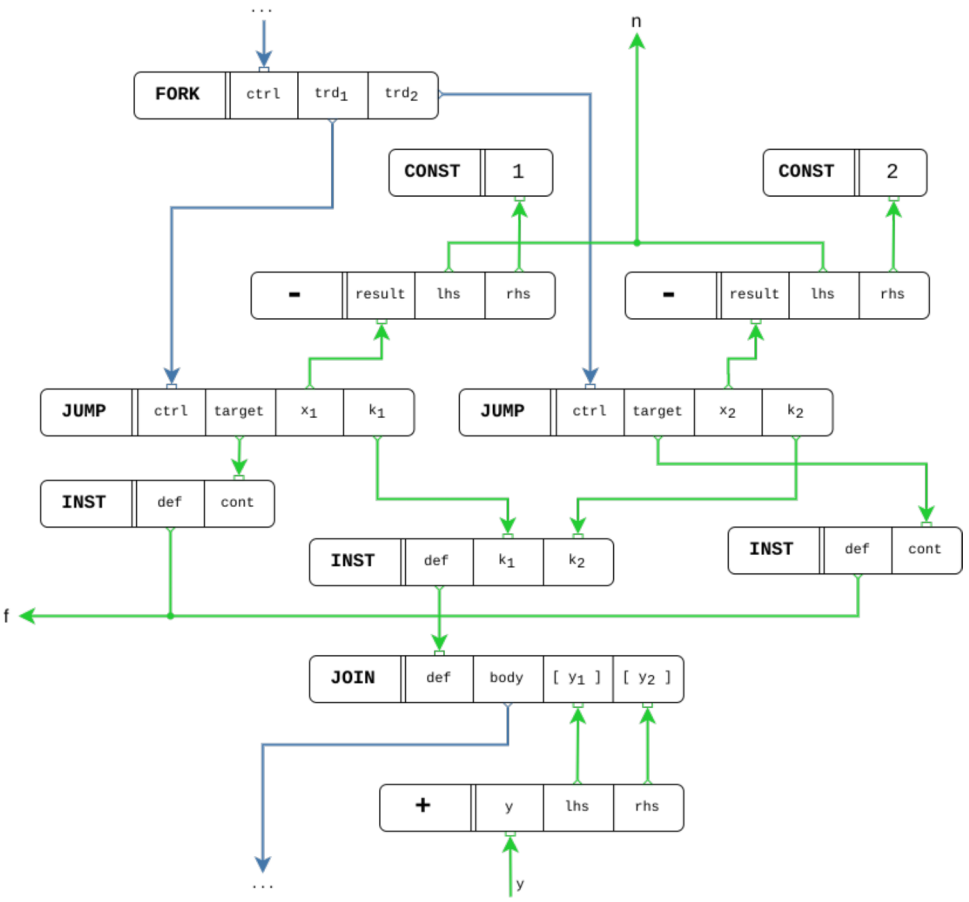
> After much debate, the language community and architecture community appear to have finally settled on release consistency as the standard memory consistency model [...] (RISC-V, 2019, ch. 8)

Although formal reasoning in the presence of data races remains a though problem (DOLAN; SIVARAMAKRISHNAN; MADHAVAPEDDY, 2018), adding this memory consistency model to the IR results only in a few extra bits on memory operations: `LOAD`s and `STORE`s can be optionally labeled as non-synchronizing atomic operations – equivalent to `memory_order_relaxed` in C – and `FENCE` nodes synchronize atomic operations based on memory dependency edges and a couple of extra bits to indicate "acquire" and/or "release" semantics. In order to support memory-mapped input and output, `LOAD`s and `STORE`s additionally carry a "volatile" bit.

While assigning concurrent semantics to memory operations requires changes to the IR, certain kinds of data-level parallelism become implicit in the sea-of-nodes-like design: two nodes with only data dependency edge slots can have their operations computed in parallel whenever there is no chain of dependencies linking them (as in the case of the two addition operators in the rightmost graph of Figure 19). Control flow, on the other hand, is inherently ordered, as in a CFG. In order to overcome this limitation, the new IR also provides mechanisms to relax the strict ordering of operations which require control flow: `FORK` nodes inform

the compiler that certain program fragments need not be executed before or after each other, and `JOIN` nodes can be used to synchronize those fragments, merging them back together into a single control flow. Figure 20 illustrates this mechanism by encoding the C equivalent of `{ y = f(n-1) + f(n-2); }`, in which the order of function calls to `f` is unspecified.

Figure 20 – Explicit concurrency using FORK and JOIN nodes.



Source: Gabriel B. Sant'Anna, 2022.

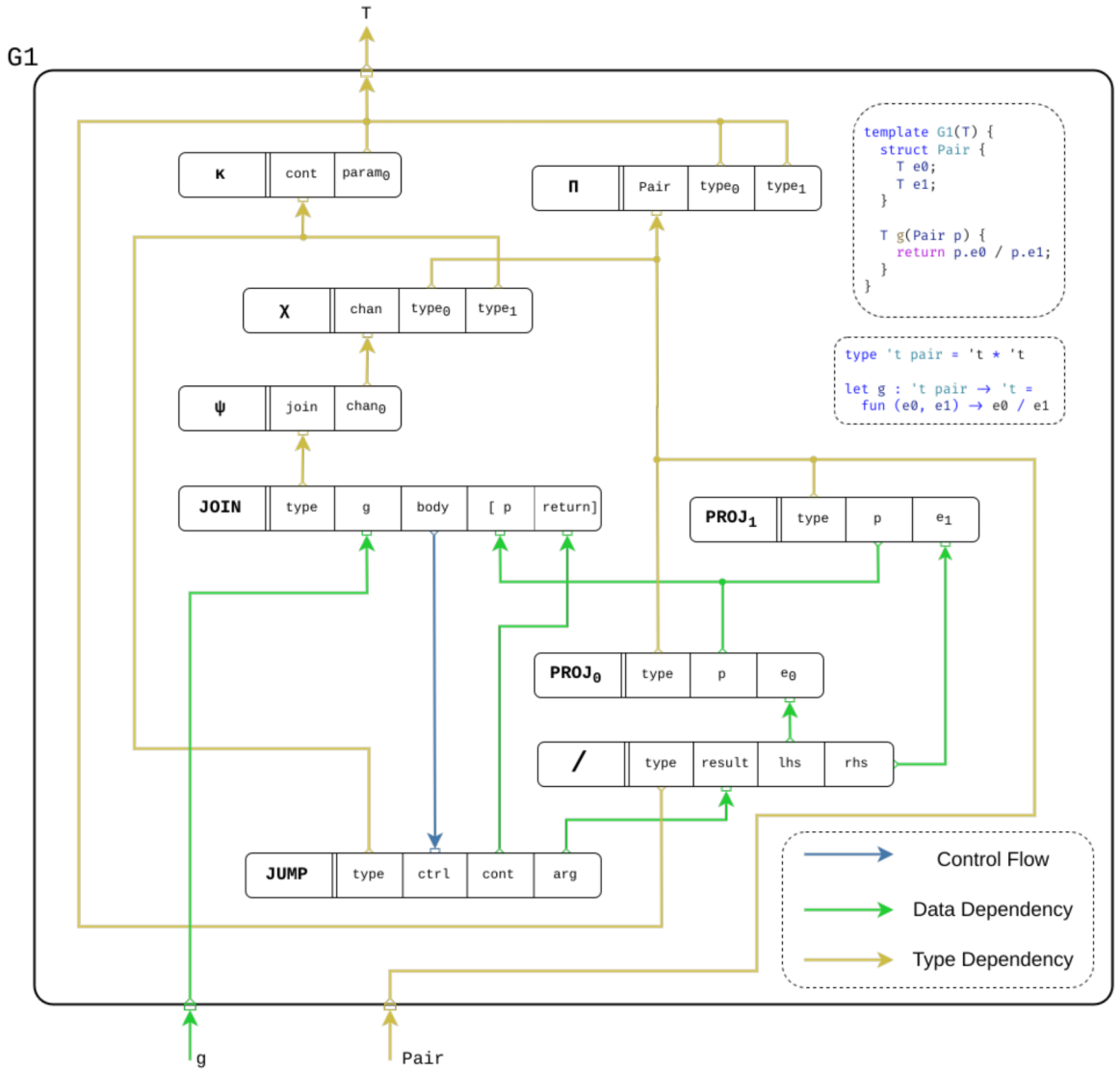
Unlike in previous examples, the `JOIN` node at the bottom of Figure 20 produces two continuations when instantiated, each corresponding to a group of parameters (indicated by square brackets). These are used as the return continuations for each separate call to `f`, both of which take control from the `FORK` node. After the two calls return by jumping to their respective continuations, program execution resumes at the `JOIN` node, which, at that point, has access to the results of each independent function call. This local concurrency mechanism matches the working of linear join patterns in the join calculus (FESSANT; MARANGET, 1998, p. 2), which can thus be viewed as a concurrent extension to CPS and whose use in a compiler IL has precedent in the work of Calvert (2015).

4.1.3 Types and Macros

According to Lattner (2002, sec. 3.3), type information enables "a broad class of high-level transformations on low-level code". Furthermore, parameterizing certain operations with the types they are expected to use and produce avoids an explosion of type-specific operators, resulting in a smaller language (STEEL, 1961a). For these reasons, the new IR is typed.

The new IR defines base types and type-level operations, much like SPIR-V. These are encoded in yet another subgraph, which, for the sake of clarity, has been omitted from previous illustrations of the IR. Figure 21 displays an example of type-level operations (labeled with Greek letters) and type dependency edges (colored in yellow) being used to encode a program exhibiting parametric polymorphism (OCaml and D equivalents are also shown in the figure).

Figure 21 – Type dependency edges and type-level operations.



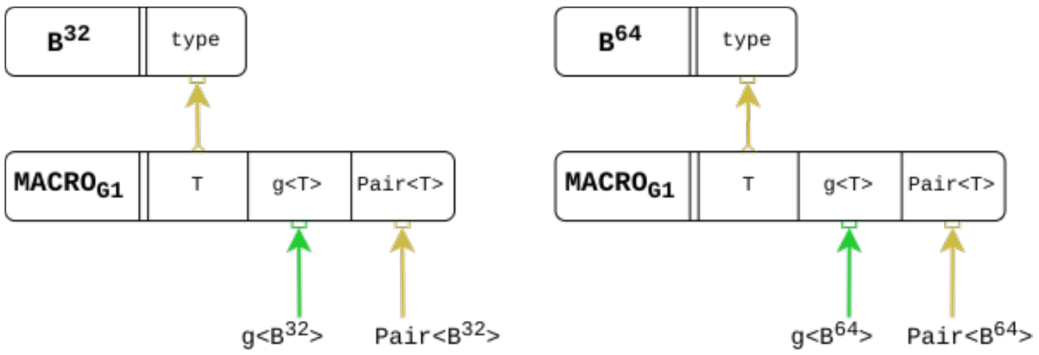
Source: Gabriel B. Sant'Anna, 2022.

In Figure 21, graph `G1` encodes a program which depends on a certain type `T`. It defines `Pair` as a product type (created with a Π node) which aggregates two elements of type `T`. Then,

it defines a function g which receives a `Pair` and returns a `T` resulting from the division of its two inner elements. The remaining type-level operations are used to specify the type of g, which is a single-channel join pattern (ψ); this channel (χ) receives a `Pair` and a continuation (κ), which in turn can be `JUMP`ed to by providing it a value of type `T`.

Along with its type system (which is detailed in Section 4.2), the new IR uses macros as an abstraction mechanism. According to Macrakis (1993a), IR macros can be used to delay the lowering of some program information (e.g., the size of target-dependent types), which enables the representation to preserve portability without fully compromising on architecture-specific optimizations. These macros are represented as nodes which encapsulate other subgraphs. Figure 22 illustrates a particular use of this mechanism, in which the IR can approximate parametric polymorphism by using `MACRO` nodes to export definitions (which can include procedures, other types and constant values) based on what type parameters they are given.

Figure 22 – Approximating parametric polymorphism through macro nodes.



Source: Gabriel B. Sant'Anna, 2022.

In Figure 22, the subgraph `G1` is used as a template, which is instantiated through `MACRO` nodes. Each of the two instances provides a different base type as input to the macro, which results in new types and procedure definitions. Hence, the program expressed by Figure 22 is equivalent to two side-by-side copies of Figure 21, with `T` substituted by either a 32- or 64-bits-wide bitvector in each case. This also illustrates how macros can save space in the IR by compacting redundant copies of similar subgraphs – a possibility previously noted by Macrakis (1993a) in the context of ANDF.

Finally, macros can be used to implement a technique from the Standard ML of New Jersey system, where "a compilation unit is represented as a function that takes a tuple of bindings for its free variables and returns a tuple representing the bindings that it has introduced" (FARVARDIN; REPPY, 2020). In practice, such "functions" correspond to `MACRO` nodes in the new IR, and linking multiple compilation units (subgraphs) together can be achieved by correctly setting up their dependency edges and expanding their `MACRO` definitions.

4.2 FORMALIZATION

4.2.1 Multigraph Structure

Formally, programs in the new IR are composed by multigraphs $\{G_1, \ldots, G_n\}$, with $1 \leq n \leq graph_{MAX}$. Let Σ be a finite set of symbols corresponding to slot labels, K_N a finite set of node kinds, and K_E a finite set of edge colors. Then, each multigraph G_i is defined by a tuple in the form $(N_i, E_i, target_i, label_i, color_i, r_i)$, where:

- N_i is a non-empty finite set of nodes, pairwise disjoint from the nodes of all other multigraphs in the program: $\forall j \in [1,n] \,.\, j = i \vee N_i \cap N_j = \emptyset$.
- $E_i \subseteq (N_i \times \Sigma)$ is a finite set of edges, identified by a pair of source node and slot label.
- $target_i : E_i \mapsto (N_i \times \Sigma)$ is a wiring function mapping each edge in E_i to a target node and slot, such that the image and domain of this function are disjoint: $\forall e \in E_i \,.\, target_i(e) \notin E_i$.
- $label_i : N_i \mapsto K_N$ is a labeling function which maps each node in N_i to a node kind.
- $color_i : E_i \mapsto K_E$ is a coloring function which maps each edge in E_i to an edge color.
- $r_i \in N_i$ is a distinguished node, which is called the root (or, "top-level") node.

The set of edge kinds K_E is defined by the finite list $\{control, data, memory, type\}$. The set of slot labels Σ, on the other hand, is only used to disambiguate different edges coming from or reaching any given node, so an exhaustive definition is not particularly useful. Finally, the set of node kinds K_N includes:

- Nodes related to control flow: {`JOIN`, `INST`, `JUMP`, `FORK`, `COND`}.
- Memory operations: {`LOAD`, `STORE`, `FENCE`}.
- Type nodes M and ρ, respectively for memory and pointer types.
- The family of fixed-width bitvector types: $\{\mathbb{B}^1, \ldots, \mathbb{B}^{w_{MAX}}\}$.
- Type combinators $\{\Pi, \kappa, \chi, \psi\}$.
- Macros $\{\texttt{MACRO}_{G_1}, \ldots, \texttt{MACRO}_{G_{graph_{MAX}}}\}$, corresponding to other multigraphs.
- A set of primitive data operations, with constant values {`CONST`}, bitwise operations {`AND`, `OR`, `XOR`, $\ll$, $\gg$}, integer arithmetic $\{+, -, \times, \div, \%\}$, comparisons $\{=, \neq, <, \geq\}$, bitvector conversions, Π-type value constructors {`TUPLE`} and Π-type projections $\{\texttt{PROJ}_0, \ldots, \texttt{PROJ}_{proj_{MAX}}\}$. Although most of these (refer to Appendix A for a full listing) could be derived from a smaller set of primitives, this set roughly corresponds to the intersection of operations available in modern IRs (including LLVM IR, WebAssembly, Thorin, SPIR-V and the CIL).

Wiring functions $target_i$ are constructed in such a way as to disallow the same node-slot pair to identify both the source and the target of an edge. Hence, it is possible to define, for every node $v \in N_i$, two disjoint sets of slot labels: $outSlots(v) \rightleftharpoons \{s \in \Sigma \mid (v,s) \in E_i\}$ and $inSlots(v) \rightleftharpoons \{t \in \Sigma \mid \exists e \in E_i \,.\, target_i(e) = (v,t)\}$. Then, for any slot $t \in inSlots(v)$, define $uses(v,t) \rightleftharpoons \{e \in E_i \mid target_i(e) = (v,t)\}$. These sets, along with node labeling and edge coloring functions $label_i$ and $color_i$, describe the local structure of v, which is said to be well-structured if and only if it respects some additional local rules:

1. Linked slots have the same color: for every slot $t \in inSlots(v)$, the set $color_i[uses(v,t)]$ of colors associated to edges which target (v,t) must have exactly one element, written $color(v,t)$. In other words, $\forall e \in uses(v,t) \,.\, color_i(e) = color(v,t)$.

2. A node's kind (partially) defines its structure: $label_i(v)$ determines possible values for $outSlots(v)$ and $inSlots(v)$, as well as valid colors for each of these slots. This rule actually consists of several formulas (listed in Appendix A), one example being:

$$
\begin{aligned}
label(v) = \texttt{JUMP} \implies & \\
& outSlots(v) \subseteq \{type,\ target,\ arg_0,\ \ldots,\ arg_{args_{MAX}}\} \\
\wedge\ & type \in outSlots(v) \ \wedge\ color(v,type) = type \\
\wedge\ & target \in outSlots(v) \ \wedge\ color(v,target) = data \\
\wedge\ & (\forall j \,.\, arg_j \in outSlots(v) \implies color(v,arg_j) \in \{data, memory\}) \\
\wedge\ & inSlots(v) \subseteq \{ctrl\} \\
\wedge\ & (ctrl \in inSlots(v) \implies color(v,ctrl) = control)
\end{aligned}
$$

3. Root nodes are labeled as MACROs: $v = r_i \implies label_i(v) = \texttt{MACRO}_{G_i}$.

For every node $v \in N_i$, define its set of out-neighbor nodes predicated by p to be $outNeighbors_p(v) \rightleftharpoons \{w \in N_i \mid \exists s \in outSlots(v) \,.\, \exists t \in \Sigma \,.\, target_i(v,s) = (w,t) \wedge p((v,s),(w,t))\}$. The relation between v and any one of its out-neighbors $w \in outNeighbors_p(v)$ is written $v \underset{p}{\rightarrow} w$. Conversely, the set of in-neighbor nodes of v, as predicated by p, is $inNeighbors_p(v) \rightleftharpoons \{u \in N_i \mid v \in outNeighbors_p(u)\}$; and thus $u \in inNeighbors_p(v)$ may be written as $v \underset{p}{\leftarrow} u$. Furthermore, any sequence of (two or more) nodes $[v_0,\ \ldots,\ v_m]$ is called a (non-trivial) path predicated by p in G_i when, for every consecutive pair (v_j, v_{j+1}) in the sequence, it holds that $v_j \underset{p}{\rightarrow} v_{j+1}$, in which case one may write $v_0 \underset{p}{\rightarrow} v_1 \underset{p}{\rightarrow} \ldots \underset{p}{\rightarrow} v_m$ or, more succinctly, $v_0 \overset{+}{\underset{p}{\rightarrow}} v_m$. Finally, when there is a path such that $v \overset{+}{\underset{p}{\rightarrow}} v$, that path is said to form a cycle (predicated by p) in G_i. When these notations are used and no predicate is specified, assume the predicate always holds.

Consider $\delta_i((u,s),(v,t))$ to be a predicate defined by the expression $\neg(label_i(v) = \texttt{JOIN} \wedge t = def)$, which holds true for any pair of linked slots in G_i, except when the target is the definition of a JOIN node. Similarly, define $flow_i(from,to) \coloneqq color_i(from) = control$ (slots are linked by a control flow edge) and $dep_i(use,def) \coloneqq \delta_i(use,def) \ \wedge\ color_i(use) \neq control$

(slots are linked by a dependency edge). Then, let $Dependents_i(f)$ be the set of nodes which transitively depend on $f \in N_i$ (be it by using a value produced in the set or by receiving control flow from it). $Dependents_i(f)$ is the smallest set determined by the inference rules:

$$\frac{}{f \in Dependents_i(f)} \qquad \frac{w \in Dependents_i(f) \quad v \xrightarrow[dep_i]{} w}{v \in Dependents_i(f)} \qquad \frac{u \in Dependents_i(f) \quad v \xleftarrow[flow_i]{} u}{v \in Dependents_i(f)}$$

Within G_i, define the relations "v depends on f" as $v\ Depends_i\ f \iff v \in Dependents_i(f)$, and "$v$ is live in f" as $v\ Live_i\ f \iff label_i(f) = JOIN \wedge f \xrightarrow[\delta_i]{+} v$.

It is now possible to formally define the notion of a "scope" in the new IR, which associates a set of nodes to either the root of a multigraph G_i or to any one of its `JOIN` nodes. The "global scope" of G_i is, by definition, the set N_i, which may also be written $Scope_i(r_i)$. For all other nodes $\{f_0, ..., f_m\}$ such that $label_i(f_j) = \texttt{JOIN}$, the sets $\{Scope_i(f_0), ..., Scope_i(f_m)\}$ are the smallest which satisfy the set of constraints given by:

$$\frac{v\ Depends_i\ f_j}{v \in Scope_i(f_j)} \qquad \frac{v\ Live_i\ f_j \qquad v \in Scope_i(g)}{Scope_i(f_j) \subseteq Scope_i(g)}$$

In other words, a node which (transitively) depends on a `JOIN` node must be within its scope. Additionally, if a node belongs to some (including the global) scope and it is live in the body of a join pattern, then the `JOIN` node's scope must be nested within this outer scope.

A multigraph G_i is said to be well-structured if and only if each of its nodes is locally well structured and the whole respects additional global rules:

1. All internal cycles must go through a join pattern: in a well-structured multigraph, it holds that $\neg\exists v \in N_i\ .\ v \xrightarrow[\neg\varphi_i]{+} v$, where $\varphi_i((u,s),(v,t)) \coloneqq label_i(v) = \texttt{JOIN} \vee v = r_i$. In other words, removing all `JOIN` nodes from a multigraph also breaks its internal cycles.

2. Scope nesting induces an antisymmetric relation: let f, g denote arbitrary `JOIN`-labeled nodes, it must hold that $f \neq g \implies \neg(Scope_i(f) \subseteq Scope_i(g) \wedge Scope_i(g) \subseteq Scope_i(f))$, that is, different "functions" must have different scopes.

3. Macros are not self-recursive: no nodes other than a graph's root may correspond to that graph's body, $\neg\exists v \in N_i - \{r_i\}\ .\ label_i(v) = \texttt{MACRO}_{G_i}$.

Since their nodes and edges are disjoint by definition, different multigraphs interact only through IR macros. Define $macros(G_i) \rightleftharpoons \{G_m \mid \exists v \in N_i - \{r_i\}\ .\ label_i(v) = \texttt{MACRO}_{G_m}\}$ to be the set of other multigraphs G_m used as macros in G_i, which defines the relation $G_m \sqsubset G_i$. Whenever $macros(G_i)$ is not empty, G_i may be called an *abstract* multigraph; otherwise it is said to be *concrete*. Turning an abstract multigraph into an equivalent concrete one happens through macro expansion, where a new multigraph G_i' is created from G_i by substituting every node labeled as $\texttt{MACRO}_{G_m}$ in G_i for a subgraph equivalent to the body of G_m.

At last, one may define full programs in the IR as tuples $(\mathbb{G}, \mathbb{I}, uid, G_{main})$, where:

- $\mathbb{G}$ is a non-empty finite set of multigraphs $\{G_1, ..., G_n\}$, as defined above.
- $\mathbb{I}$ is a set of identifiers.
- $uid : \mathbb{G} \mapsto \mathbb{I}$ is a function mapping each multigraph to its identifier in $\mathbb{I}$.
- $G_{main} \in \mathbb{G}$ is a distinguished multigraph in the program.

A program $P = (\mathbb{G}, \mathbb{I}, uid, G_{main})$ is said to be concrete (resp. abstract) in case G_{main} is concrete (resp. abstract). Hence, the "main program" is defined by G_{main}, and other multigraphs serve as macro definitions. Additionally, P is said to be well-structured if and only if each multigraph in $\mathbb{G}$ is globally well-structured and P respects additional whole-program rules:

1. Each multigraph in a program is uniquely identified: *uid* must be an injective function.
2. Macros are not recursive: consider a directed graph whose set of nodes is $\mathbb{G}$ and whose edges are defined by the relation $G_m \sqsubset G_i$; this graph must not be cyclic.

As in ANDF, IR macros were devised to enable efficient program portability by preserving "high-level" information until as late as possible, and assigning unique identifiers to each multigraph also contributes to this purpose. Consider an IR macro being used in a compiler front end as a complex operation. In a portable program, this operation would be implemented in a machine-independent manner and encoded in a separate multigraph $G_{portable}$. Still, a special identifier in $\mathbb{I}$ may be assigned to that macro. If a compiler back end recognizes the special identifier, it may swap $G_{portable}$ for $G_{efficient}$, an implementation of the macro which is more efficient but may only result in the same program behavior when compiled to a specific machine. On the other hand, a back end which does not recognize the special identifier can still compile the program with the portable implementation. In the context of the new IR, a program P is portable if it carries a definition for each macro it uses – $\mathbb{G} \supseteq \bigcup_{G_i \in \mathbb{G}} macros(G_i)$ – and expanding these definitions in G_{main} preserves the program's intended behavior.

4.2.2 Execution Model

Different compiler IRs model program execution in their own ways. While linear IRs usually mimic the behavior of sequential computers, some algebraic IRs describe program execution as a series of tree rewrites. In CFGs, a program begins in some initial basic block and sequentially executes every instruction in that block; the last instruction either terminates the program or determines another basic block in which to continue this process. In the case of SSA-form CFGs, these rules are distorted by ϕ operations, which must execute atomically when control flow reaches the block containing them – this behavior has confused compiler writers in the past (BRIGGS et al., 1998). Meanwhile, both Click's IR and the VSDG use variants of Petri nets to model control flow (CLICK, 1995; LAWRENCE, 2007).

Despite Petri nets being well suited to model both sequential and concurrent processes, Calvert (2015, sec. 3.1.3) points out that deterministic conditional branches become difficult, and indirect procedure invocations nigh impossible to express in the static structure of a classic Petri net. Calvert's work then explores the join calculus as an alternative. Due to the dynamic nature of CPS, the new IR also follows this path.

Figure 23 – The join calculus (left) and an example program (right).

x, y		variables
M		simple expressions
P, Q	::=	processes
	$\bar{x}\langle M_0, ..., M_n\rangle$	send message on x
	$P \mid Q$	parallel composition
	$def\ \ D\ \ in\ \ P$	instantiate definition(s)
J, K	::=	join patterns
	$x\langle y_0, ..., y_n\rangle$	receive message on x
	$J\ \&\ K$	synchronization
D, E	::=	definitions
	$J \triangleright P$	reaction rule
	D, E	simultaneous definitions

$$
\begin{array}{ll}
def & k_1\langle y_1\rangle\ \&\ k_2\langle y_2\rangle \triangleright \overline{use_y}\langle y_1 + y_2\rangle\ , \\
 & use_y\langle y\rangle \qquad\qquad\ \triangleright\ ... \\
in & \\
 & \overline{f_1}\langle n-1, k_1\rangle \mid \overline{f_2}\langle n-2, k_2\rangle
\end{array}
$$

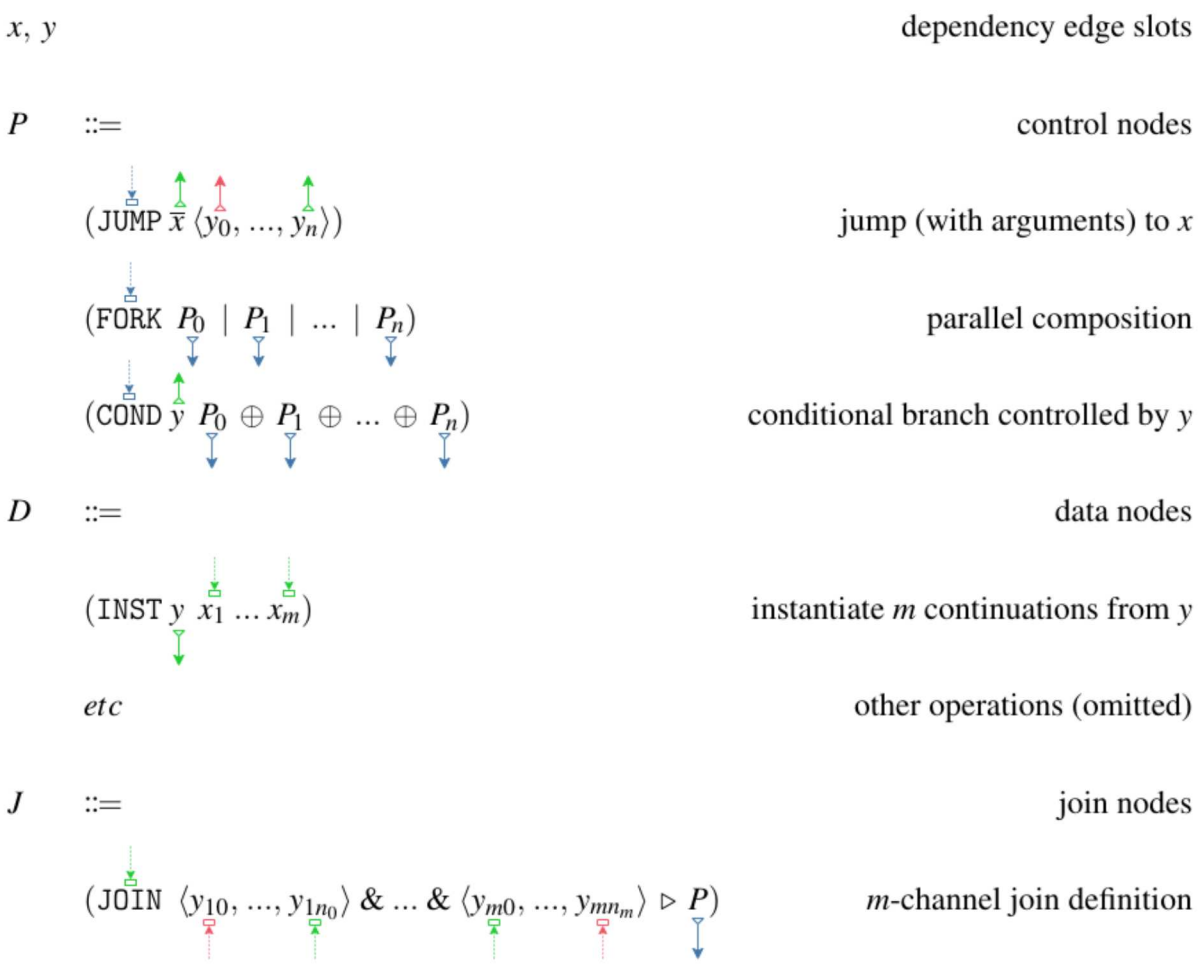
Source: Gabriel B. Sant'Anna, 2022.

In summary, programs in the join calculus are represented by multisets of live processes and reaction rules. Processes can introduce new reaction rules, fork into two or more concurrent processes, or terminate after sending a message on a channel. Live reaction rules trigger when every channel in their join pattern receives a message, in which case a new process starts with access to these messages' contents. Fournet & Gonthier (2000) provide a more detailed description of the calculus, whose core syntax is shown in Figure 23. When join patterns have a single channel, sending a message corresponds to "a `GOTO` which passes some data" (STEELE, 1976), also known as a tail call. In this context, sending a channel as part of a message *is* continuation-passing, which explains the join calculus' proximity to CPS and makes the example program in Figure 23 roughly equivalent to the multigraph illustrated by Figure 20.

In the new IR, three separate subgraphs can be identified within a well-structured multigraph: a control flow graph (through *control* edge slots), a data dependency graph (using both *data*- and *memory*- colored slots), and a type-level graph (through *type* slots); the first two of these drive the execution model. This model is illustrated in Figure 24, which reveals that non-macro nodes in the IR have been made in the image of the join calculus.

While the join calculus captures the semantics of all control flow edges in a concrete multigraph, the same does not hold for dependency edges. These represent partial results in a program, produced by primitive operations which do not directly affect control flow. Following

Figure 24 – The new IR in join calculus terms.



Source: Gabriel B. Sant'Anna, 2022.

Click's conceptual execution model, these operations produce their results in parallel and "at the speed of light" (CLICK; PALECZNY, 1995), as soon as their dependencies are available. This requires the data subgraph to form a DAG, which is always the case within the scope of a join node in a well-structured multigraph. As in Click's sea of nodes, memory operations require special treatment, since they must only produce their effects if the result is actively demanded (either directly or through a chain of dependencies) by a control operation.

4.2.3 Type System

Types in the new IR serve two purposes. The first is to reduce the number of node kinds: when nodes are parameterized by the type of their arguments and results (as in LLVM), there is no need to define multiple variants of every operation (as in WebAssembly). The second purpose is to apply further restrictions to the IR's graph structure in order to detect miscompilations. For example: `JUMP` nodes should not be able to have arbitrary bitvectors as their target continuation, and using a join definition as a subtraction operand is nonsensical. A non-goal of

the new IR's type system is safety. Whereas type-safe ILs can provide more guarantees about the behavior of well-typed programs, a universal representation must also account for unsafe programming languages – C being a notable example (LATTNER, 2002, p. 21). Therefore, the IR's type system does not have an accompanying soundness proof.

Figure 25 – A type grammar for the new IR.

J	$::=$	join types
	$\psi(X_1, ..., X_m)$	m-channel join, $m \geq 1$
X	$::=$	channel types
	$\chi(A_0, ..., A_n)$	n-parameter channel, $n \geq 0$
K	$::=$	continuation types
	$\kappa(A_0, ..., A_n)$	n-parameter continuation, $n \geq 0$
A	$::=$	parameter types
	M	memory
	S	sized type
	K	continuation type
S	$::=$	sized types
	$\mathbb{B}^w$	bitvector with w bits, $w \geq 1$
	$\Pi(S_1, ..., S_n)$	product type, $n \geq 1$
	ρ	pointer type

Source: Gabriel B. Sant'Anna, 2022.

In Figure 25, types are described syntactically, where type constructors receiving other types as parameters correspond to nodes with outgoing type dependency edges. All other non-macro nodes (those related to control and data operations) have a single *type*-colored slot, which is used to infer the type of each other slot in the node. This inference is guided by a set of typing rules, which are listed in Appendix B.

It should be noted that, in a well-structured multigraph, the type-level subgraph forms a DAG. Therefore, recursive types (e.g., binary trees) must be built with pointers. Furthermore, pointer types are "opaque", as in the most recent versions of LLVM (BLAIKIE, 2015). In summary, all dependencies typed as a pointer have the same type, even if memory operations using these pointers are consistent with respect to the types of values being loaded and stored. This reflects the fact that the IR is designed to accommodate unsafe languages, where, according to Blaikie (2015), assigning non-opaque types to pointers (for instance, having $\rho(T)$ instead of the opaque version, ρ) can be misleading for some compiler analyses and transformations.

4.3 PROTOTYPE IMPLEMENTATION

According to Braun, Buchwald & Zwinkau (2011), the main goal of a compiler IR is to support program optimizations. One way to approach this goal is making these optimizations an inherent part of the representation (BRAUN; BUCHWALD; ZWINKAU, 2011). This was kept in mind during the development of a prototype implementation of the new IR[14]. The prototype is designed to automatically apply some optimizations as the IR is being built.

4.3.1 On-the-fly Optimizations

In the classic compiler literature (AHO et al., 2006, p. 533), analyses and transformations are classified as either local or global; where the first kind is limited to a single CFG basic block. This classification is not applicable in the context of a sea-of-nodes graph, which does not have basic blocks. Global analyses are also commonly divided into intraprocedural and interprocedural (TANENBAUM et al., 1983; STANIER; WATSON, 2013), but even this distinction is not immediately clear in a CPS-based IR (KELSEY, 1995).

An alternative classification scheme considers two distinct groups: peephole analyses and global analyses; the former are limited to the extent of a fixed-size instruction "window", while the latter have access to the whole program (TANENBAUM et al., 1983). Although the size of a peephole is typically defined as an instruction count within a linear IR, Click (1995) suggests that the length of def-use chains could be used instead. Hence, in a sea-of-nodes graph, a peephole is "centered" in a certain node and extends towards its neighbors. The maximum number of edges traversed from the initial node to any other node in the peephole defines the extent of the transformation, which is global in case this number is unbounded.

Click (1995) and Braun, Buchwald & Zwinkau (2011) show that some optimizations (including global ones) can be applied on-the-fly, during the construction of an IR with the right structure. This work aimed to replicate these optimizations, but, at the time of writing, the prototype compiler is in a very early stage of development and not yet ready to produce quantitative results. Hence, this subsection is limited to the description of algorithms which implement some of those optimizations in the new IR.

The first optimization is Copy Propagation, which removes unnecessary assignments to local variables (e.g., `y = x`). Since the IR is in SSA form and has no copy operation, uses of a variable always point to its definition (in the example, uses of `y` would point directly to `x`). Therefore, this optimization is inherent to the IR (BRAUN; BUCHWALD; ZWINKAU, 2011). Operations which would effectively implement a copy when given constant arguments (addition with zero or multiplication by one, for instance) are handled during peephole optimizations.

Constant Propagation, Arithmetic Simplification and Weak Strength Reduction are combined into a single peephole optimization, which can be applied in constant time to nodes which do not involve *memory*- or *control*- colored edges. The optimization relies on the fact

[14] Source code and documentation available at `https://github.com/baioc/gyred/`

that data operations within the scope of a `JOIN` node form an acyclic subgraph: when nodes representing these operations are created in a reverse topological order (i.e., dependencies-first), each one can be individually optimized. Once a node's dependencies have been added, a peephole is drawn around it and extended to its immediate neighbors. Analysis then consists of a pattern matching process, which determines what transformation, if any, to apply.

Figure 26 displays pseudocode for an algorithm which implements this process, considering a simplified setting where an expression tree (with only unsigned integer constants and multiplication operations) is being compiled to a multigraph G. Subexpressions are visited in post-order, which is a valid reverse topological ordering of the data dependency subgraph. Finally, this algorithm could also be extended to certain control-related nodes – a `COND` node, for example, with a constant branch selector, could be constant-folded in a similar way.

Figure 26 – Combined peephole optimizations, simplified.

```
1  procedure CompileTo(expr, G)
2      if expr is an integer value then
3          return AddConstNode(G, expr.value)
4      else if expr is of the form left × right then
5          lhs ← CompileTo(expr.left, G)
6          rhs ← CompileTo(expr.right, G)
7          node ← AddMultNode(G, lhs, rhs)
8          return Peephole(G, node)
9      end if
10 end procedure
11
12 procedure Peephole(G, node)
13     if node.kind = MULT then
14         lhs ← node.lhs
15         rhs ← node.rhs
16         if lhs.kind = rhs.kind = CONST then                    // propagate constant
17             return AddConstNode(G, lhs.value × rhs.value)
18         else if rhs.kind = CONST ∧ rhs.value = 0 then          // simplify e × 0 to 0
19             return rhs
20         else if rhs.kind = CONST ∧ rhs.value = 1 then          // simplify e × 1 to e
21             return node.lhs
22         else if rhs.kind = CONST ∧ powerOf2(rhs.value) then    // reduce e × 2^n to e ≪ n
23             shift ← AddConstNode(G, log2(rhs.value))
24             return AddShlNode(G, lhs, shift)
25         else if ... then                                       // other patterns omitted for brevity
26             ...
27         end if
28     end if
29     ...
30     return node                                                // in case no rule applies, node is returned as is
31 end procedure
```

Source: Gabriel B. Sant'Anna, 2022.

Common Node Elimination has been implemented as an incremental global optimization, which avoids building certain subgraphs when an equivalent one already exists. This

equivalence check is performed every time a node is added to the IR, but it can be optimized if nodes are added in a reverse topological order. The implementation is based on hash consing, a technique used in some Lisp systems (DEUTSCH, 1976) to avoid redundant allocations by sharing values which are structurally equal.

Figure 27 – Caching structural hashes.

```
 1 procedure AddConstNode(G, constant)
 2     node ← NewNode(G)
 3     node.kind ← CONST
 4     node.value ← constant
 5     node.hash ← Hash(value)
 6     return AddNode(G, node)
 7 end procedure
 8
 9 procedure AddMultNode(G, lhs, rhs)
10     if lhs.kind = CONST ∧ rhs.kind ≠ CONST then
11         return AddMultNode(G, rhs, lhs)          // Peephole prefers constants in rhs
12     else
13         node ← NewNode(G)
14         node.kind ← MULT
15         Link(node.lhs, lhs.result)
16         Link(node.rhs, rhs.result)
17         node.hash ← lhs.hash ⊕ rhs.hash          // where ⊕ is commutative (e.g., bitwise xor)
18         return AddNode(G, node)
19     end if
20 end procedure
```

Source: Gabriel B. Sant'Anna, 2022.

In summary, every node added to the data subgraph caches its own hash code, assuming its dependencies are in the graph as well. At this point, the node becomes conceptually "frozen" and is treated as an immutable structure. Later, if an equivalent node would be added to the graph, an alias to the first one is returned instead. The implementation uses a hash table to check for duplicates, and the cached hash code avoids as many graph traversals as possible. Even in the case of hash collisions, comparing two nodes only takes time proportional to their outgoing edge count: since their dependencies have already been deduplicated by this same process, structural comparison reduces to a series of pointer (i.e., edge) comparisons. Figures 27 and 28 display pseudocode which implements this optimization. The simplified setting with only data-colored dependency edges makes this a form of Common Subexpression Elimination, although a full version could also deduplicate type-level operations and certain control-related nodes (for instance, `JUMP` nodes which send the same arguments to the same target continuation).

Another global optimization is Dead Node Elimination, which removes useless subgraphs (i.e., dead code) from the program. In summary, it uses a global analysis to determine a set of "live" nodes $\{v \in N_i \mid v = r_i \lor r_i \xrightarrow{+} v\}$. All nodes which are not in the live set are considered "dead". Then, it transforms the multigraph G_i by removing all of its dead nodes and any edges connected to them; this applies to any kind of node, including memory operations, control operations, type constructors, and even macros.

Figure 28 – Common Subexpression Elimination via hash consing.

```
1  procedure ADDNODE(G, node)
2      existing ← LOOKUP(G.hashTable, node)
3      if existing.found then
4          DELETENODE(G, node)                                  // common node eliminated
5          return existing.node
6      else
7          INSERT(G.hashTable, node)
8          return node
9      end if
10 end procedure
11
12 procedure EQVCONSTNODES(u, v)                                // invoked during hash table lookups
13     return u.value = v.value
14 end procedure
15
16 procedure EQVMULTNODES(u, v)
17     return {u.lhs, u.rhs} = {v.lhs, v.rhs}                   // order of factors does not alter the product
18 end procedure
```

Source: Gabriel B. Sant'Anna, 2022.

Due to the way in which edges are directed in the new IR, unreachable nodes correspond to dead code, which allows this optimization to be implemented as a garbage-collection algorithm. In the prototype implementation, Dead Node Elimination is automatically performed by a multigraph when it needs to expand its private node pool. Assuming that the reallocation process is relatively expensive, the garbage collector can turn this into an opportunity to move live nodes in memory such that def-use chains are packed closely together – this should improve the memory locality of certain graph traversals, in the interest of having a more efficient compiler. Figures 29 and 30 together implement a node allocator and garbage collector, where the latter applies Dead Node Elimination and compacts the multigraph at the same time.

Figure 29 – Bump allocation in a node pool.

```
1  procedure NEWNODE(G)
2      if G.cursor < G.arena.length then                        // if there are still nodes available, grab the first one
3          node ← GET(G.arena, G.cursor)
4          G.cursor ← G.cursor + 1
5          node.mark ← INITIAL
6          return node
7      else                                                     // otherwise, expand the arena, then try again
8          REALLOCATEARENA(G)
9          return NEWNODE(G)
10     end if
11 end procedure
12
13 procedure DELETENODE(G, node)
14     if node = GET(G.arena, G.cursor − 1) then
15         G.cursor ← G.cursor − 1
16     end if                                                   // non-LIFO deallocations are deferred to the garbage collector
17 end procedure
```

Source: Gabriel B. Sant'Anna, 2022.

Figure 30 – Dead Code Elimination as Garbage Collection.

```
1  procedure REALLOCATEARENA(G)
2      newArena ← ALLOCATEARENA(2 × max(1, G.arena.length))
3      newCursor ← 0
4
5      stack ← MAKESTACK()
6      for all node ∈ outNeighbors(G.root) do      // starting at the root node, mark reachable nodes
7          node.mark ← PENDING
8          PUSH(stack, node)
9      end for
10
11     while ¬EMPTY(stack) do                      // repeat until all nodes have been transferred
12         oldNode ← POP(stack)
13
14         for all neighbor ∈ outNeighbors(oldNode) do   // mark out-neighbors of a reachable node
15             if neighbor.mark = INITIAL then
16                 neighbor.mark ← PENDING
17                 PUSH(stack, neighbor)
18             end if
19         end for
20
21         newNode ← COPY(newArena, newCursor, oldNode)          // transfer the node ...
22         newCursor ← newCursor + 1
23         oldNode.forwardingPointer ← newNode          // ... and set up a forwarding pointer
24         oldNode.mark ← RELOCATED
25     end while
26     DESTROYSTACK(stack)
27
28     for all n ∈ [0, newCursor − 1] do          // then, forward all live edges to the new node pool
29         node ← GET(newArena, n)
30         node.mark ← INITIAL
31         for all neighbor ∈ outNeighbors(node) do
32             if neighbor.mark = RELOCATED then
33                 newNeighbor ← neighbor.forwardingPointer
34                 REWIRESLOTS(node, neighbor, newNeighbor)
35             end if
36         end for
37     end for
38
39     DEALLOCATEARENA(G.arena)                    // dead nodes have now been eliminated
40     G.arena ← newArena
41     G.cursor ← newCursor
42     UPDATEHASHCONSINGTABLE(G)
43 end procedure
```

Source: Gabriel B. Sant'Anna, 2022.

The pseudocode in Figure 30 implements a copying garbage collector (JONES; HOSKING; MOSS, 2011, ch. 4). Its main deviation from Cheney's classic algorithm is in the use of an auxiliary stack to traverse and copy live nodes in depth-first (instead of breadth-first) fashion.

5 CONCLUSION

> The best way to implement the future is to avoid having to predict it.
> (PIUMARTA, 2005)

5.1 SUMMARY

This work is motivated by indications of an upcoming boom in the diversity of hardware architectures (HENNESSY; PATTERSON, 2019) and programming languages (LATTNER, 2021). After a historical retrospective on the concept of UNCOL and its influence in compiler technology, it is argued that a universal IL may still be a viable solution to the (apparently inevitable) compiler construction problem. One of the main arguments for revisiting UNCOL is that even its proposed alternatives benefit from (and often require) a common IL.

The main contribution of the thesis is a survey on compiler IRs. Unlike related works, it covers functional and algebraic IRs, while describing techniques used in both modern and old compilers to increase the efficiency *and* the portability of programs. Research leading up to this survey can also be seen as a means to an end: understanding portable compiler IRs, identifying design principles and eliciting requirements to guide modern iterations on the UNCOL idea. These principles and requirements, extracted from the literature review, are described informally, which makes them hard to evaluate in practical terms. Still, a rationale for these exploratory results is communicated in the text to the best of the author's capabilities.

The author's initial hypothesis was that one or more suitable UNCOL candidates – compiler IRs which meet the requirements of a practical universal representation – would have been found during the systematic review. While that does not seem to be the case, the evolution of compiler technology over the last 60 years has highlighted some techniques (theoretical frameworks, data structures, analyses and transformations) which may facilitate the achievement of these goals. Using some of these techniques, and in an attempt to strengthen the validity of the aforementioned design principles and requirements, a new compiler IR is presented.

This work formally describes the new IR's structure. Then, its interpretation – the model of computation exposed by the IR – is expressed in terms of existing models. The resulting design mixes together ideas from many different sources, which leads to certain deviations from classic (CFG-based) approaches; one notable example is the implementation of global Dead Code Elimination as a garbage collection algorithm. Finally, the IR began to be implemented in a compiler middle end, of which an early prototype has been made publicly available as part of this thesis.

5.2 FUTURE WORK

In the author's opinion, this thesis has uncovered a couple of future work possibilities. The obvious one is further development of the new IR's prototype compiler implementation,

which is yet to reach a stage mature enough for empirical evaluation. One concern with the implicitly-optimizing approach is finding a way to expose a programming interface which does not lead to surprises when, for example, constructing an arithmetic operation node yields back an alias to an existing constant (supposedly because the requested operation was immediately constant folded). Another engineering concern lies in how to proceed with the evaluation of an IR designed to support multiple languages and machine architectures.

Jordan et al. (2013) explain why thorough experimental comparisons of compiler IRs are impractical: such an endeavor would require, for the complete set of investigated IRs, comparable implementations of all possible analyses and transformations – that, as well as an extensive set of benchmark programs to exercise all of these optimizations. Hence, new IR designs are usually evaluated by the proxy of a compiler middle end, which uses said IR to translate some high-level language into optimized machine code. Then, the special case of a portable (would-be universal) IR leads back to the $N \times M$ problem: one possible way to evaluate such an IR would be by reproducing the compiler construction problem at a small scale, choosing a small, yet diverse, set of languages on both ends. Once $N \times M$ translation pathways are made available through the middle end, empirical results can be obtained by combining standard benchmark and compiler performance metrics for each source-target pair.

A more pragmatic approach, as seen in the work of Reissmann et al. (2020), would be to use existing compiler infrastructures to test the effect of optimizations being performed on the new IR. Following their example, LLVM IR could be used as both input and output to a prototype compiler, and experimental results measured in terms of the baseline compilation pipeline. While this is certainly a faster way to evaluate certain optimizations, it does not have a clear connection to the compiler construction problem.

Another future work possibility is expanding the use of some theoretical frameworks to guide the design of IR features and transformations. Earlier in the text, the work of Felleisen (1991) was cited due to its presentation of a theory of expressiveness, which can be used to find redundancies in programming language constructs. Applying this to a compiler IR would be an innovative experiment, which in turn would require a precise description of an IR's model of computation.

In fact, the IR presented in this thesis could also benefit from a more formal treatment, particularly with respect to multiple uses of instantiated continuations and the semantics of explicit concurrency in the control-flow subgraph. Only then would it be possible to design complete algorithms for the construction (i.e., generation) and destruction (i.e., lowering) of the IR. Additionally, this text is often imprecise in its use of terms such as "equivalent", appealing to the reader's intuition in order to relate IR structure to program behavior. There are multiple formal alternatives – especially so in the context of process calculi (FOURNET; GONTHIER, 2000) – which could be used instead. Doing so would enable formal reasoning about semantics-preserving program transformations, which is precisely what compiler optimizations are supposed to be.

BIBLIOGRAPHY

ABELSON, H.; SUSSMAN, G. J. **Structure and Interpretation of Computer Programs**. 2nd. ed. MIT Press, 1996. ISBN 0262510871. Disponível em: https://mitpress.mit.edu/sites/default/files/sicp/full-text/book/book.html.

ADAMS, N. et al. ORBIT: An Optimizing Compiler for Scheme. In: **Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction**. New York, NY, USA: Association for Computing Machinery, 1986. (SIGPLAN '86), p. 219–233. ISBN 0897911970.

ADVE, S. V.; BOEHM, H.-J. Memory Models: A Case for Rethinking Parallel Languages and Hardware. **Communications of the ACM**, Association for Computing Machinery, New York, NY, USA, v. 53, n. 8, p. 90–101, aug 2010. ISSN 0001-0782. Disponível em: https://cacm.acm.org/magazines/2010/8/96610-memory-models-a-case-for-rethinking-parallel-languages-and-hardware/fulltext.

AHO, A. et al. **Compilers: Principles, Techniques, and Tools**. 2nd. ed. Boston, MA, USA: Addison Wesley, 2006. ISBN 0321486811.

APPEL, A. W. SSA is Functional Programming. **SIGPLAN Notices**, Association for Computing Machinery, New York, NY, USA, v. 33, n. 4, p. 17–20, apr 1998. ISSN 0362-1340.

AUSLANDER, M.; HOPKINS, M. An Overview of the PL.8 Compiler. **SIGPLAN Notices**, Association for Computing Machinery, New York, NY, USA, v. 17, n. 6, p. 22–31, jun 1982.

BACKUS, J. The History of Fortran I, II, and III. In: **History of Programming Languages**. New York, NY, USA: Association for Computing Machinery, 1978. p. 25–74. ISBN 0127450408. Disponível em: https://dl.acm.org/doi/pdf/10.1145/800025.1198345.

BAGLEY, P. R. Principles and Problems of a Universal Computer-Oriented Language. **The Computer Journal**, v. 4, n. 4, p. 305–312, 01 1962. ISSN 0010-4620. Disponível em: https://academic.oup.com/comjnl/article-pdf/4/4/305/8201605/040305.pdf.

BAHMANN, H. et al. Perfect Reconstructability of Control Flow from Demand Dependence Graphs. **ACM Transactions on Architecture and Code Optimization**, Association for Computing Machinery, New York, NY, USA, v. 11, n. 4, jan 2015. ISSN 1544-3566. Disponível em: https://dl.acm.org/doi/pdf/10.1145/2693261.

BAKER, H. G. CONS Should Not CONS Its Arguments, Part II: Cheney on the M.T.A. **SIGPLAN Notices**, Association for Computing Machinery, New York, NY, USA, v. 30, n. 9, p. 17–20, sep 1995. ISSN 0362-1340.

BELWAL, M.; TSB, S. Intermediate representation for heterogeneous multi-core: A survey. In: **International Conference on VLSI Systems, Architecture, Technology and Applications (VLSI-SATA)**. Bengaluru, India: IEEE, 2015. p. 1–6.

BLAIKIE, D. **Opaque Pointer Types**. San Jose, CA, USA: [s.n.], 2015. LLVM Developers' Meeting. Disponível em: https://www.youtube.com/watch?v=ePu6c4FLc9I.

BOEHM, H.-J. Threads Cannot Be Implemented as a Library. **SIGPLAN Notices**, Association for Computing Machinery, New York, NY, USA, v. 40, n. 6, p. 261–268, jun 2005. ISSN 0362-1340. Disponível em: https://www.hpl.hp.com/techreports/2004/HPL-2004-209.pdf.

BRATMAN, H. A Alternate Form of the "UNCOL Diagram". **Communications of the ACM**, Association for Computing Machinery, New York, NY, USA, v. 4, n. 3, p. 142, mar 1961.

BRAUN, M.; BUCHWALD, S.; ZWINKAU, A. **FIRM–A Graph-Based Intermediate Representation**. Chamonix, France, 2011. Workshop on Intermediate Representations. Disponível em: http://beza1e1.tuxen.de/pdfs/braun11wir.pdf.

BRIGGS, P. et al. Practical Improvements to the Construction and Destruction of Static Single Assignment Form. **Software – Practice & Experience**, John Wiley & Sons, Inc., USA, v. 28, n. 8, p. 859–881, jul 1998. ISSN 0038-0644. Disponível em: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.49.9683&rep=rep1&type=pdf.

BROOKER, R. et al. The Compiler Compiler. **Annual Review in Automatic Programming**, v. 3, p. 229–275, 1963. ISSN 0066-4138. Disponível em: http://curation.cs.manchester.ac.uk/atlas/elearn.cs.man.ac.uk/_atlas/docs/Compiler%20Compiler%20Paper%201963%20Annual%20Review%20of%20Auto%20Prog.pdf.

CALVERT, P. R. **Architecture-neutral parallelism via the Join Calculus**. Cambridge, United Kingdom, 2015. Disponível em: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-871.pdf.

CARTER, L.; FERRANTE, J.; THOMBORSON, C. Folklore confirmed: Reducible flow graphs are exponentially larger. In: **Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages**. New York, NY, USA: Association for Computing Machinery, 2003. (POPL '03), p. 106–114. ISBN 1581136285.

CATTELL, R. G. Automatic Derivation of Code Generators from Machine Descriptions. **ACM Transactions on Programming Languages and Systems**, Association for Computing Machinery, New York, NY, USA, v. 2, n. 2, p. 173–190, abr. 1980. ISSN 0164-0925.

CHISNALL, D. C Is Not a Low-Level Language: Your Computer is Not a Fast PDP-11. **ACM Queue**, Association for Computing Machinery, New York, NY, USA, v. 16, n. 2, p. 18–30, apr 2018. ISSN 1542-7730. Disponível em: https://dl.acm.org/doi/pdf/10.1145/3212477.3212479.

CHOW, F. Intermediate Representation: The Increasing Significance of Intermediate Representations in Compilers. **ACM Queue**, Association for Computing Machinery, New York, NY, USA, v. 11, n. 10, p. 30–37, oct 2013. ISSN 1542-7730. Disponível em: https://dl.acm.org/doi/pdf/10.1145/2542661.2544374.

CHURCH, A. An Unsolvable Problem of Elementary Number Theory. **American Journal of Mathematics**, Johns Hopkins University Press, v. 58, n. 2, p. 345–363, 1936. ISSN 00029327. Disponível em: https://doi.org/10.2307/2371045.

CLICK, C.; PALECZNY, M. A Simple Graph-Based Intermediate Representation. In: **Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations**. New York, NY, USA: Association for Computing Machinery, 1995. (IR '95), p. 35–49. ISBN 0897917545. Disponível em: https://www.oracle.com/technetwork/java/javase/tech/c2-ir95-150110.pdf.

CLICK, C. N. J. **Combining Analysis, Combining Optimizations**. Houston, TX, USA: Rice University, 1995. Disponível em: https://scholarship.rice.edu/bitstream/handle/1911/96451/TR95-252.pdf.

COLEMAN, S. S.; POOLE, P. C.; WAITE, W. M. The Mobile Programming System, Janus. **Software: Practice and Experience**, v. 4, n. 1, p. 5–23, 1974.

CONG, Y. et al. Compiling with Continuations, or without? Whatever. **Proceedings of the ACM on Programming Languages**, Association for Computing Machinery, New York, NY, USA, v. 3, n. ICFP, jul 2019. Disponível em: https://dl.acm.org/doi/pdf/10.1145/3341643.

CONWAY, M. E. Proposal for an uncol. **Communications of the ACM**, Association for Computing Machinery, New York, NY, USA, v. 1, n. 10, p. 5–8, oct 1958. ISSN 0001-0782.

CORMEN, T. H. et al. **Introduction to Algorithms**. 3rd. ed. USA: The MIT Press, 2009. ISBN 9780262533058.

COUSINEAU, G.; CURIEN, P.-L.; MAUNY, M. The Categorical Abstract Machine. **Science of Computer Programming**, v. 8, n. 2, p. 173–202, 1987. ISSN 0167-6423.

DEUTSCH, P. L. **An Interactive Program Verifier**. CSL-73-1. Palo Alto, CA, USA, 1976. Disponível em: https://www.softwarepreservation.org/projects/verification/pivot/Deutsch-An_Interactive_Theorem_Prover-1973.pdf.

DOLAN, S.; SIVARAMAKRISHNAN, K.; MADHAVAPEDDY, A. Bounding Data Races in Space and Time. In: **Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation**. New York, NY, USA: Association for Computing Machinery, 2018. (PLDI 2018), p. 242–255. ISBN 9781450356985.

DOWNEN, P. et al. Sequent Calculus as a Compiler Intermediate Language. **SIGPLAN Notices**, Association for Computing Machinery, New York, NY, USA, v. 51, n. 9, p. 74–88, sep 2016. ISSN 0362-1340. Disponível em: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/04/sequent-calculus-icfp16.pdf.

DUBOSCQ, G. et al. An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler. In: **Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages**. New York, NY, USA: Association for Computing Machinery, 2013. (VMIL '13), p. 1–10. ISBN 9781450326018.

ECMA-335. **ECMA-335 - Common Language Infrastructure (CLI)**. 6th. ed. [S.l.], 2012. Disponível em: https://www.ecma-international.org/publications-and-standards/standards/ecma-335/.

ECMA-94. **ECMA-94 - 8-bit single-byte coded graphic character set**. 1st. ed. [S.l.], 1985. Disponível em: https://www.ecma-international.org/publications-and-standards/standards/ecma-94/.

FARVARDIN, K.; REPPY, J. A New Backend for Standard ML of New Jersey. In: **IFL 2020: Proceedings of the 32nd Symposium on Implementation and Application of Functional Languages**. New York, NY, USA: Association for Computing Machinery, 2020. (IFL 2020), p. 55–66. ISBN 9781450389631. Disponível em: https://dl.acm.org/doi/pdf/10.1145/3462172.3462191.

FELDMAN, S. A Conversation with Alan Kay: Big Talk with the Creator of Smalltalk - and Much More. **ACM Queue**, Association for Computing Machinery, New York, NY, USA, v. 2, n. 9, p. 20–30, dec 2004. ISSN 1542-7730. Disponível em: https://queue.acm.org/detail.cfm?id=1039523.

FELLEISEN, M. On the expressive power of programming languages. **Science of Computer Programming**, v. 17, n. 1, p. 35–75, 1991. ISSN 0167-6423. Disponível em: https://doi.org/10.1016/0167-6423(91)90036-W.

FERRANTE, J.; OTTENSTEIN, K. J.; WARREN, J. D. The Program Dependence Graph and Its Use in Optimization. **ACM Transactions on Programming Languages and Systems**, Association for Computing Machinery, New York, NY, USA, v. 9, n. 3, p. 319–349, jul 1987. ISSN 0164-0925. Disponível em: https://doi.org/10.1145/24039.24041.

FESSANT, F. L.; MARANGET, L. Compiling Join-Patterns. **Electronic Notes in Theoretical Computer Science**, v. 16, n. 3, p. 205–224, 1998. ISSN 1571-0661. HLCL '98, 3rd International Workshop on High-Level Concurrent Languages (Satellite Workshop of CONCUR '98). Disponível em: https://doi.org/10.1016/S1571-0661(04)00143-4.

FLANAGAN, C. et al. The Essence of Compiling with Continuations. In: **Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation**. New York, NY, USA: Association for Computing Machinery, 1993. (PLDI '93), p. 237–247. ISBN 0897915984.

FORRESTER, J. W. Digital Computers: Present and Future Trends. In: **Papers and Discussions Presented at the Dec. 10-12, 1951, Joint AIEE-IRE Computer Conference: Review of Electronic Digital Computers**. New York, NY, USA: Association for Computing Machinery, 1951. (AIEE-IRE '51), p. 109–114. ISBN 9781450378512.

FOURNET, C.; GONTHIER, G. The Join Calculus: a Language for Distributed Mobile Programming. In: BARTHE, G. et al. (Ed.). **Applied Semantics. International Summer School, APPSEM 2000,**. Berlin, Germany: Springer, 2000. p. 268–332. ISBN 978-3-540-45699-5. Disponível em: https://www.microsoft.com/en-us/research/wp-content/uploads/2017/01/join-tutorial.pdf.

FRANZ, M. **Code-generation On-the-fly: A Key to Portable Software**. Zurich: Swiss Federal Institute of Technology in Zurich, 1994. ISBN 9783728121158.

FRANZ, M.; KISTLER, T. Slim Binaries. **Communications of the ACM**, Association for Computing Machinery, New York, NY, USA, v. 40, n. 12, p. 87–94, dec 1997.

FSF. **GNU Compiler Collection (GCC) Internals**. Free Software Foundation, Inc., 2022. Disponível em: https://gcc.gnu.org/onlinedocs/gccint/.

FUTAMURA, Y. Partial Evaluation of Computation Process – An Approach to a Compiler-Compiler. **Higher Order and Symbolic Computation**, Kluwer Academic Publishers, USA, v. 12, n. 4, p. 381–391, dec 1999. ISSN 1388-3690. Disponível em: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.10.2747&rep=rep1&type=pdf.

GANAPATHI, M.; FISCHER, C. N.; HENNESSY, J. L. Retargetable Compiler Code Generation. **ACM Computing Surveys**, Association for Computing Machinery, New York, NY, USA, v. 14, n. 4, p. 573–592, dec 1982. ISSN 0360-0300.

GENTZEN, G. Investigations into Logical Deduction. **American Philosophical Quarterly**, North American Philosophical Publications, University of Illinois Press, v. 1, n. 4, p. 288–306, 1964. ISSN 00030481. Disponível em: http://www.jstor.org/stable/20009142.

GHARACHORLOO, K. et al. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. **SIGARCH Computer Architecture News**, Association for Computing Machinery, New York, NY, USA, v. 18, n. 2SI, p. 15–26, may 1990. ISSN 0163-5964. Disponível em: https://doi.org/10.1145/325096.325102.

GILOI, W. K. Konrad Zuse's Plankalkül: The First High-Level, "non von Neumann" Programming Language. **IEEE Annals of the History of Computing**, IEEE Computer Society, Los Alamitos, CA, USA, v. 19, n. 02, p. 17–24, April 1997. ISSN 1934-1547.

GIVEN, D. **The Amsterdam Compiler Kit**. 2005. Disponível em: http://tack.sourceforge.net/.

GOLDBERG, C. J. **The Design of a Custom 32-bit RISC CPU and LLVM Compiler Backend**. Rochester, NY, USA, 2017. Disponível em: https://scholarworks.rit.edu/theses/9550.

GORN, S.; BEMER, R. W.; GREEN, J. American Standard Code for Information Interchange. **Communications of the ACM**, Association for Computing Machinery, New York, NY, USA, v. 6, n. 8, p. 422–426, aug 1963. ISSN 0001-0782.

GRAHAM, B. **SECD: Design Issues**. Calgary, Canada, 1989. Disponível em: https://prism.ucalgary.ca/handle/1880/46590.

GYLLSTROM, H. C. et al. The Universal Compiling System. **SIGPLAN Notices**, Association for Computing Machinery, New York, NY, USA, v. 14, n. 12, p. 64–70, dec 1979.

HAAS, A. et al. Bringing the Web up to Speed with WebAssembly. **SIGPLAN Notices**, Association for Computing Machinery, New York, NY, USA, v. 52, n. 6, p. 185–200, jun 2017.

HADDON, B. K.; WAITE, W. M. Experience with the Universal Intermediate Language Janus. **Software: Practice and Experience**, v. 8, n. 5, p. 601–616, 1978.

HARPER, R. **Practical Foundations for Programming Languages**. 3rd. ed. Cambridge University Press, 2016. Disponível em: http://www.cs.cmu.edu/~rwh/pfpl/2nded.pdf.

HENDERSON, P. **Functional Programming Application and Implementation**. London, United Kingdom: Prentice Hall, 1980. ISBN 0133315797.

HENNESSY, J. L.; PATTERSON, D. A. A New Golden Age for Computer Architecture. **Communications of the ACM**, Association for Computing Machinery, New York, NY, USA, v. 62, n. 2, p. 48–60, jan. 2019. ISSN 0001-0782. Disponível em: https://cacm.acm.org/magazines/2019/2/234352-a-new-golden-age-for-computer-architecture/fulltext.

HOARE, C. A. R. Hints on Programming-Language Design. In: **Essays in Computing Science**. USA: Prentice-Hall, Inc., 1989. ISBN 0132840278. Disponível em: https://dl.acm.org/doi/abs/10.5555/63445.C1104368.

HOPCROFT, J.; MOTWANI, R.; ULLMAN, J. **Introduction to Automata Theory, Languages, and Computation**. 3rd. ed. USA: Pearson, 2006. ISBN 0321455363.

HSA. **HSA Programmer's Reference Manual: HSAIL Virtual ISA and Programming Model, Compiler Writer, and Object Format (BRIG)**. Version 1.2. [S.l.], 2018. Disponível em: http://hsa.glossner.org/wp-content/uploads/2021/02/HSA-PRM-1.2.pdf.

IERUSALIMSCHY, R.; FIGUEIREDO, L. H. de; CELES, W. The Implementation of Lua 5.0. **Journal of Universal Computer Science**, Rio de Janeiro, Brazil, v. 11, n. 7, 2005.

ISO/IEC 9899. **Information technology - Programming languages - C**. N2176. [S.l.], 2017. Disponível em: https://web.archive.org/web/20181230041359/http://www.open-std.org/jtc1/sc22/wg14/www/abq/c17_updated_proposed_fdis.pdf.

JOHNSON, N. E. **Code size optimization for embedded processors**. Cambridge, United Kingdom, 2004. Disponível em: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-607.pdf.

JONES, R.; HOSKING, A.; MOSS, E. **The Garbage Collection Handbook: The Art of Automatic Memory Management**. 1st. ed. [S.l.]: Chapman and Hall/CRC, 2011. ISBN 9781420082791.

JORDAN, H. et al. INSPIRE: The Insieme Parallel Intermediate Representation. In: **Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques**. [S.l.]: IEEE Press, 2013. (PACT '13), p. 7–18.

KEGLEY, R. Software Extensibility and Portability with Macro Processors. In: **Proceedings of the 15th Annual Southeast Regional Conference**. New York, NY, USA: Association for Computing Machinery, 1977. (ACM-SE 15), p. 250–263. ISBN 9781450373029.

KELSEY, R. A. A Correspondence between Continuation Passing Style and Static Single Assignment Form. In: **Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations**. New York, NY, USA: Association for Computing Machinery, 1995. (IR '95), p. 13–22. ISBN 0897917545.

KENNEDY, A. Compiling with Continuations, Continued. **SIGPLAN Notices**, Association for Computing Machinery, New York, NY, USA, v. 42, n. 9, p. 177–190, oct 2007.

KESSENICH, J. **An Introduction to SPIR-V – A Khronos-Defined Intermediate Language for Native Representation of Graphical Shaders and Compute Kernels**. [S.l.], 2015. Disponível em: https://www.khronos.org/registry/SPIR-V/papers/WhitePaper.pdf.

KHALDI, D. et al. SPIRE : A Methodology for Sequential to Parallel Intermediate Representation Extension. In: **6ème rencontre de la communauté française de compilation**. Annecy, France: [s.n.], 2013.

KHRONOS. **SPIR Overview**. Khronos Group, 2020. Disponível em: https://www.khronos.org/spir/.

KOOPMAN, P. J. J. **Stack Computers: the new wave**. Pittsburgh, PA, USA: Ellis Horwood, 1989. Disponível em: https://users.ece.cmu.edu/~koopman/stack_computers/.

LANDIN, P. J. The Mechanical Evaluation of Expressions. **The Computer Journal**, v. 6, n. 4, p. 308–320, 01 1964. ISSN 0010-4620. Disponível em: https://dl.acm.org/doi/pdf/10.1145/3140587.3062363.

LARABEL, M. **Rust Lands Experimental Cranelift-Based Code Generator - Much Faster Debug Build Times**. Phoronix, 2020. Disponível em: https://www.phoronix.com/scan.php?page=news_item&px=Rust-Cranelift-Merged.

LATTNER, C. **LLVM: An Infrastructure for Multi-Stage Optimization**. Urbana, IL, USA: Computer Science Dept., University of Illinois at Urbana-Champaign, 2002. Disponível em: https://llvm.org/pubs/2002-12-LattnerMSThesis.pdf.

LATTNER, C. **The Golden Age of Compiler Design in an Era of HW/SW Co-design**. Lausanne, Switzerland: [s.n.], 2021. ASPLOS 2021 Keynote. Disponível em: https://www.youtube.com/watch?v=4HgShra-KnY.

LATTNER, C.; ADVE, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: **Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)**. Palo Alto, CA, USA: [s.n.], 2004. Disponível em: https://llvm.org/pubs/2004-01-30-CGO-LLVM.pdf.

LATTNER, C. et al. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In: **Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization**. Korea: IEEE Press, 2021. (CGO '21), p. 2–14. ISBN 9781728186139.

LAWRENCE, A. C. **Optimizing compilation with the Value State Dependence Graph**. Cambridge, United Kingdom, 2007. Disponível em: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-705.pdf.

LEE, P.; PLEBAN, U. A Realistic Compiler Generator Based on High-Level Semantics: Another Progress Report. In: **Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages**. New York, NY, USA: Association for Computing Machinery, 1987. (POPL '87), p. 284–295. ISBN 0897912152.

LEISSA, R.; KÖSTER, M.; HACK, S. A Graph-Based Higher-Order Intermediate Representation. In: **Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization**. San Francisco, CA, USA: IEEE Computer Society, 2015. (CGO '15), p. 202–212. ISBN 9781479981618.

LIEDTKE, J. et al. An Unconventional Proposal: Using the X86 Architecture as the Ubiquitous Virtual Standard Architecture. In: **Proceedings of the 8th ACM SIGOPS European Workshop on Support for Composing Distributed Applications**. New York, NY, USA: Association for Computing Machinery, 1998. (EW 8), p. 237–241. ISBN 9781450373173.

LINDHOLM, T. et al. **The Java Virtual Machine Specification**. Java SE 8 Edition (JSR-337). Redwood City, CA, USA, 2015. Disponível em: https://docs.oracle.com/javase/specs/jvms/se8/html/.

LLNL. **ROSE Compiler – Program Analysis and Transformation**. awrence Livermore National Laboratory, 2022. Disponível em: http://rosecompiler.org/.

LLVM. **The LLVM Compiler Infrastructure – Language Reference Manual**. LLVM Foundation, 2022. Disponível em: https://llvm.org/docs/LangRef.html.

LOGOZZO, F.; FAHNDRICH, M. On the Relative Completeness of Bytecode Analysis versus Source Code Analysis. In: **Proceedings of the International Conference on Compiler Construction**. Springer Verlag, 2008. Disponível em: https://www.microsoft.com/en-us/research/publication/on-the-relative-completeness-of-bytecode-analysis-versus-source-code-analysis/.

MACQUEEN, D.; HARPER, R.; REPPY, J. The History of Standard ML. **Proceedings of the ACM on Programming Languages**, Association for Computing Machinery, New York, NY, USA, v. 4, n. HOPL, jun 2020. Disponível em: https://dl.acm.org/doi/pdf/10.1145/3386336.

MACRAKIS, S. **Delivering applications to multiple platforms using ANDF**. [S.l.], 1993. Disponível em: http://www.tendra.org/Macrakis93-platforms.pdf.

MACRAKIS, S. **From UNCOL to ANDF: Progress in standard intermediate languages**. USA, 1993. Disponível em: http://www.tendra.org/Macrakis93-uncol.pdf.

MACRAKIS, S. **The structure of ANDF: Principles and examples**. [S.l.], 1993. Disponível em: http://www.tendra.org/Macrakis93-structure.pdf.

MAURER, L. et al. Compiling without Continuations. In: **Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation**. New York, NY, USA: Association for Computing Machinery, 2017. (PLDI 2017), p. 482–494. ISBN 9781450349888. Disponível em: https://dl.acm.org/doi/pdf/10.1145/3062341.3062380.

MCCARTHY, J. History of LISP. In: ____. **History of Programming Languages**. New York, NY, USA: Association for Computing Machinery, 1978. p. 173–185. ISBN 0127450408. Disponível em: https://dl.acm.org/doi/pdf/10.1145/800025.1198360.

MERRILL, J. **GENERIC and GIMPLE: A New Tree Representation for Entire Functions**. Ottawa, Canada, 2003. GCC Developers Summit. Disponível em: https://gcc.gnu.org/pub/gcc/summit/2003/GENERIC%20and%20GIMPLE.pdf.

MILNER, R. Elements of Interaction: Turing Award Lecture. **Communications of the ACM**, Association for Computing Machinery, New York, NY, USA, v. 36, n. 1, p. 78–89, jan 1993. ISSN 0001-0782. Disponível em: https://dl.acm.org/doi/pdf/10.1145/151233.151240.

MOGENSEN, T. Æ. **Basics Of Compiler Design)**. Copenhagen, Denmark: [s.n.], 2010. ISBN 9788799315406. Disponível em: http://hjemmesider.diku.dk/~torbenm/Basics/basics_lulu2.pdf.

NEWEY, M. C.; POOLE, P. C.; WAITE, W. M. Abstract Machine Modelling to Produce Portable Software - A Review and Evaluation. **Software: Practice and Experience**, v. 2, n. 2, p. 107–136, 1972.

NYSTROM, R. **Crafting Interpreters**. USA: Genever Benning, 2021. ISBN 9780990582939. Disponível em: http://craftinginterpreters.com/.

OFFNER, C. D. **Notes on Graph Algorithms Used in Optimizing Compilers**. Boston, MA, USA: [s.n.], 2013. Disponível em: https://www.cs.umb.edu/~offner/files/flow_graph.pdf.

OLIVA, D.; NORDIN, T.; JONES, S. P. C--: A Portable Assembly Language. In: **Proceedings of the 1997 Workshop on Implementing Functional Languages**. Springer Verlag, 1997. (LNCS, v. 1467), p. 1–19. Disponível em: https://www.microsoft.com/en-us/research/publication/c-a-portable-assembly-language/.

OTTENSTEIN, K. J.; BALLANCE, R. A.; MACCABE, A. B. The Program Dependence Web: A Representation Supporting Control-, Data-, and Demand-Driven Interpretation of Imperative Languages. **SIGPLAN Notices**, Association for Computing Machinery, New York, NY, USA, v. 25, n. 6, p. 257–271, jun 1990. ISSN 0362-1340. Disponível em: https://doi.org/10.1145/93548.93578.

PALAMIDESSI, C. Comparing the Expressive Power of the Synchronous and the Asynchronous π-Calculus. In: **Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages**. New York, NY, USA: Association for Computing Machinery, 1997. (POPL '97), p. 256–265. ISBN 0897918533. Disponível em: https://arxiv.org/pdf/1307.2062.pdf.

PALECZNY, M.; VICK, C.; CLICK, C. The Java Hotspot^{TM} Server Compiler. In: **Proceedings of the 2001 Symposium on Java Virtual Machine Research and Technology**

Symposium. Monterey, CA, USA: USENIX Association, 2001. (JVM'01, v. 1). Disponível em: https://www.usenix.org/legacy/events/jvm01/full_papers/paleczny/paleczny.pdf.

PATTERSON, D. A.; HENNESSY, J. L. **Computer Organization and Design: The Hardware/Software Interface (MIPS Edition)**. 5th. ed. USA: Morgan Kaufmann, 2013. ISBN 9780124077263.

PERKINS, D. R.; SITES, R. L. Machine-Independent PASCAL Code Optimization. **SIGPLAN Notices**, Association for Computing Machinery, New York, NY, USA, v. 14, n. 8, p. 201–207, aug 1979. ISSN 0362-1340.

PERLIS, A. J. Special Feature: Epigrams on Programming. **SIGPLAN Notices**, Association for Computing Machinery, New York, NY, USA, v. 17, n. 9, p. 7–13, sep 1982.

PERLIS, A. J. Foreword. In: **Structure and Interpretation of Computer Programs**. 2nd. ed. Cambridge, MA, USA: MIT Press, 1996. ISBN 0262510871.

PESCHANSKI, F. Parallel Computing with the Pi-Calculus. In: **Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming**. New York, NY, USA: Association for Computing Machinery, 2011. (DAMP '11), p. 45–54. ISBN 9781450304863. Disponível em: https://hal.archives-ouvertes.fr/hal-00628492/document.

PIERCE, B. C. **Types and Programming Languages**. 1st. ed. USA: The MIT Press, 2002. ISBN 0262162091.

PIUMARTA, I. **Making COLAs with Pepsi and Coke**. [S.l.], 2005. Disponível em: https://piumarta.com/software/cola/colas-whitepaper.pdf.

POUNTAIN, D. Configuring Parallel Programs – Part 1. **Byte Magazine**, v. 14, n. 13, p. 349–352, dez. 1989. ISSN 0360-5280. Disponível em: https://archive.org/details/byte-magazine-1989-12/page/n382/mode/1up.

REISSMANN, N. et al. RVSDG: An Intermediate Representation for Optimizing Compilers. **ACM Transactions on Embedded Computing Systems**, Association for Computing Machinery, New York, NY, USA, v. 19, n. 6, dec 2020. ISSN 1539-9087.

RISC-V. **The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA**. Document version 20191213. [S.l.], 2019. Disponível em: https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf.

ROSEN, B. K.; WEGMAN, M. N.; ZADECK, F. K. Global Value Numbers and Redundant Computations. In: **Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages**. New York, NY, USA: Association for Computing Machinery, 1988. (POPL '88), p. 12–27. ISBN 0897912527.

ROSS, D. T. Gestalt Programming: A New Concept in Automatic Programming. In: **Papers Presented at the February 7-9, 1956, Joint ACM-AIEE-IRE Western Computer Conference**. New York, NY, USA: Association for Computing Machinery, 1956. (AIEE-IRE '56 (Western)), p. 5–10. ISBN 9781450378581.

SAMMET, J. E.; HEMMENDINGER, D. Programming Languages. In: **Encyclopedia of Computer Science**. John Wiley and Sons Ltd., 2003. p. 1470–1475. ISBN 0470864125. Disponível em: https://dl.acm.org/doi/10.5555/1074100.1074735.

SAVAGE, J. E. **Models of Computation: Exploring the Power of Computing**. 1st. ed. USA: Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN 0201895390. Disponível em: https://cs.brown.edu/people/jsavage/book/pdfs/ModelsOfComputation.pdf.

SCHMIDT, U.; VÖLLER, R. A Multi-Language Compiler System with Automatically Generated Codegenerators. **SIGPLAN Notices**, Association for Computing Machinery, New York, NY, USA, v. 19, n. 6, p. 202–212, jun. 1984. ISSN 0362-1340.

SHIVERS, O. Control Flow Analysis in Scheme. In: **Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation**. New York, NY, USA: Association for Computing Machinery, 1988. (PLDI '88), p. 164–174.

STANIER, J.; WATSON, D. A study of irreducibility in C programs. **Software: Practice and Experience**, v. 42, n. 1, p. 117–130, 2012.

STANIER, J.; WATSON, D. Intermediate Representations in Imperative Compilers: A Survey. **ACM Computing Surveys**, Association for Computing Machinery, New York, NY, USA, v. 45, n. 3, jul 2013. ISSN 0360-0300.

STEEL, T. B. A First Version of UNCOL. In: **Papers Presented at the May 9-11, 1961, Western Joint IRE-AIEE-ACM Computer Conference**. New York, NY, USA: Association for Computing Machinery, 1961. (IRE-AIEE-ACM '61), p. 371–378. ISBN 9781450378727.

STEEL, T. B. UNCOL: The Myth and the Fact. In: GOODMAN, R. (Ed.). **Annual Review in Automatic Programming**. [S.l.]: Elsevier, 1961, (International Tracts in Computer Science and Technology and Their Application, v. 2). p. 325–344.

STEELE, G. L. **LAMBDA: The Ultimate Declarative**. USA, 1976. AI Memo 379. Disponível em: https://apps.dtic.mil/sti/pdfs/ADA034090.pdf.

STEELE, G. L. **Rabbit: A Compiler for Scheme**. USA, 1978. Disponível em: https://dspace.mit.edu/bitstream/handle/1721.1/6913/AITR-474.pdf.

STEELE, G. L.; SUSSMAN, G. J. **Lambda: The Ultimate Imperative**. USA, 1976. AI Memo 353. Disponível em: https://dspace.mit.edu/bitstream/handle/1721.1/5790/AIM-353.pdf.

STRONG, J. et al. The Problem of Programming Communication with Changing Machines: A Proposed Solution. **Communications of the ACM**, Association for Computing Machinery, New York, NY, USA, v. 1, n. 8, p. 12–18, ago. 1958. ISSN 0001-0782.

STRONG, J. et al. The Problem of Programming Communication with Changing Machines: A Proposed Solution - Part 2. **Communications of the ACM**, Association for Computing Machinery, New York, NY, USA, v. 1, n. 9, p. 9–16, sep 1958. ISSN 0001-0782.

SUIF. **SUIF Compiler System**. Stanford SUIF Compiler Group, 2005. Disponível em: https://suif.stanford.edu/.

SUSUNGI, A.; TADONKI, C. Intermediate Representations for Explicitly Parallel Programs. **ACM Computing Surveys**, Association for Computing Machinery, New York, NY, USA, v. 54, n. 5, may 2021. ISSN 0360-0300.

TANENBAUM, A. S. et al. A Practical Tool Kit for Making Portable Compilers. **Communications of the ACM**, Association for Computing Machinery, New York, NY, USA, v. 26, n. 9, p. 654–660, sep 1983. ISSN 0001-0782.

TenDRA. **The TenDRA Project**. 2022. Disponível em: http://www.tendra.org/tdf-specification/.

TURING, A. M. On Computable Numbers, with an Application to the Entscheidungsproblem. **Proceedings of the London Mathematical Society**, s2-42, n. 1, p. 230–265, 1937. Disponível em: https://londmathsoc.onlinelibrary.wiley.com/doi/epdf/10.1112/plms/s2-42.1.230.

UNGER, S.; MUELLER, F. Handling Irreducible Loops: Optimized Node Splitting versus DJ-Graphs. **ACM Transactions on Programming Languages and Systems**, Association for Computing Machinery, New York, NY, USA, v. 24, n. 4, p. 299–333, jul 2002.

UNICODE. **History of Unicode**. The Unicode Consortium, 1991. Disponível em: https://www.unicode.org/history/.

WADLER, P. Propositions as Types. **Communications of the ACM**, Association for Computing Machinery, New York, NY, USA, v. 58, n. 12, p. 75–84, nov 2015. Disponível em: https://cacm.acm.org/magazines/2015/12/194626-propositions-as-types/fulltext.

WEISE, D. et al. Value Dependence Graphs: Representation without Taxation. In: . New York, NY, USA: Association for Computing Machinery, 1994. (POPL '94), p. 297–310. ISBN 0897916360. Disponível em: https://doi.org/10.1145/174675.177907.

WILSON, R. P. et al. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. **SIGPLAN Notices**, Association for Computing Machinery, New York, NY, USA, v. 29, n. 12, p. 31–37, dec 1994. ISSN 0362-1340.

WINGO, A. **Optimizing with Persistent Data Structures**. Hebden Bridge, UK: [s.n.], 2016. LLVM Cauldron Conference. Disponível em: https://www.youtube.com/watch?v=KYoDyM6l_5s.

WIRTH, N. Recollections about the Development of Pascal. **SIGPLAN Notices - History of Programming Languages (HOPL)**, Association for Computing Machinery, New York, NY, USA, v. 28, n. 3, p. 333–342, mar 1993. ISSN 0362-1340.

WÜRTHINGER, T. et al. Practical Partial Evaluation for High-Performance Dynamic Language Runtimes. **SIGPLAN Notices**, Association for Computing Machinery, New York, NY, USA, v. 52, n. 6, p. 662–676, jun 2017. ISSN 0362-1340.

ZAKAI, A. Emscripten: An LLVM-to-JavaScript Compiler. In: **Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion**. New York, NY, USA: Association for Computing Machinery, 2011. (OOPSLA '11), p. 301–312. ISBN 9781450309424. Disponível em: https://github.com/emscripten-core/emscripten/raw/main/docs/paper.pdf.

ZHAO, J.; SARKAR, V. Intermediate Language Extensions for Parallelism. In: **Proceedings of the Compilation of the Co-Located Workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11**. New York, NY, USA: Association for Computing Machinery, 2011. (SPLASH '11 Workshops), p. 329–340. ISBN 9781450311830.

ZHAO, J.; SARKAR, V. Intermediate Language Extensions for Parallelism. In: **Proceedings of the Compilation of the Co-Located Workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11**. New York, NY, USA: Association for Computing Machinery, 2011. (SPLASH '11 Workshops), p. 329–340. ISBN 9781450311830.

INDEX

APPENDIX A – NODE STRUCTURE

Every node v in the new IR has an associated "label", drawn from a finite set K_N. The following equations describe how the label of a well-structured node partially determines its shape: what edge slots it may have and the color associated to each of these (if present).

$$
\begin{aligned}
&label(v) = \texttt{JOIN} \implies \\
&\quad outSlots(v) = \{type,\ body\} \\
&\wedge\ color(v,type) = type \ \wedge\ color(v,body) = control \\
&\wedge\ inSlots(v) \subseteq \bigcup_{i=1}^{chan_{MAX}} \bigcup_{j=0}^{args_{MAX}} param_{ij} \cup \{def\} \\
&\wedge\ (def \in inSlots(v) \implies color(v,def) = data) \\
&\wedge\ (\forall i,j\ .\ param_{ij} \in inSlots(v) \implies color(v,param_{ij}) \in \{data, memory\})
\end{aligned}
$$

$$
\begin{aligned}
&label(v) = \texttt{INST} \implies \\
&\quad outSlots(v) = \{type,\ def\} \\
&\wedge\ color(v,type) = type \ \wedge\ color(v,def) = data \\
&\wedge\ inSlots(v) \subseteq \bigcup_{i=1}^{chan_{MAX}} cont_i \\
&\wedge\ (\forall i\ .\ cont_i \in inSlots(v) \implies color(v,cont_i) = data)
\end{aligned}
$$

$$
\begin{aligned}
&label(v) = \texttt{JUMP} \implies \\
&\quad outSlots(v) \subseteq \{type,\ target,\ arg_0,\ \ldots,\ arg_{args_{MAX}}\} \\
&\wedge\ type \in outSlots(v) \ \wedge\ color(v,type) = type \\
&\wedge\ target \in outSlots(v) \ \wedge\ color(v,target) = data \\
&\wedge\ (\forall j\ .\ arg_j \in outSlots(v) \implies color(v,arg_j) \in \{data, memory\}) \\
&\wedge\ inSlots(v) \subseteq \{ctrl\} \\
&\wedge\ (ctrl \in inSlots(v) \implies color(v,ctrl) = control)
\end{aligned}
$$

$$
\begin{aligned}
&label(v) = \texttt{FORK} \implies \\
&\quad outSlots(v) \subseteq \{thread_0,\ \ldots,\ thread_{fork_{MAX}}\} \\
&\wedge\ |outSlots(v)| \ \geq\ 2 \\
&\wedge\ (\forall i\ .\ thread_i \in outSlots(v) \implies color(v,thread_i) = control) \\
&\wedge\ inSlots(v) \subseteq \{ctrl\} \\
&\wedge\ (ctrl \in inSlots(v) \implies color(v,ctrl) = control)
\end{aligned}
$$

$$
\begin{aligned}
&label(v) = \texttt{COND} \implies \\
&\quad outSlots(v) \subseteq \{type,\ sel,\ br_0,\ ...,\ br_{2^{Wmax}-1}\} \\
&\wedge\ type \in outSlots(v)\ \wedge\ color(v,type) = type \\
&\wedge\ sel \in outSlots(v)\ \wedge\ color(v,sel) = data \\
&\wedge\ (\forall i\ .\ br_i \in outSlots(v)\ \implies\ color(v,br_i) = control) \\
&\wedge\ inSlots(v) \subseteq \{ctrl\} \\
&\wedge\ (ctrl \in inSlots(v)\ \implies\ color(v,ctrl) = control)
\end{aligned}
$$

$$
\begin{aligned}
&label(v) \in \{\texttt{MACRO}_{G_1},\ ...,\ \texttt{MACRO}_{G_{graph_{MAX}}}\} \implies \\
&\quad outSlots(v) \subseteq \{out_0,\ ...,\ out_{macro_{MAX}}\} \\
&\wedge\ inSlots(v) \subseteq \{in_0,\ ...,\ in_{macro_{MAX}}\}
\end{aligned}
$$

$$
\begin{aligned}
&label(v) = \texttt{CONST} \implies \\
&\quad outSlots(v) = \{type\}\ \wedge\ color(v,type) = type \\
&\wedge\ inSlots(v) \subseteq \{value\} \\
&\wedge\ (value \in inSlots(v)\ \implies\ color(v,value) = data)
\end{aligned}
$$

$$
\begin{aligned}
&label(v) \in \{\texttt{AND},\ \texttt{OR},\ \texttt{XOR},\ \text{`}\ll\text{'},\ \text{`}\gg\text{'},\ \text{`}+\text{'},\ \text{`}-\text{'},\ \text{`}\times\text{'},\ \text{`}\div\text{'},\ \text{`}\%\text{'},\ \text{`}=\text{'},\ \text{`}\neq\text{'},\ \text{`}<\text{'},\ \text{`}\geq\text{'}\} \implies \\
&\quad outSlots(v) = \{type,\ lhs,\ rhs\} \\
&\wedge\ color(v,type) = type \\
&\wedge\ color(v,lhs) = data\ \wedge\ color(v,rhs) = data \\
&\wedge\ inSlots(v) \subseteq \{result\} \\
&\wedge\ (result \in inSlots(v)\ \implies\ color(v,result) = data)
\end{aligned}
$$

$$
\begin{aligned}
&label(v) = \texttt{BITCAST} \implies \\
&\quad outSlots(v) = \{type,\ input\} \\
&\wedge\ color(v,type) = type \\
&\wedge\ color(v,input) = data \\
&\wedge\ inSlots(v) \subseteq \{output\} \\
&\wedge\ (output \in inSlots(v)\ \implies\ color(v,output) = data)
\end{aligned}
$$

$$
\begin{aligned}
&label(v) = \texttt{MUX} \implies \\
&\qquad outSlots(v) \subseteq \{type,\ sel, input_0,\ ...,\ input_{2^{Wmax}-1}\} \\
&\quad \wedge\ type \in outSlots(v)\ \wedge\ color(v,type) = type \\
&\quad \wedge\ sel \in outSlots(v)\ \wedge\ color(v,sel) = data \\
&\quad \wedge\ (\forall i\ .\ input_i \in outSlots(v) \implies color(v,input_i) = data) \\
&\quad \wedge\ inSlots(v) \subseteq \{output\} \\
&\quad \wedge\ (output \in inSlots(v) \implies color(v,output) = data)
\end{aligned}
$$

$$
\begin{aligned}
&label(v) = \texttt{TUPLE} \implies \\
&\qquad outSlots(v) \subseteq \{type,\ field_0,\ ...,\ field_{proj_{MAX}}\} \\
&\quad \wedge\ type \in outSlots(v)\ \wedge\ color(v,type) = type \\
&\quad \wedge\ (\forall i\ .\ field_i \in outSlots(v) \implies color(v,field_i) = data) \\
&\quad \wedge\ inSlots(v) \subseteq \{tuple\} \\
&\quad \wedge\ (tuple \in inSlots(v) \implies color(v,tuple) = data)
\end{aligned}
$$

$$
\begin{aligned}
&label(v) \in \{\texttt{PROJ}_0,\ ...,\ \texttt{PROJ}_{proj_{MAX}}\} \implies \\
&\qquad outSlots(v) = \{type,\ tuple\} \\
&\quad \wedge\ color(v,type) = type \\
&\quad \wedge\ color(v,tuple) = data \\
&\quad \wedge\ inSlots(v) \subseteq \{field\} \\
&\quad \wedge\ (field \in inSlots(v) \implies color(v,field) = data)
\end{aligned}
$$

$$
\begin{aligned}
&label(v) \in \{M,\ \rho,\ \mathbb{B}^1,\ ...,\ \mathbb{B}^{w_{MAX}}\} \implies \\
&\qquad inSlots(v) \subseteq \{type\} \\
&\quad \wedge\ (type \in inSlots(v) \implies color(v,type) = type)
\end{aligned}
$$

$$
\begin{aligned}
&label(v) = \Pi \implies \\
&\qquad outSlots(v) \subseteq \{T_0,\ ...,\ T_{proj_{MAX}}\} \\
&\quad \wedge\ (\forall i\ .\ T_i \in outSlots(v) \implies color(v,T_i) = type) \\
&\quad \wedge\ inSlots(v) \subseteq \{type\} \\
&\quad \wedge\ (type \in inSlots(v) \implies color(v,type) = type)
\end{aligned}
$$

$$
\begin{aligned}
&label(v) \in \{\kappa,\ \chi\} \implies \\
&\qquad outSlots(v) \subseteq \{T_0,\ \ldots,\ T_{args_{MAX}}\} \\
&\quad \wedge\ (\forall i\ .\ T_i \in outSlots(v) \implies color(v, T_i) = type) \\
&\quad \wedge\ inSlots(v) \subseteq \{type\} \\
&\quad \wedge\ (type \in inSlots(v) \implies color(v, type) = type)
\end{aligned}
$$

$$
\begin{aligned}
&label(v) = \psi \implies \\
&\qquad outSlots(v) \subseteq \{T_0,\ \ldots,\ T_{chan_{MAX}}\} \\
&\quad \wedge\ (\forall i\ .\ T_i \in outSlots(v) \implies color(v, T_i) = type) \\
&\quad \wedge\ inSlots(v) \subseteq \{type\} \\
&\quad \wedge\ (type \in inSlots(v) \implies color(v, type) = type)
\end{aligned}
$$

$$
\begin{aligned}
&label(v) = \texttt{LOAD} \implies \\
&\qquad outSlots(v) = \{type,\ mem,\ \ ptr\} \\
&\quad \wedge\ color(v, type) = type \\
&\quad \wedge\ color(v, mem) = memory \\
&\quad \wedge\ color(v, ptr) = data \\
&\quad \wedge\ inSlots(v) \subseteq \{mem', val\} \\
&\quad \wedge\ (mem' \in inSlots(v) \implies color(v, mem') = memory) \\
&\quad \wedge\ (val \in inSlots(v) \implies color(v, val) = data)
\end{aligned}
$$

$$
\begin{aligned}
&label(v) = \texttt{STORE} \implies \\
&\qquad outSlots(v) = \{type,\ mem,\ \ ptr,\ val\} \\
&\quad \wedge\ color(v, type) = type\ \wedge\ color(v, mem) = memory \\
&\quad \wedge\ color(v, ptr) = data\ \wedge\ color(v, val) = data \\
&\quad \wedge\ inSlots(v) \subseteq \{mem'\} \\
&\quad \wedge\ (mem' \in inSlots(v) \implies color(v, mem') = memory)
\end{aligned}
$$

$$
\begin{aligned}
&label(v) = \texttt{FENCE} \implies \\
&\qquad outSlots(v) \subseteq \{mem_0,\ \ldots,\ mem_{fence_{MAX}}\} \\
&\quad \wedge\ (\forall i\ .\ mem_i \in outSlots(v) \implies color(v, mem_i) = memory) \\
&\quad \wedge\ inSlots(v) \subseteq \{mem'\} \\
&\quad \wedge\ (mem' \in inSlots(v) \implies color(v, mem') = memory)
\end{aligned}
$$

APPENDIX B – TYPING RULES

In a multigraph G_i, let v range over nodes in N_i and let T stand for any type produced by the grammar shown in Figure 25. For the sake of brevity, $\diamond$ stands for any kind of binary operation in K_N with operand slots $\{lhs, rhs\}$ and result slot $result$. Similarly, let $\preceq$ range over binary predicates, that is, integer comparison operators. Additionally, $\tau(v) = T$ indicates that the type dependency edge of a node v is wired to a type-level operator producing type T; this does not lead to ambiguities because primitive nodes in the IR which are not type operators themselves have at most one type dependency edge. Finally, $v.x : T$ denotes that slot $x \in (inSlots(v) \cup outSlots(v))$ has been assigned type T by one of the inference rules below.

$$\frac{label(v) = \texttt{JOIN} \quad \tau(v) = J = \psi(\chi(T_{10}, ..., T_{1n_1}),\ ...,\ \chi(T_{m0}, ..., T_{mn_m}))}{v.def : J \ \wedge\ \forall 1 \leq i \leq m\ .\ \forall 0 \leq j \leq n_m\ .\ (v.param_{ij} : T_{ij})}$$

$$\frac{label(v) = \texttt{INST} \quad \tau(v) = J = \psi(\chi(T_{10}, ..., T_{1n_1}),\ ...,\ \chi(T_{m0}, ..., T_{mn_m}))}{v.def : J \ \wedge\ \forall 1 \leq i \leq m\ .\ (v.cont_i : \kappa(T_{i0}, ..., T_{in_i}))}$$

$$\frac{label(v) = \texttt{JUMP} \quad \tau(v) = \kappa(T_0, ..., T_n)}{v.target : \kappa(T_0, ..., T_n) \ \wedge\ \forall 0 \leq j \leq n\ .\ (v.arg_j : T_j)} \qquad \frac{label(v) = \texttt{COND} \quad \tau(v) = \mathbb{B}^w}{v.sel : \mathbb{B}^w}$$

$$\frac{label(v) = \texttt{CONST} \quad \tau(v) = T}{v.value : T} \qquad \frac{}{color(v,x) = memory \iff v.x : M}$$

$$\frac{label(v) = \diamond \quad \tau(v) = T}{v.lhs : T \ \wedge\ v.rhs : T \ \wedge\ v.result : T} \qquad \frac{label(v) = \preceq \quad \tau(v) = T}{v.lhs : T \ \wedge\ v.rhs : T \ \wedge\ v.result : \mathbb{B}^1}$$

$$\frac{label(v) = \texttt{BITCAST} \quad v.input : T_a \quad \tau(v) = T_b}{v.output : T_b}$$

$$\frac{label(v) = \texttt{MUX} \quad v.sel : \mathbb{B}^w \quad \tau(v) = T}{v.output : T \ \wedge\ \forall 0 \leq n < 2^w\ .\ (v.input_n : T)}$$

$$\frac{label(v) = \texttt{TUPLE} \quad \tau(v) = \Pi(T_1, ..., T_n)}{v.tuple : \Pi(T_1, ..., T_n) \ \wedge\ \forall 1 \leq i \leq n\ .\ v.field_i : T_i}$$

$$\frac{label(v) = \texttt{PROJ}_i \quad \tau(v) = \Pi(T_1, ..., T_n) \quad 1 \leq i \leq n}{v.tuple : \Pi(T_1, ..., T_n) \ \wedge\ v.field : T_i}$$

$$\frac{label(v) \in \{\texttt{LOAD}, \texttt{STORE}\} \quad \tau(v) = T}{v.mem : M \ \wedge\ v.ptr : \rho \ \wedge\ v.mem' : M \ \wedge\ v.val : T} \qquad \frac{label(v) = \texttt{FENCE} \quad \tau(v) = T}{mem' : M \ \wedge\ \forall i\ .\ (v.mem_i : M)}$$

Appendix C – Preprint Article
A Survey on Intermediate Representations

Gabriel B. Sant'Anna[1]

[1] Federal University of Santa Catarina (UFSC) – Florianópolis – Brazil

baiocchi.gabriel@gmail.com

Abstract. *Developing and maintaining an optimizing compiler is far from trivial. Modern compiler infrastructures attempt to reduce that development cost by sharing the same "middle end" across various combinations of language-specific front ends and target-specific back ends. Shared Intermediate Representations (IRs) play a major role in these architectures, but their design is still not fully understood, even by practitioners in the field. This work attempts to increase that understanding by surveying various compilers and their respective IRs. Unlike related works, we cover some functional and algebraic IRs, study the use of high-level programming-languages as an intermediate translation step, and include more recently developed compiler IRs in our research.*

1. Introduction

Computer languages are means of communication between humans and machines. The communication is generally considered more effective when programs are written in a high-level programming language [Click 1995], and programs more efficient when an optimizing compiler performs the translation to machine code [Johnson 2004]. Unfortunately, developing and maintaining an optimizing compiler is, and has always been, far from trivial [Steel 1961a, Oliva et al. 1997, Chow 2013, Susungi and Tadonki 2021].

Compilers must be able to reason about non-trivial programs, to combine various analyses in order to determine whether or not to apply certain transformations, and to ensure that optimizations preserve program semantics [Reissmann et al. 2020]. Among other reasons, the complexity of the translation process leads to a multi-stage pipeline design (refer to Figure 1), in which the same program can take multiple different forms, also known as Intermediate Representations (IRs). At a higher level, it is possible to group translation stages [Aho et al. 2006] into three categories: the *front end* is responsible for enforcing the invariants of the source language; the *back end* handles the idiosyncrasies of the target encoding; and the oxymoronic "*middle end*" performs IR analyses and transformations which are often independent of both source language and target machine. IRs play a major role in modern compiler infrastructures, warranting the rise of the middle end as one of the most important parts of an optimizing compiler [Chow 2013].

Until relatively recently, most IRs were unspecified proprietary formats which could not be used between different compilers, and in some cases not even across different versions of the same compiler [Lattner 2002]. In contrast, the LLVM project [Lattner and Adve 2004] has shown that a well-known IR, with open-source tools built around it, facilitates the development of new compiler technology, to the benefit of the computing industry as well as of programming language research. Hence, pursuing a

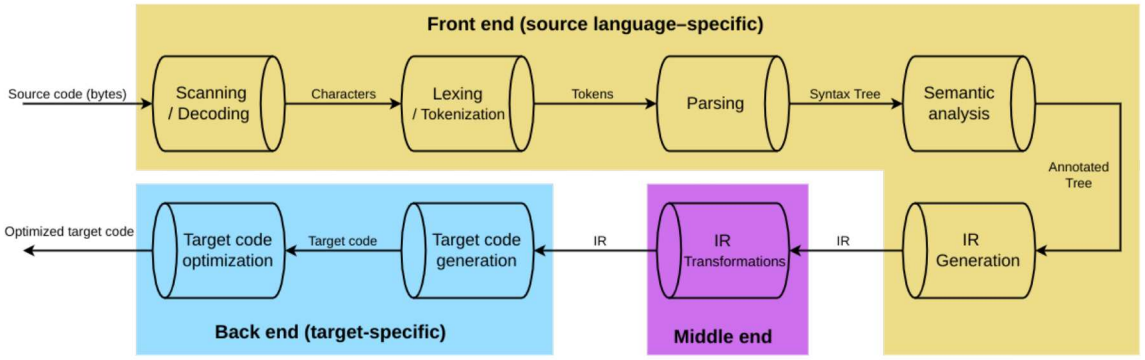


Figure 1: Logical structure of a typical optimizing compiler.

standardized universal IR, one which would enable compiler interoperability and target-independent binary distribution, is a very promising, even if idealistic, goal [Chow 2013].

Despite the importance of compiler IRs, it appears that their design is still an "art", not completely understood even by practitioners in the field of programming languages and compilers [Lattner et al. 2021]. In this context, we present a survey on IRs and their accompanying abstract machines. Unlike related works [Stanier and Watson 2013, Belwal and TSB 2015, Susungi and Tadonki 2021], the survey covers functional and algebraic IRs, while describing techniques used in both modern and older compilers to increase the efficiency *and* the portability of program encodings. This research can also be seen as a means to an end: understanding portable compiler IRs, identifying design principles and eliciting requirements to guide the design of a modern "universal IR", a concept known in the past as UNCOL (Universal Computer Oriented Language) [Strong et al. 1958, Steel 1961b, Macrakis 1993b].

2. Related Works

An overview on the topic of compiler IRs can be found in [Chow 2013]. Directly related works begin with [Stanier and Watson 2013], with a first survey on IRs used by imperative programming language compilers. [Belwal and TSB 2015] later published their research on IRs for heterogeneous multi-core computing. More recently, [Susungi and Tadonki 2021] reviewed IRs designed to represent "explicitly parallel" programs. These previous surveys narrowed their scope to IRs used and designed for certain purposes; this one is no exception, with special attention given to portable IRs, that is, to abstract program representations which may (as claimed by its authors or as observed in practice) be able to accommodate multiple high-level languages and target architectures.

Each of the three related works classifies IRs differently. [Stanier and Watson 2013] use three aspects: structure, which can be linear or graph-based; dependency information, including data flow and control flow; and program content, whether programs can be fully or only partially encoded in the IR. The classification of [Belwal and TSB 2015] is simpler: IRs can be "syntactic" (represented as trees), graph-based or a mix of the two previous options. [Susungi and Tadonki 2021], on the other hand, consider four IR categories: Intermediate Languages (ILs), graphs, Static Single Assignment (SSA) form and polyhedral representations. Meanwhile, this survey groups IRs solely by their structure: Sections 3, 4 and 5 respectively describe linear, tree-structured and graph-based IRs.

Despite being separately classified, these kinds of IRs are frequently used together in modern compilers, albeit at different stages of the translation pipeline. [Stanier and Watson 2013] identify a tendency towards: (a) parse trees and Abstract Syntax Trees (ASTs) in the front end; (b) graph-based IRs to represent whole functions; and (c) linear IRs – usually put in SSA form for the purpose of optimization – used for local "low-level" transformations.

3. Linear Representations

Linear IRs represent programs as sequences of instructions for an abstract machine. Imagining an IL in this format is particularly straightforward when the abstract machine resembles a simple stored-program computer. Thus, most ILs, including the first UNCOL ever proposed [Conway 1958], follow this pattern [Franz 1994]. It is also the preferred IR type of Virtual Machines (VMs), since a baseline implementation can operate by interpreting an individual instruction, moving to the next one in the sequence and repeating this process in a loop, essentially emulating a sequential computer. This is why linear IRs are said to be "low-level" [Lattner and Adve 2004]. When interpreted by a VM, linear ILs are called bytecode formats, since each instruction can often be encoded in a single byte [Nystrom 2021]. They are also known as P-code (portable or pseudo code), in praise of the VM-based implementation of the Pascal programming language [Wirth 1993].

Despite their simplicity, linear IRs are not without disadvantages. [Haddon and Waite 1978] describe the challenge of choosing a minimal set of portable instructions. In short, recovering the original intent of high-level code after it has been lowered to a linear IR is not trivial, especially after multiple rounds of program transformations [Logozzo and Fahndrich 2008]. Furthermore, this "primitive mismatch" problem (which also extends to data types, number of available registers and properties of memory operations, among other aspects) is unavoidable: unless the IR is to expand uncontrollably, there will always be programming language operations and machine instructions without a one-to-one mapping in the IR [Ganapathi et al. 1982].

Similar to instruction formats of real computers, linear compiler ILs have a simple, regular structure. Also like the instructions of real machines, the biggest difference between them lies in what the instructions themselves do: what sort of data they operate on, how they access memory and what effects are caused by their execution [Stanier and Watson 2013]. Despite the great variety of possible instructions, linear compiler IRs can be categorized as either *stack-based* or *register-based*.

3.1. Stack-based Bytecode

Another way to conceive linear IRs is by attempting to serialize syntax trees into a more densely packed format. For example, a simple compiler for arithmetic expressions – such as the one described by [Aho et al. 2006] – can traverse ASTs in post-order and emit bytecode at each visited node; the resulting program would be ready to be executed by a stack-based VM – like the one described by [Nystrom 2021]. Figure 2 illustrates how this process can be applied to a short program, which is parsed into an AST and serialized as a sequence of stack-based bytecode instructions. [Stanier and Watson 2013] would designate such an IL as an extension of postfix Polish (or, reverse Polish) notation. These IRs are very compact, but become difficult to optimize [Stanier and Watson 2013].

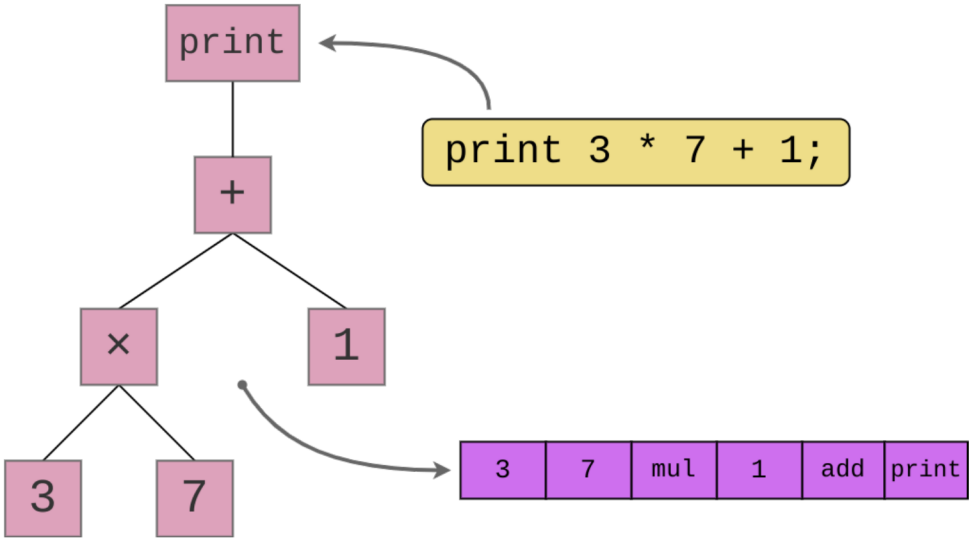


Figure 2: Translating arithmetic to stack-based bytecode.

[Koopman 1989] presented a stack computer taxonomy which can be adapted to stack-based compiler IRs. One dimension of the design space is the number of distinct stacks available to the machine: single-stack machines are simpler and VM implementations can be faster, but having multiple stacks may help prevent certain classes of security vulnerabilities by separating data and control flow manipulation [Haas et al. 2017]. The other significant aspect is the number of operands an instruction may have: while a 0-operand "pure stack" architecture is almost always manipulating the topmost stack elements, 1-operand machines – also known as single-address or "stack/accumulator" architectures – designate an additional machine register or memory address as either input or output to each instruction [Koopman 1989].

The following is a non-exhaustive list of stack-based compiler IRs:

- **The SECD machine.** [Landin 1964] designed an abstract machine to evaluate λ-calculus terms. Its 0-operand IR controlled four distinct memory devices, two of which were stacks. Later, the SECD machine was used as a central component of the LispKit compiler [Henderson 1980] and there was an effort to implement it directly on hardware [Graham 1989].

- **Janus.** Janus was an IL which shared most of UNCOL's goals. [Coleman et al. 1974] describe the system as a family of abstract machines with a common basic structure (a 1-operand stack-based architecture) and a universal IL. They achieved portability through the abstraction of high-level operations as macros, which are called in the IL and expanded in a target-specific manner when lowering to machine language [Newey et al. 1972]. This technique was rediscovered some years later, in the "Universal Compiling System" proposed by [Gyllstrom et al. 1979], although they do not describe their IR in much detail. Janus was used in the implementation of a Pascal compiler and in the design of an Algol 68 compiler [Haddon and Waite 1978].

- **U-code.** A variant of P-code designed by [Perkins and Sites 1979] to enable compiler optimizations across a wide variety of machines and languages. The 1-

operand stack-based IL was used to compile both Pascal and Fortran programs and had some degree of backwards compatibility with P-code–based compilers [Perkins and Sites 1979]. U-code had dedicated metadata instructions for the purposes of debugging, guiding optimizations and stating program requirements (e.g., defining memory alignment restrictions).

- **The Amsterdam Compiler Kit IL.** The Amsterdam Compiler Kit was a collection of compiler modules created by [Tanenbaum et al. 1983], based on the UNCOL idea. Their main IR, Encoding Machine (EM) language, is linear, but the tool kit also included graph-based global optimizations. As its name suggests, the stack-based IL was used mainly as an encoding mechanism, where the back end maintained a simulated stack and generated code with the aid of a target-specific "driving table", which described how to translate certain stack configurations into machine language. Lacking an entry in the driving table, the compiler would inject IL instructions into the program with the purpose of coercing the stack into a known configuration. [Tanenbaum et al. 1983] describe this method as being analogous to a chess-playing artificial intelligence program, since it searches through many possible sequences of "moves". There were also meta-linguistic declarations, not unlike those previously suggested by [Steel 1961a], which helped guide the "chess-playing" procedure. The Amsterdam Compiler Kit, originally a commercial product, was used for many years as the native compiler toolchain in Tanembaum's Minix operating system [Given 2005], with support for C, Plain, Pascal, Algol 68 and more than a dozen different hardware architectures. According to [Tanenbaum et al. 1983], it showed that "the old UNCOL idea is a sound way to produce compilers". The toolkit has since been released under an open-source license, but has not received much attention.

- **The Categorical Abstract Machine.** The Categorical Abstract Machine was created as a mathematically well-founded method of functional programming language implementation [Cousineau et al. 1987]. The core stack-based architecture was formally described by [Cousineau et al. 1987], who claimed the small changes proposed over the SECD machine enabled much simpler correctness proofs.

- **Java Virtual Machine (JVM) bytecode.** Many qualities associated with the Java programming language, such as machine-independence and memory safety, stem from its VM [Lindholm et al. 2015]. The JVM is a portable stack-based and object-oriented architecture specifically designed for Java. Nonetheless, compilers for other programming languages (Kotlin, Scala and Clojure, to cite a few) frequently target JVM bytecode in order to profit from the mature platform. Since most JVM implementations employ Just-In-Time (JIT) compilers, which translate bytecode to machine language at run-time, JVM bytecode can be considered an IL. Still, compilers often convert the stack-based IR into a format better suited for transformations before lowering it to machine code [Lattner and Adve 2004].

- **Microsoft's Common Intermediate Language (CIL).** An Ecma International standard defines the Common Language Infrastructure (CLI) [ECMA-335 2012], which includes a common type system, a stack-based virtual execution system and an accompanying IL, the CIL. The machine-independent infrastructure was

built with maximum portability in mind; in order to integrate multiple programming languages in the same platform, it was "designed to be large enough that it is properly expressive yet small enough that all languages can reasonably accommodate it" [ECMA-335 2012]. One way the CLI attempts to deliver practical language interoperability is by providing abstract operations (such as a "virtual calling convention") and extensible IL metadata that let compiler front ends communicate optimization-relevant information all the way to the execution engine, which can then tailor code generation decisions to the user's machine. In general, such on-the-fly code generation methods aim to counteract the efficiency losses which tend to arise when using machine-independent IRs [Franz 1994].

- **WebAssembly.** WebAssembly is a VM-bytecode pair for the Web platform. Despite currently being in a minimum-viable-product state, it was collaboratively designed by engineers from major browser vendors [Haas et al. 2017] and therefore boasts wide availability on consumer devices. The stack-based architecture is explicitly noted by [Haas et al. 2017] as merely a mechanism to easily achieve a formalization of the abstract machine and provide a compact program representation. Hence, many design decisions – notably, prohibiting irreducible control flow – aim to enable quick conversion into IRs better suited for program transformations and compiler optimizations, such as a register-based SSA [Haas et al. 2017].

3.2. Register-based Three-Address Codes

Register-based compiler IRs are often synonymous with Three-Address Codes (3ACs), although alternative architectures exist (and are mentioned below). In a typical 3AC, each instruction performs a single operation, having at most two inputs and one output [Aho et al. 2006]. An example is shown in Figure 3, which displays the small program from Figure 2 encoded in a 3AC IL.

```
%t1 = i32 1
%t3 = i32 3
%t7 = i32 7
%tm = mul %t3, %t7
%ta = add %tm, %t1
%_ = call @print(%ta)
```

Figure 3: A simple 3AC program, using LLVM-like syntax.

In most register-based IRs, the compiler assigns a named register to all intermediate results. This has the advantage of facilitating the reuse of previously computed values: one simply needs to refer to the name associated with it, as opposed to having to perform a series of memory and stack manipulation operations [Koopman 1989]. A disadvantage lies in the fact that the compiler *needs* to allocate a register for every single intermediate value in the program: if the abstract machine has a limited number of registers, this becomes a non-trivial problem [Aho et al. 2006]. According to [Aho et al. 2006], 3ACs allow programs to be transformed and rearranged much more easily than stack-based bytecodes, since the latter requires the insertion of instructions in a very specific order to achieve the effects described by the original high-level program.

There are many ways for a compiler to implement 3ACs. The most straightforward representation consists of quadruples: sequences of four-field records containing the instruction code, (up to) two argument registers and a named result [Aho et al. 2006]. Triples are a close alternative, where the result's "name" is implicitly defined by the position of the instruction in the containing sequence. The triple scheme requires less space and lets the compiler use numbers (which are faster indexes and easier to generate) instead of arbitrary names [Stanier and Watson 2013]. Since triples associate instruction indexes with their produced results, reordering operations requires the compiler to update all uses of the result of a relocated triple. Indirect triples offer a solution to that problem by uncoupling program order from result handles, such that a reference to a triple is kept stable even when the program is being transformed [Aho et al. 2006]. Figure 4 shows a single program (the same one from Figure 3) represented as quadruples, triples and indirect triples. In the last representation, the program was slightly reordered.

```
PROGRAM:                    PROGRAM:                  0: <const, i32, 1>
<const, i32, 1,   %t1>      [0: const, i32,  1 ]      1: <const, i32, 3>
<const, i32, 3,   %t3>      [1: const, i32,  3 ]      2: <const, i32, 7>
<const, i32, 7,   %t7>      [2: const, i32,  7 ]      3: <mul,   (1), (2)>
<mul,   %t3, %t7, %tm>      [3: mul,   (1), (2)]      4: <add,   (3), (0)>
<add,   %tm, %t1, %ta>      [4: add,   (3), (0)]      5: <print, (4), _>
<print, %ta, _,   _>        [5: print, (4),  _ ]      PROG': [1;2;3;0;4;5]
```

Figure 4: Quadruples (left), triples (center) and indirect triples (right).

Register-based compiler IRs include, but are not limited to:

- **Conway's UNCOL.** In the first proposed UNCOL design, [Conway 1958] partially describes a register-based IR. He believed that minimalistic abstract machines, with fewer registers and instructions, would provide greater opportunity for code optimization. Thus, his IR follows a single-address architecture, where operations specify only their output locations, and operands are always read from predefined "argument-storage" registers.
- **Steel's UNCOL.** As one of the authors of the UNCOL report, [Steel 1961a] provides further detail on how a universal IR could be envisioned. Like Conway, Steel points towards a single-address machine. He indicates the need to describe, for every item of data expressed in the IL, properties such as type, numerical range, precision and storage allocation, arguing that (at least some of) these could be achieved with a mechanism to define the lexical and syntactical structure of language elements. Steel also suggests that some instructions should aid compiler optimizations through meta-linguistic declarations such as "the next 20 statements form a loop". Finally, a striking feature of Steel's UNCOL was that commands could have different behavior based on the data types being operated on. This type-based instruction overloading results in a smaller IR: a single addition command, for example, would work with both integers and floating-point numbers. Like Conway's UNCOL, this IL was never implemented.
- **IBM's PL.8 compiler.** The PL.8 project was a compiler which could accept multiple source languages (Pascal and a variant of PL/I) and produce op-

timized code for several different machines, notably by the means of a common IR "closely matching the computational semantics of the target machines" [Auslander and Hopkins 1982]. Its IR was a 3AC with an unlimited number of virtual registers. This is a notable feature of the IR's design because it may have prompted the invention of SSA form, especially since [Rosen et al. 1988] were familiar with IBM's PL.8 infrastructure.

- **The Stanford University Intermediate Format (SUIF).** SUIF was a compiler framework heavily influenced by the concept of a universal IR. It is divided into a "kernel", which consists of IR generation and manipulation procedures, and a "toolkit" with front end and back end modules using the IR as a common interface [Wilson et al. 1994]. SUIF's IR is described as a "mixed-level" program representation, being for the most part a 3AC, but also including tree-like constructs for loops, conditionals and array accesses [Wilson et al. 1994]. The infrastructure was released for free as an attempt to solve the compiler construction problem in research. It succeeded only to some degree: there were multiple research projects based on SUIF, with front ends implemented for C, Fortran, Java and C++ [SUIF 2005]. However, its place in research has since been occupied by the LLVM project [Farvardin and Reppy 2020].

- **GCC's Register Transfer Language (RTL).** GCC is another well known compiler collection and infrastructure, notable for being used in the Linux kernel [FSF 2022]. According to [Lattner 2021], GCC's open-source nature created a turning point in the compiler industry around the 1990s (back in a time when most compilers where commercial products, all mutually incompatible), enabling collaboration from academia and industry, reducing language community fragmentation and to a great extent achieving the $N + M$ scaling of multiple front ends and back ends. GCC officially supports C, C++, Objective-C, Fortran, Ada, Go and D, for countless back ends [FSF 2022]. Still, modern compilers tend to gravitate towards the LLVM infrastructure, mainly due to its modular architecture and better-specified (even if slightly unstable) IL. GCC has multiple IRs [Merrill 2003], the oldest one being RTL, a 3AC with an infinite number of virtual registers. Unlike LLVM IR, RTL's SSA form is optional [FSF 2022].

- **The LLVM Project (and derivatives).** LLVM is a modern compiler framework originally designed by [Lattner 2002] to tackle the challenges originated by recently developed programming practices, namely the support for dynamic extensions to long-running programs and multi-language software development. A solution is envisioned as the combination of capabilities previously considered to be mutually exclusive: while classic compilers provide ahead-of-time code optimization and a language-agnostic runtime model, VM-based approaches such as the JVM or the CLI can implement profile-guided JIT transformations by preserving program information until run-time [Lattner and Adve 2004]. Providing all of these features is LLVM's goal, which, according to [Lattner and Adve 2004], is achieved through its innovative IR design and the open-source compiler infrastructure built around it. LLVM's IR can be described as a register-based 3AC (although the implementation places the linear IR inside Control Flow Graph (CFG) nodes) in SSA form, extended with "high-level" type information. The

LLVM infrastructure has been successfully applied in academia and in the industry, with countless programming languages implemented on it due to its maturity and ability to emit optimized code for many target architectures [LLVM 2022]. Although earlier versions of LLVM were quite minimalistic, with only 31 opcodes [Lattner and Adve 2004], it has been in a constant state of change and expansion: the current version (LLVM 13) of the IL reference documentation [LLVM 2022] can fill over 300 printed pages. This instability and constantly increasing complexity has led some groups to create similar IRs, isolated on purpose from the main LLVM infrastructure. Examples include Khronos' SPIR-V [Khronos 2020], which can be described as a fork of LLVM IR specialized for parallel computation and graphics processing, and Cranelift, which has been considered as a faster (in terms of compile time) alternative to LLVM for debug builds [Larabel 2020].

- **The Join Calculus Abstract Machine.** [Calvert 2015] proposes using the join calculus as the foundation for a parallel abstract machine and LLVM-like linear compiler IR. He argues that the resulting design could be used as a universal IR, especially since, among other members of the process calculi family, the join calculus seems to be particularly suitable for the efficient implementation of programming languages [Le Fessant and Maranget 1998, Fournet and Gonthier 2000].
- **Heterogeneous System Architecture (HSA) IL.** HSA IL is described as a low-level virtual instruction set for the architectures supported by the HSA Foundation [HSA 2018]. According to [Chow 2013], although the register-based IL's extensive specification could be a step forward in what concerns universal IRs, the complexity of the HSA programming model makes it unlikely that compilers will use it as anything other than a target for HSA-compatible devices. For instance, a limited number of registers [HSA 2018] implies that most, if not all, program transformations may need to invoke a register allocator.

4. Tree-structured Representations

Trees are some of the most common compiler data structures. Although mainly utilized to represent the syntactic structure of high-level languages, tree-based IRs are also used for macro expansion, type checking and some program transformations [Aho et al. 2006]. A simple high-level program represented as an AST has been shown in Figure 2.

4.1. Generic Tree Encodings

Trees are generally considered language-dependent compiler IRs, since each node in the tree normally corresponds to a specific programming language construct [Aho et al. 2006]. Still, a few works propose language-independent tree-structured ILs:

- **Architecture Neutral Distribution Format (ANDF).** ANDF has been described as an "architecture- and language-neutral distribution format resembling a compiler intermediate language", aiming to succeed where UNCOL proposals had previously failed [Macrakis 1993b]. ANDF was essentially a compact binary format used to encode program trees in a standard manner. Defined by the Open Software Foundation, the system took a very pragmatic position on the problem of portability: "ANDF is not a tool for making non-portable software into portable

software [an unsolvable problem], but a tool for distributing portable software" [Macrakis 1993b]. The major mechanism used to achieve practical language and machine independence was a macro system reminiscent of Lisp's syntax macros. In ANDF, static conditional branches, identifier references, constant literals, type layout and even other macros could be identified and stored separately from the main program. This allowed the system to retain portability and preserve the potential for both language- and machine-specific optimizations, since such delayed definitions could be "linked in" and expanded only when appropriate, up until the moment the program was loaded [Macrakis 1993c]. ANDF was only ever implemented in UNIX systems, and for the C programming language [Macrakis 1993a]. The implementation, documented as TenDRA, was later released as an open-source compiler framework [TenDRA 2022]. Stavros Macrakis (personal communication, 2021) suggests ANDF was not widely adopted mainly because of: (a) lack of interest from UNIX vendors; and (b) the compiler construction problem becoming less relevant as the computer market was dominated by specific (software and hardware) systems.

- **Semantic Dictionary Encoding.** Proposed by [Franz 1994], "semantic dictionaries" can encode general ASTs in a compact tabular form. It was used as Oberon's module interchange format, also known as "slim binaries". It achieved portability by distributing high-level source programs, while the "fat binary" alternative was to package multiple machine-specific objects in the same executable file [Franz and Kistler 1997].
- **The ROSE compiler.** ROSE is an open compiler infrastructure for the development of program analysis tools and source-to-source transformations. Given its goals, a generic tree-based IR is a natural fit. The framework has been used to develop tools and research projects for C, C++, Fortran, Java, Python and PHP, among others [LLNL 2022].

4.2. Programming Languages as Compiler ILs

Some high-level programming languages have been used as compiler ILs. Instead of generating machine code (or VM bytecode) directly, compilers can emit a program encoded in another high-level language and then call one of its existing back ends (or use an existing VM implementation). Typically, this is done to avoid re-implementing architecture-specific transformations and code generation [Oliva et al. 1997], but also provides a way to reuse developer tools (debuggers, for instance) in a newly-created programming language. This approach is hereby grouped with other tree-structured IRs, since trees are the standard computer-oriented representation of high-level languages.

Although not the only programming language to have been repurposed as a compiler target, C is the most notorious. It has been used in countless implementations of other high-level languages, including C++, Ada, Cedar, Eiffel, Modula-3, Pascal, Standard ML [Macrakis 1993b], Scheme and Haskell [Oliva et al. 1997]. Alan Kay even considers C to have become a "*de facto* UNCOL" [Feldman 2004], since the standard is portable and compilers are available on most platforms. Another programming language being targeted by various compilers is JavaScript. According to [Haas et al. 2017], this is mainly due to "historical accident", since JavaScript was (until the very recent advent of We-

bAssembly) the only computer language with standardized support in the Web platform. Figure 5 illustrates the use of C and JavaScript as compiler targets.

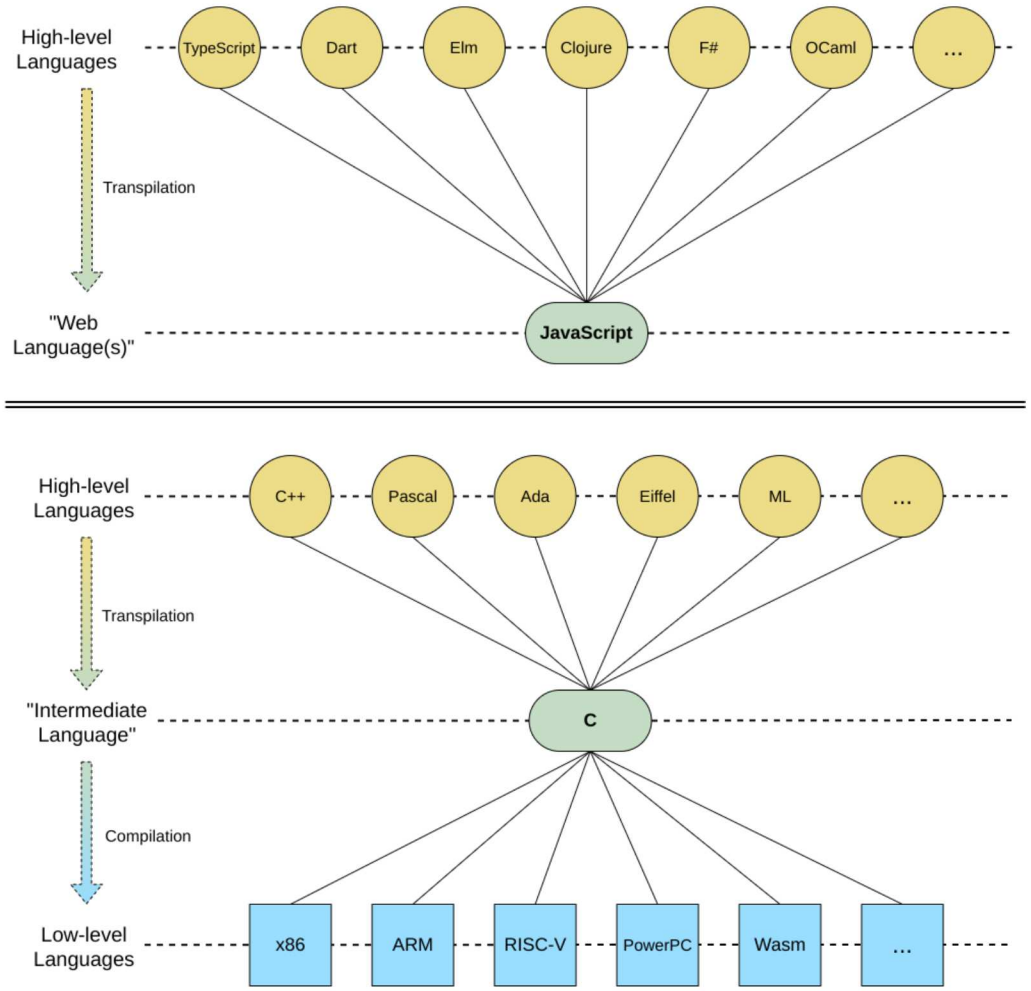


Figure 5: JavaScript's monopoly and C's ubiquity.

High-level programming languages are designed to be written and understood by humans. This leads to a set of design decisions which make them undesirable as compiler IRs [Chow 2013]: text encodings are far less compact – and also much harder to parse and transform – than binary formats [Haas et al. 2017]; some high-level languages simply lack the mechanisms to perform lower-level operations needed by other languages (JavaScript, for example, does not have integers); and the semantics of a "target" programming language are more often than not incompatible with those of other "source" programming languages. This last point has been noted as the main reason not to use C as a compiler IL.

[Oliva et al. 1997] argue that C is not suitable to represent functional languages, mainly because C compilers do not ensure Tail Call Optimization (TCO), lack information needed for precise garbage collection and cannot encode – neither discover by themselves – some of the program properties guaranteed by functional languages (such as data immutability or pointer aliasing restrictions) which would otherwise enable more optimizations during code generation. Furthermore, [Macrakis 1993b] considers C to

be under-specified: aspects which would be relevant to an optimizing compiler, such as supported integer ranges or the behavior of out-of-bounds array accesses, are either implementation-defined or not defined at all. Of course, Turing-completeness means that none of this prevents a transpiler from generating portable C programs which emulate the abstract machine exposed by other languages. This, however, would introduce unacceptable performance degradation when the models of computation do not match [Haddon and Waite 1978]. According to [Macrakis 1993b], emulating another high-level language requires C generators to commit to a concrete implementation of every single construct in the source language; whichever strategy is chosen will almost certainly not be appropriate for all target architectures and cannot be adapted by the C compiler being used as a back end, since it knows nothing about the higher-level source language.

In this category of compiler ILs, a final candidate is Lisp: according to [Steele 1976], "a carefully chosen small dialect of LISP would be a good UNCOL, that is, a good intermediate compilation language". [Steele 1978] implemented this idea in his Scheme compiler, Rabbit, a seminal work presenting the first use of Continuation-Passing Style (CPS) in a compiler. CPS is a representation in which control points are explicitly named, manipulated and invoked [Steele 1976]. Steele's proposal relies on Scheme's guaranteed TCO and its strong proximity to the lexical-scoped applicative-order λ-calculus. These qualities, combined with Lisp macros and an imperative assignment operator, allow for a straightforward implementation of most programming language constructs; thus, Scheme was its own IL [Steele 1976].

The Rabbit compiler operated through source-to-source transformations, lowering high-level functional Scheme code into an imperative-style CPS IL, which happened to be a subset of Scheme itself. Further research on the topic of compiling with continuations adopted the same source-to-source explanation of CPS [Flanagan et al. 1993, Kennedy 2007, Maurer et al. 2017]. Although Steele's work was primarily research-oriented, [Adams et al. 1986] used its design as a base for their optimizing Scheme compiler, Orbit. This later work proved that CPS transformations could be an effective tool in optimizing compilers, since Orbit was used to produce code competitive with C, Pascal, Modula-3 and Lisp systems of the time [Adams et al. 1986].

At last, it should be noted that Lisp's syntax appears to be more suitable for a compiler IL than that of most other programming languages, since it trivially maps to ASTs[1]. Regardless, [Macrakis 1993b] points out that Scheme as a compiler IL shares some flaws with other high-level language options, notably due to Lisp systems' reliance on garbage collection and lack of concrete data definition facilities.

4.3. Algebraic ILs

Another class of representations whose use in compilers has been proposed consists of "algebraic" ILs. These are formal languages from the lineage of term rewriting systems (e.g., mathematical logic), which have become foundational for programming language and type theory [Pierce 2002]. While there have been operational machine-like approaches (such as the SECD machine, mentioned in section 3) to defining these models of computation, the usual method is more syntax-directed, with ASTs used to express pro-

[1] In fact, the infamous S-expression notation, which ended up becoming Lisp's standard programming syntax, was originally a computer-oriented (i.e., not human-oriented) representation [McCarthy 1978].

grams, as well as their execution [Harper 2016]. Manipulating code through an algebraic representation has the advantage of enabling the application of well-known mathematical proof methods to programming languages [Pierce 2002]. The λ-calculus is one such algebraic language; its syntax is displayed in Figure 6. It could also be considered a family of languages (or calculi), due to its many variants and extensions [Pierce 2002].

$M,\ N ::=$	terms
x	variable
$\lambda x.M$	abstraction
$M\ N$	application

$P,\ Q ::=$	processes
$\mathbf{0}$	null process
$\nu x.P$	new local channel x
$x\langle y\rangle.P$	receive message y on x
$\overline{x}\langle y\rangle.P$	send message y on x
$\tau.P$	silent action
$P \mid Q$	parallel composition
$P + Q$	choice operator
$!P$	replication

Figure 6: The λ-calculus (left) and the π-calculus (right).

[Milner 1993] has claimed that, although λ-functions are "an essential ingredient" of sequential program semantics, handling concurrency requires a different framework. Albeit notoriously hard for humans to reason about, concurrent programs are an important part of modern computing [Peschanski 2011]. This had already been predicted in the 1950s, as mentioned by [Strong et al. 1958]: "everyone looks with dread at the possible computers of the next decade, which will be simultaneously executing multiple asynchronous stored programs".

The need for mechanisms to formally reason about concurrent processes has fostered the family of process algebras (or, process calculi), notably including Milner's π-calculus (also shown in Figure 6) and its many variations [Palamidessi 1997]. According to [Fournet and Gonthier 2000], calling these formal systems calculi implies in the possibility of using them for equational reasoning (i.e., to calculate). This requires a notion of "equality", an operational equivalence relation between programs, and many such relations have been proposed for process calculi [Fournet and Gonthier 2000]. These are particularly interesting to compiler writers, who must ensure that transformations and optimizations preserve program equivalences.

Algebraic ILs are usually associated with compilers of functional programming languages, possibly due to their strong relation with the λ-calculus. Examples include:

- **π-threads.** [Peschanski 2011] proposes a variation of the π-calculus as a parallel abstract machine model and compiler IL. He provides some safety proof outlines in the resulting model, but no benchmarks are shown.
- **Haskell Core.** Haskell's main IL, Core, is based on System Fω, a typed variant of the λ-calculus [Downen et al. 2016]. The compiler maintains this functional IL in Administrative Normal Form (ANF), an alternative to CPS which admits many of the same transformations [Maurer et al. 2017].
- **Sequent Core.** Under the Curry-Howard correspondence, the classic λ-calculus is equivalent to Gentzen's natural deduction logic [Wadler 2015]. However, natural

deduction was defined along with another reduction system, the sequent calculus [Gentzen 1964]. [Downen et al. 2016] describe Sequent Core as an experimental Haskell IL based on sequent calculi. The result comes closer to CPS form than to Core's traditional ANF, enabling more optimizations [Downen et al. 2016].

5. Graph-based Representations

Directed graphs (or, just graphs) are one of the most pervasive structures in Discrete Mathematics and Computer Science. A graph can be defined as a pair (N, E), where N is a set of nodes (or, vertices) and $E \subseteq (N \times N)$ a set of edges (alternatively, arcs), consisting in ordered pairs of nodes. A path in a graph is any sequence of nodes $[v_0, ..., v_n]$ such that every consecutive pair (v_i, v_{i+1}) is an edge in E. When all paths beginning at node v_0 and eventually reaching node v_n pass through an intermediate node v_d, it is said that v_d dominates v_n with respect to v_0. Furthermore, any two nodes (a, b) are said to be strongly connected if there is a path (however long) from a to b; when such a path can only be formed by adding reverse edges to the graph (that is, adding edges (v, u) for any subset of (u, v) already present in E), a and b are said to be weakly connected; otherwise they are disconnected. By extension, an entire graph can be called strongly (or weakly) connected when all of its nodes are strongly (resp. weakly) connected to all other nodes in the graph. A Strongly Connected Component (SSC) in a graph $G = (N, E)$ is any subgraph $G' = (N' \subseteq N,\ E' \subseteq E)$ which is itself a strongly connected graph. Finally, if a directed graph contains at least one node strongly connected to itself through a non-trivial (i.e., non-empty) path, the graph is said to be cyclic, as this implies the existence of a path starting at that node and eventually circling back to it; otherwise, it can be called a Directed Acyclic Graph (DAG). [Offner 2013] details the applications of graphs in optimizing compilers.

Graphs are very general mathematical structures: nodes in a graph may be associated to objects of any kind, and edges can be used to represent arbitrary relations between such objects. In the context of compilers, graphs have been used to represent various relations between parts of programs, such as "flows to" (control flows from a subroutine to the next), "depends on" (an instruction's result depends on its arguments), "calls" (e.g., procedure `foo` may call procedure `bar`), "happens before" (one memory operation happens before the other), etc [Stanier and Watson 2013, Shivers 1988, Adve and Boehm 2010]. Each distinct set of relations being expressed gives rise to a different graph-based IR.

5.1. Control Flow Graphs

Control Flow Graphs (CFGs) are classic IRs used in optimizing compilers to partition procedures into nodes and express control flow between them [Aho et al. 2006]. Nodes in a CFG are "basic blocks", which, according to [Aho et al. 2006], consist in linear instruction sequences with a single entry and no exit points until the very last instruction. Then, graph edges are used to link basic blocks to every other node the program could branch off to by the end of that instruction sequence. This leads to the classification of compiler optimizations as either local (to a basic block) or global (but still within the same procedure) [Aho et al. 2006]. Figures 7 and 8 show a C program and an equivalent CFG.

```c
int f(int n)
{
    int p = 1;
    int c = 0;

    for (int i = 0; i < n; ++i) {
        int t = c;
        c = p + c;
        p = t;
    }

    return c;
}
```

Figure 7: C program with an input-controlled loop.

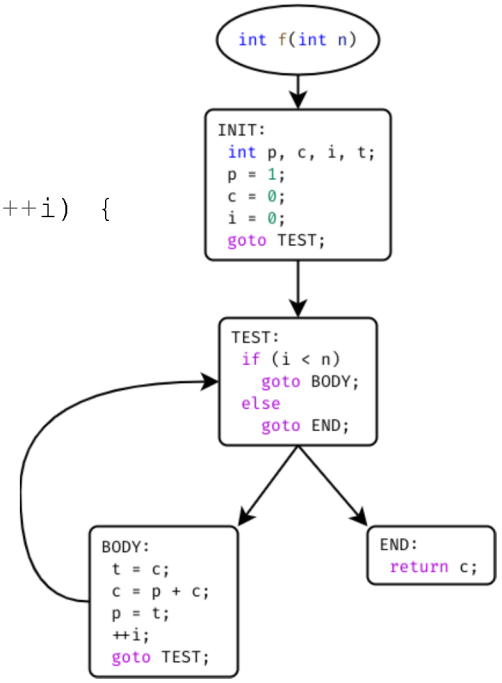


Figure 8: A CFG for Figure 7.

As exemplified in Figure 8, cyclic CFGs can represent the branching behavior of programs with loops. However, not all loops are created equal. Computer languages with `goto` or equivalent constructs may be used to write programs displaying irreducible control flow [Stanier and Watson 2012]. Irreducible programs can be identified as such when (a) their CFG contains a cycle and (b) there is no node in the cycle which dominates all others (in the cycle, with respect to the procedure's entry point). More informally, irreducible control flow is caused by "jumping inside a loop" [Unger and Mueller 2002]. However, even in programming languages without `goto`, this property may "naturally" arise as a result of program transformations and compiler optimizations [Stanier and Watson 2012].

Irreducibility is often considered a problem because many compiler algorithms either do not work or have different asymptotic complexity in its presence [Stanier and Watson 2012]. For this reason, some IRs (e.g., WebAssembly) make irreducible control flow impossible to represent. Therefore, compilers which lower optimized programs into an IR (or programming language, in JavaScript's case) which cannot represent irreducible control flow have to implement non-trivial workarounds [Unger and Mueller 2002]. Although standard techniques to do so exist in the compiler literature [Aho et al. 2006], they either cause an exponential size blowup at the IR level [Carter et al. 2003] or require complex analyses to modify the program, adding guard predicates and state variables [Zakai 2011, Bahmann et al. 2015]. Lastly, although irreducible flow can be detected and eliminated within individual procedures, doing so across function calls is not possible in general [Shivers 1988].

5.2. Data Flow Graphs

CFGs are chiefly concerned with control flow, the "shape" of a program. They do not directly represent data flow information, especially not across basic blocks.

Hence, Data Flow Graphs (DFGs) are the traditional approach to representing relations between definitions and uses of program values – also known as "def-use" chains [Stanier and Watson 2013]. Figure 9 shows a DFG derived from Figure 7.

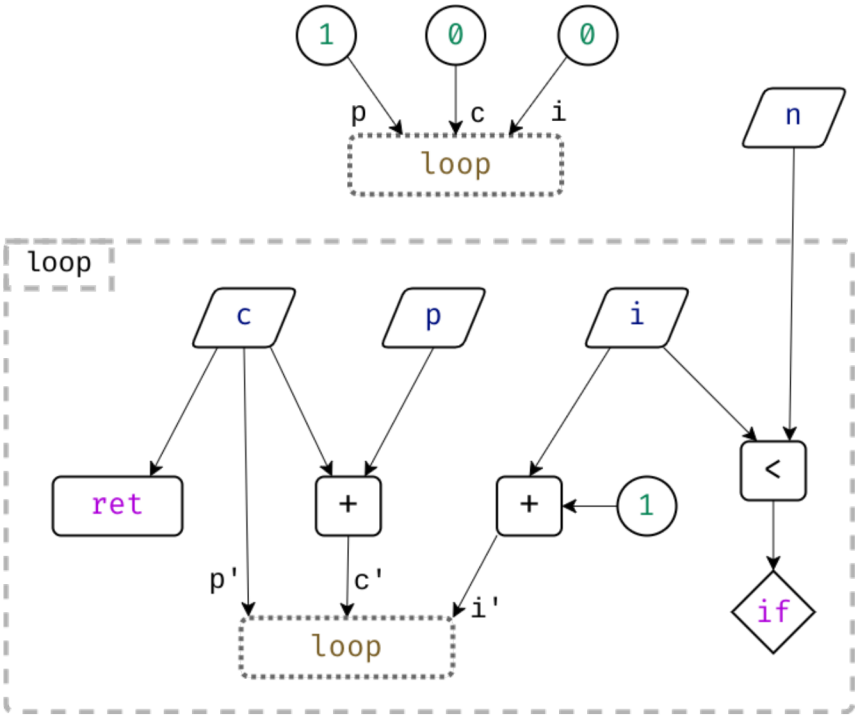


Figure 9: Acyclic DFG for the looping program of Figure 7.

Since data dependencies are generally expected (and sometimes required) to be acyclic, so are DFGs. In most computer languages, this raises the problem of having to express many different values associated with the same variable name. For instance, imperative assignments such as `c = p + c` (as seen in Figure 7, line 8) would apparently require a cyclic DFG, since data stored in a variable may depend on its own previous definition. Fortunately, there are techniques (e.g., SSA form) which can be used to "break" apparent data flow cycles.

Most programs cannot be represented by a DFG alone. This becomes clear when comparing Figure 9 with the original program: the illustrated DFG cannot represent the relation between the conditional form (indicated by the node labeled as "`if`") and its mutually-exclusive branches (one which returns from the function and the other which continues its internal computation), nor does it clearly indicate that nodes `p, c, i` correspond to inputs of the `loop` node (whose body is outlined within Figure 9 for the sake of clarity). Therefore, the DFG does not substitute the CFG, but rather enriches it with information that is useful for many analyses and transformations [Aho et al. 2006].

5.3. Hybrid Dependence Graphs

Unfortunately, having both a CFG and a DFG forces compilers to keep the separate structures consistent with each other across program transformations, which can be costly [Stanier and Watson 2013]. Given the importance of def-use – and its converse, "use-def" – information for compiler optimizations, this has motivated the creation of alternative IRs

and techniques to combine control and data flow information in a single IR. One such approach is what [Lawrence 2007] refers to as an "SSA CFG", that is, a CFG which respects SSA form in the contents of its basic blocks. According to [Stanier and Watson 2013], maintaining IRs in SSA form has become standard practice in modern compilers.

Static Single Assignment (SSA) is a technique invented by [Rosen et al. 1988] to enable the efficient representation of use-def and def-use relations. It should be noted that SSA, by itself, is not an IR, but a program property. Therefore, it can be achieved in any representation, including linear 3ACs, graphs-based IRs and even high-level programming languages [Stanier and Watson 2013]. For example, Figure 10 displays the same program from Figures 7 and 8, now encoded in LLVM IR (left) and as a Scheme program (right). These programs are equivalent and respect SSA form, as explained below.

```
define i32 @f(i32 %n) {
INIT:
  br label %TEST

TEST:
  %p1 = phi i32 [1, %INIT], [%c1, %BODY]
  %c1 = phi i32 [0, %INIT], [%c2, %BODY]
  %i1 = phi i32 [0, %INIT], [%i2, %BODY]
  %b0 = icmp slt i32 %i1, %n
  br i1 %b0, label %BODY, label %END

BODY:
  %c2 = add i32 %p1, %c1
  %i2 = add i32 %i1, 1
  br label %TEST

END:
  ret i32 %c1
}
```

```
(define (f n ret)

  (define (loop p c i)
    (define (BODY)
      (let ((p2 c)
            (c2 (+ p c))
            (i2 (+ i 1)))
        (loop p2 c2 i2)))

    (define (END)
      (ret c))

    (begin ; TEST
      (if (< i n)
          (BODY)
          (END))))

  (begin ; INIT
    (loop 1 0 0)))
```

Figure 10: Imperative ϕs *versus* functional blocks.

As the acronym suggests, a program in SSA form assigns all of its "variables" precisely once, creating a local mapping from names to program values. Furthermore, no variable may be used without being defined first (in a CFG, this can be rephrased as "every basic block which uses a value is dominated by the basic block which defines it"). These properties make it trivial to find the definition of a value when, later in the program, it is being used in some operation [Lattner 2002].

In order to handle different mutable assignments to the same variable, SSA uses a special ϕ operator. ϕ nodes (when represented in a graph-based IR) or ϕ functions (in a register-based IR) use control flow information to resolve the correct value of variables within their basic blocks [Stanier and Watson 2013]. Intuitively, ϕ nodes extend basic blocks with formal arguments, much like parameters in a procedure. In fact, some compilers (e.g., MLIR) achieve SSA form through such "extended basic blocks", instead of the equivalent ϕ nodes [Lattner et al. 2021]. Figure 10 (left) displays LLVM syntax for ϕ functions: each operand to a `phi` pairs a value and a block label, and its result depends on which block preceded the current one during program execution.

Programs expressed in a "purely functional" manner – with no mutable variable assignments – are in SSA form by definition. This similarity has been observed by [Kelsey 1995], who proved that SSA corresponds to a restricted version of CPS by describing conversion methods between both representations. [Appel 1998] states that SSA *is* functional programming, rediscovered by developers of imperative language compilers. To illustrate this point, Figure 10 displays two equivalent SSA-form programs; while the LLVM versions requires ϕ instructions, the Scheme one uses pure functions as extended basic blocks – notice that it displays the same number of "block arguments" (three) as there are `phi` instructions in the LLVM IR program. This Scheme program was manually produced as an intermediate step when building the acyclic DFG shown in Figure 9. Basing the DFG on the LLVM version instead would have resulted in an equivalent, yet cyclic, graph, with a ϕ node at every cycle, clearly marking a "temporal dependency", that is, the influence of control information in the data flow graph.

Maintaining SSA form within a CFG is not the only way to combine control and data flow information in a single IR. Within this group of hybrid graph-based IRs, it is also possible to identify a chronological succession of so-called "dependence graph" designs, each one aiming to improve upon its predecessors:

- **The Program Dependence Graph (PDG)** – Originally developed by [Ferrante et al. 1987], the PDG is a directed multigraph[2] combining a CFG and a data *dependency* graph. The latter differs from a data *flow* graph because edges follow the use-def direction, instead of def-use. The PDG is a useful representation for many compiler optimizations [Lawrence 2007, Stanier and Watson 2013]. However, maintaining its invariants across program transformations has a high cost in comparison to the alternatives, and it complicates other analyses due to aliasing and side-effect problems [Johnson 2004]. [Stanier and Watson 2013] note that they were not aware of any compiler primarily using the PDG, despite it being the most cited IR in their survey. A more detailed critique of the PDG is provided in the work of [Johnson 2004].
- **The Program Dependence Web (PDW)** – [Ottenstein et al. 1990] designed the PDW by augmenting the PDG with "gated SSA form", which (unlike classic SSA) is a representation that admits direct program interpretation. Unfortunately, the PDW puts an even higher burden on compilers, since its construction requires five passes over the PDG and it cannot directly represent irreducible control flow [Johnson 2004, Stanier and Watson 2013, Reissmann et al. 2020].
- **The Value Dependence Graph (VDG)** – According to [Weise et al. 1994], "dependence graph" IRs can be viewed as different approaches to removing the limitations of a CFG. They propose the VDG as a next step in this direction, aiming to represent programs only with dependency information and a few special operators. The IR was deemed unsuitable for non-experimental use because it may fail to preserve program termination behavior [Johnson 2004, Stanier and Watson 2013, Reissmann et al. 2020].
- **The Value State Dependence Graph (VSDG)** – In order to address the VDG's

[2]Multigraphs extend graphs with the ability to identify multiple different edges with the same nodes at their extremities. This is particularly useful for dependency graph IRs, which may attribute different meanings to two edges leaving a certain node, even if both connect to a common node on the other side.

termination problem, [Johnson 2004] augmented it with state dependency edges. The resulting IR is similar to the PDG in the sense that it combines control and data flow information in a single graph, but without the ordering restrictions of a CFG. Like the VDG, the VSDG is implicitly in SSA form, which eliminates certain problems found in the PDG [Johnson 2004]. According to [Lawrence 2007], while SSA CFGs correspond to functional programs under an *eager* evaluation strategy, VSDGs correspond to functional programs under a *lazy* evaluation strategy. While this aids some optimizations, it makes it hard to efficiently preserve the semantics of effectful computations, which are not functional in nature [Reissmann et al. 2020]. Figure 11 displays a VSDG.

- **The Regionalized Value State Dependence Graph (RVSDG)** – [Lawrence 2007] argues that the VSDG's problem when representing effectful computations can be solved by augmenting it with the concept of graph "regions". The RVSDG is a multigraph with a hierarchical structure; some of its nodes introduce regions, and regions can contain entire subgraphs. This also leads to an IR which can capture more abstract notions directly in its structure. For example, θ nodes in an RVSDG explicitly indicate that the subgraph within their region constitutes the body of a loop [Reissmann et al. 2020]. Meanwhile, detecting the same loop in a CFG would require a graph traversal. Another key aspect of the RVSDG is that it provides interprocedural constructs in the IR itself: it can "represent a program as a unified data structure where a def-use dependency of one function on another is modeled the same way as the def-use dependency of scalar quantities" [Reissmann et al. 2020]. Despite having been initially described by [Lawrence 2007], it seems that the RVSDG was not implemented and empirically evaluated until the work of [Reissmann et al. 2020] on the jlm compiler. Figure 12 illustrates an example RVSDG, which is equivalent to the program previously shown in Figures 7–11.

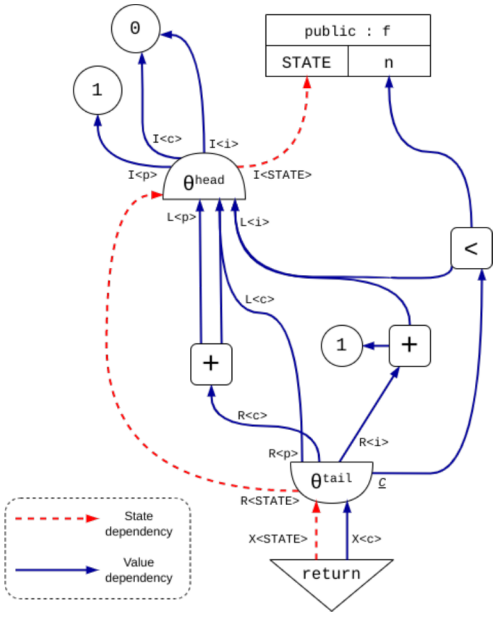


Figure 11: VSDG equivalent to Figures 7–10.

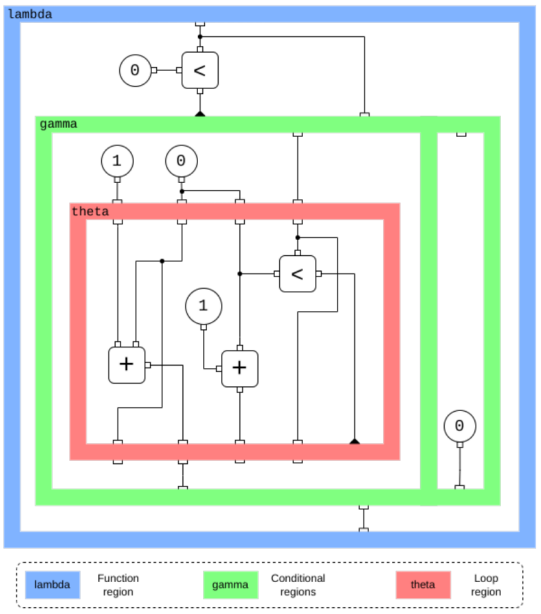


Figure 12: An RVSDG, equivalent to Figure 11.

Separate from the *-DG line of research (from PDG to RVSDG) is Click's IR [Click and Paleczny 1995]. According to [Johnson 2004], it can be seen as a variation of the PDG which combines a control subgraph and a data subgraph, connecting them at `PHI` nodes (which use control information to resolve data flow) or `IF` nodes (which use program values to define control flow). Unlike the PDG, it is always in SSA form. Click's IR is also known as "the sea of nodes" [Click 1995], since it maps each operation to a node without putting it in any specific "place" in the graph; nodes are "floating around", ordered only by their dependency relations (the same could also be said of the DFG and the VSDG). This is in contrast to the CFG, which enforces a total order of instructions within basic blocks.

In order to model effectful computations, Click's IR treats memory as a special kind of value which is produced by load and store operations [Click and Paleczny 1995]. It also defines its own notion of a "region", which allows it to represent the same control flow information that could be found in a CFG [Click and Paleczny 1995]. Unlike in the RVSDG, however, regions in Click's IR correspond to simple nodes (albeit with additional edges linking them to the aforementioned `PHI` nodes); this avoids the additional structural restrictions of an RVSDG, which requires all of its regions to nest hierarchically and hence cannot represent irreducible control without performing restructuring transformations [Bahmann et al. 2015].

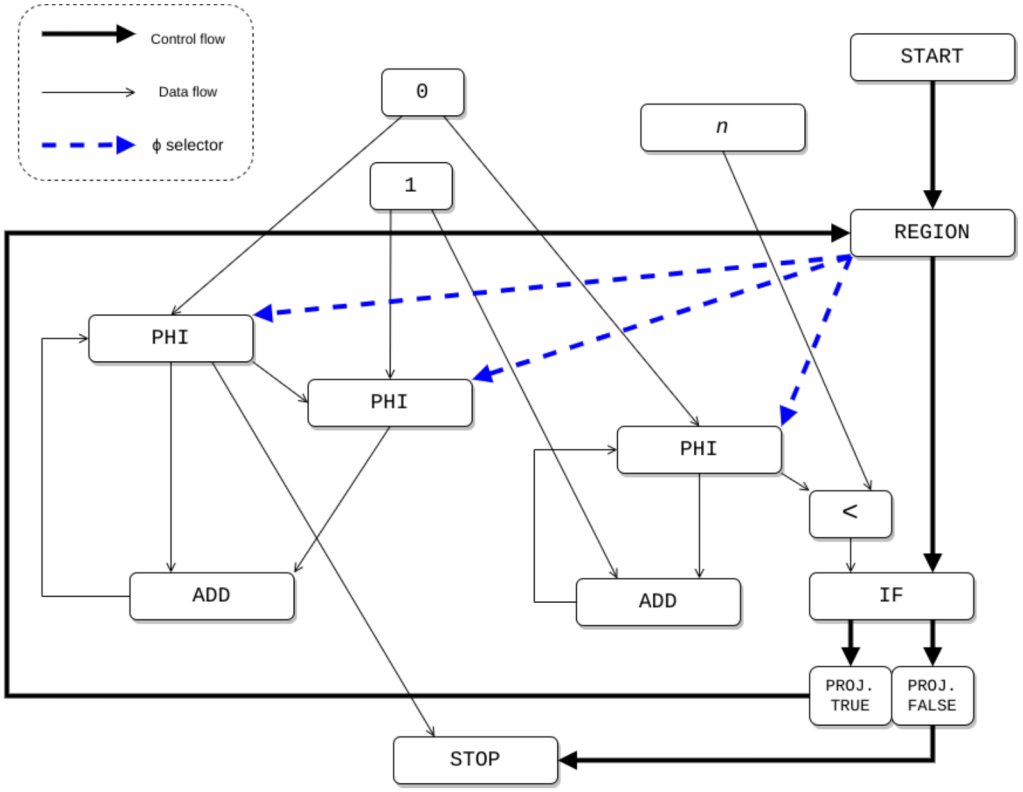


Figure 13: Click's sea of nodes.

Click's sea of nodes is an SSA graph which can be built and optimized while a program is being parsed [Click 1995]. This makes it particularly desirable in JIT compilers such as the Java HotSpot Server Compiler [Paleczny et al. 2001] and libFirm [Braun et al. 2011], which use adaptations of Click's IR. [Braun et al. 2011] explain how

the sea of nodes design leads to "implicit" and "inherent" transformations; in other words, the mere act of representing a program in the IR can lead to optimizations such as Copy Propagation, Dead Code Elimination and Common Subexpression Elimination. Figure 13 shows the previously discussed C program represented in Click's IR.

An idea similar to the sea of nodes can be found in Thorin, an IR which aims to effectively represent and optimize both imperative and functional programs, exploiting the correspondence between SSA and CPS to model control flow in an uniform manner [Leißa et al. 2015]. Unlike other functional IRs, Thorin encodes CPS in a graph instead of the more traditional tree-based approach. According to [Leißa et al. 2015], this avoids the compile-time and implementation overhead of resolving name conflicts during program transformations. In fact, the sea-of-nodes-like design completely eliminates the need to explicitly encode names and scope nesting; the former are substituted by use-def edges in the SSA graph, and the latter can be found by analyzing the sets of nodes linked by these edges [Leißa et al. 2015].

The influence of Click's IR is also noted in MLIR [Lattner et al. 2021], a framework designed for the construction of domain-specific compilers. MLIR can be viewed as an extremely extensible SSA-form IR, designed to represent and optimize programs at multiple levels of abstraction. It has a generic textual encoding, but can be grouped with other graph-based IRs due to its ability to assign custom semantics to "regions" of basic blocks – the most common one being the semantics of an SSA CFG [Lattner et al. 2021].

6. Conclusion

This work is motivated by indications of an upcoming boom in the diversity of hardware architectures [Hennessy and Patterson 2019] and programming languages [Lattner 2021]. Its main contribution is a survey on portable compiler IRs, which are increasingly important aspects of modern compilers. Unlike related works, this one covers functional and algebraic IRs, while describing techniques used in both modern and old compilers to increase the efficiency *and* the portability of program encodings.

Through an extensive literature review, it was possible to identify methodological gaps related to the design and evaluation of compiler IRs: we lack IR design principles and evaluation methods, especially in the context of a multi-language, multi-target "universal IR". Perhaps related to these issues is the fact that most compiler IRs lack a formalization of their structure (including exchange formats) and interpretation (the model of computation exposed by the IR).

We estimate that modern compiler infrastructures, including but not limited to LLVM and MLIR, could be improved by applying more principled techniques and guidelines to the design of IR features and transformations. One such theoretical framework can be found in the work of [Felleisen 1991], which presents a theory of expressiveness that can be used to find redundancies in programming language constructs. Applying this to compiler IRs would be an innovative experiment by itself.

In summary, future work possibilities include developing better evaluation methods for compiler IRs, from both empiric and theoretical points of view.

References

[Adams et al. 1986] Adams, N., Kranz, D., Kelsey, R., Rees, J., Hudak, P., and Philbin, J. (1986). ORBIT: An Optimizing Compiler for Scheme. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '86, page 219–233, New York, NY, USA. Association for Computing Machinery.

[Adve and Boehm 2010] Adve, S. V. and Boehm, H.-J. (2010). Memory Models: A Case for Rethinking Parallel Languages and Hardware. *Communications of the ACM*, 53(8):90–101.

[Aho et al. 2006] Aho, A., Lam, M., Sethi, R., and Ullman, J. (2006). *Compilers: Principles, Techniques, and Tools*. Addison Wesley, Boston, MA, USA, 2nd edition.

[Appel 1998] Appel, A. W. (1998). SSA is Functional Programming. *SIGPLAN Notices*, 33(4):17–20.

[Auslander and Hopkins 1982] Auslander, M. and Hopkins, M. (1982). An Overview of the PL.8 Compiler. *SIGPLAN Notices*, 17(6):22–31.

[Bahmann et al. 2015] Bahmann, H., Reissmann, N., Jahre, M., and Meyer, J. C. (2015). Perfect Reconstructability of Control Flow from Demand Dependence Graphs. *ACM Transactions on Architecture and Code Optimization*, 11(4).

[Belwal and TSB 2015] Belwal, M. and TSB, S. (2015). Intermediate representation for heterogeneous multi-core: A survey. In *International Conference on VLSI Systems, Architecture, Technology and Applications (VLSI-SATA)*, pages 1–6, Bengaluru, India. IEEE.

[Braun et al. 2011] Braun, M., Buchwald, S., and Zwinkau, A. (2011). FIRM–A Graph-Based Intermediate Representation. Technical report, Chamonix, France. Workshop on Intermediate Representations.

[Calvert 2015] Calvert, P. R. (2015). Architecture-neutral parallelism via the Join Calculus. Technical Report UCAM-CL-TR-871, University of Cambridge, Computer Laboratory, Cambridge, United Kingdom.

[Carter et al. 2003] Carter, L., Ferrante, J., and Thomborson, C. (2003). Folklore confirmed: Reducible flow graphs are exponentially larger. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, page 106–114, New York, NY, USA. Association for Computing Machinery.

[Chow 2013] Chow, F. (2013). Intermediate Representation: The Increasing Significance of Intermediate Representations in Compilers. *ACM Queue*, 11(10):30–37.

[Click and Paleczny 1995] Click, C. and Paleczny, M. (1995). A Simple Graph-Based Intermediate Representation. In *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations*, IR '95, page 35–49, New York, NY, USA. Association for Computing Machinery.

[Click 1995] Click, C. N. J. (1995). *Combining Analysis, Combining Optimizations*. Rice University, Houston, TX, USA.

[Coleman et al. 1974] Coleman, S. S., Poole, P. C., and Waite, W. M. (1974). The Mobile Programming System, Janus. *Software: Practice and Experience*, 4(1):5–23.

[Conway 1958] Conway, M. E. (1958). Proposal for an uncol. *Communications of the ACM*, 1(10):5–8.

[Cousineau et al. 1987] Cousineau, G., Curien, P.-L., and Mauny, M. (1987). The Categorical Abstract Machine. *Science of Computer Programming*, 8(2):173–202.

[Downen et al. 2016] Downen, P., Maurer, L., Ariola, Z. M., and Peyton Jones, S. (2016). Sequent Calculus as a Compiler Intermediate Language. *SIGPLAN Notices*, 51(9):74–88.

[ECMA-335 2012] ECMA-335 (2012). ECMA-335 - Common Language Infrastructure (CLI). Standard, Ecma International.

[Farvardin and Reppy 2020] Farvardin, K. and Reppy, J. (2020). A New Backend for Standard ML of New Jersey. In *IFL 2020: Proceedings of the 32nd Symposium on Implementation and Application of Functional Languages*, IFL 2020, page 55–66, New York, NY, USA. Association for Computing Machinery.

[Feldman 2004] Feldman, S. (2004). A Conversation with Alan Kay: Big Talk with the Creator of Smalltalk - and Much More. *ACM Queue*, 2(9):20–30.

[Felleisen 1991] Felleisen, M. (1991). On the expressive power of programming languages. *Science of Computer Programming*, 17(1):35–75.

[Ferrante et al. 1987] Ferrante, J., Ottenstein, K. J., and Warren, J. D. (1987). The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349.

[Flanagan et al. 1993] Flanagan, C., Sabry, A., Duba, B. F., and Felleisen, M. (1993). 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. Association for Computing Machinery.

[Fournet and Gonthier 2000] Fournet, C. and Gonthier, G. (2000). The Join Calculus: a Language for Distributed Mobile Programming. In Barthe, G., Dybjer, P., Pinto, L., and Saraiva, J., editors, *Applied Semantics. International Summer School, APPSEM 2000,*, pages 268–332, Berlin, Germany. Springer.

[Franz 1994] Franz, M. (1994). *Code-generation On-the-fly: A Key to Portable Software*. Swiss Federal Institute of Technology in Zurich, Zurich.

[Franz and Kistler 1997] Franz, M. and Kistler, T. (1997). Slim Binaries. *Communications of the ACM*, 40(12):87–94.

[FSF 2022] FSF (2022). GNU Compiler Collection (GCC) Internals.

[Ganapathi et al. 1982] Ganapathi, M., Fischer, C. N., and Hennessy, J. L. (1982). Retargetable Compiler Code Generation. *ACM Computing Surveys*, 14(4):573–592.

[Gentzen 1964] Gentzen, G. (1964). Investigations into Logical Deduction. *American Philosophical Quarterly*, 1(4):288–306.

[Given 2005] Given, D. (2005). The Amsterdam Compiler Kit.

[Graham 1989] Graham, B. (1989). SECD: Design Issues. Technical report, University of Calgary, Calgary, Canada.

[Gyllstrom et al. 1979] Gyllstrom, H. C., Knippel, R. C., Ragland, L. C., and Spackman, K. E. (1979). The Universal Compiling System. *SIGPLAN Notices*, 14(12):64–70.

[Haas et al. 2017] Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., Wagner, L., Zakai, A., and Bastien, J. (2017). Bringing the Web up to Speed with WebAssembly. *SIGPLAN Notices*, 52(6):185–200.

[Haddon and Waite 1978] Haddon, B. K. and Waite, W. M. (1978). Experience with the Universal Intermediate Language Janus. *Software: Practice and Experience*, 8(5):601–616.

[Harper 2016] Harper, R. (2016). *Practical Foundations for Programming Languages*. Cambridge University Press, 3rd edition.

[Henderson 1980] Henderson, P. (1980). *Functional Programming Application and Implementation*. Prentice Hall, London, United Kingdom.

[Hennessy and Patterson 2019] Hennessy, J. L. and Patterson, D. A. (2019). A New Golden Age for Computer Architecture. *Communications of the ACM*, 62(2):48–60.

[HSA 2018] HSA (2018). HSA Programmer's Reference Manual: HSAIL Virtual ISA and Programming Model, Compiler Writer, and Object Format (BRIG). Standard, HSA Foundation.

[Johnson 2004] Johnson, N. E. (2004). Code size optimization for embedded processors. Technical Report UCAM-CL-TR-607, University of Cambridge, Computer Laboratory, Cambridge, United Kingdom.

[Kelsey 1995] Kelsey, R. A. (1995). A Correspondence between Continuation Passing Style and Static Single Assignment Form. In *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations*, IR '95, page 13–22, New York, NY, USA. Association for Computing Machinery.

[Kennedy 2007] Kennedy, A. (2007). Compiling with Continuations, Continued. *SIGPLAN Notices*, 42(9):177–190.

[Khronos 2020] Khronos (2020). SPIR Overview.

[Koopman 1989] Koopman, P. J. J. (1989). *Stack Computers: the new wave*. Ellis Horwood, Pittsburgh, PA, USA.

[Landin 1964] Landin, P. J. (1964). The Mechanical Evaluation of Expressions. *The Computer Journal*, 6(4):308–320.

[Larabel 2020] Larabel, M. (2020). Rust Lands Experimental Cranelift-Based Code Generator - Much Faster Debug Build Times.

[Lattner 2002] Lattner, C. (2002). *LLVM: An Infrastructure for Multi-Stage Optimization*. Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, USA.

[Lattner 2021] Lattner, C. (2021). The Golden Age of Compiler Design in an Era of HW/SW Co-design. ASPLOS 2021 Keynote.

[Lattner and Adve 2004] Lattner, C. and Adve, V. (2004). LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004*

International Symposium on Code Generation and Optimization (CGO'04), Palo Alto, CA, USA.

[Lattner et al. 2021] Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R., Shpeisman, T., Vasilache, N., and Zinenko, O. (2021). MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '21, page 2–14, Korea. IEEE Press.

[Lawrence 2007] Lawrence, A. C. (2007). Optimizing compilation with the Value State Dependence Graph. Technical Report UCAM-CL-TR-705, University of Cambridge, Computer Laboratory, Cambridge, United Kingdom.

[Le Fessant and Maranget 1998] Le Fessant, F. and Maranget, L. (1998). Compiling Join-Patterns. *Electronic Notes in Theoretical Computer Science*, 16(3):205–224. HLCL '98, 3rd International Workshop on High-Level Concurrent Languages (Satellite Workshop of CONCUR '98).

[Leißa et al. 2015] Leißa, R., Köster, M., and Hack, S. (2015). A Graph-Based Higher-Order Intermediate Representation. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, page 202–212, San Francisco, CA, USA. IEEE Computer Society.

[Lindholm et al. 2015] Lindholm, T., Yellin, F., Bracha, G., and Buckley, A. (2015). The Java Virtual Machine Specification. Technical report, Oracle, Redwood City, CA, USA.

[LLNL 2022] LLNL (2022). ROSE Compiler – Program Analysis and Transformation.

[LLVM 2022] LLVM (2022). The LLVM Compiler Infrastructure – Language Reference Manual.

[Logozzo and Fahndrich 2008] Logozzo, F. and Fahndrich, M. (2008). On the Relative Completeness of Bytecode Analysis versus Source Code Analysis. In *Proceedings of the International Conference on Compiler Construction*. Springer Verlag.

[Macrakis 1993a] Macrakis, S. (1993a). Delivering applications to multiple platforms using ANDF. Technical report.

[Macrakis 1993b] Macrakis, S. (1993b). From UNCOL to ANDF: Progress in standard intermediate languages. Technical report, Open Software Foundation, USA.

[Macrakis 1993c] Macrakis, S. (1993c). The structure of ANDF: Principles and examples. Technical report.

[Maurer et al. 2017] Maurer, L., Downen, P., Ariola, Z. M., and Peyton Jones, S. (2017). Compiling without Continuations. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 482–494, New York, NY, USA. Association for Computing Machinery.

[McCarthy 1978] McCarthy, J. (1978). *History of LISP*, page 173–185. Association for Computing Machinery, New York, NY, USA.

[Merrill 2003] Merrill, J. (2003). GENERIC and GIMPLE: A New Tree Representation for Entire Functions. Technical report, Ottawa, Canada. GCC Developers Summit.

[Milner 1993] Milner, R. (1993). Elements of Interaction: Turing Award Lecture. *Communications of the ACM*, 36(1):78–89.

[Newey et al. 1972] Newey, M. C., Poole, P. C., and Waite, W. M. (1972). Abstract Machine Modelling to Produce Portable Software - A Review and Evaluation. *Software: Practice and Experience*, 2(2):107–136.

[Nystrom 2021] Nystrom, R. (2021). *Crafting Interpreters*. Genever Benning, USA.

[Offner 2013] Offner, C. D. (2013). Notes on Graph Algorithms Used in Optimizing Compilers.

[Oliva et al. 1997] Oliva, D., Nordin, T., and Peyton Jones, S. (1997). C--: A Portable Assembly Language. In *Proceedings of the 1997 Workshop on Implementing Functional Languages*, volume 1467 of *LNCS*, page 1–19. Springer Verlag.

[Ottenstein et al. 1990] Ottenstein, K. J., Ballance, R. A., and MacCabe, A. B. (1990). The Program Dependence Web: A Representation Supporting Control-, Data-, and Demand-Driven Interpretation of Imperative Languages. *SIGPLAN Notices*, 25(6):257–271.

[Palamidessi 1997] Palamidessi, C. (1997). Comparing the Expressive Power of the Synchronous and the Asynchronous π-Calculus. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, page 256–265, New York, NY, USA. Association for Computing Machinery.

[Paleczny et al. 2001] Paleczny, M., Vick, C., and Click, C. (2001). The Java Hotspot^{TM} Server Compiler. In *Proceedings of the 2001 Symposium on Java Virtual Machine Research and Technology Symposium*, volume 1 of *JVM'01*, Monterey, CA, USA. USENIX Association.

[Perkins and Sites 1979] Perkins, D. R. and Sites, R. L. (1979). Machine-Independent PASCAL Code Optimization. *SIGPLAN Notices*, 14(8):201–207.

[Peschanski 2011] Peschanski, F. (2011). Parallel Computing with the Pi-Calculus. In *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming*, DAMP '11, page 45–54, New York, NY, USA. Association for Computing Machinery.

[Pierce 2002] Pierce, B. C. (2002). *Types and Programming Languages*. The MIT Press, USA, 1st edition.

[Reissmann et al. 2020] Reissmann, N., Meyer, J. C., Bahmann, H., and Själander, M. (2020). RVSDG: An Intermediate Representation for Optimizing Compilers. *ACM Transactions on Embedded Computing Systems*, 19(6).

[Rosen et al. 1988] Rosen, B. K., Wegman, M. N., and Zadeck, F. K. (1988). Global Value Numbers and Redundant Computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, page 12–27, New York, NY, USA. Association for Computing Machinery.

[Shivers 1988] Shivers, O. (1988). 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. Association for Computing Machinery.

[Stanier and Watson 2012] Stanier, J. and Watson, D. (2012). A study of irreducibility in C programs. *Software: Practice and Experience*, 42(1):117–130.

[Stanier and Watson 2013] Stanier, J. and Watson, D. (2013). Intermediate Representations in Imperative Compilers: A Survey. *ACM Computing Surveys*, 45(3).

[Steel 1961a] Steel, T. B. (1961a). A First Version of UNCOL. In *Papers Presented at the May 9-11, 1961, Western Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM '61, page 371–378, New York, NY, USA. Association for Computing Machinery.

[Steel 1961b] Steel, T. B. (1961b). UNCOL: The Myth and the Fact. In Goodman, R., editor, *Annual Review in Automatic Programming*, volume 2 of *International Tracts in Computer Science and Technology and Their Application*, pages 325–344. Elsevier.

[Steele 1976] Steele, G. L. (1976). LAMBDA: The Ultimate Declarative. Technical report, USA. AI Memo 379.

[Steele 1978] Steele, G. L. (1978). Rabbit: A Compiler for Scheme. Technical report, USA.

[Strong et al. 1958] Strong, J., Wegstein, J., Tritter, A., Olsztyn, J., Mock, O., and Steel, T. (1958). The Problem of Programming Communication with Changing Machines: A Proposed Solution. *Communications of the ACM*, 1(8):12–18.

[SUIF 2005] SUIF (2005). SUIF Compiler System.

[Susungi and Tadonki 2021] Susungi, A. and Tadonki, C. (2021). Intermediate Representations for Explicitly Parallel Programs. *ACM Computing Surveys*, 54(5).

[Tanenbaum et al. 1983] Tanenbaum, A. S., van Staveren, H., Keizer, E. G., and Stevenson, J. W. (1983). A Practical Tool Kit for Making Portable Compilers. *Communications of the ACM*, 26(9):654–660.

[TenDRA 2022] TenDRA (2022). The TenDRA Project.

[Unger and Mueller 2002] Unger, S. and Mueller, F. (2002). Handling Irreducible Loops: Optimized Node Splitting versus DJ-Graphs. *ACM Transactions on Programming Languages and Systems*, 24(4):299–333.

[Wadler 2015] Wadler, P. (2015). Propositions as Types. *Communications of the ACM*, 58(12):75–84.

[Weise et al. 1994] Weise, D., Crew, R. F., Ernst, M., and Steensgaard, B. (1994). Value Dependence Graphs: Representation without Taxation. POPL '94, page 297–310, New York, NY, USA. Association for Computing Machinery.

[Wilson et al. 1994] Wilson, R. P., French, R. S., Wilson, C. S., Amarasinghe, S. P., Anderson, J. M., Tjiang, S. W. K., Liao, S.-W., Tseng, C.-W., Hall, M. W., Lam, M. S., and Hennessy, J. L. (1994). SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. *SIGPLAN Notices*, 29(12):31–37.

[Wirth 1993] Wirth, N. (1993). Recollections about the Development of Pascal. *SIGPLAN Notices - History of Programming Languages (HOPL)*, 28(3):333–342.

[Zakai 2011] Zakai, A. (2011). Emscripten: An LLVM-to-JavaScript Compiler. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '11, page 301–312, New York, NY, USA. Association for Computing Machinery.