FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	P4 MLIR Midend
Student:	Bc. Šimon Bařinka
Supervisor:	Ing. Viktor Puš, Ph.D., MBA
Study program:	Informatics
Branch / specialization:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	until the end of summer semester 2024/2025

Instructions

Familiarize yourself with the P4 language [0], the domain of intermediate representations (IRs) suitable for code optimizations, and the MLIR project [1].
Using MLIR, design and implement IR suitable for transformations and optimizations of P4-specific constructs.
Describe and implement translation from the existing P4 abstract syntax tree IR [2] to the new IR.
Describe and implement selected optimizations of the P4-specific constructs using the new IR. These constructs include but are not limited to the Packet parser block and Match-action control block.
Describe and implement translation of the main parts of the new IR to another existing program representation (LLVM [3] or CIRCT [4]).

[0] - https://p4.org
[1] - https://mlir.llvm.org
[2] - https://github.com/p4lang/p4c
[3] - https://llvm.org
[4] - https://circt.llvm.org

Electronically approved by doc. Ing. Jan Janoušek, Ph.D. on 5 October 2023 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master’s thesis

P4 MLIR Midend

Bc. Šimon Bařinka

Department of Theoretical Computer Science

Supervisor: Ing. Viktor Puš, Ph.D., MBA

February 15, 2024

Acknowledgements

I sincerely thank my family and friends for their unwavering support during my studies. I extend special thanks to my supervisor, Ing. Viktor Puš, Ph.D., MBA, for joining me on the uncharted path of fusing P4 and MLIR.

Declaration

In Prague on February 15, 2024 ....

Citation of this thesis

Bařinka, Šimon. *P4 MLIR Midend.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

Abstrakt

Jazyk P4 je používán pro programování konfigurovatelných síťových zařízení. V této práci zkoumáme využití MLIR k vytvoření rozšiřitelného P4 překladače, který je schopen pracovat s několika hardwarovými a P4 architekturami. Součástí práce je návrh P4 midend architektury, která je založena na kombinaci MLIR dialektů modelujících různé stupně abstrakce. Naše midend architektura umožňuje vytvořit optimální model pro implementaci různých IR transformací, k čemuž využíváme MLIR a jeho schopnost postupného překladu. Všestrannost naší architektury demonstrujeme implementací několika optimalizací, které využívají sémantiky zvolené P4 architektury. V neposlední řadě popisujeme překlad P4 AST do našich MLIR dialektů, a jejich následný export do LLVM IR.

Klíčová slova překladač, midend, IR, P4, MLIR, LLVM

Abstract

P4 is a language for programming configurable network devices. In this thesis, we explore leveraging MLIR to create an extensible P4 compiler, capable of accommodating diverse P4 architectures and hardware backends. We design an extensible P4 midend architecture, powered by MLIR dialects operating at multiple levels of abstraction. We utilize MLIR's progressive IR lowering to provide an optimal IR model for various P4 compiler passes. To demonstrate the flexibility and reasoning power of the proposed P4 midend, we implement multiple P4 architecture-specific optimizations. Finally, to provide context of a full compilation pipeline we describe the conversion of P4 AST to our dialects and their export into LLVM IR.

Keywords compiler, midend, IR, P4, MLIR, LLVM

Contents

List of Figures

Listings

Chapter 1

Introduction

"The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise."
— Edsger W. Dijkstra [2]

1.1 Motivation

The Need for Language-Specific Midends

The compilation pipeline is commonly described as a sequence of three components: *frontend*, *midend*, and *backend*. As the input programming language progresses through these phases, it takes on various forms, known as *intermediate representations (IRs)*.

A rigorously defined set of IRs allows to decouple frontend, midend, and backend, enabling each component to focus on its specific responsibilities: The frontend and backend concentrate on understanding the source and target languages, respectively, while the midend acts as an intermediary transformer from IR to IR, enhancing the IR in a predefined manner[3].

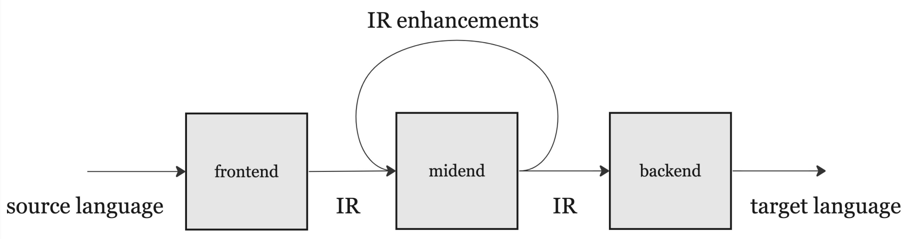

Figure 1.1: Three-component compilation pipeline.

The three-component compiler architecture not only provides a clear mental model but has also laid the foundation for technologies such as LLVM[4] or

JVM[5]. These technologies define a robust IR that can be targeted by programming languages looking to benefit from readily available midend and backend, therefore avoiding the high cost of reimplementing the whole compilation pipeline.

In the context of LLVM, to develop a substantially complete compiler for a new programming language, we must only provide a language-specific frontend. This frontend is primarily responsible for (1) the input validation, ensuring adherence to a language specification, and for (2) generation of semantically-equivalent output expressed in terms of the LLVM IR. From this point, the remaining part of the compilation pipeline is provided by LLVM. The LLVM midend executes diverse IR transformations, followed by the backend-facilitated conversion into one of the available target languages, such as x86-64 instruction set.

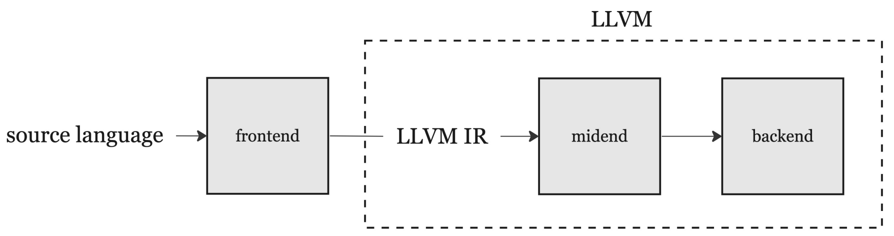

Figure 1.2: Compilation pipeline with reusable LLVM midend and backend.

To act as a common denominator, LLVM IR provides a fairly low-level abstraction, often described as “C with vectors”[6]. This design offers a straightforward lowering path from both high-level and low-level abstractions, thus ensuring that LLVM IR can be targeted by a wide array of programming languages.

As a consequence, LLVM IR acts as a double-edged sword. On the one hand, LLVM IR greatly contributes to the reusability of the LLVM system. On the other hand, LLVM IR is not capable of properly modeling constructs of high-level languages, causing language-specific information to be lost during the conversion.

This loss of source-level information makes LLVM IR an unsuitable abstraction for analyses and optimizations which rely on specifics of high-level languages. Such compiler passes are either not implemented at all[7], or they perform sub-optimally with unnecessarily complex implementations[8].

In the quest to provide new features or enhance existing ones, language engineers often find it necessary to introduce another level of language-specific abstraction which stands between frontend and LLVM. Such abstraction introduces a custom IR, which, together with a supporting infrastructure, forms a separate component, commonly referred to as a *language-specific midend.*

The use of language-specific midends is exemplified within ecosystems of many popular languages. For instance, Swift named its midend SIL[9], Rust uses

MIR[7], and to better handle template instantiation, C++ is transformed into Clang AST[10].

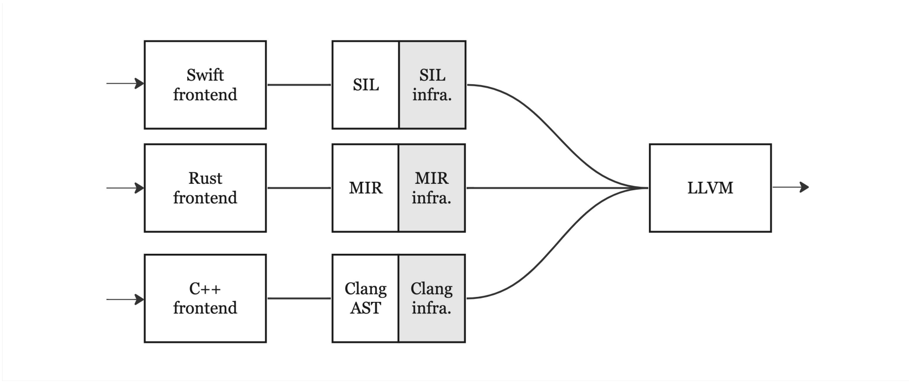

Figure 1.3: Examples of existing language-specific midends supported by custom infrastructures.

Language-specific midends enable better reasoning about high-level constructs, offer additional optimization opportunities, enhance static analysis, and solve language-specific issues. Thus offering great value despite the substantial initial investment, further development, maintenance, and allocation of engineering effort.

The Advent of MLIR

The emergence of language-specific midends stems from the unique requirements of each language. However, the supporting infrastructures implemented for these midends share many similarities. They commonly provide support for general IR inspection and transformation, implementing algorithms for IR traversal, IR rewrites, source location propagation, def-use chains computation, and many more. Additionally, compiler maintainers frequently find it necessary to develop many convenience tools, such as code generators and solutions for IR documentation.

Although these similarities do not create a direct duplication, an engineering effort can be made towards generalization and reusability. This effort led to the introduction of *Multi-Level Intermediate Representation (MLIR)*[6].

MLIR provides a highly extensible IR (sometimes referred to as *meta-IR*[6]), along with a flexible supporting infrastructure and utilities. MLIR identifies the parts of an infrastructure which are not unique to a language, and provides them as reusable extensible components. On the other hand, for the *truly* language-specific parts, MLIR provides tools for their definition and straightforward integration into the rest of MLIR. MLIR is developed as a sub-project of the LLVM Compiler Infrastructure project[4].

Contrary to LLVM, MLIR does *not* attempt to unify all language-specific midends into a single system, rather it embraces the necessity for uniqueness and extensibility. MLIR provides a foundation on top of which new midends can be built, allowing engineers to focus on the language-specific domain, instead of addressing already solved compiler problems.

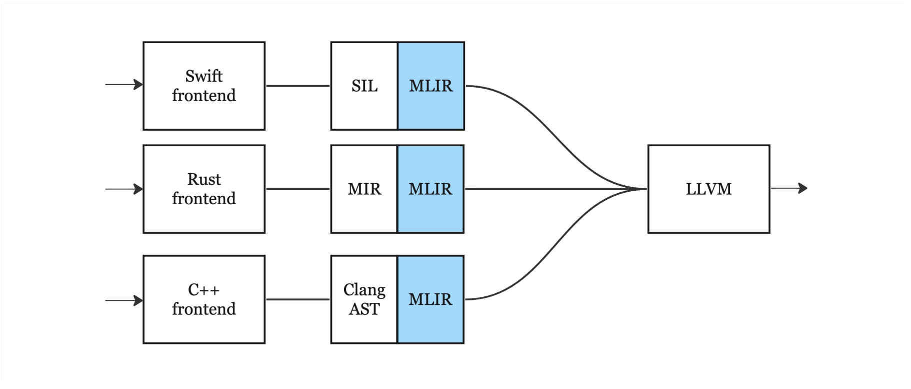

Figure 1.4: Example of possible adoption of MLIR within existing language-specific midends.

MLIR's meta-IR extensions are called *dialects*. Dialects provide encapsulation for IR components with a common goal. For instance, the *FIR*[11] dialect aims to closely model the Fortran language, or we could create a dialect which models the Python language.

Besides modeling entire programming languages, dialects can cover considerably smaller scopes. For instance, the *Linalg*[12] dialect provides IR components for modeling operations of linear algebra and the *Arith*[13] dialect encapsulates IR for basic arithmetic operations on integers and floating-point numbers.

The support for co-existence of these smaller dialects is one of the MLIR's key design points. A single IR can contain an arbitrary combination of dialects, even from different levels of abstraction. In many aspects, such a modeling expressiveness introduces a novel approach for the field of compiler design, allowing to tailor the IR for the exact requirements of each compiler pass.

In practical scenarios, language-specific midends combine multitude of both custom and existing MLIR dialects. This way, a new dialect must be created only for the abstractions unique to the language. On the other hand, if the modeled language additionally contains more common constructs, existing dialects can be reused.

P4's inherent adaptability

P4 is a domain-specific language for programming configurable networking devices. A P4 program describes transformations of networking packets and interactions with the host packet processing system. Driven by the networking industry requirements, P4 is used on a wide array of heterogeneous hardware architectures such as general-purpose CPUs, Field-Programmable Gate Arrays (FPGAs), System-on-Chips (SoCs), Network Processors, or Application-Specific Integrated Circuits (ASICs)[14].

To adapt to the diverse constraints and heterogeneity of the underlying hardware, P4 maintains flexibility and extensibility at its core. P4 allows a significant portion of its semantics to be customized for the intricacies of the execution architecture. This adaptability is achieved through the *P4 architecture model*[15], where a P4 architecture acts as a contract between a P4 program and the underlying execution unit.

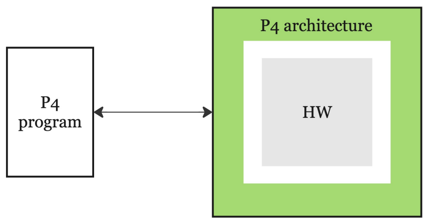

Figure 1.5: P4 architecture serves as an abstraction over a P4-programmable hardware.

Due to the P4's architecture model, P4 can be seen as having two layers of specification. Firstly, the P4 specification defines the core language (i.e., grammar and semantics) and enumerates the parts which must be defined or constrained by a P4 architecture. Secondly, the chosen P4 architecture fills in the parts of the core P4 specification which are left up to the P4 architecture. Importantly, the P4 architecture must work within the boundaries set by P4, meaning the P4 architecture specification cannot contradict the core P4 specification.

In general, a particular P4 architecture consists of P4 source files which declare architecture-specific interfaces, and a written specification with behavioral diagrams.

While existing P4 architectures are often vendor-specific and closely tied to a single hardware unit (e.g., Intel Tofino ASIC comes with the Tofino Native Architecture[16]), the P4 community supports a multitude of independent P4 architectures as well. For instance, *Portable Switch Architecture (PSA)*[1] models data flows and functionalities commonly found on networking switches. Similarly, *Portable NIC Architecture (PNA)*[17] models capabilities of a conventional network interface card.

The main goal of independent P4 architectures is to prevent development of multiple architectures which all provide a very similar abstraction. To offer a set of foundational architectures, PSA and PNA provide functionality commonly-found in conventional networking devices, therefore being a plausible P4 architecture to adopt for a wide array of programmable devices.

A P4 program is tightly coupled with the P4 architecture it was written for, which makes the use of an independent architecture a convenient choice for both P4 programmers and hardware vendors. Contrary to custom P4 architectures, an independent architecture has a greater chance to be supported by a greater number of P4-programmable devices, making a P4 program written for it much more portable. On the other hand, a hardware vendor which decides to support an independent architecture becomes automatically compatible with a set of already written P4 programs.

Apart from providing a common interface, an independent P4 architecture can serve as a foundation for other custom architectures. For instance, we can create a new P4 architecture which builds on top of PSA, exposing PSA's functionality together with capabilities unique to a specific device.

Fusion of P4 and MLIR

Considering the P4 architecture model and the P4's need to model a diverse range of hardware architectures, we argue that the adoption of MLIR into the P4 ecosystem would significantly benefit the P4 community.

By leveraging MLIR's capability to intermix dialects, we can develop a P4-specific midend which is capable of modeling P4 at various levels of abstraction. These abstractions can be decoupled into separate MLIR dialects and co-exist within a single IR, which allows to model a P4 program in an optimal way for various compiler passes.

Due to the P4's architecture model, the natural way to model the P4-specific dialects is by creating a single global P4 dialect, and a single P4 architecture dialect for each P4 architecture.

P4 dialect models the P4 language as defined by the P4 specification. P4 dialect makes no assumption about the chosen P4 architecture. P4 dialect allows to execute general compiler passes, which do not require the context of the architecture.

P4 architecture dialect (e.g., PSA or PNA dialect) models the semantics of a particular P4 architecture. In general, a P4 architecture dialect cannot model an entire P4 program, and must rely on the P4 dialect in aspects which are not defined or constrained by the architecture.

Although, during compilation, P4 and P4 architecture dialects can co-exist in a single IR, their respective developments can proceed with no mutual coordination. Moreover, if a new P4 architecture aims to build on top of an existing one, the new architecture dialect can simply declare dependency on any of the existing dialects, and thus reuse IR elements and implemented compiler passes.

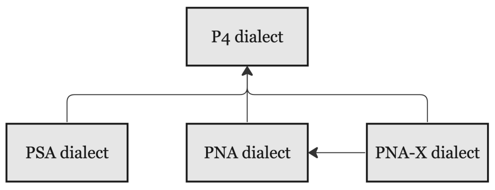

Figure 1.6: Dependency graph of possible P4 MLIR dialects. PNA-X builds on top of PNA, therefore their respective dialects declare corresponding dependency.

The adoption of MLIR provides a solid foundation for improvement of the compilation pipeline. Existing solutions typically operate at a single fixed point of abstraction, executing compiler passes on the P4 Abstract Syntax Tree (AST) which closely maps to P4's syntax. A hardware vendor then chooses a P4 architecture (possibly multiple) and creates a target-specific midend and backend which consume P4 AST.

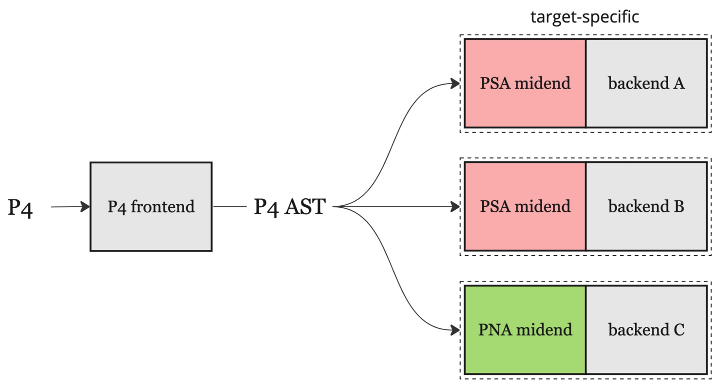

Figure 1.7: Conventional P4 compilation pipelines with PSA midend duplication.

Such approach prohibits the reuse of P4 architecture midends, which end up either duplicated across multiple backends or the P4 architecture-specific optimization and analyses opportunities are simply ignored.

Moreover, due to its one-to-one mapping to the input P4 program, P4 AST captures P4 constructs which primarily exist for the convenience of a human programmer, including implicit data flows and casts. While such information is appropriate for validation tasks such as semantic analysis, the subsequent compiler passes are more straightforward to implement on IR without implicit operations.

Using MLIR dialects, a more nuanced lowering path can be adopted through a progressive conversion across P4 AST, P4 dialect and P4 architecture dialect. This approach enables each compiler pass to operate on an optimal level of abstraction. For instance, once the semantic analysis (operating on P4 AST) is finished, all implicit data flow can be modeled explicitly within the P4 dialect, simplifying the data flow analysis.

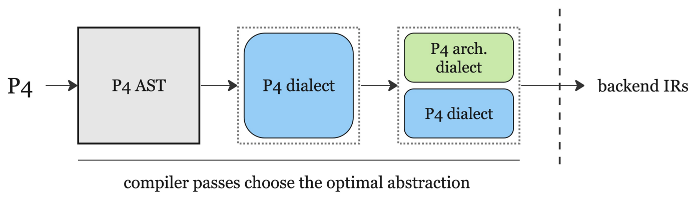

Figure 1.8: P4's progressively lowered IR across AST and dialects.

Within a progressively lowered compilation pipeline, a hardware vendor must still provide its hardware architecture-specific backend. However, such backend can reuse existing P4 architecture-specific midends together with their optimizations and analyses. Additionally, vendors have a choice to either export P4 dialects into their own system (e.g., LLVM), or build their backends using MLIR as well.

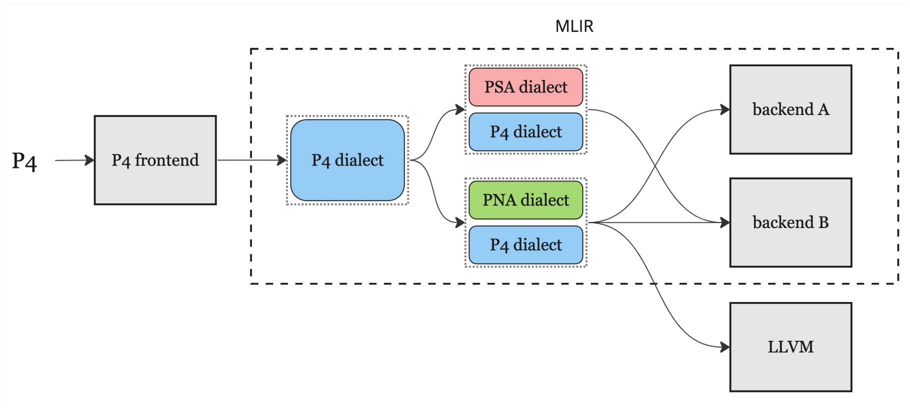

Figure 1.9: P4 compilation pipelines with reusable P4 architecture midends, built on top of MLIR dialects.

If appropriate, an MLIR-powered backend can combine P4 dialect, P4 architecture dialect and hardware-specific dialects within a single IR, allowing for unlimited expressiveness to accommodate arbitrary hardware-specific compiler passes.

Finally, the adoption of MLIR not only allows for more expressive and reusable compilation pipeline, but also connects the P4 and MLIR ecosystems. This allows the P4 community to benefit from compiler industry advancements even outside the P4, along with potentially flattening the learning curve for newcomers looking to contribute into the P4's toolchain.

1.2 Goal

This work investigates the application of MLIR in constructing a language-specific midend for the P4 programming language. To provide a context of a full compilation pipeline, we inset our P4 MLIR midend in between a readily available P4 frontend and an LLVM backend.

Firstly, we design the *P4 dialect* which closely models P4 in a P4 architecture-independent way. P4 dialect aims to identify implicit operations and data flows within P4, and instead model them *explicitly*. This key design point reduces the number of special cases that need to be considered and manually handled during optimizations and analyses, therefore simplifying complexity and implementation of compiler passes.

P4 dialect elements are introduced in a context of its mapping to P4 AST nodes produced by *p4c*[18], a reference P4 compiler.

Secondly, we describe a *"toy"* P4 architecture called *Simple Architecture (SA)*. Importantly, we use SA as an example P4 architecture for which we create a P4 architecture dialect named *SA dialect*. SA dialect aims to model the semantics and interfaces of SA. Using SA dialect, we demonstrate the key design principles necessary to implement a full-fledged P4 architecture dialect which is capable of modeling architecture's semantics.

To better understand interactions between P4 dialect and SA dialect, along with demonstrating the flexibility of P4 MLIR midend, we show P4 architecture-specific optimizations which are enabled by the combination of P4 dialect and SA dialect in a single IR.

Thirdly, to demonstrate a continuation of a compilation pipeline after the P4 midend, we export subsets of P4 dialect and SA dialect into LLVM IR.

The individual components designed in this work follow a natural flow of a compilation pipeline, and thus can be visualized by a simple diagram:

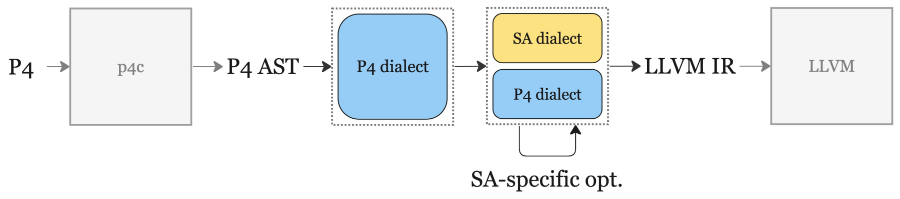

Figure 1.10: P4 MLIR midend specialized with SA dialect and inset in between p4c and LLVM.

It is worth noting that p4c and LLVM components remain grayed out to better indicate scope of this work. Even though we use p4c to produce P4 AST, we do not aim to either describe inner-workings of p4c or a verification of a P4 program. Similarly, we only use LLVM as a target of conversion from P4 dialects.

Finally, before we dive into the outlined specifics of the P4 MLIR midend, we begin with introductory chapters for P4 and MLIR.

Chapter 2

P4 Introduction

2.1 The Need for Another Language

Traditional networking devices were designed for data transmission within computer networks using a finite set of protocols. Using a predefined set of protocols meant that all device's functionality could be directly baked into its circuitry. Although this heavily limited extensibility of the device, the lack of programmability allowed to process packets in speeds two magnitudes faster than general CPUs[19].

The TCP/IP family of protocols remains a stable choice for routing from a network node to the network edge. On the other hand, the landscape of protocols used *within* a network is marked by constant growth[19]. As a result, the need for appropriate accommodation of ever-changing protocol set grew with the size of data centers and the adoption of cloud-computing[20].

However, expanding and updating the set of protocols contradicts the nature of *fixed-function* networking devices. To update the functionality of such networking units, network operators often had to wait for several years until hardware vendors updated their devices with the required functionality[20].

RMT

The demand for flexibility gave rise to the *Software-Defined Networking (SDN)* architecture, which introduced a significantly higher level of programmability of networking devices[21]. Nevertheless, the initial network processors were still a magnitude slower than the fixed-function devices. It wasn't until the adoption of the *Reconfigurable Match Tables (RMT)* hardware architecture that programmable ASICs were able to match the speeds of their fixed-function counterparts[19].

RMT is a carefully crafted hardware architecture, which enables to configure the devices with the contemporary protocols, while providing just enough functionality for accommodating the future ones.

To limit the gap between the speed of fixed-function and programmable devices, RMT forwards the packet through a pipeline consisting of fixed number of stages and ALUs. Each stage *parallelly* applies its ALUs onto the packet's fields and sends the transformed packet towards the next stage. To limit the costs of manufacturing RMT hardware, the number of stages is highly limited, with the original proposal accounting for 32 stages for the incoming (ingress) packet and 32 stages for the outgoing (egress) packet[19].

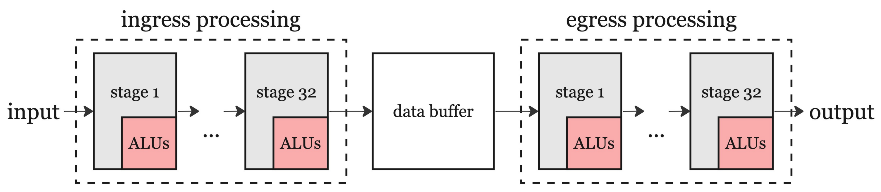

Figure 2.1: Ingress and egress stages of the RMT pipeline architecture.

The limited resources and the need for parallelism present a significant challenge for the compiler. To map a sequential program onto RMT hardware, the compiler must be able to analyze the data dependencies within a program and, wherever feasible, minimize them.

The Language for RMT

Bearing RMT's constraints in mind, the authors of P4 initially explored the use of established technologies, such as *Click*[22]. Click is a C++ framework designed for describing packet processing within a CPU kernel. While capable of describing packet transformations, the use of C++ essentially provides an *unconstrained expressiveness.* Such flexibility makes accurately inferring data dependencies within a written program a challenging task, often requiring overly conservative assumptions, which limit the complexity of programs that fit within the RMT's stages[14].

Aiming to limit unnecessary data dependencies while maintaining fair amount of flexibility, the P4 programming language was introduced in 2014 in [14]. Building on top of the foundation of the first release, a subsequent iteration, called $P4_{16}$, was introduced in 2016 in [15]. Simultaneously, the initial version was renamed to $P4_{14}$.

While sharing many similarities, $P4_{16}$ and $P4_{14}$ are not source compatible, and this study further considers only $P4_{16}$.

2.2 P4 Overview

P4 is a programming language designed to describe how packets undergo transformations within programmable packet-processing systems, such as hardware or software switches, network interface cards, or routers[14].

P4 is protocol-agnostic. It does not provide built-in implementation even for the most popular networking protocols, such as Ethernet or IP[15].

Despite being a domain-specific language tailored for the networking industry, P4 is an imperative language incorporating concepts commonly found in other general-purpose languages: P4 is statically typed and contains expressions, statements, declarations, mutable variables and constructs resembling functions.

Note on scope:

While this chapter draws from the $P4_{16}$ specification [15], our objective is not to replicate it. Rather, we gradually implement a standalone P4 program for an illustrative routing protocol.

Instead of providing an exhaustive specification, our focus is on introducing the domain of P4 and its fundamental concepts, which are crucial for understanding the chapters that follow.

P4's Informal Specification

P4 has evolved without a formal foundation. Its language specification primarily consists of informal prose, behavioral diagrams, and pseudocode. Such approach not only limits the formal reasoning about the language and its type system, but also leaves numerous aspects of the language up to the interpretation of each implementation.

Even though the P4 specification provides a satisfactory documentation for a P4 programmer, compilers are better built on solid formalisms of language theory. However, the need for the formal foundation was soon recognized by the P4 community and materialized within the *Petr4* [23] project.

Petr4 provides a formal framework for the **subset** of the P4 programming language. It systematically fills the holes of the original P4 specification and provides formal judgments about the core P4 constructs.

As such, for the purpose of this work, we choose to *not* divide P4 into the original P4 and Petr4's subset. We rather refrain from making any formal statements about the P4's properties, and instead provide only behavioral descriptions which correspond to the P4's specification.

2.3 Types

P4 contains base types and derived types familiar to those commonly found in other general-purpose languages:

- `void` - represents no values and can appear only in few restricted places of a P4 program.
- `bool` - represents either the `false` value or the `true` value.

- `bit<W>` - represents an unsigned integer (also called a *bit-string*) of an arbitrary bit width `W`. The value of `W` must be a compile-time known non-negative integer.

- `int<W>` - represents a signed integer of an arbitrary bit width `W`. P4's signed integers use two's complement encoding. The value of `W` must be a compile-time known positive integer.

- `int` - represents an arbitrary-precision signed integer. The `int` type is reserved for compile-time known integer values.

- `string` - represents a character string. P4 does not define any operations on strings nor allows to declare parameters or variables of type `string`. P4 strings can appear in few restricted places as constant string literals.

- `enum` - represents a type with several named constant values. An `enum` declaration introduces an identifier for a new `enum` type, along with its named values.

- `struct` - represents a type with several typed and named field variables. A `struct` declaration introduces an identifier for a new `struct` type, along with its fields.

- `tuple<T, ..., F>` - represents a value which holds values of types `T, ..., F`.

— *P4*$_{16}$ *specification* [15]

Additionally, P4 contains multiple P4-specific types suited for its networking domain: `match_kind`, `varbit`, `header`, `header` stacks, `header_union`, `extern`, `parser`, `control`, `package` and type specializations.

We examine some of these domain-specific types together with operations on the already introduced types later in the context of other P4 constructs.

2.4 Parser

The packet's path through a P4 program starts at the P4 parser block. The parser's primary purpose is to parse an incoming packet data (unstructured sequence of bits) into structured P4 data types which can be handled further by P4.

Before we begin with the parser's definition, we need to specify the structure of a packet that our parser accepts. Any packet which does not correspond to such structure must be rejected.

For the purpose of the P4 introduction, we implement a **highly simplified** IP routing algorithm. We call such routing the *Simple IP Routing (SIPR)*. The SIPR protocol operates on the SIPR packets.

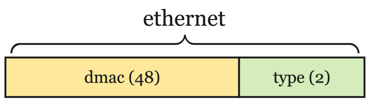

Figure 2.2: The structure of the SIPR Ethernet header.

SIPR Packet

The SIPR packet starts with the *Ethernet-like* header, consisting of 48-bit destination MAC address (*dmac*) and 2-bit *type* field. This header is visualized in Figure 2.2.

After the Ethernet-like part, depending on the *type* field, the packet can continue in two possible ways. If the *type* field contains the $0x0$ value, the packet continues with the *IPv4-like* header. On the other hand, if the *type* field contains the $0x1$ value, the packet continues with the *IPv6-like* header.

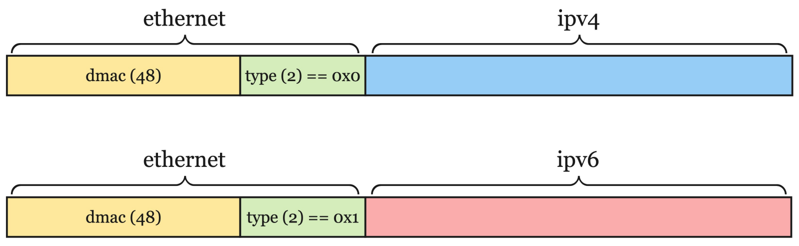

Figure 2.3: The structure of the SIPR packet.

Redundant *type* values:

While the SIPR protocol defines the *type* field to be two bits wide, the range of the accepted values (i.e., $0x0$ and $0x1$) can be represented with a single bit. We reserve the additional bit for future use cases. Such future-proof redundancy is typical across many networking protocols[24].

Finally, we need to define how the IPv4-like and IPv6-like headers look inside. Their structure is very similar; they both contain three fields: the source address, the destination address, and the time-to-live (*TTL*) field. The only difference is in the bit-length of the addresses: While the IPv4-like header uses 32-bit addresses, the IPv6-like header uses much wider 128-bit addresses. The detailed SIPR packet can be seen in Figure 2.4.

SIPR Packet Data Structures

To create a P4 parser for the SIPR packet, we first define data structures which can store the parsed SIPR headers. As described in the previous section, SIPR protocol recognizes three types of headers: *Ethernet*, *IPv4*, and *IPv6*[1].

[1]From now on, we omit the *"-like"* suffix.

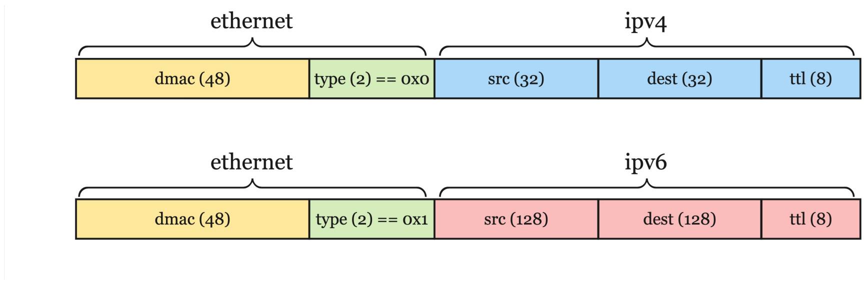

Figure 2.4: The structure of the SIPR packet with detailed IPv4 and IPv6 headers.

In P4, to encapsulate fields of a single header, we use the `header` type. We thus create three new `header` types with appropriate fields.

```
header Ethernet {
    bit<48> dmac;
    bit<2> type;
}

header IPv4 {
    bit<32> src;
    bit<32> dest;
    int<8> ttl;
}

header IPv6 {
    bit<128> src;
    bit<128> dest;
    int<8> ttl;
}
```

Listing 2.1: Declarations of the SIPR P4 headers.

For the individual fields we use bit-strings (`bit<W>`) and signed integers (`int<W>`) with bit widths as defined by the SIPR protocol.

To provide a more convenient way of manipulation, all P4 headers are traditionally encapsulated into a single P4 `struct`. We therefore declare the `SiprPacket` `struct`, which contains all of our headers.

```
header Ethernet { /*...*/ }
header IPv4 { /*...*/ }
header IPv6 { /*...*/ }

struct SiprPacket {
    Ethernet ethernet;
    IPv4 ipv4;
    IPv6 ipv6;
}
```

Listing 2.2: Declaration of the SIPR packet P4 structure.

A P4 `struct` does not assume any concrete arrangement. Its header fields can be therefore declared in an arbitrary order. In contrast, the individual fields within a single `header` must appear in the correct order, as defined by the implemented protocol.

SIPR Parser

With the declaration of the `SiprPacket` `struct` completed, we are ready to define our P4 SIPR parser.

We start by defining the `SiprParser` parser block. The `SiprParser` accepts an instance of `packet_in` object and an instance of the `SiprPacket` `struct`.

```
parser SiprParser(packet_in wire, out SiprPacket pkt) {
    state start {}
}
```

Listing 2.3: Definition of the P4 SIPR parser (1).

The `packet_in` is an *extern object*[2] provided by the *P4 core library*. It offers a stateful interface over the incoming unstructured packet bits. On the other hand, the `SiprPacket pkt` parameter is an output parameter (indicated by the `out` keyword) which serves as the storage for the parsed SIPR packet headers.

When the parser is invoked with these two required arguments, its execution begins within the `start` parser state. As we explore later, in addition to the compulsory `start` state, each P4 parser can contain any number of additional states. Together with transitions between them, these states form the so-called *parse graph*.

Given that we know the SIPR packet begins with the Ethernet header, we start by invoking the `extract` method of the `packet_in wire` object.

```
parser SiprParser(packet_in wire, out SiprPacket pkt) {
    state start {
        wire.extract(pkt.ethernet);
    }
}
```

Listing 2.4: Definition of the P4 SIPR parser (2).

The `extract` method is one of the fundamental P4 primitives for the packet parsing. Given a header H, the `extract` method reads just enough bits from the incoming packet buffer to fill all H's fields. In our case, we pass-in the `pkt.ethernet`, which causes a read of fifty bits from the received packet. These bits are used for the initialization of the `bit<48> dmac` and `bit<2> type` fields inside the Ethernet `header`.

With the Ethernet header parsed, we can now read and manipulate the header's fields within our program. To continue, we use the Ethernet's *type* field to

[2]P4 extern objects are a broad topic which we explore later in this work.

determine the next header. As defined by the SIPR protocol, either the *type*'s value is equal to $0x0$ and the packet continues with the IPv4 header, or its value is equal to $0x1$ and the next expected header is the IPv6 header.

To implement such a form of conditional parsing, we must extend our parse graph with two more states and provide appropriate transitions between them. To do that we use the *conditional transition statement*.

A conditional transition chooses the next parser state depending on a provided value and a set of patterns. In our case, we match the `pkt.ethernet.type` field against a set of simple patterns — $0x0$ and $0x1$. Depending on the result, the execution transitions into one of the newly defined parser states — `parseIPv4` or `parseIPv6`.

```
parser SiprParser(packet_in wire, out SiprPacket pkt) {
    state parseIPv4 {}

    state parseIPv6 {}

    state start {
        wire.extract(pkt.ethernet);
        transition select(pkt.ethernet.type) {
            0x0: parseIPv4;
            0x1: parseIPv6;
        }
    }
}
```

Listing 2.5: Definition of the P4 SIPR parser (3).

To finish our SIPR packet parser, we have two remaining tasks. Firstly, we need to complete the bodies of the `parseIPv4` and `parseIPv6` states. Secondly, we must define the behavior if the incoming packet does not have the expected SIPR structure.

For both of these tasks, we start by introducing two new parser states — the `accept` state and the `reject` state. These states are provided implicitly by the P4 language. Moreover, transitioning to either of these states terminates the execution of the parser. The `accept` state indicates successful parsing, while the `reject` state signals parsing failure.

To complete the `parseIPv4` and `parseIPv6` states, we first extract the next header into the `pkt.ipv4` field and the `pkt.ipv6` field, respectively. Then, to indicate successful parsing, we transition to the `accept` state using the *unconditional transition statement*. Similar to the conditional transition, the unconditional transition is prefixed with the `transition` keyword. However, instead of listing several patterns, we directly specify the destination parser state.

```
parser SiprParser(packet_in wire, out SiprPacket pkt) {
    state parseIPv4 {
        wire.extract(pkt.ipv4);
        transition accept; // <-- unconditional transition
    }

    state parseIPv6 {
        wire.extract(pkt.ipv6);
        transition accept; // <-- unconditional transition
    }

    state start {
        wire.extract(pkt.ethernet);
        transition select(pkt.ethernet.type) {
            0x0: parseIPv4;
            0x1: parseIPv6;
        }
    }
}
```

Listing 2.6: Definition of the P4 SIPR parser (4).

Finally, we need to handle situations where the incoming packet does not have the correct SIPR structure. Such packets should be handled by transitioning into the `reject` state, indicating parsing failure. Looking at the SIPR packet format, our parser can receive only two types of invalid packets.

Firstly, the incoming packet can be too short. In such a case, one of the `extract` method invocations fails to read enough bits to fill the destination header. Conveniently, the extract method handles such situation *automatically* by transitioning into the `reject` state.

Secondly, the Ethernet's *type* field might not be equal to any of the supported values — $0x0$ or $0x1$. The SIPR protocol defines the *type* field to be two bits wide, as a result, such situation can possibly occur.

This issue is best handled within the existing conditional transition. One of the possible solutions is to provide patterns for all invalid values with a transition into the `reject` state. However, P4 provides a much more compact solution. To specify a pattern which is selected only if none the other patterns match, we use the `default` pattern.

```
/* ... */
state start {
    wire.extract(pkt.ethernet);
    transition select(pkt.ethernet.type) {
        0x0: parseIPv4;
        0x1: parseIPv6;
        default: reject; // <-- default pattern
    }
}
```

Listing 2.7: Definition of the P4 SIPR parser (5).

Adding the `default` pattern with a transition into the `reject` state concludes the implementation of the SIPR parser. To sum up, our parser rejects all packets which do not match the defined SIPR packet structure. On the other hand, in case of a valid SIPR packet, we store the parsed headers into the output `SiprPacket pkt` parameter and transition into the `accept` state, indicating parsing success.

```
parser SiprParser(packet_in wire, out SiprPacket pkt) {
    state parseIPv4 {
        wire.extract(pkt.ipv4);
        transition accept;
    }

    state parseIPv6 {
        wire.extract(pkt.ipv6);
        transition accept;
    }

    state start {
        wire.extract(pkt.ethernet);
        transition select(pkt.ethernet.type) {
            0x0: parseIPv4;
            0x1: parseIPv6;
            default: reject;
        }
    }
}
```

Listing 2.8: The P4 SIPR parser.

2.5 Match-Action Pipeline

During parsing, we validated the incoming SIPR packet and initialized the `SiprPacket` structure. However, to perform any form of packet routing, parsing alone is not sufficient; we need to execute additional logic. Fundamentally, any routing algorithm consists of performing packet transformations and choosing the correct output port.

In P4, packet transformations and output port selection belong to the *match-action pipeline*. As we will explain later, the name corresponds to the match-action computational model, characterized by executing a sequence of configurable match-action units.

Before we implement the SIPR match-action pipeline, we need to describe the specifics of the SIPR algorithm. Specifically, how are the fields of the incoming SIPR packet used to determine the output port, and how is the SIPR packet transformed.

SIPR Algorithm

A computer network consists of a multitude of interconnected nodes. In general, a network forms an *incomplete* graph. As a result, to send a packet from node A to node B, we cannot rely on the existence of a *direct* connection between A and B. Instead, such a connection leads through several intermediary nodes. To mediate a connection between A and B, each node must adhere to a specific *routing* protocol.

Although we have already described the SIPR packet structure, the exact routing algorithm to which each node must adhere to remains undefined. The SIPR protocol dictates to each network node what transformations to execute and how to perform the next *"hop"*, which brings the SIPR packet one step closer to the destination node.

As we have defined, each SIPR packet contains two types of destination addresses: the MAC address and the IPv4/IPv6 address. Additionally, we assume that each network node has limited knowledge about the network topology:

1. Each node knows its own unique MAC, IPv4, and IPv6 addresses.
2. Each node knows MAC addresses of its *direct* neighbors. These associations are maintained as a table of *MAC-neighbor* pairs. We mark this table *MAC* → *neighbor*.
3. Each node stores two tables which map every possible destination IPv4/IPv6 address to the MAC address of the *"next-hop"* neighbor. We mark these tables *IPv4* → *MAC* and *IPv6* → *MAC*.

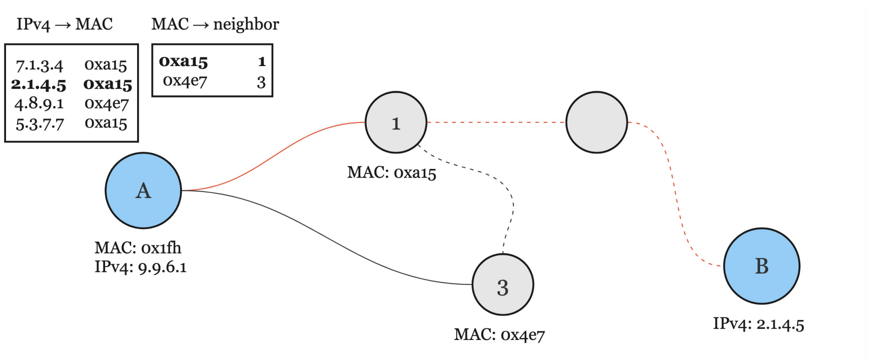

Figure 2.5: The network topology knowledge from the *A*'s viewpoint. *A* uses red connections to send packet to the 2.1.4.5 (*B*) IPv4 address, for which it uses bold records in its tables. The IPv6 information is omitted for brevity.

Upon receiving a SIPR packet *P*, the network node firstly compares its MAC address with *P*'s destination MAC address. If the MAC addresses do not match, the packet *P* is discarded. The node additionally checks whether *P*'s *time-to-live* field is greater than zero; if not, the packet *P* is also discarded.

Secondly, the *time-to-live* field of packet *P* is decremented by one. This operation, along with the initial test of positivity, prevents a packet from entering an infinite cycle within a network.

Thirdly, depending on the type of *P*, the node either chooses the *IPv4* → *MAC* or the *IPv6* → *MAC* lookup table. Using the selected table *T*, the node changes *P*'s destination MAC address. The new next-hop destination MAC address is retrieved from table *T* using *P*'s destination IP address.

Lastly, using the updated P's MAC destination address, the node selects an appropriate output port using the $MAC \rightarrow neighbor$ table. Subsequently, the node forwards the modified packet P to the next-hop node, where the SIPR algorithm repeats.

SIPR Match-Action Pipeline

With the SIPR algorithm fully defined, we can translate its mechanisms into the match-action computational model. In P4, match-action pipelines are embodied by the *control block* constructs, defined using the `control` keyword.

```
control SiprPipe(inout SiprPacket pkt, out bit<2> port) {
        apply {}
}
```

Listing 2.9: Definition of the P4 SIPR match-action pipeline (1).

Similarly to the parser block, we begin by defining the control block parameters. Our SIPR match-action pipeline is expected to firstly transform the SIPR packet according to the SIPR algorithm, and secondly select the appropriate output port. To accomplish this, the `SiprPipe` must be invoked with instances of the `SiprPacket pkt` and `bit<2> port` parameters.

Using the `SiprPacket pkt` parameter, the `SiprPipe` receives the parsed SIPR headers from our `SiprParser`. Indicated by the `inout` parameter direction, the `SiprPacket pkt` serves both as the input and output of the `SiprPipe`'s invocation.

To select the final output port, `SiprPipe` must set the second `port` output parameter during its execution.

Similarly to the parser's `start` state, the execution of every P4 control block begins inside its *apply* method, which we define using the `apply` keyword. To define our apply method's body we use the *if-statement* which performs the positivity test of the SIPR packet's *time-to-live* field.

```
control SiprPipe(inout SiprPacket pkt, out bit<2> port) {
        apply {
             if (pkt.ttl <= 0) {
                 port = 0;
                 return;
             }
        }
}
```

Listing 2.10: Definition of the P4 SIPR match-action pipeline (2).

P4's if-statement uses standard syntax and semantics familiar from many C-like programming languages. Therefore, if the *time-to-live* field is less than or equal to zero, we enter the if-statement's body, set the output `port` to zero to indicate the packet shall be discarded, and finish the `SiprPipe`'s execution by using the *return statement*.

Even though setting the output port to zero to discard the packet might seem counter-intuitive, P4 does not provide a native discard operation. Instead, we reserve the zero port to be the *"discard-port"*, leaving us with three more *"real"* ports.

The second check that the SIPR algorithm needs to perform is the MAC address test. The packet's destination MAC address must match with the device's MAC address. While we could hard-code the device's MAC address directly into the P4 program, we would need to recompile the program for each new device. To make our P4 program more practical, we utilize the main tool of the match-action computational model — the configurable match-action units. In P4, match-action units are embodied by the P4 *tables*.

Who invokes the `SiprParser` and `SiprPipe`?

At this point in our P4 introduction, there are few natural questions that we have not answered yet, such as:

- Where do we invoke our `SiprParser` and `SiprPipe`?
- Where is the `SiprPacket` instance created and how do we pass it into the `SiprParser` and `SiprPipe`?
- What if our hardware has more than four ports and therefore two bits are insufficient to cover all of our ports?

All these questions (and many similar ones) will be addressed by the *P4 architecture model*, which we cover later in this chapter. Until that point, we encourage the reader to consider these specifics as implementation details and accept them as facts:

- Our P4 program starts with the `SiprParser` and continues with the `SiprPipe`. Both blocks operate on the same instance of a `SiprPacket`.
- The `bit<2>` is enough to cover all of the device ports, including the discard-port.

SIPR Tables

Apart from offering a natural computational model for packet transformations, P4 tables introduce a possibility of runtime re-programmability.

Computer networks are dynamic, ever-changing systems. Throughout the lifespan of a network, individual nodes can be both added and removed. Moreover, a network's workload is often highly variable and unpredictable. For example, if a network monitoring detects a congested node, the network traffic might need to be rerouted through a chain of more, but less busy nodes. The network must be able to respond to these situations quickly and effectively.

Addressing such dynamic issues through recompilation of the entire P4 program would be highly impractical and time-consuming. Instead, P4 tables provide a form of quick re-programmability even after compilation. In our scenario, we could use a P4 table to perform the SIPR MAC address check, enabling us to compile our P4 program only once and reuse it for each new device.

Each P4 table maintains a *map* which stores a multitude of records. Each record consists of a value (*pattern*) and its associated action with action's arguments.

To execute a P4 table, we must provide a value. This value is compared with the patterns inside the map. If a match is found, the action (and its arguments) associated with the matched pattern gets executed. If there is no match, a default action is executed (can be a no-op action).

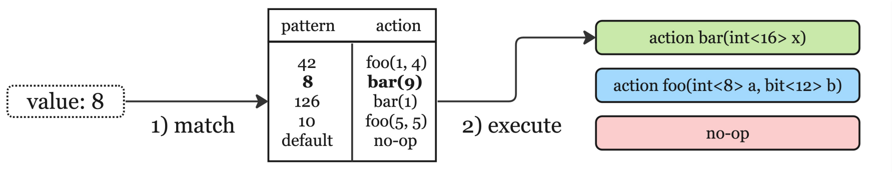

Figure 2.6: Example of a match-action unit execution.

The promised re-programmability of a P4 table is facilitated by manipulating its map. Records in the map can be dynamically both added and removed, enabling the execution of different actions during runtime. Adding nor removing a record does not require a program recompilation or lengthy packet-processing halts.

To define an action that can be executed after a pattern match, P4 provides the *P4 action* construct. Returning to our original goal of performing the SIPR MAC address check, we add a single `MacTest` table and a `discard` action above our apply method.

```
control SiprPipe(inout SiprPacket pkt, out bit<2> port) {
        action discard() {
            port = 0;
        }

        table MacTest { /* ... */ }

        apply {
            if (pkt.ttl <= 0) {
                discard(); // <-- replaced 'port = 0'
                return;
            }
        }
}
```

Listing 2.11: Definition of the P4 SIPR match-action pipeline (3).

We already know what to do if we need to discard a packet — we must set the `port` output parameter to zero. Therefore, inside the `discard` action, we simply execute the statement `port = 0`. Additionally, P4 actions are essentially independent of P4 tables and can be called like familiar C-like functions. Therefore, we add the `discard` action call into our existing if-statement to better describe the intent.

P4 is not isolated-from-above:

It is worth noting that in our previous snippet we were able to set the `port` value, even though the `port` variable is not `discard`'s parameter.

While we could have modified the `discard` action to contain a new parameter and provide the `port` at the call site, the P4 language conveniently allows referencing all symbols that are both declared lexically above and within the same curly-braced block.

More formally, the P4 language does not define its constructs as *"isolated-from-above"*, meaning it does not create artificial barriers in its lexical scoping.

Before we define the body of the `MacTest` table and execute it within our program, we need to discuss the type of records we need to store inside the `MacTest`'s map. Given the device's MAC address `DEVICE_MAC` and the packet's destination MAC address `PKT_MAC`, we execute the no-op action if the addresses match and `discard` action otherwise. Consequently, the map will contain two records which associate `DEVICE_MAC` with the no-op action and the default record associated with the `discard` action.

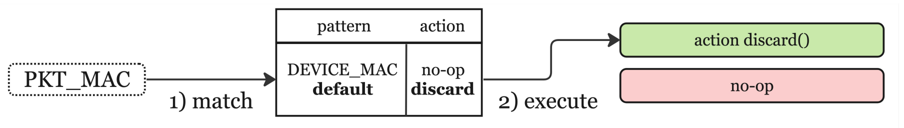

Figure 2.7: Example of `MacTest`'s runtime invocation.

A P4 table requires the definition of many of its structural properties beforehand. While the table's records remain dynamic, the structure of an individual record must be specified statically during compilation. This is achieved by listing a sequence of key-value pairs as a part of the table's body.

```
/* ... */
action discard() {
    port = 0;
}

table MacTest {
    key = { pkt.ethernet.dmac: exact; }
    actions = { discard; NoAction; }
    default_action = discard;
    size = 1;
}
/* ... */
```

Listing 2.12: Definition of the P4 SIPR match-action pipeline (4).

The `key` property defines the runtime value used to lookup a matching pattern in the table's map. In our case, we want to check if the packet's destination MAC address equals the device's MAC address, so we use the `pkt.ethernet.dmac` value. Additionally, the `pkt.ethernet.dmac` value is accompanied by the `exact` identifier, which specifies the matching algorithm. Aligned with our goal, the `exact` algorithm specifies that the pattern is looked up using a simple equals comparison.

Although using a single runtime value with the `exact` algorithm is sufficient for this thesis, the `key` property offers significantly more flexibility. The pattern can consist of multiple components, where each of them is matched using a different algorithm.

The `actions` property constraints the set of actions which can be associated with a pattern within the table's map. In our scenario, the `MacTest`'s records only need to use the `discard` action and the no-op action. The no-op action is conveniently provided to us by the P4 core library as the `NoAction` action.

The `default_action` property chooses the action which is executed if no other patterns match. As we discussed, in such a case the `MacTest` table must discard the packet. Therefore, the `MacTest`'s default action is the `discard` action.

The `size` property limits the maximum number of records that can fit into the table's map. Typically, this number is initially estimated and adjusted as necessary later on. In the special case of the `MacTest` table, we anticipate only a single record (with the default record not being included in this count).

The dual purpose of the `key` property:

In the list of P4 table's properties, the `key` property can be thought of as having a double purpose. Firstly, it defines the runtime value which is provided during the table's invocation. Secondly, the data type of the `key` property value constraints the data type of the pattern within each record.

In case of the `MacTest` table and its `pkt.ethernet.dmac` key, the `MacTest`'s patterns must have the `bit<48>` type.

With the `MacTest` P4 table fully defined, we can insert the table's invocation under the `ttl` test. Even though P4 table definitions do not contain an apply method, the table is executed using the `.apply()` syntax.

As described, the apply method firstly compares the runtime value (defined by the `key` property) with the patterns inside the table's map. Secondly, the action associated with the matched pattern is executed.

```
control SiprPipe(inout SiprPacket pkt, out bit<2> port) {
        action discard() {
            port = 0;
        }

        table MacTest {
            key = { pkt.ethernet.dmac: exact; }
            actions = { discard; NoAction; }
            default_action = discard;
            size = 1;
        }

       apply {
            if (pkt.ttl <= 0) {
                discard();
                return;
            }
            MacTest.apply() // <-- table invocation
       }
}
```

Listing 2.13: Definition of the P4 SIPR match-action pipeline (5).

The final unclear concept concerning the P4 tables is the precise mechanism of manipulating the table's map. In our context, we have yet to describe how we insert the record which associates the device-specific `DEVICE_MAC` value with the `discard` action.

Although the low-level details are hardware-dependent, the manipulation of the table's map is facilitated through what is known as *the control plane*. The control plane is a program that communicates with a running P4 program in various ways. While the concept of a control plane is not unique to the P4 ecosystem, in P4 terms, it handles the manipulation of the P4 table maps. As we see later, the communication between the control plane software and the P4 program extends beyond runtime configuration of the P4 tables.

In practical scenarios, the control plane software typically runs on an auxiliary CPU alongside the P4-programmable silicon[3]. This software is usually written in a general-purpose programming language and communicates with the P4-programmable processor through a hardware interface.

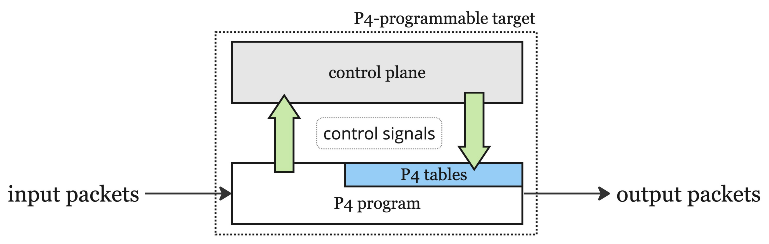

Figure 2.8: A P4-programmable target with a control plane.

[3]Traditionally called the *data plane*.

SIPR Headers Validity

With the packet's destination MAC address and *time-to-live* fields verified, we can continue with finding the next-hop MAC address and sending the SIPR packet out on its way.

As covered earlier, each SIPR node should maintain *IPv4* → *MAC* and *IPv6* → *MAC* tables, which provide the next-hop MAC address using the destination IPv4 or IPv6 address. Therefore, we first create the `Ipv4ToMac` and `Ipv6ToMac` P4 tables. Then, we invoke one of these two tables, depending on the type of the incoming SIPR packet.

```
control SiprPipe(inout SiprPacket pkt, out bit<2> port) {
    /* ... */

    table Ipv4ToMac {
        /* ... */
    }

    table Ipv6ToMac {
        /* ... */
    }

    apply {
        if (pkt.ttl <= 0) {
            discard();
            return;
        }
        MacTest.apply()

        if (/* packet is IPv4 */) {
            Ipv4ToMac.apply();
        } else if (/* packet is IPv6 */) {
            Ipv6ToMac.apply();
        }
    }
}
```

Listing 2.14: Definition of the P4 SIPR match-action pipeline (6).

To determine the type of the incoming packet within the `SiprPipe`, we can leverage the structure of our parse graph. To recapitulate, depending on the type of the incoming SIPR packet, the `SiprParser` either visits the `parseIPv4` or `parseIPv6` parser states. Within these states, we then execute `wire.extract(pkt.ipv4)` or `wire.extract(pkt.ipv6)` respectively.

We could therefore use the mutual exclusivity of the `parseIPv4` and `parseIPv6` states, to recognize the type of the incoming packet within the `SiprPipe`. To do that we must first learn about the *validity flag* of P4 headers.

Each P4 header instance contains a hidden validity bit of type `bool`, whose value is initially `false`. The value can be manually manipulated using the `setValid()` and `setInvalid()` methods. Importantly for our use case, the `extract` method, apart from filling the destination header with the parsed data, sets the header's validity bit to `true`.

```
parser SiprParser(packet_in wire, out SiprPacket pkt) {
    state parseIPv4 {
        // sets the validity bit of the pkt.ipv4 header
        wire.extract(pkt.ipv4);
        transition accept;
    }

    state parseIPv6 {
        // sets the validity bit of the pkt.ipv4 header
        wire.extract(pkt.ipv6);
        transition accept;
    }

    state start {
        /* ... */
    }
}
```

Listing 2.15: The *parseIPv4* and *parseIPv6* parser states which set the validity bits of their respective headers.

The SIPR parse graph ensures that only one of the IPv4 and IPv6 headers can have their validity bit set to `true`. We can therefore invoke either the `Ipv4ToMac` or `Ipv6ToMac` tables depending on the validity bits of the `pkt.ipv4` and `pkt.ipv6` headers. The validity bit can be queried using the `isValid()` method.

```
control SiprPipe(inout SiprPacket pkt, out bit<2> port) {
    /* ... */

    table Ipv4ToMac {
        /* ... */
    }

    table Ipv6ToMac {
        /* ... */
    }

    apply {
        if (pkt.ttl <= 0) {
            discard();
            return;
        }
        MacTest.apply()

        if (pkt.ipv4.isValid()) {
            Ipv4ToMac.apply();
        } else if (pkt.ipv6.isValid()) {
            Ipv6ToMac.apply();
        }
    }
}
```

Listing 2.16: Definition of the P4 SIPR match-action pipeline (7).

To finish the implementation of the `Ipv4ToMac` and `Ipv6ToMac` tables we list their properties and add a new `setNextHopDmac` action. Both of the new tables list the `setNextHopDmac` action within their `actions` property, along with the no-op `NoAction` action.

```
/* ... */

action setNextHopDmac(bit<48> nextHop) {
    /* ... */
}

table Ipv4ToMac {
    key = { pkt.ipv4.dest : exact; }
    actions = { setNextHopDmac, NoAction; }
    default_action = NoAction;
    size = 1024;
}

table Ipv6ToMac {
    key = { pkt.ipv6.dest : exact; }
    actions = { setNextHopDmac, NoAction; }
    default_action = NoAction;
    size = 1024;
}

/* ... */
```

Listing 2.17: Definition of the P4 SIPR match-action pipeline (8).

It is worth noting, that the only difference between `Ipv4ToMac` and `Ipv6ToMac` tables lies in their `key` properties. The `Ipv4ToMac` table must be provided with the packet's IPv4 destination address (`pkt.ipv4.dest`). On the other hand, the `Ipv6ToMac` requires the `pkt.ipv6.dest` value.

As we mentioned, a single P4 table record contains three components: a pattern, the associated action, and the action's arguments. In P4, the associated action's arguments are provided through the action parameters.

The `setNextHopDmac` action receives the next-hop destination MAC address via its `nextHop` parameter. To finish the `setNextHopDmac`'s body, we transform the SIPR packet by assigning the new MAC address into the `pkt.ethernet.dmac` field.

```
control SiprPipe(inout SiprPacket pkt, out bit<2> port) {
    /* ... */

    action setNextHopDmac(bit<48> nextHop) {
        pkt.ethernet.dmac = nextHop;
    }

    /* ... */
}
```

Listing 2.18: Definition of the P4 SIPR match-action pipeline (8).

Using the control-plane software, both `Ipv4ToMac` and `Ipv6ToMac` can now be filled with records, which associate destination IP addresses with the next-hop MAC address.

Finally, with the next-hop MAC address assigned to the `pkt.ethrnet.dmac` field, we must select an appropriate output port for the packet. This involves assigning the correct `bit<2>` value to the `port` parameter of the `SiprPipe` control block.

For this task, we once again employ a new P4 table. The new `MacToPort` table must store the *MAC* → *neighbor* mappings, which we have already discussed during the SIPR algorithm introduction. Importantly, we assume that the *“neighbor”* value corresponds with the required output port.

```
control SiprPipe(inout SiprPacket pkt, out bit<2> port) {
    /* ... */

    action setNewPort(bit<2> newPort) {
        port = newPort;
    }

    table MacToPort {
        key = { pkt.ethernet.dmac : exact; }
        actions = { setNewPort; NoAction; }
        default_action = NoAction;
        size = 1024;
    }

    apply {
        /* ... */

        if (pkt.ipv4.isValid()) {
            Ipv4ToMac.apply();
        } else if (pkt.ipv6.isValid()) {
            Ipv6ToMac.apply();
        }

        MacToPort.apply();
    }
}
```

Listing 2.19: Definition of the P4 SIPR match-action pipeline (8).

Similarly to the `setNextHopDmac` action, we use the `setNewPort` action, whose `newPort` parameter can be used to transmit the output port from the into the P4 program. The `newPort` parameter is then directly assigned into the `SiprPipe`’s `port` parameter.

Since the output port selection is the last operation of the SIPR protocol, we insert the `MacToPort` invocation as the last statement of the `SiprPipe`’s apply method.

SIPR Local Elements

Before moving further, it is crucial that we address a subtle issue within our SIPR match-action pipeline. This issue becomes evident upon reviewing the `SiprPipe`’s apply method and the `MacTest` table, particularly focusing on the `discard` action that may be invoked by the `MacTest` table.

```
control SiprPipe(inout SiprPacket pkt, out bit<2> port) {

    action discard() {
        port = 0;
    }

    /* ... */

    action setNewPort(bit<2> newPort) {
        port = newPort;
    }

    /* ... */

    apply {
        if (pkt.ttl <= 0) {
            discard();
            return;
        }
        MacTest.apply() // <-- may invoke ‘discard‘

        /* ... */

        MacToPort.apply(); // <-- may invoke ‘setNewPort‘
    }
}
```

Listing 2.20: Definition of the P4 SIPR match-action pipeline (9).

As we know, the `MacTest` table compares the `DEVICE_MAC` address with the `PKT_MAC` address. If these addresses do not match, the `discard` action is triggered, setting the output port to the zero *“discard-port”*.

The issue arises when, after selecting the *“discard-port”*, the execution continues within the apply method. As expected, the program executes the rest of the apply method's statements, including the final `MacToPort` table invocation. Unfortunately, this `MacToPort` invocation invocation may overwrite the output port to a completely different port, potentially resulting in the packet not being discarded.

To address this issue, we need to conditionally return early from the apply method, depending on whether the `discard` action was invoked or not. One possible method to achieve this is by setting a boolean flag from within the `discard` action and returning from the apply method depending on this flag. This can be achieved by using a new P4 concept — *local elements*.

A local element is a variable initialized before the execution of the apply method. Similarly to tables and actions, local elements are defined above the apply method. Since none of the P4 constructs are isolated-from-above, local elements can be referenced and assigned throughout the entire control block, including actions and the apply method.

```
control SiprPipe(inout SiprPacket pkt, out bit<2> port) {

    bool shouldExit = false; // <-- local element definition

    action discard() {
        port = 0;
        shouldExit = true; // <-- set the flag
    }

    /* ... */

    apply {
        if (pkt.ttl <= 0) {
            discard();
            return;
        }
        MacTest.apply()

        // exit from the SiprPipe if the packet is discarded
        if (shouldExit) {
            return;
        }

        /* ... */
    }
}
```

Listing 2.21: Definition of the P4 SIPR match-action pipeline (11).

We define a boolean local element called `shouldExit`, initialized to `false`, and set it to `true` within the `discard` action. After invoking the `MacTest` table, we check the value of the `shouldExit` and return from the `SiprPipe` accordingly. This ensures that if the packet needs to be discarded, we exit the `SiprPipe` early, preventing the overwriting of the output *"discard-port"*.

By adding the `shouldExit` local element, we finalized the implementation of the SIPR match-action pipeline. The `SiprPipe` control block implements the defined behavior of the SIPR protocol: upon receiving a parsed SIPR packet, it either discards the packet or sets its next-hop address together with the correct output port, bringing it closer to its final destination.

```
control SiprPipe(inout SiprPacket pkt, out bit<2> port) {
    bool shouldExit = false;

    action discard() {
        port = 0; shouldExit = true;
    }

    table MacTest {
        key = { pkt.ethernet.dmac: exact; }
        actions = { discard; NoAction; }
        default_action = discard;
        size = 1;
    }

    action setNextHopDmac(bit<48> nextHop) {
        port.ethernet.dmac = nextHop;
    }

    table Ipv4ToMac {
        key = { pkt.ipv4.dest : exact; }
        actions = { setNextHopDmac, NoAction; }
        default_action = NoAction;
        size = 1024;
    }

    table Ipv6ToMac {
        key = { pkt.ipv6.dest : exact; }
        actions = { setNextHopDmac, NoAction; }
        default_action = NoAction;
        size = 1024;
    }

    action setNewPort(bit<2> newPort) {
        port = newPort;
    }

    table MacToPort {
        key = { pkt.ethernet.dmac : exact; }
        actions = { setNewPort; NoAction; }
        default_action = NoAction;
        size = 1024;
    }

    apply {
        if (pkt.ttl <= 0) {
            discard();
            return;
        }

        MacTest.apply()
        if (shouldExit) { return; }

        if (pkt.ipv4.isValid()) { Ipv4ToMac.apply(); }
        else if (pkt.ipv6.isValid()) { Ipv6ToMac.apply(); }

        MacToPort.apply();
    }
}
```

Listing 2.22: The P4 SIPR match-action pipeline.

2.6 Deparser

After the packet's route through the parser and match-action pipeline, it might appear that our task is complete, and the P4-programmable device can forward the packet. However, the P4 programming language offers an additional layer of flexibility: After the match-action pipeline, we have to determine which SIPR headers to transmit from our device and in what order. This process is commonly known as *deparsing*.

In our specific scenario, deparsing is straightforward. We transmit the SIPR headers in the same order they were received, starting with the Ethernet header followed by either IPv4 or IPv6 header.

Similarly to the match-action pipeline, the deparser is implemented as a control block construct.

```
control SiprDeparser(SiprPacket pkt, packet_out wire) {
    apply {}
}
```

Listing 2.23: Definition of the P4 SIPR deparser (1).

The `SiprDeparser` is invoked with the `SiprPacket` instance, which is the output of the preceding `SiprPipe`. Additionally, our deparser receives the `packet_out` extern object.

The `packet_out` object serves as the counterpart to the `packet_in` extern object, which facilitated an interface over the incoming packet in our `SiprParser`. On the ther hand, the `packet_out` provides an interface over the outgoing packet.

The `packet_out` object offers a single `emit` method, which accepts a single header argument. This argument is then serialized and copied into the outgoing packet buffer.

```
control SiprDeparser(SiprPacket pkt, packet_out wire) {
    apply {
        wire.emit(pkt.ethernet);
        wire.emit(pkt.ipv4);
        wire.emit(pkt.ipv6);
    }
}
```

Listing 2.24: Definition of the P4 SIPR deparser (2).

The order of `emit` method calls determines the order of headers in the outgoing packet. Moreover, before copying a header, the `emit` method verifies the header's validity bit. If the bit is set to `true`, indicating the header is valid and initialized, it is then copied into the outgoing packet buffer, otherwise it does nothing. In our case, this behavior allows for a simple sequence of `emit` calls without the need for additional control flow. The Ethernet header is emitted unconditionally, followed by either the IPv4 or IPv6 header, depending on the type of the SIPR packet which we have received in the parser.

While packet deparsing often involves intricate logic, our particular case is straightforward and includes just three `emit` method calls. To sum up, the `SiprDeparser` emits the Ethernet header followed by the IPv4 or IPv6 header, depending on the packet type. Once the apply method of the `SiprDeparser` finishes, the SIPR packet proceeds on its path towards the next network node, as defined by the selected output port.

2.7 P4 Architecture Model

The P4 specification does not define the full semantics of the P4 language, instead it leaves many aspects to be filled by the selected *P4 architecture*. While these missing pieces span across all levels of the P4 language, they primarily consist of interactions between individual control and parser blocks.

To understand the practical implications of the *P4 architecture model* it is helpful to look at this concept from the individual viewpoints of both hardware vendors and P4 programmers.

A hardware vendor develops a packet-processing device. To enable P4 programmability, the hardware vendor selects one or more P4 architectures. Subsequently, the hardware vendor creates a P4 compiler which supports writing a P4 program against the selected P4 architectures.

P4 programmers seeking to implement a set of networking protocols begin by selecting an existing P4 architecture. Subsequently, they write a P4 program against the chosen architecture. Finally, to execute their P4 program, they select a P4-programmable device equipped with a compiler supporting the selected P4 architecture.

It is worth noting, that in practical scenarios P4 programmers may be constrained to a single hardware device, forcing them to choose a P4 architecture supported by that particular device (or rather its compiler).

The natural follow-up question would be, which P4 architecture we used for our SIPR protocol implementation: Our SIPR program was written against an ad-hoc P4 architecture, which we have implicitly defined throughout this chapter. Since we reference our ad-hoc P4 architecture further in this work, for clarity, we name it the *Simple Architecture (SA)*. The key aspects of the SA can be summarized as follows:

- **Structure**: Our P4 program must consist of the parser, match-action pipeline, and deparser components. The parser is represented by the P4 parser block, while the match-action pipeline and deparser are represented by the P4 control block.
- **Control flow**: The SA firstly invokes the parser followed by the match-action pipeline and deparser.
- **Data flow**: All of the components operate on a single instance of a user-defined packet `struct`. Additionally, the parser receives a `packet_in` object,

the match-action pipeline receives a `bit<2>` object, and the deparser receives a `packet_out` object.

- **Parameter semantics**: Assigning a value to the match-action pipeline's `bit<2>` parameter selects the packet's output port. Moreover, selecting the zero port causes the packet to be discarded, skipping the execution of the deparser.
- **Parser semantics**: Within the parser, transitioning into the `reject` state causes the packet to be immediately discarded, bypassing the match-action pipeline and deparser.

Conceptually, every P4 architecture defines a fixed pipeline with several P4-programmable components. Additionally, an architecture creates and maintains several P4 objects which are passed into its components as arguments. In case of the SA, the fixed pipeline is a simple three-component architecture, with architecture-maintained objects corresponding to the parameters of the `SiprParser`, `SiprPipe`, and `SiprDeparser`. The only user-defined component parameter is the packet `struct`, which depends on the implemented protocols.

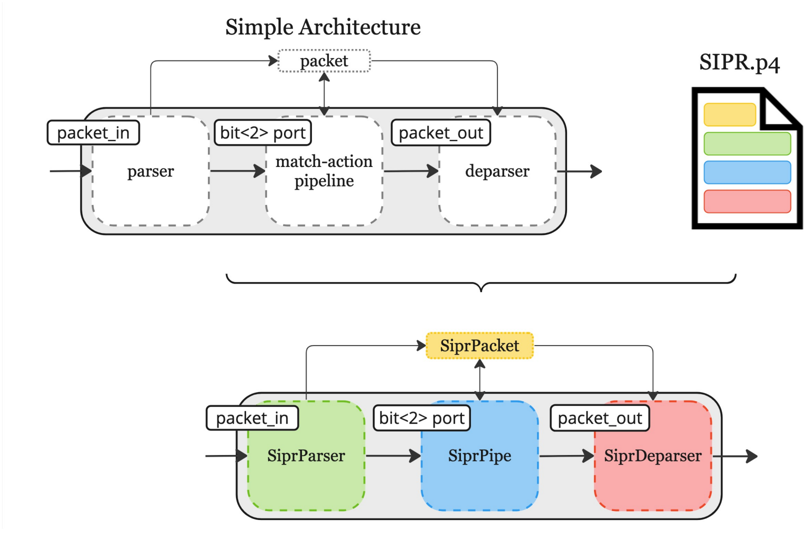

Figure 2.9: The SA pipeline programmed with the SIPR implementation.

To provide broader context, in Figure 2.10 we show the pipeline of the Portable Switch Architecture (PSA), which we have already mentioned in Chapter 1 as one of the open P4 architectures, maintained by the P4 community.

Contrary to the SA, the PSA pipeline provides much greater number of P4-programmable components, which are additionally inter-connected through recirculate channels. The recirculate channels allow to send a packet into the preceding stages, allowing to implement loops in the packet's path through the pipeline. In terms of architecture-maintained objects, PSA offers a set of metadata structures, which enable a fine-grained control over the packet's path, with access to utility information, such as the timestamp of the packet's arrival.

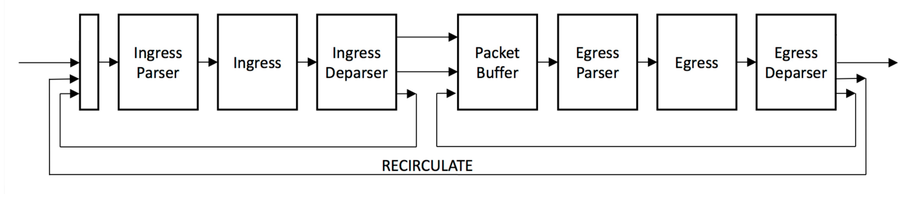

Figure 2.10: The Portable Switch Architecture (PSA) pipeline. Taken from [1].

While we have chosen the SA architecture for the implementation of the SIPR protocol, given the simplicity of the SIPR algorithm, we can imagine that an alternative implementation could be written against the PSA architecture as well.

The uncanny resemblance of RMT and PSA

In the introduction of this chapter, we introduced the RMT hardware architecture, which enabled efficient programmability of networking devices. Upon reexamining the RMT's architecture diagram in Figure 2.1, we can observe a striking resemblance to the PSA pipeline in Figure 2.10. This is not a coincidence. The PSA architecture is one of the first independent architectures[1], hence it is thoughtfully designed to provide a natural translation path to the prevalent hardware architecture of programmable packet-processing systems — the RMT architecture[19].

The semantics of a P4 architecture are typically provided as a written specification, often supplemented with behavioral diagrams and pseudocode. Importantly for the next section, each P4 architecture is accompanied by the *architecture-description file*. This file contains P4 declarations of the so-called *externs*.

SIPR Externs

Apart from defining a fixed pipeline programming model, a P4 architecture exposes functionalities unique to particular hardware units. Within a P4 program, these functionalities can be used through interfaces declared by the so-called *P4 extern objects*[4].

These hardware features often include specialized stateless units aiding in packet transformations, such as checksum units. Moreover, externs can expose *stateful* functionalities, which can persist a state across individual packet pipeline executions.

[4]Also called simply *externs*.

In terms of P4 constructs which we have covered in this chapter up to this point, only P4 tables are able to maintain their state across multiple received packets, while being mutable at the same time. As a result, if we assume constant P4 tables, P4 programs could be thought of as *"pure"*. This implies, that the value of the outgoing packet depends solely on the value of the received packet.

While the stateless programming model can simplify program reasoning for both compiler and programmer, it falls short in conveniently solving numerous networking problems, such as various network monitoring tasks.

To demonstrate the practicality of stateful externs in implementing monitoring tasks, we integrate a basic network traffic statistic into our SIPR program. Our goal is to count the successfully forwarded IPv4 and IPv6 SIPR packets.

For this purpose, we add the `Counter` extern into the SA. We define the `Counter` to be an abstraction of a persistent integer, which can be initialized to zero, and subsequently incremented by one multiple times.

```
extern Counter {
    // constructor which initializes counter to zero
    Counter();

    // increments the counter by one
    void increment();
}
```

Listing 2.25: The SA Counter extern declaration.

The fundamental characteristic of all P4 externs is their absence of P4 implementation. A P4 extern solely provides its interfaces, which include the extern's name, constructor names, method names, method parameters, and method return types. On the other hand, the semantics of an extern are specified within the architecture's documentation. Consequently, it is the responsibility of a compiler to ensure adherence to this documentation during program runtime.

To obtain an instance of an extern, it must be instantiated using one of its constructors. In case of the `Counter` extern, we defined a single constructor, which has no parameters and initializes the state of the counter instance to zero.

```
control SiprDeparser(SiprPacket pkt, packet_out wire) {
    Counter() ipv4Counter; // <-- IPv4 Counter instantiation
    Counter() ipv6Counter; // <-- IPv6 Counter instantiation

    apply {
        wire.emit(pkt.ethernet);
        wire.emit(pkt.ipv4);
        wire.emit(pkt.ipv6);
    }
}
```

Listing 2.26: Definition of the monitored P4 SIPR deparser (1).

A P4 extern must be instantiated either in the global scope or as a part of a parser or control block. It's important to note that although our `Counter` object is instantiated lexically in the same place as P4 local elements, it is initialized only once and maintains its state across individual packets. In contrast, a local element is initialized with its initial value before each apply method invocation.

After instantiation, we can proceed to invoke the extern's methods. Therefore, we call the `increment` method of one of our counters, depending on the type of the SIPR packet that is being deparsed.

```
control SiprDeparser(SiprPacket pkt, packet_out wire) {
    Counter() ipv4Counter; // <-- IPv4 counter instantiation
    Counter() ipv6Counter; // <-- IPv6 counter instantiation

    apply {
        if (pkt.ipv4.isValid()) {
            ipv4Counter.increment();
        } else {
            ipv6Counter.increment();
        }

        wire.emit(pkt.ethernet);
        wire.emit(pkt.ipv4);
        wire.emit(pkt.ipv6);
    }
}
```

Listing 2.27: Definition of the monitored P4 SIPR deparser (2).

Currently, the `Counter` extern does not include a method to return its state to a P4 program. We assume that a counter's state is queried and processed solely by a control plane. Such communication between a control plane and an extern is not part of a P4 program, and therefore, additional *"getter"* method within a counter's declaration is not necessary.

Finally, it is important to note that declarations of P4 externs are *not* meant to be created by a P4 programmer. A P4 programmer merely imports the so-called *architecture-description file* which contains all the extern declarations offered by the used P4 architecture.

Chapter 3

MLIR Introduction

MLIR streamlines the development of *reusable* and *extensible* compiler infrastructure. It offers a cost-effective approach to defining new IRs, and allows to represent the program's semantics by the combination of IR *dialects* operating on different levels of abstraction. MLIR achieves this by:

1. Standardizing the *Static Single Assignment*-based IR structure.
2. Providing a declarative framework for defining new IR elements and IR transformations.
3. Providing a wide range of common infrastructure components.

3.1 Dialects

MLIR introduces the concept of IR dialects. A dialect is a named collection of IR elements. By convention, elements of a single dialect model a common abstraction. This abstraction can range from an entire programming language, such as Fortran dialect, to a small set of arithmetic operations, such as floating-point numbers dialect.

Using dialects, MLIR presents a novel approach to program modeling. Firstly, a single program can be represented by the *co-existence* of several dialects, where each of them operates at a different level of abstraction. For example, we can imagine a point in our compilation pipeline, where we model the program's semantics using the combination of high-level concepts, such as the structured control flow dialect, and machine-level abstractions, like the ARM instructions dialect.

```
scf.while(%true) {
    %res1 = arm.add(%a, %b) : (i64, i64) -> i64
    %res2 = arm.sub(%res1, %1) : (i64, i64) -> i64
    %res3 = arm.mov(%res2) : (i64) -> i64
}
```

Listing 3.1: Example of the co-existence of the structured control flow dialect with the ARM dialect.

The second MLIR's novel design point is the idea of *progressive lowering*. MLIR enables to craft a fine-grained compilation pipeline, through which the IR progresses via small *partial* conversions. Specifically, this involves converting a single higher-level dialect to another lower-level dialect.

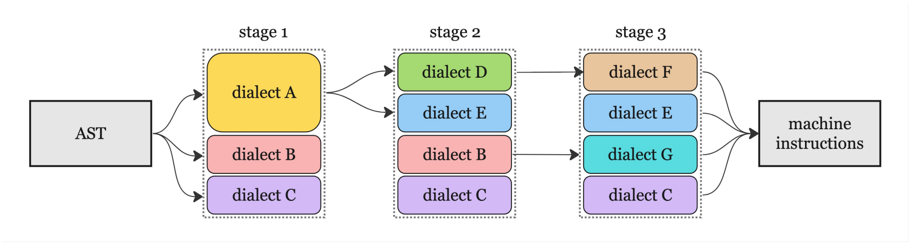

Figure 3.1: Example of a progressively lowered MLIR pipeline.

In contrast, IRs in traditional pipelines provide inextensible fixed-point abstractions.

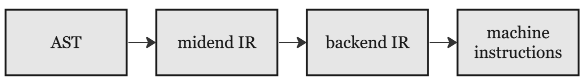

Figure 3.2: Example of IR conversions in a conventional pipeline.

The concepts of dialect inter-mixing and progressive lowering serve an important purpose: They enable to shift away from compiler passes which have to compromise and adapt to the underlying rigid IR. Instead, for each pass MLIR allows to pick the ideal collection of dialects which model the input program in a way that allows for easier implementation and optimal performance.

To prevent fragmentation and constant reinvention of even the most basic IR elements, MLIR provides a multitude of built-in dialects. As we see later, these built-in dialects span various levels of abstraction.

MLIR allows to both reuse existing dialects and develop new ones. To facilitate a common interface necessary for the co-existence of different dialects within a single IR, MLIR chooses to standardize the general structure of each dialect and its components.

Each MLIR dialect consists of three basic abstractions — *types*, *operations* and *attributes*. These abstractions are structured into the hierarchy of *regions* and *basic blocks*. Moreover, as we see later, MLIR's abstractions operate on *Static Single Assignment (SSA)* values[5].

[5]We provide SSA overview in Appendix A.

3.2 MLIR Basics

"The unit of semantics in MLIR is an 'operation', often referred to as Op. Everything from 'instruction' to 'function' to 'module' are modeled as Ops in this system." [6]

To model the semantics of the standard integer addition, we can use the `addi` operation. The `addi` Op is a part of the built-in `arith` dialect.

```
%a = "arith.addi"(%b, %c) : (i64, i64) -> i64
```

Listing 3.2: arith.addi operation.

The preceding snippet illustrates a canonical example of an Op usage. In MLIR, every operation is assigned a unique opcode, consisting of two dot-separated components. The first component denotes the Op's dialect (`arith`), while the second component represents the Op's name (`addi`).

Ops can both accept and produce zero or more values. In the textual representation, all MLIR values are prefixed with the `%` symbol. Our `addi` operation accepts values `%b` and `%c`, and produces value `%a`. All MLIR values adhere to the SSA form, implying that they must be defined (assigned) before use, and cannot be redefined (reassigned).

All uses of an operation specify the types of their arguments (also called *operands*) and produced values. In our example, the `addi` operation is provided with two 64-bit integers, and produces a value of the same type. This is indicated by the `(i64, i64) -> i64` suffix.

In the following snippet, the result of our addition is being compared against itself. To compare integers, we can use the `cmpi` operation which again comes from the `arith` dialect. The `cmpi` Op produces a boolean value.

```
%a = "arith.addi"(%b, %c) : (i64, i64) -> i64
%d = "arith.cmpi"(%a, %a) {pred = 5 : i64} : (i64, i64) -> bool
```

Listing 3.3: *arith.cmpi* operation.

The `arith.cmpi` operation again specifies the types of its arguments and results (`(i64, i64) -> bool`). In addition, the `arith.cmpi` Op must indicate the type of the performed comparison. The `cmpi` Op does so by using one of the MLIR's fundamental abstractions — *attributes*.

Each operation contains a map of its attributes (can be empty). An attribute represents a *compile-time known* value which is associated with a name and a type. The set of allowed attributes and their semantics are defined by the operation itself.

The `cmpi` Op requires a single `i64` attribute named `pred`. Setting the `pred` attribute's value to `5` instructs `cmpi` to perform a signed greater-than or equal comparison. The range of allowed `pred` values and their corresponding semantics can be found in the `cmpi`'s specification.

Textual Representation

MLIR represents its abstractions using in-memory objects that can be programmatically manipulated using MLIR's infrastructure. Additionally, MLIR provides the option to visualize these in-memory structures using the textual format, as demonstrated in our previous two examples. While acknowledging that MLIR is primarily a programmatic framework, in this work, we exclusively use its textual representation due to its better suitability for the written format.

MLIR's objective is to make its textual representation human-readable and debuggable[6]. As a result, MLIR allows to customize the textual representation, making its dialects succinct and more convenient to read. The representation which we have used so far is referred to as *generic*. The generic textual representation aims to closely mirror its in-memory counterpart.

On the other hand, the customized representation is referred to as *pretty-printed*. The pretty-printed version enables a more concise form without redundant information. In the case of the built-in `cmpi` Op, the comparison of both of its forms can be seen in the following snippet.

```
// Generic form
%d = "arith.cmpi"(%a, %a) {pred = 5 : i64} : (i64, i64) -> bool

// Pretty-printed form
%d = arith.cmpi sge, %a, %a : i64
```

Listing 3.4: The generic and pretty-printed textual forms of the *arith.cmpi* operation.

Besides omitting redundant type information, the `cmpi` Op takes advantage of the pretty-printed format and replaces the "`{pred = 5 : i64}`" representation of its attribute by a simple `sge` string, which allows to deduce the comparison type without reading the `cmpi`'s specification first.

Although the MLIR allows for a lot of flexibility, the pretty-printed version must still follow certain rules. For instance, both components of the opcode cannot be omitted. In addition, contrary to the generic form, the pretty printed opcode is not enclosed into quotation marks.

3.3 Constants

Traditional programming languages operate both on symbolic names (e.g., variables) and constant data (e.g, integer literal, string literal). In MLIR, symbolic names correspond to the `%`-prefixed names. However, the representation of constant data is a bit more nuanced.

To avoid creating a special abstraction for literals, MLIR uses combination of Ops and attributes. This design avoids a need for special handling of SSA values and literals, therefore simplifying implementation of compiler passes.

As an example, the `arith` dialect offers the `arith.constant` operation which produces an SSA value from its compile-time known attribute. For instance, to create a floating-point literal, we use an attribute of type `f32`.

```
// Floating-point constant
%a = arith.constant 4.2 : f32

// Equivalent generic form
%a = "arith.constant"() {value = 4.2 : f32} : () -> f32
```

Listing 3.5: The generic and pretty-printed textual forms of the *arith.constant* operation.

3.4 Regions

An MLIR operation can have multiple attached regions. The primary purpose of regions is to allow modeling nested constructs within the IR. Although regions are essential for MLIR's expressiveness, none of the Ops we have covered so far can contain regions. The closest operations with regions can be found within the built-in `scf` (structured control flow) dialect. Specifically, the `scf.if` Op.

```
"scf.if"(%a) ({
    %b = arith.constant 4 : i32
    %c = arith.addi(%b, %b) : i32
    scf.yield
}) : (bool) -> ()
```

Listing 3.6: The *scf.if* operation.

The `scf.if` operation contains a single region, beginning with "`({`" and ending with "`})`". The specification of the `scf.if` Op states that the attached region is executed if and only if the provided boolean operand has `true` value. This corresponds with the usual semantics of if-statements known from other programming languages. Additionally, the `scf.if` dictates that the attached region must be terminated with the `scf.yield` operation.

To model the usual semantics of the "*else block*", the `scf.if` Op allows to attach a second region. In this case, by passing an SSA value into the `scf.yield` Op, the `scf.if` Op allows to produce values as a result of the `scf.if` evaluation. Such usage is presented in the generic form in the following snippet.

```
%result = "scf.if"(%a) ({
    %b = arith.constant 4 : i32
    %c = arith.constant 2 : i32
    %d = arith.addi(%c, %c) : i32
    scf.yield %d : i32
}, {
    %e = arith.constant 5 : i32
    scf.yield %e : i32
}) : (bool) -> (i32)
```

Listing 3.7: The *scf.if* operation with else block and result value.

The presence of multiple regions underscores the importance of pretty-printed representations to improve readability. Similarly to the `cmpi` Op, the `scf.if` offers a pretty-printed format as well. While semantically equivalent to the preceding snippet, the custom representation resembles the conventional syntax of if-statements.

```
%result = scf.if %a -> i32 {
    %b = arith.constant 4 : i32
    %c = arith.constant 2 : i32
    %d = arith.addi(%c, %c) : i32
    scf.yield %d : i32
} else {
    %e = arith.constant 5 : i32
    scf.yield %e : i32
}
```

Listing 3.8: The pretty-printed form of the *scf.if* operation.

The `scf.if`'s custom representation does not include any type information for the provided condition operand (`%a`). Omitting information from the Op's custom textual representation is permitted as long as it does not introduce any ambiguity. In our case, the `scf.if` Op does not allow any other type of condition value other than `bool`.

Note:

Further in this study, if not stated otherwise, we prefer to use the more compact pretty-printed form of Op's representation.

3.5 Functions with Symbols

MLIR provides a built-in `func` dialect for expressing function definitions and function calls. To express these abstractions, MLIR uses the `func.func` and `func.call` operations together with the concept of *symbols*.

```
// Function declaration Op, declaring a symbol `addTwo`
func.func @addTwo(%x: i64) -> (i64) {
    %two = arith.constant 2 : i64
    %result = arith.addi(%x, %two) : i64
    func.return %result : i64
}

%0 = arith.constant 0 : i64
// Function call Op, referencing the symbol `addTwo`
%1 = func.call @addTwo(%0) : (i64, i64) -> i64
```

Listing 3.9: Example usage of the *func* dialect.

The `func` operations have semantics familiar from other C-like languages. The `func.func` operation defines a function, whose body is expressed using a single attached region. To execute the `func`'s region, the `func.func` operation must be referenced through the `func.call` Op. To facilitate Op references, MLIR offers the mechanism of *symbols*.

The MLIR symbols infrastructure provides a mechanism for referring to operations which declare a symbol. This allows for referring to operations by name from other Ops.

Additionally, the symbol infrastructure verifies several invariants of the IR. Firstly, MLIR checks that no two operations declare a duplicate symbol, and secondly MLIR checks that each symbol reference refers to an existing symbol declaration.

Interestingly, symbols do not belong to MLIR's core abstractions. Instead, they are constructed using fundamental concepts like Ops and attributes. Symbol references and declarations are attached to operations through attributes of a string-like type, and subsequently pretty-printed with the *@-prefix*.

3.6 Unstructured Control Flow

So far, to express the structured control flow, we have used the `scf.if` Op. However, MLIR aims to provide modeling capabilities for both higher and lower levels of abstraction. To offer control flow constructs that are conceptually closer to the hardware-level concepts, MLIR allows operations to be organized into control flow graphs (CFGs) and basic blocks.

In fact, basic blocks have been used in all of our MLIR examples thus far. Each region contains one or more basic blocks, with each basic block holding a sequence of Ops. Returning to the example of the `scf.if` operation, we can observe this underlying nested structure of regions and basic blocks.

```
%result = scf.if %a -> i32 {
    // <-- region start
        // <-- basic block start
            %b = arith.constant 4 : i32
            %c = arith.constant 2 : i32
            %d = arith.addi(%c, %c) : i32
            scf.yield %d : i32
        // <-- basic block end
    // <-- region end
} else {
    // <-- region start
        // <-- basic block start
            %e = arith.constant 5 : i32
            scf.yield %e : i32
        // <-- basic block end
    // <-- region end
}
```

Listing 3.10: The *scf.if* operation with highlighted hierarchy of regions and blocks.

To represent more complex CFGs within a region, we need to include additional basic blocks with jumps between them. Operations representing either conditional or unconditional jumps can be found inside the MLIR's `cf` (control flow) dialect.

The MLIR's built-in `cf` dialect offers a set of *terminator*[6] operations which express a transition between basic blocks of a single CFG. In this work, we use the `cf.cond_br` and `cf.br` Op, which model conditional and unconditional jumps, respectively.

In the following snippet, to demonstrate the usage of the `cf.cond_br` Op, we present two semantically-equivalent functions, each utilizing a different type of control flow.

```
// Structured control flow version
func.func @negate(%flag: bool) -> (bool) {
    scf.if %flag -> bool {
        %f = arith.constant false : bool
        func.return %f : bool
    } else {
        %t = arith.constant true : bool
        func.return %t : bool
    }
}

// Unstructured control flow version
func.func @negate(%flag: bool) -> (bool) {
    ^bb0:
        // Jump to ^bb1 if %flag is true, to ^bb2 otherwise
        cf.cond_br %flag, ^bb1, ^bb2

    ^bb1:
        %f = arith.constant false : bool
        func.return %f : bool

    ^bb2:
        %t = arith.constant true : bool
        func.return %t : bool
}
```

Listing 3.11: An illustrative function demonstrating difference between MLIR's structured and unstructured control flow Ops.

As depicted in the preceding snippet, when a region contains *one or more* basic blocks, each block must declare a label. These labels can then be referenced from the `cf.cond_br` Op to indicate the target of a jump.

Finally, it is worth noting, that both structured and unstructured types of control flow can co-exist, even within a single basic block. For instance, consider a `scf.if` Op modeling structured control flow inside a basic block which is part of a bigger CFG.

Basic Block Parameters

MLIR's basic blocks can have zero or more typed parameters. While these parameters can be used to model different concepts, its main goal is to model SSA's ϕ-nodes.

[6]Terminator operation can be used only as the last Op of a basic block.

Consider the following pseudocode function containing a ϕ-node.

```
function addOneOrTwo(num, flag)
    add$_1$ = 0
    if flag == true
        add$_2$ = 1
    else
        add$_3$ = 2
    add$_4$ = $\phi$(add$_2$, add$_3$)
    return num + add$_4$
```

Listing 3.12: The *addOneOrTwo* function expressed using pseudocode with *phi*-nodes.

In terms of MLIR, the `addOneOrTwo` function can be equivalently expressed using the unconditional branch Op `cf.br` with ϕ-nodes converted into basic block parameters.

```
func.func @addOneOrTwo(%num: i32, %flag: bool) -> (i32) {
    ^bb0:
        %add1 = arith.constant 0 : i32
        cf.cond_br %flag, ^bb1, ^bb2

    ^bb1:
        %2 = arith.constant 1 : i32
        cf.br ^bb3(%1: i32) // Jump to ^bb3 with argument %1

    ^bb2:
        %2 = arith.constant 2 : i32
        cf.br ^bb3(%2: i32) // Jump to ^bb3 with argument %2

    // ^bb3 has one integer parameter
    ^bb3(%arg : i32):
        %result = arith.addi(%num, %arg) : i32
        func.return %result : i32
}
```

Listing 3.13: The *addOneOrTwo* function expressed with the *cf* dialect.

Semantically, basic block parameters are equivalent to ϕ-nodes[6]. MLIR chooses basic block parameters, since unlike ϕ-nodes, basic block parameters are typically not treated as pseudo-instructions, and therefore do not require a special handling in various compiler passes[4].

3.7 Visibility of Values

MLIR's operations are organized into a non-trivial recursive hierarchy of basic blocks and regions. Therefore, it is worthwhile to discuss the rules of value visibility. The question of value visibility in MLIR revolves around determining which SSA values can be referenced from specific parts of an MLIR program.

For an SSA value reference to be valid it must comply with *all* of the following types of visibility:

1. **Region visibility**: *"Region-based visibility is defined based on simple nesting of regions: a value must be defined lexically above and inside the region of the use"*. The region visibility corresponds with the familiar notion of lexical scoping from other C-like languages.
2. **SSA dominance visibility**: All control paths which evaluate the SSA value reference must evaluate its definition first.
3. **Semantic visibility**: An operation can impose arbitrary restrictions on the SSA values contained in its regions. For instance, `func.func` is defined as *isolated-from-above*, indicating that no SSA value can reference SSA values defined outside of the `func.func`'s region.

— MLIR specification [6]

The following snippet demonstrates several examples which do not follow the listed rules.

```
/* Region visibility */
// error: %0 must be defined lexically above its use
%1 = arith.addi(%0, %0) : i32
func.func @foo1() {
    %0 = arith.constant 0 : i32
}

/* SSA dominance visibility */
func.func @foo2(%flag: bool) {
    ^bb0:
        %0 = arith.constant 0 : i32
        cf.cond_br %flag, ^bb1, ^bb2

    ^bb1:
        %1 = arith.constant 0 : i32
        cf.br ^bb2

    ^bb2:
        // allowed: every control path defines %0
        %2 = arith.addi(%0, %0) : i32
        // error: not every control path goes through ^bb1
        %3 = arith.addi(%1, %1) : i32
        func.return
}


/* Semantic visibility */
%0 = arith.constant 0 : i32
func.func @foo1() {
    // error: func.func Ops are defined as isolated-from-above
    %1 = arith.addi(%0, %0) : i32
}
```

Listing 3.14: Examples of invalid SSA value references

3.8 Type System

Every MLIR value has a type, indicated either by the operation which produced the value, or by a block parameter.

```
%1 = "arith.addi"(%arg1, %arg2) : (i32, i32) -> i32

func.func @foo() {
    ^bb0(%arg: i64):
        /* ... */
}
```

Listing 3.15: Sources of types of values. Relevant types and values are highlighted in red.

Every MLIR operation indicates expected types of its operands.

```
%1 = "arith.addi"(%arg1, %arg2) : (i32, i32) -> i32
```

Listing 3.16: Expected types of operands. Relevant types highlighted in red.

Type of the operand must match exactly with the type expected by the operation — MLIR does not define any implicit conversion rules.

```
%1 = arith.constant 4 : i32
%2 = arith.constant 2 : i64
 // error: type mismatch
%3 = arith.addi(%1, %2) : (i32, i32) -> i32
```

Listing 3.17: MLIR does not perform implicit type conversions. Type mismatch highlighted in red.

3.9 Extending MLIR

All operations, types, and attributes presented thus far are part of MLIR's built-in dialects. Built-in dialects aim to provide abstractions commonly found across many domains. Such abstractions include commonly used types (e.g., `i32`, `i64`, `f32`, `bool`), arithmetic operations (e.g., `arith` dialect), and control flow constructs (e.g., `scf` dialect, `cf` dialect).

Moreover, MLIR offers built-in dialects that closely model commonly targeted systems, including LLVM. For instance, the `llvm` dialect directly corresponds to the LLVM IR. Additionally, MLIR's infrastructure provides seamless conversion of such dialects into their respective target systems. Therefore, if our ultimate objective is to generate LLVM IR, we can first convert our dialects into the `llvm` dialect, and then utilize the conversion to LLVM IR provided by MLIR.

MLIR currently provides more than forty built-in dialects[25].

In cases when none of the built-in dialects are sufficient for expressing the desired abstractions, MLIR provides rich infrastructure for creating new dialects consisting of custom operations, types, and attributes. Conceptually, custom dialects are on par with MLIR's built-in dialects, enabling seamless co-existence of built-in dialects, our custom dialects, and third-party custom dialects.

To demonstrate a small example of a custom dialect, we create a miniature `stl` dialect, which models some of the semantics of the C++ `std::vector` data structure.

The `std::vector` is C++'s version of an automatically resizable array. Importantly for our scenario, the `std::vector` can be initialized as either empty or with several elements.

```
// Empty initialization
std::vector<int> vec1;

// Initialization with several elements
std::vector<int> vec2{4, 1, 9, 10};
```

Listing 3.18: Initialization of an std::vector object.

After the initialization, a new element can be added using the `push_back` method.

```
std::vector<int> vec;
vec.push_back(12);
```

Listing 3.19: Demonstration of appending an element into an *std::vector* object.

Using our `stl` dialect, we would like to model solely these two operations: initialization of an `std::vector` object and the `push_back` method. Consider the following C++ snippet.

```
std::vector<int> data;
data.push_back(1);
data.push_back(2);
data.push_back(3);
process(data); // Pass data further
```

Listing 3.20: Illustration of filling *std::vector* using the *push_back* method.

The preceding snippet can be represented using a combination of our custom `stl` dialect and the existing built-in dialects.

```
%data0 = stl.vector_constant [] : !stl.vector<i32>
%1 = arith.constant 1 : i32
%2 = arith.constant 2 : i32
%3 = arith.constant 3 : i32
%data1 = stl.push_back(%data0, %1) : !stl.vector<i32>
%data2 = stl.push_back(%data1, %2) : !stl.vector<i32>
%data3 = stl.push_back(%data2, %3) : !stl.vector<i32>
func.call @process(%data3) : (!stl.vector<i32>) -> ()
```

Listing 3.21: Filling an empty *std::vector* using *push_back*. Expressed in MLIR.

Firstly, the `stl` dialect contains a new parameterized type named `vector<T>`. Since our original C++ snippet works with integers, we choose `T = i32`. As we can also see in our snippet, all MLIR's custom types are prefixed with an exclamation mark followed by the name of the type's dialect.

Secondly, the `stl` dialect defines the `vector_constant` operation, which creates a value of type `!stl.vector<T>`. The `vector_constant` Op contains an array attribute, whose elements (which can be empty) are used to initialize the produced `!stl.vector<T>` value.

Finally, to model the `push_back` method, we define the `push_back` operation, which accepts the receiving `stl.vector<T>` value and a new element of type `T`. The `push_back` Op produces a new vector with the provided value appended.

To create values of type `i32`, and to call the `process` function, we use the built-in `arith` and `func` dialects.

With the semantics of our C++ snippet expressed completely in terms of MLIR dialects, we can implement diverse compiler passes, which leverage the `stl` dialect to reason about our program. For instance, we can implement an optimization pass which converts our original snippet with several `push_back` calls into a direct vector initialization.

```
%data = stl.vector_constant [1, 2, 3] : !stl.vector<i32>
func.call @process(%data) : (!stl.vector<i32>) -> ()
```

Listing 3.22: Creating a directly initialized *std::vector*. Expressed in MLIR.

Do we really need a special dialect?

It is important to understand that optimizing the provided C++ snippet (Listing 3.20) is not a straightforward transformation. Our optimization pass relies on the knowledge of the precise semantics of the `stl` Ops. For instance, in LLVM IR, calls to the `push_back` method are treated as regular opaque function calls, which call into a function whose body can be defined in a different translation unit. On the other hand, the `stl.push_back` Op is fully self-contained and elementary, with no other Ops needed to express its semantics.

The need to analyze possibly extensive function body restricts the ability to fully understand the `push_back`'s semantics and side effects. As of version 17.0.1, the LLVM-based C++ compiler Clang, cannot perform such optimization for the x86-64 architecture, even with the -O3 and -lto optimization options enabled. Instead of generating a single allocation for three integers, Clang emits three separate memory allocation calls.

TableGen, Constraints, Interfaces

MLIR is implemented using C++. As a result, new custom dialects and compiler passes must be implemented using C++ as well. To avoid intricacies of the C++ language, MLIR provides a *TableGen*-based domain specific language to simplify implementation of new dialects and compiler passes.

TableGen is a code generation tool created as a part of the LLVM project. LLVM uses TableGen to facilitate C++ code generation for commonly implemented constructs, such as LLVM IR elements[4]. Based on the successful adoption within the LLVM project, TableGen was employed for MLIR as well, providing a streamlined way of creating new compiler passes and dialects.

Although this text does not aim to provide comprehensive TableGen overview, we provide an illustrative example to better understand the necessary steps for defining a new MLIR Op. For this purpose, we'll use an example of creating the `push_back` operation, which was discussed in the previous section.

```
def STL_PushBackOp : STL_Op<"push_back",
        // Op's interfaces and multi-entity constraints
        [NoMemoryEffect, AppendVectorTypeConstraint]> {

  // One line documentation summary of this Op
  let summary =
    "Returns a copy of the input vector with a new element
    appended.";

  // Full specification of this Op
  let description = [{
    "Creates a copy of the ‘data‘ input vector and adds ‘value‘
    at the end of the new vector. The new vector is returned.
    Element type of ‘data‘ and type of ‘value‘ must be the same.
    "

    Serves to model the standard ‘std::vector::push_back‘
    method.
  }];

  // Ops can have a list of named and typed arguments
  let arguments = (ins STL_Vector:$data, AnyType:$value);

  // And a list of named and typed results
  let results = (outs STL_Vector:$output);
}
```

Listing 3.23: Definition of the *push_back* operation using TableGen.

Starting at the top of our snippet, new MLIR operation must define its dialect and mnemonic. Mnemonic is the operation's name used in the MLIR's textual format.

```
STL_Op<"push_back",
```

Listing 3.24: Definition of the *push_back*'s mnemonic and dialect using TableGen.

Secondly, using the *summary* and *description* fields, the operation can define its one line documentation, respectively the full specification of its semantics.

```
let summary = "/* ... */";
let description = "/* ... */";
```

Listing 3.25: Op's specification uses *summary* and *description* fields.

The MLIR specification describes these two fields as the standard way for documenting the individual dialect components[6]. Conveniently, MLIR provides a documentation generator tool which utilizes the *summary* and *description* fields to generate documentation for the entire dialect.

Next, the operation defines its named and typed SSA parameters and results.

```
let arguments = (ins STL_Vector:$data, AnyType:$value);
let results = (outs STL_Vector:$output);
```

Listing 3.26: Definition of the *push_back* Op's inputs and results using TableGen.

The `push_back` operation accepts a vector parameterized by the type `T` and a new value of the same type `T`. Then, it produces a value of the same type as the input vector value.

While specifying a single type for an input or result value is possible (e.g., `i32` or `i64`), MLIR uses a more general concept of *type constraints*.

Constraints describe a set of allowed types. The most permissive constraint is the `AnyType` constraint, which the `push_back` operation uses for the type of the appended value. The `AnyType` is an MLIR's built-in constraint and allows the value to be of any type. Another example of a built-in constraint is the `AnyInteger` constraint, which allows all integral types like `i16` or `i64`.

Additionally, the `push_back` operation uses the custom `STL_Vector` constraint for the input and output vector values. While we do not detail the implementation of the `STL_Vector`, the `STL_Vector` constraint limits the value's type to be of the `!stl.vector<T>` type with arbitrary element type `T`.

A constraint specified directly next to an input or result value can define a set of possible types for a *single* value. However, for the `push_back` operation, single-value constraints are not sufficient. We need to use a constraint that defines a relationship between *multiple* values. Specifically, we need to ensure that the types of the input and output vectors are the same, and that the element type `T` of the vectors matches the type of the appended value.

These requirements demand to define a *multi-entity constraint*. Multi-entity constraints are again defined via TableGen and specified during the Op's definition. In our scenario, we can call such constraint `AppendVectorTypeConstraint` and include it into a list right after the `push_back`'s mnemonic.

```
def STL_PushBackOp : STL_Op<"push_back",
        [NoMemoryEffect, AppendVectorTypeConstraint]> {
```

Listing 3.27: Operations can constraint its values using multi-entity constraints, such as the *AppendVectorTypeConstraint*.

Finally, in addition to multi-entity constraints, an operation can specify its *interfaces*. MLIR interfaces programmatically convey the semantics of an operation to compiler passes. In the case of the `push_back` Op, we specifiy the `NoMemoryEffect` interface.

```
def STL_PushBackOp : STL_Op<"push_back",
        [NoMemoryEffect, AppendVectorTypeConstraint> {
```

Listing 3.28: Operations can indicate its semantics to compiler passes using interfaces, such as the *NoMemoryEffect* interface.

The `NoMemoryEffect` is a built-in interface which indicates that the operation does not manipulate an external memory system. Compiler passes that rely on this fact can leverage this interface to reason about the operation. Importantly, adding built-in MLIR interfaces such as the `NoMemoryEffect` to custom Ops enables existing compiler passes to reason about new operations without modifying the compiler pass. For instance, we could design a compiler pass which traverses an MLIR program and eliminates every Op which causes no side effects and its resultant value is not used. Such pass could rely exclusively on the `NoMemoryEffect` interface to recognize such removable operations. This approach allows for reuse of transformations and optimizations with minimal additional effort.

While not presented in this text, TableGen allows to define many more properties of operations, such as the pretty-printed format or additional verifications of the Op's usage.

Chapter 4

P4 Dialect

The objective of the *P4 dialect* is to serve as a straightforward translation target for the P4 AST produced by *p4c*. Consequently, the P4 dialect preserves the general structure of the P4 language, avoiding any loss of relevant source information. This design enables optimizations and analyses that rely on the structure of the original P4 program.

On the other hand, the P4 dialect takes advantage of the fact that it is not meant to be written by hand and removes several P4 concepts primarily designed for the convenience of human programmers. As we see later, this involves explicitly capturing P4's implicit data flows and simplifying P4's calling convention.

These design choices simplify subsequent data flow analysis, reduce the number of cases that require special care by dialect passes, and allow for parallel processing of the IR.

The P4 dialect is introduced in terms of its mapping from the P4 language. Even though the translation into the P4 dialect is performed from P4 AST, we choose to present these conversions directly from the P4 language. Skipping the intermediary P4 AST is better suited for the text format, while at the same time does not introduce any ambiguity due to the close one-to-one mapping of the P4 language to P4 AST.

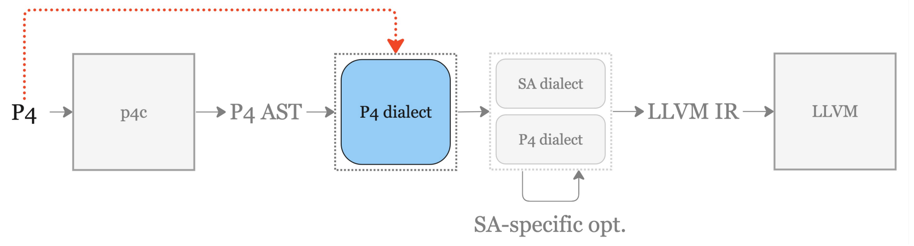

Figure 4.1: This chapter introduces the P4 dialect elements in a context of their mapping to the P4 language constructs.

P4's reference compiler *p4c*:

The P4 community maintains a reference P4 compiler called *p4c*.

"P4c is modular; it provides a standard frontend and midend which can be combined with a target-specific backend to create a complete P4 compiler."[18]

As described in Chapter 1, our dialects aim to provide an alternative solution to some of the key design points of p4c.

In contrast to our MLIR-based solution, p4c offers a *fixed-point* IR in the form of the *P4 Abstract Syntax Tree (P4 AST)*. The P4 AST closely mirrors the P4 language, even modeling concepts that introduce unnecessary complexity into subsequent passes. The P4 AST is supported by a custom infrastructure developed and maintained by the P4 community. Moreover, p4c delegates all P4 architecture-specific compiler passes to a hardware-specific backend.

In this work, we only utilize the frontend part of p4c. Specifically, we use the P4 AST generated by the p4c frontend as input into our MLIR-based midend. We assume the input P4 AST represents a valid P4 program.

4.1 Types

P4 dialect mirrors P4's types, offering a semantically equivalent counterparts for all P4 types, with the exception of the void type. Additionally, P4 dialect introduces the *reference type*, whose motivation and semantics we provide later in this chapter.

The majority of P4 dialect types appear in the text that follows, and their mapping to the P4 types is apparent (e.g., `header MyHeader` → `!p4.header<"MyHeader">`). Therefore, we list here only the types that require additional comments.

Integers: P4's signed and unsigned fixed-width integers are modeled using MLIR's built-in signed and unsigned fixed-width integers denoted as `uiW` and `siW`. For instance, `bit<12>` → `ui12` and `int<8>` → `si8`.

Booleans: For P4 booleans, the P4 dialect uses MLIR's built-in signless 1-bit integer denoted as `i1`, with 1 representing `true` and 0 representing `false`. In the textual format, MLIR allows printing the `i1` type as `bool`, along with `true` and `false` values.

Void: The P4 dialect does not include the void type. P4 extern methods and functions that return a value of type void are modeled to have no return value, therefore having no return type.

4.2 Action

To define an action, P4 dialect introduces the `p4.action` operation. The action Op declares a symbol and has a single attached region representing the action's body. This region must contain one or more basic blocks, representing CFG of the action's body.

```
action myAction() {}
```

```
p4.action @myAction() {
  p4.return
}
```

Listing 4.1: Action definition.

To return from an action P4 dialect uses the `p4.return` terminator Op. Contrary to P4, return operation is not implicitly assumed if it is missing. In case of a missing return Op, IR's verification fails.

To transfer control flow to the action's region, or in other words, to call the action, the P4 dialect utilizes the `p4.call` operation. This operation references the target action using the action's symbol. Similarly to P4, actions and action calls do not return a value.

```
myAction();
```

```
p4.call @myAction() : () -> ()
```

Listing 4.2: Action call.

Action operations contain zero or more SSA value parameters. Unlike in P4, parameters in the P4 dialect do not have a direction.

```
action myAction(in int<8> arg0, in bit<16> arg1) {}
```

```
p4.action @myAction(%arg0: si8, %arg1: ui16) {
  p4.return
}
```

Listing 4.3: Action definition with parameters.

To produce values for an action call with arguments, we can use the `p4.constant` operation which produces a constant of a given type and value. Similarly to the `arith.constant` Op, the type and value are provided as compile-time known MLIR attributes[7].

[7] We cannot reuse the `arith.constant` Op, since it allows only integral and floating-point constants which is not sufficient for P4.

```
myAction(4, 2);
```

```
%1 = p4.constant 4 : si8
%2 = p4.constant 2 : ui16
p4.call @myAction(%1, %2) : (si8, ui16) -> ()
```

Listing 4.4: Action call with constant arguments.

Although we have already introduced the P4's `in`, `inout`, and `out` parameter directions, we have yet to describe the precise semantics of P4's calling convention. P4 follows the *copy-in/copy-out* calling convention. We intentionally defer the introduction of the copy-in/copy-out convention until later in this chapter to describe it in the broader context of its non-trivial implications for the P4 dialect.

4.3 Control Block

The P4 control block is translated into the `p4.control` operation. Similarly to the `p4.action` Op, the `p4.control` Op declares a symbol and contains a single attached region. This region must include a single `p4.apply` operation, representing P4's apply method. As with action Ops, the apply operation must explicitly include the `p4.return` Op.

```
control MyControl(in int<8> arg0, in bit<16> arg1) {
    apply {}
}
```

```
p4.control @MyControl {

    p4.apply(%arg0: si8, %arg1: ui16) {
        p4.return
    }

}
```

Listing 4.5: Control block definition.

Importantly, as can be seen in the preceding snippet, P4 dialect lexically moves the control block's arguments, making them direct parameters of the apply method, instead of the control block.

This decision allows to define the `p4.apply` Op with the *isolated-from-above* interface. The isolated-from-above interface is MLIR's built-in interface which indicates that the `p4.apply` Op cannot reference SSA values defined outside of its region. As we will see later, we use the isolated-from-above interface for `p4.action` Ops as well.

Apart from verifying that the operation does not truly reference any external SSA values, MLIR treats all isolated-from-above operations as independent IR units with no shared data structures. This allows transformations of each isolated-from-above Op *in parallel* without any need for data structure locking, effectively speeding up the compilation[6].

On the other hand, this change in structure requires a corresponding adjustment in the structure of all constructs defined within a control block. Consider the following P4 snippet.

```
control MyControl(in int<8> arg0) {

    action myAction(in int<8> arg1) {
        int<8> localVar = arg0 + arg1;
    }

    apply {
        myAction(5);
    }

}
```

Listing 4.6: P4 action referencing a control block argument.

Here, `myAction` relies on the fact that P4 actions are **not** isolated-from-above and directly references the `arg0` parameter of the enclosing control block. However, converting control block parameters into the apply method parameters results in the `arg0` parameter declaration no longer being visible inside the `myAction` action Op.

To address this issue, we must first synthesize new parameters for all action operations referencing the control block's parameters. Secondly, we must pass in the appropriate arguments at action call sites.

```
p4.control @MyControl {

    // Action with new ‘arg0‘ synthesized parameter
    p4.action @myAction(%arg0: si8, %arg1: si8) {
        %localVar = p4.add(%arg0, %arg1) : (si8, si8) -> si8
    }

    p4.apply(%arg0: si8, %arg1: ui16) {
        %five = p4.constant 5 : si5
        // Action call with new synthesized argument
        p4.call @myAction(%arg0, %five) : (si8, si8) -> si8
    }

}
```

Listing 4.7: P4 dialect action with synthesized parameter.

In addition to the previously discussed parallelization benefits, another advantage of this structural change is the consolidation of parameter handling throughout the entire dialect. Specifically, each P4 dialect pass now only needs to handle a single source of action inputs, simplifying the implementation of compiler passes. This benefit becomes even more evident when considering the implications of the P4 local elements, which we discuss later.

While for demonstration purposes we have focused solely on actions and the apply method, as we will see later, a similar synthesis of parameters and arguments is required for tables as well.

4.4 Control Flow

Rationale

As discussed in Chapter 3, MLIR offers the flexibility to model control flow within a region either using structured operations, such as those provided by the built-in `scf` dialect, or through unstructured CFG with basic blocks and arbitrary transitions between them, which are offered by the built-in `cf` dialect.

The semantics of P4's control flow constructs do not differ from those of other general-purpose languages, we therefore choose to not introduce special P4 dialect control flow operations. Instead, we leverage the `scf` and `cf` dialects.

P4 dialect operations can be combined with both structured and unstructured control flow operations. In practical scenarios, structured control flow is initially used for high-level analysis, with a gradual transition to unstructured basic blocks once the structure is no longer necessary. The exact combination of structured and unstructured models should be guided by the requirements of implemented analyses, transformations, and optimizations.

Arguably, the translation of P4 into unstructured basic blocks is more intriguing, and as such, it is the only one detailed in this text.

Unstructured Control Flow

```
action myAction(int<8> arg) {
    int<8> localVar;
    if (arg > 0) {
        localVar = 4;
    } else {
        localVar = 2;
    }
}
```

```
p4.action @myAction(%arg : si8) {
^bb0:
    %zero = p4.constant 0 : si8
    %flag = p4.cmp(%arg, %zero) gt : (si8, si8) -> i1
    cf.cond_br %flag, ^bb1, ^bb2

^bb1:
    %four = p4.constant 4 : si8
    cf.br ^bb3

^bb2:
    %two = p4.constant 2 : si8
    cf.br ^bb3

^bb3:
    p4.return
}
```

Listing 4.8: Action with an if-statement.

As can be seen in the above snippet, to model an if-statement, the P4 dialect uses operations from the built-in `cf` dialect. In particular, it uses the `cf.cond_br`

Op to model conditional jumps and the `cf.br` Op for unconditional jumps.

The P4 if-statement is modeled using four basic blocks: an entry block with a conditional jump (`^bb0`), along with if (`^bb1`) and else (`^bb2`) blocks that contain unconditional jumps into the last merge block (`^bb3`).

In our example, P4 uses the result of the `arg > 0` comparison as the condition value. For such comparison, the P4 dialect provides the `p4.cmp` operation. The value produced by the `p4.cmp` Op is then passed into the `cf.cond_br` Op which performs the transition into another basic block.

It is worth noting that this example demonstrates the interaction between MLIR's built-in dialect and our custom P4 dialect for the first time. Since both the `p4.cmp` and `cf.cond_br` operations operate on values of the same `i1` type, the interaction maintains clear semantics. However, if the `p4.cmp` operation produced a value of a different type, the P4 dialect would need to provide a special cast operation to perform the conversion between the different type and the `i1` type.

4.5 SSA Form Conversion

MLIR requires that its IR follows the SSA form. However, in general, a P4 program is not in SSA form because P4 allows reassignment of variables, which contradicts the SSA's requirement that each variable is assigned only once.

As will be discussed later, the P4 dialect offers operations that initially allocate all values into external memory instead of SSA registers. This approach allows for a straightforward conversion of P4 values into MLIR values. However, allocating all values into memory instead of SSA registers effectively eliminates all benefits acquired by using the SSA form.

Although a subsequent compiler pass could convert the memory allocated values into SSA register values[8], we instead choose to create an ad-hoc SSA form over the P4 AST. Using this ad-hoc SSA form, we can translate the P4 AST into the P4 dialect with values already allocated into SSA registers.

Consider the following snippet with the reassignment of the `A` variable and its subsequent use after the if-statement.

```
action myAction(in int<8> arg) {
    int<8> A;
    if (arg > 0) {
        A = 4;
    } else {
        A = 2;
    }
    int<8> result = arg + A // <-- ‘A‘ use
}
```

Listing 4.9: P4 action with a reassignment of the A variable.

[8]This approach is demonstrated by LLVM and its *MemToReg* pass[4].

While operating on the input P4 AST, the creation of the ad-hoc SSA form follows the standard SSA conversion algorithm described in Appendix A. The ad-hoc SSA form for the preceding P4 snippet can be visualized by adding an integral subscript to variables that were assigned multiple times and placing ϕ-nodes where the control flow merges multiple versions of the same variable.

```
action myAction(in int<8> arg) {
    int<8> A1 = undefined;
    if (arg > 0) {
        A2 = 4;
    } else {
        A3 = 2;
    }
    A4 = phi(A2, A3);
    int<8> result = arg + A4
}
```

Listing 4.10: Ad-hoc SSA form of the Listing 4.9.

This ad-hoc SSA overlay over our original P4 snippet can be directly converted into P4 dialect, as shown in the following snippet. Although we have discussed the process of translating if-statements in the previous chapter, in this case, we need to use a basic block parameter for the merge block (`^bb3`). This basic block parameter corresponds to the only ϕ-node in the preceding snippet. As discussed in Section 3.6, basic block parameters are an equivalent representation of ϕ-nodes.

```
p4.action @myAction(%arg : si8) {
    %A_1 = p4.undefined : si8
    %flag = p4.cmp(%arg, %A_1) gt : (si8, si8) -> i1
    cf.cond_br %flag, ^bb1, ^bb2

^bb1:
    %A_2 = p4.constant 4 : si8
    cf.br ^bb3(%A_2 : si8)

^bb2:
    %A_3 = p4.constant 2 : si8
    cf.br ^bb3(%A_3 : si8)

^bb3(%A_4: si8):
    %res = p4.add(%arg, %A_4) : (si8, si8) -> si8
    p4.return
}
```

Listing 4.11: P4 dialect action with basic block parameters.

Additionally, P4 allows declaring a variable without an initial value, in which case the value of the variable is undefined. For this purpose, the P4 dialect introduces the `p4.undefined` operation, which produces an SSA value of the given type with an undefined value. The usage of the `p4.undefined` operation can be seen in the preceding snippet as well.

4.6 Translating P4's Calling Convention

The P4 language follows the *copy-in/copy-out* calling convention, which has non-trivial implications for the P4 dialect. Before we cover these implications, it is important to describe the copy-in/copy-out semantics.

Copy-in/Copy-out Semantics

Each P4 parameter is assigned a type and can either be *directionless* or one of three potential directions: `in`, `out`, or `inout`. These directions are specified before the type in a parameter declaration; if no direction is explicitly stated, the parameter is treated as directionless.

The `in` and directionless parameters cannot be written, only read. On the other hand, both `out` and `inout` parameters can be read and written. Contrary to the `inout` parameter, the initial value of the `out` parameter is *undefined*.

At the call site, if an argument is bound to an an `out` or `inout` parameter, it must be an *l-value*, meaning it cannot be a temporary value, rather it must be a reference to a memory storage.

```
action myAction(inout int<8> arg) { /* ... */ }

int<8> val = 5;
myAction(val); // <-- allowed: val is an l-value
myAction(2 + 3) // <-- error: '2 + 3' produces a temporary value
```

Listing 4.12: An inout parameter must be bound to an l-value.

During an invocation, arguments are passed in according to the following steps.

1. Arguments are evaluated from left to right as they appear in the function call expression.
2. Before evaluating the next argument, the value (can be an l-value) of the argument is saved into a temporary.
3. The function is invoked with the temporaries as arguments. We are guaranteed that the temporaries that are passed as arguments are never aliased to each other, so this synthesized function call can be implemented using call-by-reference.
4. On function return, the temporaries that correspond to `out` or `inout` arguments are copied in order from left to right into the l-values saved in step 2.

— *P4$_{16}$ specification* [15]

As can be seen in the following snippet, the listed steps can be illustrated by a simple P4 action call with a semantically-equivalent pseudocode. Within the pseudocode, we assume a call-by-reference calling convention.

```
action foo(in int<8> x, out int<8> y, inout int<8> z);

int<8> a = 1;
int<8> b = 20;
foo(a, a, b);
```

```
a = 1
b = 20

// step 1 and 2
tmp1 = a
tmp2 = a
tmp3 = b

// step 3
foo(tmp1, tmp2, tmp3)

// step 4
a = tmp2
b = tmp3
```

Listing 4.13: Example of a call using copy-in/copy-out semantics.

The primary reason for using the copy-in/copy-out semantics instead of the more common call-by-reference semantics is to limit the possible side-effects of calling P4 *externs*. The P4 language does not impose restrictions on the power of externs. Therefore, hypothetically, if call-by-reference was used, a P4 extern could store a reference to its arguments, which could be manipulated asynchronously from the control plane or during the invocation of different externs. This would severely limit a compiler's ability to reason about a P4 program, restricting analysis and optimizations[15].

By using the copy-in/copy-out calling convention, we ensure that the extern receives only a *copy* of its argument. Consequently, it becomes impossible for a P4 extern to store a reference to a P4 variable. As a result, externs are only able to modify their arguments during their invocation, enabling the P4 compiler to reason about values within a P4 program.

In Parameters

With the P4's copy-in/copy-out calling convention clarified, we can discuss its implications for conversion into MLIR, starting with the conversion of `in` parameters.

All MLIR values adhere to the SSA form and are therefore immutable (they can only be assigned once). These semantics directly correspond with the semantics of `in` parameters, making their conversion into the P4 dialect straightforward. We simply pass the corresponding SSA value into the `p4.call` operation.

```
action myAction(in int<8> arg) { /* ... */ }

myAction(10);
```

```
p4.action @myAction(%arg: si8) { /* ... */ }

%10 = p4.constant 10 : si8
p4.call @myAction(%10) : (si8) -> ()
```

Listing 4.14: Conversion of an in parameter.

It is worth noting that we effectively skipped the second step of the copy-in/copy-out calling convention, which requires copying the argument into a temporary variable before the invocation. The primary purpose of this step is to prevent overwriting of the variable during the evaluation of subsequent arguments. However, as mentioned, all MLIR values are immutable, and therefore the value cannot change before the invocation.

Reference Type

The conversion of `out`, `inout`, and directionless P4 parameters requires introduction of a new type.

The P4 dialect provides a reference type parameterized with type `T`, denoted as `!p4.ref<T>` (e.g., `!p4.ref<si32>`). Unlike other P4 dialect types, `!p4.ref<T>` does not correspond to any P4 type. The `!p4.ref<T>` type represents a reference to a mutable memory cell, capable of storing values of type `T`.

To allocate a mutable memory cell of type `T`, the P4 dialect defines the `p4.alloc` operation, which produces an SSA value of type `!p4.ref<T>`.

```
%ref = p4.alloc : !p4.ref<ui16>
```

Listing 4.15: Allocation of *ui16* memory cell.

Initially, the value stored inside the allocated memory cell is undefined. To store and load a value into the cell, the P4 dialect uses the `p4.store` and `p4.load` operations. These operations accept a reference to the memory cell, and in the case of `p4.store`, an SSA value that should be stored into the cell.

```
// Allocate and initialize
%ref = p4.alloc : !p4.ref<ui16>
%42 = p4.constant 42 : ui16
p4.store(%ref, %42) : (!p4.ref<ui16>, ui16) -> ()

// Load
%value = p4.load(%ref) : !p4.ref<ui16> -> ui16
```

Listing 4.16: Example store and load operations manipulating a memory cell.

The memory cell allocated via the `p4.alloc` operation remains allocated until the control flow leaves the enclosing region. Such semantics correspond to the stack allocation known from memory models of general-purpose languages.

To align with the heavily constrained nature of the P4 language and prevent introducing unnecessary expressiveness, the `!p4.ref<T>` type is subject to significant constraints regarding its usage.

Firstly, the reference type can be used as an operand only for a highly limited number of P4 dialect operations. Secondly, the reference type is not permitted to contain nested references, such as `!p4.ref<!p4.ref<si32>>`. Further, the reference type cannot be stored within compound types such as structures and headers. And finally, `!p4.ref<T>` is not allowed to be returned from externs nor stored within a control plane.

InOut and ***Out*** Parameters

Having introduced the `!p4.ref<T>` type, we can describe the conversion of P4's `out` and `inout` parameters. To translate both `out` and `inout` parameters, we must ensure that writes of P4 variables that occur within an action can be observed after the action invocation as well. As a result, instead of using the original value type `T`, we convert the type of the parameter to be `!p4.ref<T>`.

```
action foo(out int<8> arg1, inout int<8> arg2) {
    /* ... */
}
```

```
p4.action @foo(%arg1: !p4.ref<si8>, %arg2: !p4.ref<si8>) {
    /* ... */
}
```

Listing 4.17: During out and inout parameter conversions, value types must be converted to reference types.

Subsequently, as can be seen in the following snippet, the operations operating on the reference parameters must use the `p4.store` and `p4.load` operations to manipulate the referenced memory.

```
action foo(out int<8> arg1, inout int<8> arg2) {
    arg1 = 2;
    arg2 = arg2 + 1;
}
```

```
p4.action @foo(%arg1: !p4.ref<si8>, %arg2: !p4.ref<si8>) {
    // arg1 = 2;
    %2 = p4.constant 2 : si8
    p4.store(%arg1, %2) : (!p4.ref<si8>, si8) -> ()

    // arg2 = arg2 + 2;
    %tmp1 = p4.load(%arg2) : !p4.ref<si8> -> si8
    %tmp2 = p4.add(%tmp1, %2) : (si8, si8) -> si8
    p4.store(%arg2, %tmp2) : (!p4.ref<si8>, si8) -> ()
}
```

Listing 4.18: Reference parameters must manipulate their memory cells with store and load operations.

At the call site, contrary to the arguments bound to the `in` parameters, the `out` and `inout` arguments require additional handling to adhere to the copy-in/copy-out calling convention.

The individual steps are outlined in the following snippet. To pass an `out` or `inout` argument into a call operation, the copy-in/copy-out mechanism is followed precisely, with a slight variation outlined for the `out` parameter.

```
action foo(inout int<8> arg) { /* ... */ }

int<8> var = 42;
foo(var);
```

```
p4.action @foo(%arg: !p4.ref<si8>) { /* ... */ }

// Allocate and initialize the argument.
%var = p4.alloc : !p4.ref<si8>
%42 = p4.constant 42 : si8
p4.store(%var, %42) : (!p4.ref<si8>, si8) -> ()

// step 1 and 2:
// Alloc and init a temporary storage with the argument's value.
// Initialization is not necessary for an ‘out‘ argument.
%tmp = p4.alloc : !p4.ref<si8>
p4.store(%tmp, %42) : (!p4.ref<si8>, si8) -> ()

// step 3:
// Call the action with the temporary storage as an argument.
p4.call @foo(%tmp) : (!p4.ref<si8>) -> ()

// step 4:
// Copy the value from the temporary storage
// back to the original argument.
%res = p4.load(%tmp) : !p4.ref<si8> -> si8
p4.store(%var, %res) : (!p4.ref<si8>, si8) -> ()
```

Listing 4.19: Calling an action with a single inout parameter.

While initially, the calling convention is followed precisely, subsequent compiler passes can detect unnecessary operations for the particular call site and eliminate them.

Directionless Parameters

The algorithm for the conversion of directionless parameters depends on the type of the parameter.

If the type of the parameter is an extern object, the corresponding P4 dialect type must be encapsulated into the `!p4.ref<T>` type. This scenario will be described in more detail in the section dedicated to the conversion of P4 externs.

In any other case, the directionless parameters are converted as SSA values with the corresponding type, with no special handling at the call site. In other words, they are handled as if they were `in` parameters.

```
action myAction(int<8> arg) { /* ... */ }
```

```
p4.action @myAction(%arg: si8) { /* ... */ }
```

Listing 4.20: Conversion of a directionless parameter

4.7 Tables

Definition of a P4 table is composed of a set of key-value pairs. As we discussed in Chapter 2, the P4 language specifies several keys along with their corresponding type of values and semantics. These keys include: `key`, `actions`, `default_action`, and `size`.

In the P4 dialect, a table declaration is modeled using the `p4.table` operation, which declares a symbol and contains a single attached region. Following the structure of the P4 language, the `p4.table` Op must be used within a `p4.control`'s region.

```
control MyControl(in int<8> arg0, in bit<24> arg1)

    action myAction1() {}
    action myAction2() {}

    // Table definition
    table myTable {
        key = { arg0: exact; arg1: lpm; }
        actions = { myAction1; myAction2; }
        size = 1024 + 500 + 500;
        default_action = myAction1;
    }

    apply {}

}
```

```
p4.control @MyControl {

    p4.action @myAction1() { /* ... */ }
    p4.action @myAction2() { /* ... */ }

    // Table operation
    p4.table @myTable( /* ... */ ) {
        /* ... */
    }

    p4.apply(%arg0: si8, %arg1: ui24) {
        p4.return
    }

}
```

Listing 4.21: Table definition.

In the above snippet, while the P4 table is fully defined, the `p4.table`'s region remains empty. We now go over each of the table's properties and briefly discuss their conversion into the P4 dialect.

Key Property

For the `key` table property, the P4 dialect provides the `p4.table_keys_list` operation, acting as a container for the individual key components represented by the `p4.table_key` operations.

```
key = { arg0: exact; arg1: lpm; }
```

```
p4.table_keys_list {
    p4.table_key @exact {
      p4.init(%arg0) : (si8)
    }
    p4.table_key @lpm {
      p4.init(%arg1) : (ui24)
    }
}
```

Listing 4.22: Table key property conversion.

The `p4.table_key` Op contains a symbol referencing the type of the match algorithm (e.g., `@exact` or `@lpm`) and a single attached region to define the specific SSA value used for the key component. This region must be terminated by passing the SSA value into the new `p4.init` operation.

The `p4.init` Op is not exclusive to the `p4.table_key` operation; it is used in various contexts within the P4 dialect to generally *"mark"* an SSA value at the end of a region. We will meet the `p4.init` Op at several other parts of this text.

Size Property

In the P4 dialect, the `size` property is represented with the `p4.table_size` operation with a single region. Similarly to the `key` property, this region must be terminated with the `p4.init` operation. Following the P4's constraints, the `p4.init` within the `p4.table_size`'s region must receive a compile-time known integral value.

```
size = 1024 + 500 + 500;
```

```
p4.table_size {
    %1 = p4.constant 1024 : si64
    %2 = p4.constant 500 : si64
    %3 = p4.add(%1, %2) : (si64, si64) -> si64
    %4 = p4.add(%3, %2) : (si64, si64) -> si64
    p4.init(%4) : (si64)
}
```

Listing 4.23: Table size property conversion.

Action Properties

To represent P4's `actions` and `default_action` table properties, the P4 dialect provides the `p4.table_actions_list` and `p4.table_default_action` operations.

```
actions = { myAction1; myAction2; }
default_action = myAction1;
```

```
p4.table_actions_list {
    p4.table_action @myAction1
    p4.table_action @myAction2
}
p4.table_default_action {
    p4.table_action @myAction1
}
```

Listing 4.24: Conversion of the actions and default_action table properties.

To define the specific actions within the regions of these operations, the P4 dialect uses the `p4.table_action` Op with a symbol reference.

Table Invocation

To invoke a table, the P4 dialect uses the `p4.call_table` operation, which references the symbol declared by the `p4.table` Op. Contrary to the `p4.call` operation (which we used to invoke actions), the `p4.call_table` can be used *only* from within the `p4.apply`'s region.

```
control MyControl(in int<8> arg0, in bit<16> arg1) {

    table myTable { /* ... */ }

    apply {
        myTable.apply();
    }
}
```

```
p4.control @MyControl {

    p4.table @myTable(%arg0: si8, %arg1: ui16) { /* ... */ }

    p4.apply(%arg0: si8, %arg1: ui16) {
        p4.call_table @myTable(%arg0, %arg1)
        p4.return
    }
}
```

Listing 4.25: Conversion of a table invocation.

In the above snippet, the primary distinction is that the P4 dialect's tables declare their parameters. These parameters are declared as a part of the `p4.table` operation. As a result, while the P4's `.apply()` method call does not accept any arguments, the corresponding `p4.call_table` is provided with two SSA operands.

Similarly to the `p4.action` operation, the `p4.table` Op is defined as isolated-from-above. Consequently, if the table definition uses any of the control block's parameters, the `p4.table` operation must use newly synthesized parameters.

As already discussed regarding the `p4.action` Op, the isolated-from-above interface allows parallel compilation and a simplified implementation of compiler passes, which now have only a single source of input values.

Note:

As P4 dialect types become more complex, their textual representation becomes more verbose, even when using the pretty-printed form. Including all input and result types of every MLIR operation in a snippet can be impractical. Therefore, in the following text, if a type is easily deducible, it is omitted for brevity.

To conclude the conversion of tables, we provide the complete example that we used throughout this section, including the `p4.table`'s region and invocation.

```
control MyControl(in int<8> arg0, in bit<16> arg1) {

    table myTable {
        key = { arg0: exact; arg1: lpm; }
        actions = { myAction1; myAction2; }
        default_action = myAction1;
        size = 1024 + 500 + 500;
    }

    apply { myTable.apply(); }
}
```

```
p4.control @MyControl {

    p4.table @myTable(%arg0: si8, %arg1: ui16) {
        p4.table_keys_list {
            p4.table_key @exact {
              p4.init(%arg0) : (si8)
            }
            p4.table_key @lpm {
              p4.init(%arg1) : (ui24)
            }
        }
        p4.table_actions_list {
            p4.table_action @myAction1
            p4.table_action @myAction2
        }
        p4.table_default_action {
            p4.table_action @myAction1
        }
        p4.table_size {
            %1 = p4.constant 1024 : si64
            %2 = p4.constant 500 : si64
            %3 = p4.add(%1, %2) : (si64, si64) -> si64
            %4 = p4.add(%3, %2) : (si64, si64) -> si64
            p4.init(%4) : (si64)
        }
    }

    p4.apply(%arg0: si8, %arg1: ui16) {
        p4.call_table @myTable(%arg0, %arg1)
        p4.return
    }
}
```

Listing 4.26: Conversion of a table and its invocation.

4.8 Structures and Headers

In the P4 dialect, similarly to the P4 language, before a header and struct types can be referenced, they must be declared first. For this purpose, the P4 dialect introduces the `p4.struct` and `p4.header` operations, which declare a symbol and a single region. This region contains member declarations, which declare the symbol and type of each member.

```
struct MyStruct {
    MyHeader hdr;
    bit<16> f1;
}

header MyHeader {
    int<8> f1;
    int<8> f2;
}
```

```
p4.header @MyHeader {
    p4.member_decl @f1 : si8
    p4.member_decl @f2 : si8
    p4.valid_bit_decl @__valid : i1
}

p4.struct @MyStruct {
    p4.member_decl @hdr : !p4.header<"MyHeader">
    p4.member_decl @f1 : ui16
}
```

Listing 4.27: Definition of a new struct and header types.

Once the types are declared using the `p4.header` and `p4.struct` operations, type references of the declared symbols become valid. For example, if a `p4.struct` Op declares the `@MyStruct` symbol, a type reference `!p4.struct<"MyStruct">` becomes valid. Similarly, after a p4.header `@MyHeader` operation, we can use a type reference of the form `!p4.header<"MyHeader">`.

Additionally, contrary to the P4 language, the `p4.header`'s region must declare an extra `@__valid` member using the `p4.valid_bit_decl` operation. The valid bit member is required to be of type `i1` and lacks a direct counterpart in the original P4 program.

The `@__valid` member models the hidden validity bit of a P4 header, which is manipulated through the `.setValid()`, `.setInvalid()`, and `.isValid()` method calls. Making the validity bit a regular header member aims to reduce the number of necessary Ops and simplify subsequent compiler passes, which can treat the validity bit manipulations as ordinary member writes and reads.

Initialization

The P4 language allows the creation of objects of composite types (i.e., headers and structs) either as *uninitialized* or *initialized*. In the P4 dialect, to create uninitialized composite objects, we reuse the `p4.uninitialized` Op, which is already used to create uninitialized variables of base types. On the other hand, to initialize all members of a composite object, the P4 language provides a succinct tuple syntax, which is modeled within the P4 dialect with the `p4.tuple` Op. The following snippet demonstrates both types of composite object creation.

```
// Uninitialized 'MyStruct' value.
MyStruct myStruct;

// Inititialized 'MyHeader' value.
MyHeader myHdr = {4, 2};
```

```
// Uninitialized 'MyStruct' value.
%myStruct = p4.uninitialized : !p4.struct<"MyStruct">

// Inititialized 'MyHeader' value.
%4 = p4.constant 4 : si8
%2 = p4.constant 2 : si8
%t = p4.constant true : bool
%myHdr = p4.tuple(%4, %2, %t) : !p4.header<"MyHeader">
```

Listing 4.28: Example of definition of struct and header values.

According to the P4 specification, when a header is initialized using the tuple syntax, its validity bit is automatically set to `true`[15]. This behavior is reflected within the above MLIR snippet by explicitly initializing the header's last `@__valid` field to `true` as well.

Member Access and Write

Besides creating SSA struct and header values, the P4 dialect allows to allocate values of composite types into memory as well. The type of allocation determines which P4 dialect operations are used to access and modify their members.

To retrieve a member of an SSA struct or header value, the P4 dialect introduces the `p4.get_member` operation. Given a composite value and a symbol of its member, the `p4.get_member` Op copies the member value into an SSA register.

```
MyStruct myStruct;
bit<16> var = myStruct.f1;
```

```
%myStruct = p4.uninitialized : !p4.struct<"MyStruct">
%var = p4.get_member(%myStruct) @f1 : ui16
```

Listing 4.29: Member access of an SSA allocated struct value.

To access a member of a memory allocated composite value, the P4 dialect uses the `p4.get_member_ref` operation, which accepts a member symbol and a `!p4.ref<T>` reference of a composite value. The `p4.get_member_ref` Op produces a `!p4.ref<T>` reference of the member. This member reference can then be loaded into an SSA register using the previously introduced `p4.load` Op.

```
MyStruct myStruct;
bit<16> var = myStruct.f1;
```

```
%myStruct = p4.alloc : !p4.ref<!p4.struct<"MyStruct">>
%ref = p4.get_member_ref(%myStruct) @f1 : !p4.ref<ui16>
%var = p4.load(%ref) : (!p4.ref<ui16>) -> ui16
```

Listing 4.30: Member access of a memory allocated struct.

Similarly, to modify a member of a memory allocated composite value, the P4 dialect uses the `p4.store` Op.

```
MyHeader myHdr;
myHdr.f1 = 50;
```

```
%myHdr = p4.alloc : !p4.ref<!p4.header<"MyHeader">>
%ref = p4.get_member_ref(%myHdr) @f1 : !p4.ref<si8>
%50 = p4.constant 50 : si8
p4.store(%ref, %50) : (!p4.ref<si8>, si8)
```

Listing 4.31: Member write of a memory allocated header.

It is worth noting that the P4 dialect does not currently allow modification of members of SSA register allocated values.

4.9 Local Elements

As covered in Chapter 2, the P4 language allows to initialize a set of variables before the apply method invocation. These variables, called *local elements*, are lexically defined within the control block, among tables and actions.

Local elements can then be referenced from all other constructs which are lexically below and within the control block. For example, in our SIPR protocol implementation, we used the `shouldExit` local element, which was initialized to `false` and set to `true` inside the `discard` action. The value of the `shouldExit` local element was then used to control an early exit from the apply method.

Considering we have decided to define the P4 dialect Ops as isolated-from-above, the local elements are no longer visible from apply methods, actions, nor tables. They will therefore need a special handling during translation.

Similarly to the action arguments and tables, we choose to synthesize new parameters which are used to pass the local elements into the isolated-from-above constructs as arguments. Consider the following example, where the `myAction` action modifies a local element of the enclosing control block. As a result, the corresponding `p4.action` Op must contain an additional parameter.

```
control MyControl() {
    // Local element definition
    bit<16> localElem = 42;

    action myAction() {
        localElem = 10;
    }

    apply { myAction(); }
}
```

```
p4.control @MyControl {

    p4.action @myAction(%localElem: !p4.ref<ui16>) {
        %10 = p4.constant 10 : ui16
        p4.store(%localElem, %10) : (!p4.ref<ui16>, ui16)
        p4.return
    }

    p4.apply() { /* ... */ }
}
```

Listing 4.32: Conversion of an action which references a local element.

Crucially, the above MLIR snippet does not model the local element initialization; i.e., there are no corresponding Ops above the `p4.action` operation. For the P4 dialect, we choose to move local elements initialization to the beginning of the `p4.apply`'s region.

This design effectively *eliminates the concept of local elements from the P4 dialect*, simplifying all subsequent compiler passes, which can treat local elements as ordinary local variables. Consider the following snippet, which fills the `p4.apply`'s region which we have omitted in the preceding snippet.

```
control MyControl() {
    bit<16> localElem = 42;

    action myAction() { /* ... */ }

    apply { myAction(); }
}
```

```
p4.control @MyControl {
    p4.action @myAction(%localElem: !p4.ref<ui16>) { /* ... */ }

    p4.apply() {
        // Local element initialization
        %42 = p4.constant 42 : ui16
        %localElem = p4.alloc : !p4.ref<ui16>
        p4.store(%localElem, %42) : (!p4.ref<ui16>, ui16)

        p4.call @myAction(%localElem) : (!p4.ref<ui16>)
        p4.return
    }
}
```

Listing 4.33: Conversion of local elements initialization.

Once the local element is initialized, its value is passed into the action in place of the newly synthesized arguments. This way the local element can be modified from within the isolated-from-above action.

4.10 Externs

P4 externs declare interfaces for functionalities which are backed by the used P4 architecture. The semantics of P4 externs are either documented by the P4 architecture itself (e.g., the `Counter` extern in our Simple Architecture), or, in case of the `packet_in` and `packet_out` externs, directly by the P4 specification. The P4 source code merely declares available externs, which provides a convenient way for a compiler to verify their use.

An extern declaration consists of its name, methods, and constructors. In the P4 dialect, the P4's structure of extern declarations is replicated by the `p4.extern_class`, `p4.extern`, and `p4.constructor` operations. To demonstrate the conversion of P4 externs, we consider our SA `Counter` extern, with a slightly altered functionality. This new `Counter` provides a constructor to initialize its state with an arbitrary `int<8>` value. It also offers the `add` methods which increment the `Counter` by one or by an arbitrary `int<8>` value. Finally, to enable querying of the counter's state from within the P4 program, we add the `get` method which returns the current counter state.

```
extern Counter {
    Counter(int<8> init);
    void add(int<8> value);
    void add();
    int<8> get();
}
```

```
p4.extern_class @Counter {
    p4.constructor @Counter_1(si8)
    p4.extern @add_1(si8)
    p4.extern @add_0()
    p4.extern @get_1() -> si8
}
```

Listing 4.34: Conversion of the modified Counter extern.

Interestingly, the P4 dialect counterpart modifies the names of the `Counter`'s constructor and methods. This alteration is necessary due to the fact that P4 allows to overload methods and constructors, using the same name for multiple declarations inside a single extern. On the other hand, the MLIR's symbol system mandates a unique name for each of the symbols. As a result, to each MLIR symbol we simply add a number of its parameters as a suffix, which makes sure the names within the extern remain unique. This simple naming scheme is possible due to the fact that the P4 specification allows overloading of methods and constructors based solely on the *number* of input parameters.

Instantiation

P4 allows instantiation of extern object variables either in a global namespace or as a local element.

```
// Global extern instantiation
Control(10) globalCounter;

control MyControl() {
    // Local element instantiation
    Counter(0) counter;

    apply { /* ... */ }
}
```

Listing 4.35: Instantiation of a Counter object as a local element and a global.

To define a global object, the P4 dialect introduces the `p4.global` operation.

```
p4.global @globalCounter : !p4.extern_class<"Counter"> {
    %1 = p4.constant 10 : si8
    p4.init @Counter_1 !p4.extern_class<"Counter"> (%1) : (si8)
}
```

Listing 4.36: Global Counter extern object definition.

The `p4.global` operation declares a symbol and a data type of the object. Additionally, the `p4.global` Op contains a single attached region which contains Ops instantiating the object. This region must be terminated by the `p4.init` Op, which specifies the symbol of the used constructor and its arguments.

In case of the object instantiation within a control block, we can no longer perform the same structural transformation which we did for the local elements. For the local elements, we could have moved their initialization Ops to the beginning of the apply method, where they are evaluated during *every* apply method invocation (for each new packet). On the other hand, the P4 extern objects are stateful, meaning its initialization happens only *once*, before the first packet arrives.

Therefore, in the P4 dialect, the objects within a control block stay lexically among tables and actions. For their initialization we reuse the `p4.member_decl` Op, which otherwise behaves exactly the same as the `p4.global` Op.

```
p4.control @MyControl() {
    // Counter extern object definition
    p4.member_decl @counter : !p4.extern_class<"Counter"> {
        %1 = p4.constant 0 : si8
        p4.init @Counter_1 !p4.extern_class<"Counter"> (%1)
    }

    p4.apply { /* ... */ }
}
```

Listing 4.37: Instantiation of a Counter object within a control block.

Calling Extern Methods

To invoke a method of an extern object, we introduce the `p4.call_method` into the P4 dialect. This operation expects a `!p4.ref<T>` reference to the extern object value, the method's symbol, and the method's arguments.

Since both types of extern object definitions declare a symbol and not an SSA value, to obtain a `!p4.ref<T>` reference, we must first use a materializing Op. The type of a materializing Op depends on the type of the object definition.

To materialize a global object value, we use the new `p4.addressof` Op, which simply accepts a symbol of the object instance and produces a reference to it.

```
p4.global @cnt : !p4.extern_class<"Counter"> { /* ... */ }

%ref = p4.addressof @cnt : !p4.ref<!p4.extern_class<"Counter">>
```

Listing 4.38: Global Counter object materialization.

This materialized reference can then be passed into the `p4.call_method` Op to invoke an extern method.

```
%res = p4.call_method %ref @get_0 ()
                : (!p4.ref<!p4.extern_class<"Counter">>) -> si8
```

Listing 4.39: Calling a method of a Counter instance.

The process of materializing an extern object defined within a control block is analogical to the materialization of a global extern object. The only difference is in the used Op, where instead of the `p4.addressof` Op we reuse the `p4.get_member_ref` Op.

```
%ref = p4.get_member_of @cnt
                : !p4.ref<!p4.extern_class<"Counter">>
```

Listing 4.40: Materialization of a Counter object defined within a control block.

4.11 Packet Externs

While most externs are provided by P4 architectures, the P4 language itself introduces abstractions over packet data through the `packet_in` and `packet_out` extern objects. These represent an incoming packet and outgoing packet buffer, respectively.

The semantics of `packet_in` and `packet_out` externs are fully defined by the P4 standard. As a result, the P4 dialect can provide a special set of elements which enable to better capture the semantics of these objects within the IR. Such elements can be utilized by compiler passes which are better implemented over a model which makes a distinction between packet externs and other, P4 architecture, externs.

These elements include new data types for each of the externs, `!packet_in` and `!packet_out`, along with two new operations, `p4.extract` and `p4.emit`, which serve to model the extract and emit methods, respectively.

The usage of these elements will be demonstrated in the following section.

4.12 Parser Block

Similarly to the control block, the P4 dialect conversion of the parser block mostly follows the general structure of its P4 counterpart. However, we acknowledge that the P4 dialect is not meant to be handwritten and therefore does not need to capture many aspects of P4 that exist primarily for the convenience of human programmers.

One change the P4 dialect makes in the structure of a P4 parser block is the addition of the apply method. A P4 parser begins its execution inside the `start` state. Additionally, similarly to the control block, a P4 parser allows the definition of local elements, which are lexically defined above all states, but are semantically initialized before each parser invocation.

For `p4.control`, we moved local elements to the beginning of the apply method and converted them into ordinary local variables. To perform the same transformation for the parser block, we need a starting block where we can insert the initialization operations. While one might think to insert them at the beginning of the `start` state, it is important to remember that parser states express a graph, and graphs can contain cycles. If the `start` state is a part of a cycle, we end up re-initializing local elements on each visit to the `start` state.

For this reason, we synthesize an apply method, which is invoked once for each packet. In addition to initializing local elements, it contains an unconditional transition into the `start` state. This design explicitly marks the initial state of the parse graph, instead of relying on a state with a special name.

```
parser MyParser(packet_in wire, out MyHeader pkt) {

    int<8> localElem = 10;

    state start { /* ... */ }
}
```

```
p4.parser @MyParser {
    p4.state @start(%wire: !p4.packet_in, %localElem: si8) {
        /* ... */
    }

    p4.apply(%wire: !p4.packet_in,
             %pkt: !p4.ref<!p4.header<"MyHeader">>) {
      %localElem = p4.constant 10 : si8
      p4.transition @start(%wire, %localElem) : ()
    }
}
```

Listing 4.41: Parser block definition.

It is important to note that all key design decisions introduced for the `p4.control` Op apply to the `p4.parser` Op as well: The parser block parameters are moved directly into the apply method, and both the apply method and parser states are defined as isolated-from-above. Consequently, this necessitates the introduction of parameters for the parser states, which are missing in their P4 counterparts.

4.13 Parser Transitions

P4 provides both unconditional and conditional state transitions, where every parser state must terminate with one of these transitions. Additionally, P4 defines implicit `accept` and `reject` states, signifying parsing success and failure, respectively.

In the P4 dialect, the unconditional parser state transition is represented using the `p4.transition` Op, along with special operations for transitioning into the `accept` and `reject` states.

```
// Transition into the ‘myState‘ state
transition myState;

// Transition into the implicit ‘accept‘ state
transition accept;

// Transition into the implicit ‘reject‘ state
transition reject;
```

```
// Transition into the ‘myState‘ state
p4.transition @myState(/* ... */)

// Transition into the implicit ‘accept‘ state
p4.parser_accept

// Transition into the implicit ‘reject‘ state
p4.parser_reject
```

Listing 4.42: Examples of unconditional parser state transitions.

The conditional transitions involve listing possible cases, where each case consists of a pattern and a target parser state. If the provided values successfully match one of these patterns, a transition into the indicated state occurs.

```
state start {
    bit<8> var1 = 4;
    int<16> var2 = 20;
    transition select (var1, var2) {
                (1..10, 3): myState1;
                (1, 2): myState2;
                _: reject; // <-- default case
    }
}
```

Listing 4.43: Example of a P4 conditional transition.

To support conditional transitions, the P4 dialect introduces multiple new operations. The following snippets gradually build the P4 dialect representation of the conditional transition from the preceding snippet.

The `p4.select_transition` operation serves as a container for possible cases. Each case is defined by the `p4.select_transition_case` Op. Additionally, the list of cases must include a single `p4.select_transition_default_case` operation, which contains operation indicating where to transition if none of the other cases match.

```
%var1 = p4.constant 4 : ui8
%var2 = p4.constant 20 : si16
p4.select_transition %var1, %var2 : (ui8, si16) {
        p4.select_transition_case {
            /* ... */
        }
        p4.select_transition_case {
            /* ... */
        }
        p4.select_transition_default_case {
            p4.parser_reject
        }
}
```

Listing 4.44: Conditional transition with the case bodies elided.

Continuing into a `p4.select_transition_case` operation, a single case comprises of a `p4.select_transition_pattern` and a transition Op. In this example, both patterns consist of two components. Therefore, `p4.select_transition_pattern` must mark two SSA values of type `!p4.set`, which represents a set of numbers. Such set is created using the `p4.range(%from, %to)` operation, producing a set of all numbers from the `[%from, %to]` interval.

```
// (1..10, 3): myState1;
p4.select_transition_pattern {
    %1 = p4.constant 1 : ui8
    %10 = p4.constant 10 : ui8
    %3 = p4.constant 3 : si16

    // (1..10)
    %set1 = p4.range(%1, %10) : (ui8, ui8) -> !p4.set<ui8>

    // 3
    %set2 = p4.range(%3, %3) : (si16, si16) -> !p4.set<si16>

    // (1..10, 3)
    p4.init(%set1, %set2) : (!p4.set<ui8>, !p4.set<si16>)
}
```

Listing 4.45: Definition of a single pattern.

Finally, with the pattern created, we can add the second part of each transition case — the transition Op. The complete case with the pattern and transition operation can be seen in the following snippet.

```
p4.select_transition_case {
    p4.select_transition_pattern {
        %1 = p4.constant 1 : ui8
        %10 = p4.constant 10 : ui8
        %3 = p4.constant 3 : si16
        %set1 = p4.range(%1, %10) : (ui8, ui8) -> !p4.set<ui8>
        %set2 = p4.range(%3, %3) : (si16, si16) -> !p4.set<si16>
        p4.init(%set1, %set2) : (!p4.set<ui8>, !p4.set<si16>)
    }
    p4.transition @myState1(/* ... */)
}
```

Listing 4.46: P4 dialect - Definition of a complete transition case

Chapter 5

Simple Architecture Dialect

The P4 language specification presents numerous opportunities to extend or further constrain its core features. These flexible aspects of the specification are intended to be precisely defined by the P4 architecture in use.

In addition to precisely defining the semantics of the core language features, a P4 architecture specifies how the packet processing pipeline looks: which blocks it contains and which of these blocks are programmable by P4. For example, the Simple Architecture (SA) defines three programmable blocks: the parser, match-action pipeline, and deparser. The SA also specifies the order in which these blocks are executed: The packet is initially processed by the parser and then continues through the match-action pipeline before being sent to the deparser.

A P4 architecture also specifies various exceptional control flows throughout the pipeline and how these exceptional control flows are triggered. For example, the SA specifies that transitioning into the reject state within the parser causes the packet to be immediately discarded. Also, assigning zero to the `bit<2> port` in the match-action pipeline parameter causes the packet to be discarded as well, skipping the deparser.

Moreover, a P4 architecture defines externs, which expose additional functionality that cannot be simply expressed by P4 and is expected to be backed by specialized hardware units. The SA offers the `Counter` extern, which provides an interface for a persistent counter that maintains its state across individual processed packets.

All of these properties of a P4 architecture offer various opportunities for a compiler. Such opportunities can include program analysis, as well as optimizations that leverage detailed knowledge of the pipeline's control flow. For example, if for a given program, the SA parser transitions into the `reject` state for all packets whose field `foo` is zero, then checking if `foo == 0` within the deparser will always result in `false` and can therefore be replaced by a constant Op.

Such program transformations are highly dependent on the specifics of the chosen P4 architecture and should thus be part of a P4 architecture dialect.

A P4 architecture dialect should primarily be viewed as a set of compiler passes which exploit additional information that is provided by the chosen P4 architecture. This perspective suggests the kind of IR elements that an architecture dialect should contain: IR elements that allow for modeling the program in a way that is natural for the required compiler passes.

Using this simple guide, we will demonstrate a concrete scenario for the Simple Architecture. We will create the SA dialect, which can co-exist with the P4 dialect and assist in implementing transformations utilizing the SA semantics.

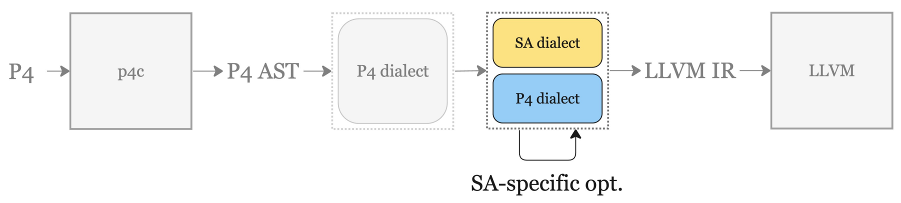

Figure 5.1: This chapter introduces the SA dialect elements and demonstrates its usage during optimizations.

5.1 Counter

The SA provides a single extern object named `Counter`. `Counter` provides an abstraction of a counter which persists its state across multiple packets. The P4 dialect declaration of the `Counter` declaration looks as follows.

```
p4.extern_class @Counter {
    p4.constructor @Counter_1(ui8)
    p4.extern @increment_0()
    p4.extern @add_1(ui8)
    p4.extern @get_0() -> ui8
}
```

Listing 5.1: P4 dialect declaration of the Counter extern

The `Counter`'s constructor initializes its state to the given value. Subsequently, an instance of the `Counter` extern can be incremented by one, incremented by a given value, or queried for its state. Additionally, at this point, we add an extra piece of specification for the `Counter`: The SA `Counter` defines that if its state overflows, it results in undefined behavior. In other words, our program transformations can assume that an instance of a `Counter` never overflows.

Upon examining the `Counter` declaration, it becomes evident that its interface was designed with human programmers in mind. The `increment` method appears redundant, as its semantics can be equivalently expressed by invoking the `add` method with a value of 1 as an argument.

Furthermore, the P4 dialect's approach of listing the entire `Counter` interface via declaration may become cumbersome during implementation of IR passes.

Although technically sound, this approach could lead to code being driven by numerous string comparisons in practice. For instance, to identify the `Counter` type, one would need to verify that a given `!p4.extern_class<"Counter">` type indeed contains the string `"Counter"`. Similarly, to recognize a call of the `add` method, one would need to inspect whether the `p4.call_method` Op references the string `"add_1"`. Such an approach could potentially complicate and introduce errors in the implementation of compiler passes.

To address these practical issues, we can introduce a new `Counter` type and a set of native `Counter` Ops within the SA dialect. This approach significantly enhances the convenience and type safety[9] of implementing SA compiler passes.

We name the new SA dialect type `!p4sa.counter`, and in the following snippet, we demonstrate the new SA dialect `Counter` Ops alongside their mapping to the P4 dialect Ops.

```
// %cnt = p4.addressof @counter
//                 : !p4.ref<!p4.extern_class<"Counter">>
%cnt = p4.addressof @counter : !p4.ref<!p4sa.counter>

// p4.call_method %cnt @increment_0()
%1 = p4.constant 1 : ui8
p4sa.counter_add(%cnt, %1)

// call_method %cnt @add_1(%5)
%5 = p4.constant 5 : ui8
p4sa.counter_add(%cnt, %5)

// %count = call_method %cnt @get_0() : ui8
%count = p4sa.counter_get(%cnt) : ui8
```

Listing 5.2: SA dialect conversion patterns for the Counter.

It is worth noting, that the SA dialect does not contain a corresponding Op for the `increment` method. As we have already explained, this method is redundant and can be replaced by calling the `p4sa.counter_add` Op with 1 as an argument.

Furthermore, while not detailed in this text, the SA dialect includes a compiler pass that traverses the IR, and replaces the P4 dialect `Counter` Ops with their native SA dialect counterparts. This pass also removes the P4 dialect `Counter` declaration, which is now redundant.

5.2 SA Dialect Optimizations

With the introduction of the `Counter` SA dialect Ops, we can dive into a concrete example of dialect optimizations. However, it is important to note that we have not specified the concrete hardware architecture on which our programs will run; we only know that our P4 programs are written against the Simple Architecture. Consequently, the scope of optimizations we can implement and offer as part of the SA dialect is limited. Generally, we should focus on transformations that benefit all possible hardware targets. Classic examples of such IR transformations include dead code elimination and constant propagation.

[9]Each MLIR Op corresponds to a C++ class.

To illustrate these transformations, we can consider the following P4 snippet programmed against the SA architecture. This snippet already utilizes the SA dialect `Counter` Ops.

```
p4.global @counter : !p4.ref<!p4sa.counter>

p4.control @MyPipe {
    p4.apply(%hdr: !p4.ref<!p4.header<"MyHeader">>,
             %port: !p4.ref<ui2>) {
        /* ... */

        // Store the output port into a header field
        %ref = p4.get_member_ref(%hdr) @bridged_port
        p4.store(%ref, %port)

        p4.return
    }
}

p4.control @MyDeparser {
    p4.apply(%hdr: !p4.ref<!p4.header<"MyHeader">>, /* ... */) {
        // Increment the counter by 1
        %cnt = p4.addressof @counter
        %1 = p4.constant 1
        p4sa.counter_add(%cnt, %1)

        // Check if counter > 0
        %0 = p4.constant 0
        %state = p4sa.counter_get(%cnt)
        %res1 = p4.cmp(%state, %0) gt : bool

        // Check if port > 0
        %ref = p4.get_member_ref(%hdr) @bridged_port
        %port = p4.load(%ref)
        %res2 = p4.cmp(%port, %0) gt : bool

        // If counter > 0 && port > 0, increment counter by 2
        %res3 = p4.and(%res1, %res2) : bool
        scf.if %res3 {
            %2 = p4.constant 2
            p4sa.counter_add(%cnt, %2)
        }

        p4.return
    }
}
```

Listing 5.3: Example of SA dialect optimizations (1).

The snippet demonstrates an excerpt from a complete program, showing implementation of the SA match-action pipeline and deparser. Firstly, at the very top, we have defined an instance of a `Counter` extern. Further, within the match-action pipeline, we store the output port into the `bridged_port` field of the packet. Subsequently, the packet progresses within the deparser. Inside the deparser, we first increment the counter by one, and if the state of the counter and the `bridged_port` header field are greater than zero, we additionally increment the counter by two.

An initial observation can be made about the comparison of the output port (stored within the `bridged_port` header field) within the deparser. As the Simple Architecture defines, if the match-action pipeline sets the output port to zero, the packet is discarded and does not continue into the deparser. Consequently, the value of the output port inside the deparser can never be zero, and the result of checking the port for positivity is always `true`. Therefore, the `p4.cmp` operation can be simply replaced by the `p4.constant` Op, and the Ops retrieving the `bridged_port` field eliminated. The result of these transformations are shown in the following snippet.

```
p4.global @counter : !p4.ref<!p4sa.counter>

p4.control @MyDeparser {
    p4.apply(%hdr: !p4.ref<!p4.header<"MyHeader">>, /* ... */) {
        // Increment the counter by 1
        %cnt = p4.addressof @counter
        %1 = p4.constant 1
        p4sa.counter_add(%cnt, %1)

        // Check if counter > 0
        %0 = p4.constant 0
        %state = p4sa.counter_get(%cnt)
        %res1 = p4.cmp(%state, %0) gt : bool

        // Check if port > 0 <-- optimized
        %res2 = p4.constant true : bool

        // If counter > 0 && port > 0, increment counter by 2
        %res3 = p4.and(%res1, %res2) : bool
        scf.if %res3 {
            %2 = p4.constant 2
            p4sa.counter_add(%cnt, %2)
        }

        p4.return
    }
}
```

Listing 5.4: Example of SA dialect optimizations (2).

The second observation uses the SA dialect's understanding of the `Counter` extern's semantics. In our snippet, we increment the counter extern by one and subsequently check its state for positivity. Given that the counter tracks its state as an unsigned integer, and considering our definition in this chapter that overflow results in undefined behavior, we can deduce that such a comparison must always be `true`. Therefore, similarly to the previous case, we can replace this comparison with a constant.

```
p4.global @counter : !p4.ref<!p4sa.counter>

p4.control @MyDeparser {
    p4.apply(%hdr: !p4.ref<!p4.header<"MyHeader">>, /* ... */) {
        // Increment the counter by 1
        %cnt = p4.addressof @counter
        %1 = p4.constant 1
        p4sa.counter_add(%cnt, %1)

        // Check if counter > 0 <-- optimized
        %res1 = p4.constant true : bool

        // Check if port > 0 <-- optimized
        %res2 = p4.constant true : bool

        // If counter > 0 && port > 0, increment counter by 2
        %res3 = p4.and(%res1, %res2) : bool
        scf.if %res3 {
            %2 = p4.constant 2
            p4sa.counter_add(%cnt, %2)
        }

        p4.return
    }
}
```

Listing 5.5: Example of SA dialect optimizations (3).

Further, the `p4.and` Op now receives two constant `true` values. Such operation can be folded into a `true` constant, resulting in converting the conditional counter increment to an unconditional one.

```
p4.global @counter : !p4.ref<!p4sa.counter>

p4.control @MyDeparser {
    p4.apply(%hdr: !p4.ref<!p4.header<"MyHeader">>, /* ... */) {
        // Increment the counter by 1
        %cnt = p4.addressof @counter
        %1 = p4.constant 1
        p4sa.counter_add(%cnt, %1)

        // Increment the counter by 2
        %2 = p4.constant 2
        p4sa.counter_add(%cnt, %2)

        p4.return
    }
}
```

Listing 5.6: Example of SA dialect optimizations (4).

Finally, since our dialect understands the exact effects of the `p4sa.counter_add` Op, the two sequential `p4sa.counter_add` operations can be safely merged into a single one, resulting in the final version of the optimized deparser.

```
p4.global @counter : !p4.ref<!p4sa.counter>

p4.control @MyDeparser {
    p4.apply(%hdr: !p4.ref<!p4.header<"MyHeader">>, /* ... */) {
        // Increment the counter by 3
        %cnt = p4.addressof @counter
        %3 = p4.constant 3
        p4sa.counter_add(%cnt, %3)

        p4.return
    }
}
```

Listing 5.7: Example of SA dialect optimizations (5).

The shown transformations are primarily supposed to demonstrate a general philosophy of P4 architecture dialects. We aim to showcase what kind of compiler passes a P4 architecture dialect should contain, how can a dialect use the context of the used P4 architecture, and how can such passes be supported by additional IR elements. In practical scenarios, while these transformations remain part of a dialect, they should be integrated into generalized data-flow and pattern matching algorithms.

Chapter 6

Compiling P4/SA Dialects to LLVM

This chapter aims to illustrate a conversion of a *subset* of the P4 and SA dialects into a representation which models abstractions closer to the underlying hardware. Such lowering must address actual implementation details of various P4 constructs, and together with preceding chapters forms a complete walkthrough of a compilation pipeline.

For the final conversion, we choose the LLVM. The LLVM system offers a diverse range of IR components which are conceptually close to hardware instructions. Importantly for our scenario, the LLVM IR offers a possibility to showcase a conversion from P4-specific dialects into representations which were not designed with P4 in mind.

6.1 LLVM Dialect

While the LLVM is an external system[10], it is popular enough that the MLIR community deemed it worthwhile to offer a possibility to use the MLIR infrastructure even when working with LLVM. This is achieved by offering an LLVM dialect as one of the built-in dialects.

The `llvm` dialect serves as a wrapper around a subset of LLVM's IR elements[26]. While functioning within the constraints of MLIR, the `llvm` dialect offers a collection of Ops which map one-to-one to their LLVM counterparts.

```
// LLVM IR
%a = add i32 5, 7

// LLVM dialect
%5 = llvm.mlir.constant(5 : i32)
%7 = llvm.mlir.constant(7 : i32)
%a = llvm.add %5, %7 : i32
```

Listing 6.1: Example of adding two numbers in the LLVM IR and its mapping to the LLVM dialect.

[10]This is not exactly true, MLIR is an LLVM sub-project.

Apart from the dialect elements, the `llvm` dialect also contains a conversion pass which exports an IR of `llvm` Ops into the textual representation of the actual LLVM IR. The LLVM IR can then be loaded into the LLVM and processed further. As a result, we can formulate the objective of this chapter as transforming an IR expressed with P4 and SA dialects into IR containing only the `llvm` dialect.

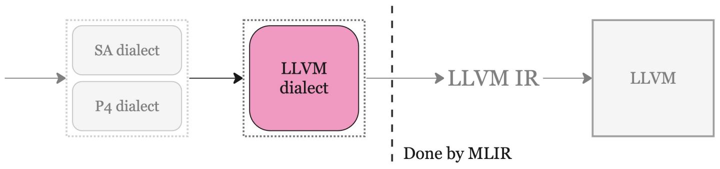

Figure 6.1: In this chapter we convert the P4/SA dialects into the LLVM dialect.

6.2 Action

Even though the LLVM IR aims to be conceptually close to machine instructions, it still provides a concept of functions. Although our P4 dialects do not directly feature functions, they do include abstractions similar to functional blocks, such as actions, states, and apply methods. All of these P4 constructs can be mapped onto the LLVM's understanding of functions, as we initially demonstrate with the `p4.action` operation.

```
p4.action @myAction(%arg: si8) {
    %1 = p4.constant 10 : si8
    %2 = p4.add(%arg, %1) : (si8, si8) -> si8
    p4.return
}
```

```
llvm.func @myAction(%arg: i8) -> () {
    %1 = llvm.mlir.constant 10 : i8
    %2 = llvm.add %arg0, %1 : i8
    llvm.return
}
```

Listing 6.2: Conversion of *p4.action* into the llvm.func Op.

To convert the `p4.action` Op, we use the `llvm.func` Op with no return type. Within the action body, the `p4.constant`, `p4.add`, and `p4.return` Ops naturally correspond to the `llvm.mlir.constant`, `llvm.add`, and `llvm.return` Ops. Interestingly, the `llvm.mlir.constant` Op is additionally prefixed with the `mlir` string. This is `llvm`'s dialect way of indicating that an Op does not have a direct counterpart within the LLVM IR instructions. LLVM IR models constants as a first-class concept (see Listing 6.1), whereas, as we know, the MLIR must use an Op producing a value to express a constant. Therefore, the `llvm.mlir.constant` serves as a bridging Op between these two models.

To call an `llvm.func`, we use the `llvm.call` Op.

```
%42 = p4.constant 42 : si8
p4.call @myAction(%42)
```

```
%42 = llvm.mlir.constant 42 : i8
llvm.call @myAction(%42) : (i8) -> ()
```

Listing 6.3: Conversion of *p4.call* into the *llvm.call* Op.

As we can see, the `llvm` dialect reuses the system of MLIR symbols to declare a name of a callable Op, which can then be referenced from a call operation.

6.3 Integral Data Types

Importantly, as we can see in the preceding snippets, during the conversion into the `llvm` dialect, we had to change the data types of integral values. While the P4 dialect versions use the `si8` (which originates from the P4's `int<8>`), the `llvm` versions use the signless version — `i8`. This conversion corresponds with the LLVM's understanding of integral data types, which does no distinguish between signed and unsigned versions of integers. Instead, LLVM encodes all integer constants using the 2's complement encoding, but then treats them as plain strings of bits. It is then up to the used operation to interpret these strings as signed or unsigned. Therefore, as we demonstrate in the following snippet, during the conversion from the P4/SA dialects, we must be mindful of the used integral types and choose the correct `llvm` Op accordingly.

```
// Signed division
%10s = p4.constant 10 : si8
%5s = p4.constant 5 : si8
%sres = p4.div(%10s, %5s) : si8

// Unsigned division
%10u = p4.constant 10 : ui8
%5u = p4.constant 5 : ui8
%ures = p4.div(%10u, %5u) : ui8
```

```
%10 = llvm.mlir.constant 10 : i8
%5 = llvm.mlir.constant 5 : i8

// Signed division
%sres = llvm.sdiv(%10, %5) : i8

// Unsigned division
%ures = llvm.udiv(%10, %5) : i8
```

Listing 6.4: Conversion of the *p4.div* must consider the types of operands to choose the correct llvm Op.

However, some arithmetic operations work the same for both integral types, and do not need to differentiate between signed and unsigned versions. For such operations, as was the case for the `p4.add` and `llvm.add` Ops, we would emit the same operation for both signed and unsigned operands.

6.4 Control Block

Conversion of the `p4.control` operation with no stateful externs yields no `llvm` Ops. On the other hand, the `p4.control`'s body elements are moved into the global scope. To prevent name conflicts which can now occur after such tranformation (e.g., two different control blocks with equally named actions), we prepend the resultant `llvm.func` names with `"__"`[11] and the name of the enclosing control block. Additionally, the `p4.apply` is converted into an `llvm` Op as if it was an p4.action called `@__apply`. We demonstrate this transformation with the following snippet, where we convert the `MyControl`'s body into `llvm.func` Ops and move them into the global scope.

```
p4.control @MyControl {

    p4.action @myAction(%arg0 : si8) {
        /* ... */
    }

    p4.apply(%arg0 : si8) {
        p4.call @myAction(%arg0) : (si8) -> ()
        p4.return
    }

}
```

```
llvm.func @__MyControl_myAction(%arg0 : i8) {
    /* ... */
}

llvm.func @__MyControl___apply(%arg0 : i8) {
    llvm.call @__MyControl_myAction(%arg0) : (i8) -> ()
    llvm.return
}
```

Listing 6.5: Conversion of the *p4.control* into multiple *llvm.func* Ops in the global scope.

It is worth noting, that moving P4 actions into the global scope can be a fairly complex process with many nuances. Nevertheless, our P4 dialects made the key design decision to transform control and parser parameters into the apply method parameters, to convert all functional P4 constructs into isolated-from-above Ops, and to eliminate the concept of local elements. All of these preceding transformations make the control block's conversion into the `llvm` dialect a relatively straightforward process.

[11]In P4, no user-defined symbol can begin with double underscore, we therefore use it to prevent conflicts with other already existing global symbols.

6.5 Allocation

The P4 dialects offer an *"escape-hatch"* from the SSA's immutability in the form of the `p4.alloc` Op. The `p4.alloc` Op allocates a mutable memory cell and provides a `!p4.ref<T>` reference to it. Through this reference value, the memory cell can be read and written using the `p4.load` and `p4.store` Ops. The memory allocated through the `p4.alloc` is automatically released when execution control leaves the region in which it was allocated.

The memory model of the P4 dialects mirrors the behavior of the *stack* memory known from general-purpose machines. Consequently, the `p4.alloc` can be directly converted into the `llvm.alloca` operation, which allocates memory on the stack of the machine modeled by the LLVM system. The `llvm.alloca` Op produces a value of type `!llvm.ptr<T>`. While the `!llvm.ptr<T>` type is generally more expressive than our `!p4.ref<T>` type (e.g., an `!llvm.ptr<T>` value can be an operand of pointer arithmetic operations), all semantics of the `!p4.ref<T>` type can be expressed by the `!llvm.ptr<T>` type, and as such, the `!p4.ref<T>` type can be directly converted to the `!llvm.ptr<T>` type. Finally, to read and write the memory referenced by the `!llvm.ptr<T>` value, we use the `llvm.store` and `llvm.load` Ops.

```
// Allocate
%ref = p4.alloc : !p4.ref<si8>

// Store and load
%50 = p4.constant 50 : si8
p4.store(%ref, %50)
%val = p4.load(%ref) : si8
```

```
// Allocate
%count = arith.constant 1 : i32
%ptr = llvm.alloca %count x i8 : (i32) -> !llvm.ptr<i8>

// Store and load
%50 = llvm.mlir.constant 50 : si8
llvm.store %50, %ptr : !llvm.ptr<i8>, i8
%val = llvm.load %ptr : !llvm.ptr<i8> -> i8
```

Listing 6.6: Conversion of memory allocation and memory manipulation Ops.

As can be seen in the preceding snippet, the `llvm`'s `alloca` operation requires an additional integer value to indicate the number of allocated values. This value will always be *one* for all types introduced in this work.

6.6 Structures and Headers

P4 headers and structures can be expressed using the LLVM structs. However, while our P4 dialects reference composite types through a reference of a declaration Op, the LLVM IR and `llvm` dialect omit the type declarations and list both the name and the types of fields during each type reference.

```
p4.struct @MyStruct {
    p4.member_decl @f1 : si8
    p4.member_decl @f2 : si16
    p4.member_decl @f3 : ui3
}

$str = foo.get : !p4.struct<"MyStruct">
```

```
$str = foo.get : !llvm.struct<"MyStruct", (i8, i16, i3)>
```

Listing 6.7: Conversion of a P4 dialect's composite type into an LLVM struct.

With the LLVM's way of referencing a struct type, the IR's textual representation can become quickly cluttered with lengthy type references. Luckily, MLIR offers possibility to define a type alias, which effectively allows us to reference a structure without listing all of its members. We can therefore create such a type alias definition in place of our header and struct declarations.

```
// Alias definitions
!MyHeader = type !llvm.struct<"MyHeader", (i8, i16, i1)>
!MyStruct = type !llvm.struct<"MyStruct", (!MyHeader, i10)>

// Alias references
$0 = foo.get : !MyStruct
$1 = foo.get : !MyHeader
```

Listing 6.8: Definition and usage of MLIR's type aliases.

Member Access and Write

Similarly to the P4 dialect's structs and headers, to access members of LLVM structs, the `llvm` dialect uses different operations depending on whether the struct value is allocated into an SSA register or a stack memory. As a result, the P4 dialect's `p4.get_member` and `p4.get_member_ref` can be mapped to the `llvm`'s `llvm.extractvalue` and `llvm.getelementptr` Ops respectively.

```
// Register-allocated struct member access
%val = p4.get_member(%str) @f1 : !p4.struct<"MyStruct"> -> si8

// Memory-allocated struct member access
%ref = p4.get_member_ref(%addr) @f1 :
            !p4.ref<!p4.struct<"MyStruct">> -> !p4.ref<si8>
```

```
// Register-allocated struct member access
$val = llvm.extractvalue %str[0] : !MyStruct ->  i8

// Memory-allocated struct member access
%0 = llvm.mlir.constant 0 : i32
%ptr = llvm.getelementptr %addr[%0] :
            (!llvm.ptr<!MyStruct>, i32) -> !llvm.ptr<i8>
```

Listing 6.9: Conversion of the member access Ops.

During the conversion of the member access Ops, the main distinction is the indication of the accessed member. While the P4 dialect accesses members using their symbols, the LLVM dialect uses the index from the beginning of the structure. Therefore, if the P4 dialect Ops use `@f1` to access the first member of the `MyStruct`, the LLVM dialect uses index zero.

Finally, to modify a struct's field, similarly to the P4 dialect, we need to use the `llvm.store` Op with the `!llvm.ptr<T>` value produced by the `llvm.getelementptr`.

6.7 Counter

In general, the backend is the first stage of the compilation pipeline, where it is necessary to consider the actual implementation of P4 externs. Conversion of a P4 extern to LLVM IR (or any other backend IR) heavily relies on the extern's semantics and therefore cannot be universally generalized. However, with LLVM, we consistently have the option to either directly implement the extern's semantics using LLVM's instructions, or to delegate the implementation by invoking certain library functions. For the Simple Architecture's `Counter` extern, we will illustrate the former approach.

As LLVM IR models an unconstrained general-purpose hardware, the `Counter` can be directly represented as a first-class LLVM data type. For converting a global `Counter` definition, we use the `llvm.mlir.global` Op, specifying its type as `i8`, and initialize it with an appropriate value.

```
p4.global @counter : !p4sa.counter {
    %10 = p4.constant 10 : si8
    p4.init(%10) : (si8)
}
```

```
llvm.mlir.global @counter(10 : i8) : i8
```

Listing 6.10: Conversion of a global Counter instance to an llvm's global *i8*.

To convert SA dialect operations which manipulate a counter instance, we first use the `llvm.mlir.addressof` to retrieve a pointer, and then use other already discussed `llvm`'s Ops to manipulate the referenced `i8` value.

```
llvm.mlir.global @counter(10 : i8) : i8
%ptr = llvm.mlir.addressof @counter : !llvm.ptr<i8>

// p4sa.counter_get
%curr = llvm.load %ptr : !llvm.ptr<i8>, i8

// p4sa.counter_add(%3)
%3 = llvm.mlir.constant 3 : i8
%next = llvm.add(%curr, %3)
llvm.store %next, %ptr : i8, !llvm.ptr<i8>
```

Listing 6.11: Conversion of the SA dialect's Counter Ops.

Chapter 7

Evaluation

For the evaluation purposes, we need to first identify the primary contributions of our work and the metrics we can use to assess them.

The key contribution of this work lies in the methodology of integrating the P4 and MLIR ecosystems. This involves designing the P4 dialect, introducing the concept of P4 architecture dialects, and outlining how these dialects, along with other MLIR dialects, can coexist within a single compilation pipeline.

On the other hand, we deem the Simple Architecture dialect and the conversion of our dialects into the LLVM dialect as primarily demonstrative. While the SA dialect follows the key principles for creating a full-fledged P4 architecture, the Simple Architecture is merely a minimalistic example created for the purposes of this thesis. Similarly, the conversion of our dialects into the LLVM dialect considers only a limited subset of P4 constructs.

However, within the scope of our work, objectively evaluating our contributions presents significant challenges. To understand the difficulty of this task, we can outline several possible evaluation methods and the challenges they bring.

One approach involves passing a non-trivial P4 program through our compilation pipeline based on P4 dialects, performing various optimizations, and measuring metrics of the final IR, such as the number of branching Ops. This method's drawback lies in its expansive scope, requiring the development of analysis algorithms and optimization frameworks tailored to P4/SA dialects, which would vastly extend the intended focus of this text.

Alternatively, we could examine runtime statistics like packet throughput or latency of a compiled P4 program, but this too conflicts with the project's scope. It would be necessary to develop a substantial runtime environment to accommodate P4 constructs converted into the LLVM representation. Such environment would include interfaces for extracting the incoming packets, services implementing the match-action table semantics, and interfaces to emit packet bits to an outgoing packet buffer. Additionally, there would need to be a driver program which allocates necessary resources, and invokes the control and parser blocks as defined by the Simple Architecture.

Nevertheless, while we do not evaluate our dialects using a robust metric, there are still several objective judgments that we can make about our MLIR-based approach, together with several comparisons to the *p4c* (the reference P4 compiler).

7.1 P4 Dialect Design

Apart from being able to serve as a robust compilation target for an arbitrary P4 program, the P4 dialect makes several design decisions which aim to simplify complexity of IR transformations.

Firstly, the P4 dialect eliminates the concept of *local elements*. In Section 4.9 we have demonstrated a simple conversion algorithm which can turn the P4 local elements into semantically-equivalent local variables.

Secondly, the P4 dialect turns the control block, parser block, actions, states, tables, and apply methods into *isolated-from-above* constructs. This means that a compiler pass can work only with a single source of region input SSA values. As described in Section 4.3, this design point necessitated to transform the control and parser block parameters into the apply method parameters.

7.2 Separation of the P4 Architecture Development

Our proposed approach of creating a single global P4 architecture-independent dialect, and moving an architecture-dependent passes and IR elements into a separate dialect provides a straightforward solution to encapsulating a P4 architecture development into a single logical unit.

As discussed in Chapter 1, the *p4c* compiler delegates all of the P4 architecture-dependent IR transformations to the backend, causing great duplication across various hardware architectures. Having a dedicated dialect for all architecture-dependent code structures reduces both cognitive load and chances of introducing unwanted dependencies which would otherwise reduce the possibility of reusing the architecture-dependent elements across backends. Additionally, thanks to the MLIR, packaging and distributing a P4 architecture dialect is a straightforward and tested operation maintained by the MLIR community.

7.3 Practicality

We have successfully created an example P4 architecture dialect (SA dialect), and implemented a partial conversion of our P4 dialects to the LLVM dialect. While we have marked these contributions as primarily demonstrative, it nevertheless indicates that our approach is practical and that our MLIR-based IR model can withstand various transformations and conversions across ecosystems.

7.4 MLIR Ecosystem

The sole decision to use the MLIR framework brings several benefits by itself. While we have provided a detailed motivation for the use of the MLIR framework in Chapter 1, we briefly list the key advantages here as well.

Firstly, by providing infrastructure, standardized ways of documentation, utility tools, and code generators, MLIR allows the P4 community to focus on the P4-specific domain, instead of solving already solved problems.

Secondly, with the support for combination of dialects and the progressive lowering, the MLIR allows to create a flexible IR model that promotes adaptability to compiler passes.

Lastly, contrary to the P4 AST, MLIR uses SSA form, whose advantages are backed by scientific literature[27, 28, 29, 30], and its integration into popular systems, such as LLVM, successfully withstood the test of time.

Conclusion

We embarked on developing a P4 compilation pipeline that is modular, can adapt to various compiler passes, and can efficiently support P4's architecture model. Our aim was to create an IR capable of modeling the program's semantics in a way that is ideal for each of the required IR transformations, rather than offering a single fixed-point abstraction. To achieve this, we decided to use the MLIR framework and its novel approach to compiler design, leveraging dialect inter-mixing and progressive lowering.

Following the introductory chapters on P4 and MLIR, we designed the P4 dialect, which includes IR components to model a P4 program in an architecture-independent manner. In spite of maintaining the general structure of P4, the P4 dialect makes several design decisions aimed at simplifying compiler pass implementation and enabling parallel IR processing within MLIR.

Subsequently, we described the principles guiding P4 architecture dialects and developed a dialect for the Simple Architecture. Furthermore, we demonstrated a series of program optimizations utilizing both SA and P4 dialects.

Moreover, we outlined the conversion of several P4 constructs into the LLVM dialect, which demonstrated the translation of P4-specific abstractions into a backend dialect not originally created with P4 in mind.

Lastly, we briefly compared our MLIR-based approach with a P4 AST-based compilation pipeline, reiterating on the multitude of advantages of our P4 MLIR dialects.

Bibliography

[1] Portable Switch Architecture. Available from: `https://p4.org/p4-spec/docs/PSA.html`

[2] Dijkstra, E. W. The Humble Programmer. *Commun. ACM*, volume 15, no. 10, oct 1972: p. 859–866, ISSN 0001-0782, doi:10.1145/355604.361591. Available from: `https://doi.org/10.1145/355604.361591`

[3] Torczon, L.; Cooper, K. *Engineering A Compiler*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., second edition, 2007, ISBN 012088478X.

[4] Lattner, C.; Adve, V. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. San Jose, CA, USA, Mar 2004, pp. 75–88.

[5] Lindholm, T.; Yellin, F.; et al. *The Java Virtual Machine Specification, Java SE 8 Edition*. Addison-Wesley Professional, first edition, 2014, ISBN 013390590X.

[6] Lattner, C.; Amini, M.; et al. MLIR: A Compiler Infrastructure for the End of Moore's Law. 2020, `2002.11054`.

[7] Matsakis, N. Introducing MIR. 2016. Available from: `https://blog.rust-lang.org/2016/04/19/MIR.html`

[8] Khaldi, D.; Jouvelot, P.; et al. LLVM parallel intermediate representation: design and evaluation using OpenSHMEM communications. 11 2015, pp. 1–8, doi:10.1145/2833157.2833158.

[9] Swift Intermediate Language (SIL). Available from: `https://github.com/apple/swift/blob/main/docs/SIL.rst`

[10] Introduction to the Clang AST. Available from: `https://clang.llvm.org/docs/IntroductionToTheClangAST.html`

[11] Flang. Available from: `https://releases.llvm.org/12.0.1/tools/flang/docs`

[12] Linalg dialect. Available from: `https://mlir.llvm.org/docs/Dialects/Linalg`

[13] Arith dialect. Available from: `https://mlir.llvm.org/docs/Dialects/ArithOps`

[14] Bosshart, P.; Daly, D.; et al. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM Comput. Commun. Rev.*, volume 44, no. 3, jul 2014: p. 87–95, ISSN 0146-4833, doi:10.1145/2656877.2656890. Available from: `https://doi.org/10.1145/2656877.2656890`

[15] P416 Language Specification. Available from: `https://p4.org/p4-spec/docs/P4-16-v1.1.0-spec.html`

[16] Tofino Native Architecture. Available from: `https://opennetworking.org/wp-content/uploads/2021/05/2021-P4-WS-Vladimir-Gurevich-Slides.pdf`

[17] Portable NIC Architecture. Available from: `https://p4.org/p4-spec/docs/PNA.html`

[18] p4c. Available from: `https://github.com/p4lang/p4c`

[19] Bosshart, P.; Gibb, G.; et al. Forwarding metamorphosis: fast programmable match-action processing in hardware for SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, 2013, ISBN 9781450320566, p. 99–110. Available from: `https://doi.org/10.1145/2486001.2486011`

[20] Wong, M. D.; Varma, A. K.; et al. Testing Compilers for Programmable Switches through Switch Hardware Simulation. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '20, New York, NY, USA: Association for Computing Machinery, 2020, ISBN 9781450379489, p. 211–217, doi:10.1145/3386367.3431309. Available from: `https://doi.org/10.1145/3386367.3431309`

[21] Omayma, B.; Laaziz, Y. Software Defined Networks (SDN): the new Era of Networking. 12 2015.

[22] Kohler, E.; Morris, R.; et al. The Click Modular Router. *ACM Transactions on Computer Systems*, volume 18, 05 2001, doi:10.1145/354871.354874.

[23] Doenges, R.; Arashloo, M. T.; et al. Petr4: Formal Foundations for P4 Data Planes. *Proc. ACM Program. Lang.*, volume 5, no. POPL, jan 2021, doi:10.1145/3434322. Available from: `https://doi.org/10.1145/3434322`

[24] Santitoro, R. Metro Ethernet Services - A Technical Overview. 2003.

[25] MLIR. Available from: `https://mlir.llvm.org`

[26] LLVM dialect. Available from: `https://mlir.llvm.org/docs/Dialects/LLVM`

[27] Cytron, R.; Ferrante, J.; et al. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, volume 13, no. 4, oct 1991: p. 451–490, ISSN 0164-0925, doi:10.1145/115372.115320. Available from: `https://doi.org/10.1145/115372.115320`

[28] Cytron, R.; Ferrante, J.; et al. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, New York, NY, USA: Association for Computing Machinery, 1989, ISBN 0897912942, p. 25–35, doi:10.1145/75277.75280. Available from: `https://doi.org/10.1145/75277.75280`

[29] Boissinot, B.; Hack, S.; et al. Fast liveness checking for ssa-form programs. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '08, New York, NY, USA: Association for Computing Machinery, 2008, ISBN 9781595939784, p. 35–44, doi:10.1145/1356058.1356064. Available from: `https://doi.org/10.1145/1356058.1356064`

[30] Knobe, K.; Sarkar, V. Array SSA form and its use in parallelization. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, New York, NY, USA: Association for Computing Machinery, 1998, ISBN 0897919793, p. 107–120, doi:10.1145/268946.268956. Available from: `https://doi.org/10.1145/268946.268956`

[31] Chase, D. R.; Wegman, M.; et al. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, New York, NY, USA: Association for Computing Machinery, 1990, ISBN 0897913647, p. 296–310, doi:10.1145/93542.93585. Available from: `https://doi.org/10.1145/93542.93585`

[32] Sreedhar, V. C.; Ju, R. D.-C.; et al. Translating Out of Static Single Assignment Form. In *Proceedings of the 6th International Symposium on Static Analysis*, SAS '99, Berlin, Heidelberg: Springer-Verlag, 1999, ISBN 3540664599, p. 194–210.

Appendix A

Static Single Assignment

"A single program is defined to be in SSA form if each variable is a target of exactly one assignment statement in the program text."[27]

For instance, none of the P4 actions in the following snippet are in SSA form.

```
action foo() {
    bit<8> var = 2;
    var = 4; // <-- 'var' reassigned
}

action bar() {
    bit<8> var;
    if (var > 3) {
        var = 3; // <-- 'var' reassigned
    }
}

action baz(inout bit<8> var) {
    var = 2;  // <-- 'var' reassigned
}
```

Listing A.1: Examples of P4 actions that are not in SSA form.

SSA form considers only simple scalar, unaliased variables[28]. While many extensions exist, such as [30] or [31], SSA form does not consider derived types like structures, arrays or pointers.

Since each variable has exactly one value, which was assigned to it at a single place of a program, SSA form makes reasoning about many aspects of the data flow of a program trivial, simplifying queries about use-definition and definition-use chains.

As a consequence, SSA form of a program provides data flow information in a form which makes many compiler optimizations easier to perform. For instance, whenever we apply a constant propagation algorithm on SSA code, the use of a variable can be replaced by its value directly, since there is only one definition of that variable in the entire program[29].

Even though the initial program representation is not in SSA form, it is still possible to benefit from SSA form by using the so-called ϕ-nodes.

The algorithm proposed in [28] converts a program into a *Control Flow Graph (CFG)* representation first, and places ϕ-nodes at the start of the basic blocks where two possible versions of a variable merge.

For instance, consider the following pseudocode (which is not in SSA form).

```
X = input()
V = 0
if X < 3
    V = 4
else
    V = 2
X = V
```

Listing A.2: Example of a program that cannot be converted to SSA form without the introduction of a ϕ-node.

If we attempt to rename each assigned variable to a unique name without introducing the ϕ-nodes first, we will be able to successfully rename all occurrences of variable `X`, since at each use of variable `X`, there is only one possible source assignment. However, we cannot do the same with variable `V`. Without a ϕ-node, we cannot determine whether the read `V` value on the last line is coming either from the `V = 4` or the `V = 2` assignment. The version of the read variable depends on the input during the program's execution.

To address this issue, we need to place a single ϕ-node before the last `X = V` assignment.

```
X = input()
V = 0
if X < 3
    V = 4
else
    V = 2
V = phi(V, V)
X = V
```

Listing A.3: Example of a program with a ϕ-node.

For each parent basic block, this ϕ-node accepts an input variable `V`. Next, the result of the ϕ-node is assigned back to the variable `V`.

Finally, as demonstrated in the following snippet, we can rename each variable correctly by renaming the input variables in the ϕ-node according to the version from the parent basic block.

```
$X_1$ = input()
$V_1$ = 0
if $X_1$ < 3
    $V_2$ = 4
else
    $V_3$ = 2
$V_4$ = phi($V_2$, $V_3$)
$X_2$ = $V_4$
```

Listing A.4: Example of a program with a ϕ-node and renamed variables.

The ϕ-node is a special kind of function placed at the start of a basic block B, having as many input arguments as there are parents of B. During evaluation, the ϕ-node selects the input argument corresponding to the parent from which the control flow entered B, and returns it unaltered.

While the ϕ-node has clear semantics, conventional hardware does not implement such function nor is able to simulate it efficiently. Therefore, before lowering to the machine code, ϕ-nodes must be transformed into several copy operations. A naive solution places a copy operation into each of the preceding basic blocks.

```
X = input()
V = 0
if X < 3
    V = 4
    V = V
else
    V = 2
    V = V
X = V
```

Listing A.5: Example of a naive conversion from the SSA form using copy assignments.

While such solution is always correct, [27] describes a more efficient algorithm, which as proved by [32] creates only necessary copy operations.

Appendix B

Contents of enclosed medium

```
src.....................................................implementation
text.........................................the thesis text directory
  thesis.pdf..........................the thesis text in PDF format
  src ............................ the directory of LaTeX source codes
```