TU WIEN Informatics

Mitigating Runtime Errors in Infrastructure as Code Programs Using Static Type Analysis

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Andreas Weichselbaum, BSc
Registration Number 01525900

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dipl.-Ing. Dr.sc. Jürgen Cito, BSc
Assistance: Dipl.-Ing. Michael Schröder, BSc
Dr. Daniel Sokolowski

Vienna, 25th October, 2024

Andreas Weichselbaum | Jürgen Cito

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Andreas Weichselbaum, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 25. Oktober 2024

Andreas Weichselbaum

Danksagung

Zunächst möchte ich meinem Betreuer, Associate Prof. Dipl.-Ing. Dr.sc. Jürgen Cito, BSc, meinen Dank für seine Unterstützung und sein wertvolles Feedback während der gesamten Dauer dieses Projekts aussprechen. Seine Expertise und Ermutigung waren entscheidend für die Gestaltung dieser Arbeit.

Mein Dank gilt auch Dipl.-Ing. Michael Schröder, BSc, für seine wertvolle Zeit, seinen Einsatz und seine Einsichten in die Typentheorie. Sein Feedback hat die Klarheit und Struktur dieser Arbeit erheblich verbessert.

Ebenso möchte ich Dr. Daniel Sokolowski und Prof. Dr. Guido Salvaneschi für ihre frühzeitige Unterstützung bei der Verfeinerung des Umfangs meiner Arbeit danken. Ihre Expertise und Unterstützung waren von unschätzbarem Wert und übertrafen sogar den großzügigen Zugang, den sie mir zu Daten, Skripten und Drafts unveröffentlichter Arbeiten gewährten.

Acknowledgements

First of all, I would like to express my gratitude to my supervisor, Associate Prof. Dipl.-Ing. Dr.sc. Jürgen Cito BSc, for his guidance, support, and insightful feedback throughout the duration of this project. His expertise and encouragement were instrumental in shaping this thesis.

I would like to extend my gratitude to Dipl.-Ing. Michael Schröder, BSc, for his valuable time, effort, and insights on type theory. His feedback significantly enhanced the clarity and structure of the background chapter.

I am also thankful to Dr. Daniel Sokolowski and Prof. Dr. Guido Salvaneschi for their early guidance in refining the scope of my thesis. Their expertise and support were invaluable, surpassing even the generous access they provided to data, scripts, and draft versions of unreleased papers.

Kurzfassung

Infrastructure as Code (IaC) ist essenziell für die Verwaltung von Cloud-Infrastrukturen, aber Laufzeitfehler treten selbst in den am sorgfältigsten geschriebenen Programmen auf, was zu teuren Ausfallzeiten und Sicherheitslücken führt. Trotz der weit verbreiteten Nutzung von Tools wie Pulumi zeigt unsere Arbeit, dass eine bedeutende Möglichkeit zur Reduzierung dieser Fehler durch statische Typensysteme ungenutzt bleibt. Diese Arbeit befasst sich mit der kritischen Frage, wie die statische Typentheorie genutzt werden kann, um Laufzeitfehler in IaC-Programmen zu verringern.

Wir finden und unterteilen echte Laufzeitfehler in drei Kategorien zunehmender Komplexität — **Enumerations**, **Refinements** und **Dependencies** — und verbinden sie mit Enums, Refinement Types und Dependent Types aus der statischen Typentheorie. Zur Formalisierung dieser Typen führen wir einen minimalen Lambda-Kalkül $\lambda_{\rightarrow}$ und darauf basierende Erweiterungen ein, um diese Fehler statisch zu erkennen. Unsere Analyse zeigt, dass viele Laufzeitfehler in IaC-Programmen durch statische Typanalyse verhindert werden können.

Durch eine umfassende Evaluierung von Pulumi, einem führenden IaC-Tool, das mehrere Programmiersprachen unterstützt, stellen wir fest, dass Pulumi statische Typinformationen oft nicht nutzt, selbst wenn sie verfügbar sind. Konkret zeigen wir, dass bis zu 48.68% der AWS-Repositories, 22.29% der Azure-Repositories und 28.45% der GCP-Repositories aus einem Datensatz von 1023 Repositories von Enums profitieren würden – diese werden jedoch in den am häufigsten genutzten Pulumi-Provider-Varianten ignoriert. Darüber hinaus zeigen wir, dass Pulumi Refinement Types in Form von Validierungsparametern, die in den Cloud-Provider-APIs vorhanden sind, übersieht und somit eine Möglichkeit zur Vermeidung von Laufzeitfehlern verpasst.

Wir zeigen außerdem, dass während Dependent Types die komplexesten Fehler, bei denen Parameter voneinander abhängen, abfangen könnten, weder Cloud-Provider-APIs, noch Pulumi, noch die von Pulumi unterstützten Programmiersprachen native Unterstützung für Dependent Types bieten.

Diese Arbeit schließt mit der Empfehlung, diese statische Typinformation zu verwenden, um die Zuverlässigkeit von IaC-Programmen zu verbessern, Laufzeitfehler deutlich zu reduzieren und das Management von Cloud-Infrastrukturen zu optimieren.

Abstract

Infrastructure as Code (IaC) is essential for managing Cloud infrastructure, but runtime errors occur even in the most carefully written programs, leading to costly downtime and vulnerabilities. Despite the widespread adoption of tools like Pulumi, our investigation reveals that a significant opportunity to mitigate these errors through static type systems is being missed. This thesis addresses the critical issue of how to leverage static type theory to reduce runtime errors in IaC programs.

We find and divide real runtime errors into three categories of escalating levels of complexity — **Enumerations**, **Refinements**, and **Dependencies** — and connect them to enums, refinement types, and dependent types from static type theory. To formalise them, we introduce a minimal lambda calculus, $\lambda_{\rightarrow}$, and extensions to it to recognise these errors statically. Our analysis shows that many runtime errors in IaC programs can be prevented by statically enforcing enums, refinement types and dependent types.

Through an extensive evaluation of Pulumi, a leading IaC tool supporting multiple programming languages, we find that Pulumi often fails to use static type information, even when it is readily available. Specifically, we show that up to 48.68% of AWS repositories, 22.29% of Azure repositories, and 28.45% of GCP repositories of a dataset of 1023 repositories would benefit from enum types—yet these are ignored in Pulumi's most common provider variants. Additionally, we demonstrate that Pulumi overlooks refinement type information in the form of validation constraints present in Cloud provider APIs, missing an opportunity to prevent runtime errors.

We show that, while dependent types could mitigate the most complex errors involving interdependent parameters, neither Cloud provider APIs, Pulumi, nor the programming languages it supports offer native support for dependent types.

This thesis concludes by advocating for the adoption of these type systems to enhance the reliability of IaC programs, making significant strides toward reducing runtime errors and improving Cloud infrastructure management.

Contents

CHAPTER 1

Introduction and Problem Description

Infrastructure as Code (IaC) [92] has revolutionised the way modern software systems are deployed and managed, offering significant advantages [12, 73, 111] in automating and scaling infrastructure. However, as IaC practices become more prevalent [12], it is increasingly critical to ensure the correctness and reliability of IaC programs. A fundamental challenge in this domain is to manage and mitigate runtime errors. These errors can lead to costly and disruptive failures [27, 56].

Static type safety is a powerful tool [91, 80] for improving the reliability of software systems by catching errors at compile time rather than at runtime. In conventional programming, static type systems help prevent a range of issues by enforcing type constraints and validating type correctness before the code is executed. However, the integration of static type safety into IaC programs presents challenges as well as opportunities for improvement.

Pulumi [20] is a prominent IaC tool that leverages conventional programming languages to define and manage Cloud resources. Unlike other IaC tools, such as Ansible [10], Chef [6], Terraform [28], or Puppet [22], which use a domain-specific language, or AWS Cloud Development Kit (CDK) [7] and CDK for Terraform (CDKTF) [34], which offer limited capabilities when compared to Pulumi (see Section 2.6.1), Pulumi allows users to write infrastructure code in familiar languages such as C#, Go, Java, TypeScript, and Python for multiple different Cloud providers. This integration provides a powerful means to use existing type systems but also introduces complexities related to ensuring comprehensive type safety across diverse programming environments and Cloud ecosystems.

This thesis addresses the need for improved static type safety in IaC programs by focusing on Pulumi. We explore the kinds of runtime errors commonly encountered in IaC programs and investigate how these errors can be mitigated through improved static

type analysis and type system enhancements. Our approach involves categorising errors, formulating a minimal lambda calculus framework to represent and address these errors formally, and evaluating the applicability of this framework within the Pulumi ecosystem with concrete code examples.

The core objectives of this research are to answer the following research questions:

- **RQ1**: What are common errors in IaC programs that could be prevented statically?
- **RQ2**: How can such errors be prevented?

By advancing static type safety in IaC programs, particularly in the context of Pulumi, this thesis aims to contribute to more reliable and maintainable infrastructure management practices. In contrast to the existing scientific literature (see Section 2.1) that focusses on errors in IaC programs, our research is unique in a number of ways.

- This thesis focuses on a concrete subset of the most prevalent errors in IaC programs, that is, type-related, erroneous infrastructure configuration.
- We highlight a proper correspondence between a set of well-defined error categories, based on real IaC runtime errors, and static types.
- We offer a comprehensive "end-to-end" perspective on **(1)** how type-related runtime errors can be expressed as static types in type theory, **(2)** how these types are implemented in existing programming languages, and **(3)** their applicability within Pulumi, a specific IaC tool.
- Finally, we assess the potential impact of improving static type safety in Pulumi based on its integration with three major public Cloud providers.

The remainder of the thesis is structured as follows. It begins with a background chapter that provides foundational knowledge on types, including enums, refinement types, and dependent types, and introduces Pulumi in detail, as well as related work. The next chapter starts with an analysis of existing runtime errors in IaC. Afterwards, we establish a dataset of Pulumi repositories and conduct some data analysis on it. The succeeding chapter continues with strategies for improving static type safety idiomatically within Pulumi using enums, refinement types, and dependent types, by using the established dataset. We continue with evaluating and discussing our findings, before the final chapter summarises our results.

CHAPTER 2

Background and Related Work

This chapter starts by presenting the current state of the art in scientific literature concerning errors in IaC. It continues by introducing a minimal, simply typed lambda calculus to define typing problems in a formal way. We present how to formulate errors in our error categories formally and define typing rules, with which the characteristic errors can be handled statically. We discuss IaC and Programming Languages IaC in general and the role that Pulumi plays within it. Finally, this chapter concludes with a detailed description of Pulumi, for which we will analyse if the runtime errors from our categories can be mitigated statically.

2.1 Related Work

Infrastructure as Code is an active research area with a community researching IaC best-practices, patterns, errors, properties of, and verification tools for IaC programs. However, the majority [103] of the research goes towards proposing additional frameworks or tools for existing IaC tools and libraries.

Rahman et al. [102] categorise defects in IaC programs and found that most defects are related to erroneous assignments and configurations, implying that handling this kind of errors more effectively will provide great benefit to the reliability of the configured infrastructure. Notably, our thesis focuses on the subset of this set of erroneous assignments, that is, type-related runtime errors.

Sharma et al. [110] and Schwarz et al. [108] defined code smells for Chef, an IaC tool with strong integration into traditional server management tools. Although the research focuses on Chef, it is argued that the proposed code smells are agnostic to the IaC technology in use.

Using a machine learning algorithm, Chen et al. [57] identified IaC errors from git commit histories and proposed a JSON-based rule set of 30 rules, which was able to find 60%

of all identified error patterns in Puppet artefacts. Similarly, Dai et al. [62] created an analysis framework to find risky code patterns in Ansible playbooks. However, this approach requires a third-party analyser, searching for error patterns using the predefined rules, instead of relying on established static code validation, like static type systems of programming languages.

Other research focusses on providing formal frameworks and tools to assess desired qualities of IaC solutions, like idempotency and determinism. Shambaugh et al. [109] create a formal verification tool to analyse idempotency, determinism and other properties in Puppet, an IaC tool to configure physical and virtual machines, while Hummer et al. [77] design a similar verification tool for Chef.

Sokolowski et al. [114, 115, 113] conducted extensive research on automated IaC program testing and verification with a focus on Pulumi programs by proposing Automated Configuration Testing (ACT) and creating ProTI, an implementation of ACT for Pulumi.

2.2 Types

In general, types restrict the possible values that can be used in certain contexts. From a programmer's point of view, that context is a computer program, where types restrict the values of input parameters to and output parameters of functions, variables, or constants. The underlying formal theory for types systems of programming languages is called type theory. Type theory [60, 107] is a branch of logic based on types that first appeared as an alternative to **set theory**, after discovering a paradox in set theory that causes the theory to lead to contradictions, commonly known as **Russell's paradox**. Russell further presented type theory as sufficiently expressible to act as a foundation of mathematics [101].

A type theory consists of a handful of components that form a framework to test "well-typedness" of terms within the theory. These include, syntax definition of the formal language in the theory and typing rules, which will be explained in the following sections.

2.2.1 Type Theory Syntax

One of the most prevalent formal notations for a type theory is the simply typed lambda calculus [58] by Alonzo Church, which is usually denoted in Backus-Naur form. We will introduce a very basic form of the simply typed lambda calculus, which we will call $\lambda_{\rightarrow}$.

Types	τ	::=	Bool	
		|	Int	
		|	...	
		|	$\tau \to \sigma$	type of functions
Terms	e	::=	x	variable
		|	$\lambda x : \tau.e$	abstraction, i.e. function definition
		|	$e1\, e2$	function application
Contexts	Γ	::=	$\emptyset$	
		|	$\Gamma, x : \tau$	context containing variable binding x to type τ

Table 2.1: Syntax of $\lambda_{\to}$

Table 2.1 shows the syntax of the simply typed lambda calculus. It consists of **types**, **terms** and **contexts**.

- **Types** can be any kind of primitve type like Bool or Int, in additional to function types $\tau \to \sigma$, which represents functions mapping values of type τ to values of type σ.
- **Terms** may be variables x, abstractions $\lambda x : \tau.e$, function applications $e1\, e2$, atomic values or conditional terms.
- **Contexts** are sets with variable bindings. They are either the empty set $\emptyset$ or contain bindings of variables x to types τ.

2.2.2 Typing Rules

Typing rules are the formal rules for assigning types to terms. They usually take the form of Gentzen-style inference rules [71], where the premises and consequences are divided by a horizontal line. A set of typing rules can look similar to the following.

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau}$$

(a) T-Var

$$\frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \lambda x : \sigma.e : \sigma \to \tau}$$

(b) T-Abs

$$\frac{\Gamma \vdash e_1 : \sigma \to \tau \qquad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1\, e_2 : \tau}$$

(c) T-App

Figure 2.1: Typing Rules of $\lambda_{\to}$

The typing rules are explained in the following enumeration. Note that we do not use all the typing rules that are typically used in simply typed lambda calculus. For a more

comprehensive coverage of the simply typed lambda calculus, the reader is referred to Benjamin C. Pierce's **Types and Programming Languages** [99].

1. **T-Var**: If $(x : \tau) \in \Gamma$, then x is well-typed under τ in context Γ.
2. **T-Abs**: If $e : \tau$ is well-typed in context Γ, which includes $x : \sigma$, then abstraction $\lambda x : \sigma.e$ has type $\sigma \to \tau$ in Γ
3. **T-App**: If $e1 : \sigma \to \tau$ is a function and well-typed in Γ and $e2 : \sigma$ is a term and well-typed in Γ, $e1\, e2$, i.e., applying $e1$ to $e2$, has type τ in Γ.

Usually, a type system additionally contains typing rules for primitive types, for example, to define which type a constant has. These typing rules are merely consequences without premises (or with the premises set to $\top$). Figure 2.2 shows typing rules for constants. For convenience sake, we included **T-Cons** to denote the intuitive notation of a constant belonging to a matching type, if not defined more precisely.

$$\frac{\top}{\Gamma \vdash true : Bool}$$

(a) T-True

$$\frac{\top}{\Gamma \vdash false : Bool}$$

(b) T-False

$$\frac{(\text{c is a constant of type } \tau)}{\Gamma \vdash c : \tau}$$

(c) T-Cons

Figure 2.2: Typing Rules for Giving Types to Constants

2.2.3 Example

At this point, our small type system is already able to express basic static typing. Consider the trivial TypeScript code in Listing 2.1

```
1 function identity (x: number): number {return x}
2 var a:number;
3 identity(a)
```

Listing 2.1: Example for Function Application

Certainly a programmer would want the static type checker of TypeScript to validate if the invocation of `identity(a)` is well-typed. In the Lambda Calculus, functions calls are typed by **T-App**. The type checker expands the proof tree "upwards" by applying the typing rules. If the term can be derived from rules, the program is considered well-typed. At the time of the function call in line 7 in Listing 2.1, we already bound the variable `a` to the type `number`, therefore $\Gamma = \{a : Int\}$.

$$\dfrac{\dfrac{\dfrac{x : Int \in \{a : Int, x : Int\}}{\{a : Int, x : Int\} \vdash x : Int}}{\{a : Int\} \vdash \lambda x : Int.x : Int \to Int} \quad \dfrac{(a : Int) \in \{a : Int\}}{\{a : Int\} \vdash a : Int}}{\{a : Int\} \vdash (\lambda x : Int.x)\, a : Int}$$

Figure 2.3: Complete Proof Tree for Proofing that the Identity Function is Well-Typed

The proof tree in Figure 2.3 shows that the identity function for **number** values implemented in Listing 2.1 is correctly typed. It also shows that even proofs for simple programs produce large proof trees. While being occasionally useful for checking type safety, the presented simply typed lambda calculus fails to validate anything but the simplest types. More precisely, it has no means of addressing the typing errors discussed in Section 3.3. The following sections will build upon this basic lambda calculus and add new syntax and typing rules, to be able to handle enums, refinement types, and dependent types.

2.3 Enums

In type theory, enums are a special case of what are called (labelled) **variants**. A variant is a type that encompasses one or multiple other types, which are labelled by the so-called **field labels**. For example, consider the variant type $IpAddress = \langle ip4 : Ip4Address, ip6 : Ip6Address \rangle$. Intuitively, an ip address can be either an ip4 address or an ip6 address. An example value for type $IpAddress$ would be written as $address = \langle ip4 = i \rangle \ as \ IpAddress$, where i is a value of type $Ip4Address$. Enumerations, or enums, are a simpler version of variants in that every type T_i in $\langle l_i : T_i \{i \in \dots n\} \rangle$ is $Unit$.

Unit is a type that represents a singleton. There is only one value that has the type Unit (with a capital U) and that is the value unit (with a lowercase u), which is formalised by the typing rule **T-Unit** below.

As in $\lambda_{\to}$, we will introduce the syntax and the typing rules needed to formalise enums, before presenting an example.

2.3.1 Syntax

Table 2.2 shows extensions for the syntax definition τ of $\lambda_{\to}$ that we need to formalise enums.

Types	τ	::=	$\ldots$	
		$\mid$	$Unit$	
		$\mid$	$\langle l_i : T_i \ ^{i \in 1 \ldots n} \rangle$	type of variants
Terms	e	::=	$\ldots$	
		$\mid$	$unit$	
		$\mid$	$\langle l = t \rangle$ as T	tagging

Table 2.2: Syntax Extensions for τ and e to support Enums

2.3.2 Typing Rules

As with the syntax extensions, the typing rules in Figure 2.4 are extensions to the typing rules we used for $\lambda_{\rightarrow}$.

$$\frac{}{\Gamma \vdash unit : Unit}$$

(a) T-Unit

$$\frac{\Gamma \vdash t_j : T_j}{\Gamma \vdash \langle l_j = t_j \rangle \ as \ \langle l_i : T_i \ ^{i \in 1 \ldots n} \rangle : \langle l_i : T_i \ ^{i \in 1 \ldots n} \rangle}$$

(b) T-Variant

Figure 2.4: Typing Rules Extension for the Simply Typed Lambda Calculus to Support Enums

The two additional typing rules are explained in the following enumeration.

- **T-Unit**: value **unit** is of type **Unit**.
- **T-Variant**: If t_j has type T_j in context Γ, the term $\langle lj = tj \rangle$ can be considered to belong to the broader variant type $\langle l_i : T_i \ ^{i \in 1 \ldots n} \rangle$, provided T_j matches the type labeled by l_j in the variant type.

2.3.3 Example

Listing 2.2 shows another simple code example, this time containing enums. The code defines an enum with two possible values and a function that flips between those two values.

```
1 enum Coin {HEAD, TAIL}
2 function flip(c: Coin) {return c == Coin.HEAD ? Coin.TAIL : Coin.HEAD}
3 flip(Coin.HEAD)
```

Listing 2.2: Example for Function Application with Enums

Figure 2.5 shows a complete proof of the typing in Listing 2.2 , where type $Coin$ is defined as $Coin = \langle HEAD : Unit, TAIL : Unit \rangle$. Note that the structure is very similar to the proof in Figure 2.3, even though the implementation of the function is completely different. In fact, the implementation does not even matter in our current form of the calculus, since we are only dealing with **uninterpreted functions**, which have no semantic, other than being a function.

$$\dfrac{\dfrac{\dfrac{x : Coin \in \{x : Coin\}}{\{x : Coin\} \vdash x : Coin}}{\{\} \vdash \lambda x : Coin.x : Coin \to Coin} \quad \dfrac{\{\} \vdash unit : Unit}{\{\} \vdash \langle HEAD = unit \rangle \text{ as } Coin : Coin}}{\{\} \vdash ((\lambda x : Coin.x)\ (\langle HEAD = unit \rangle \text{ as } Coin)) : \text{Coin}}$$

Figure 2.5: Complete Proof Tree for Proofing that the Program in Listing 2.2 is Well Typed

2.4 Refinement Types

Refinement types [67, 78, 119], **predicate subtypes** [106] or **liquid types** [104, 105] describe types in programming languages that are defined by a base type b and a decidable logical predicate p. A value v is valid for the refinement type t, if v is of type b and p evaluates to true. As with enums in Section 2.3, we will extend the syntax and set of typing rules of our simply typed lambda calculus with a minimal set of additions to be able to represent refinement types within our calculus. For a more comprehensive presentation of the refinement types, we recommend the work of **Jhala** and **Vazou** [78], including the refinement type implementation of Liquid Haskell [120].

2.4.1 Syntax

Again, we extend the syntax of $\lambda_{\to}$ to formalise refinement types. Note that now a type is made up of two parts, basic types and refinements, summarised in Table 2.3.

Basic Types	b	$:=$	int,float,string...	as defined by the type system
Refinements	r	$:=$	$\{v : p\}$	a predicate p, parameterised by v
Types	τ	$:=$	...	
		$\mid$	$b\{r\}$	refined base

Table 2.3: Syntax Extensions for τ and e to Support Refinement Types

- **Basic Types** b are primitive types directly exposed by the type system.
- **Refinements** r are SMT-solvable predicates p parameterised by v, e.g. $\{v : p = 5 \leq v\}$

- **Types** τ are refined base types $b\{r\}$ like the type $int : \{v : 0 \leq v\}$, which represents $\mathbb{N}$

2.4.2 Typing Rules

The following typing rules in Figure 2.6 extend the typing rules we used for $\lambda_{\rightarrow}$.

$$\frac{\textbf{SmtValid}(c)}{\emptyset \vdash c} \qquad \frac{\Gamma \vdash \forall x : b.p \implies c}{\Gamma, x : bx : p \vdash c}$$

(a) Ent-Emp (b) Ent-Ext

Figure 2.6: Typing Rules Extension for the Simply Typed Lambda Calculus to Support Refinement Types

As with enums, there are two additional typing rules that we need to validate refinement types.

1. **Ent-Emp**: if a logical constraint c is deemed valid by a SMT solver, c is valid under the empty context $\emptyset$

2. **Ent-Ext**: if for all x of type b predicate p implies c under context Γ, c is valid, after adding the variable binding $x : b\{x : p\}$ to context Γ.

2.4.3 Predicates

Refinement types augment base types by an additional logical predicate p. The validity of p implies that the program is well-typed. In theory, p could be drawn from any decidable logic. In practise, many refinement type verification implementations leverage SMT solvers [93], to check type validity and therefore p has to be decidable by an SMT solver. Satisfiability Modulo Theories (SMT) generalise the Boolean Satisfiability Problem (SAT) to mathematical formulas, which may include different kinds of numbers and strings and even data structures, like lists or arrays.

Table 2.4 summarises the syntax of predicates in Liquid Haskell [120], a refinement implementation for Haskell. To stay decidable, predicates do not possess the full expressibility of an arbitrary Haskell program, but rather are taken from the set of the quantifier-free fragment of linear arithmetic with uninterpreted functions (QF-UFLIA) [53].

Constants	c	$:=$	$0, 1, 2, \ldots$	
Variables	v	$:=$	$x, y, z, \ldots$	
Expressions	e	$:=$	v	variable
		$\mid$	c	constant
		$\mid$	$(e + e)$	addition
		$\mid$	$(e - e)$	subtraction
		$\mid$	$(c * e)$	multiplication by constant
		$\mid$	$(v\ e1\ e2\ \ldots\ en)$	uninterpreted function application
		$\mid$	$(if\ p\ then\ e\ else\ e)$	if-then-else
Relations	r	$:=$	$==$	equality
		$\mid$	$/=$	disequality
		$\mid$	$>=$	greater than or equal
		$\mid$	$<=$	less than or equal
		$\mid$	$>$	greater than
		$\mid$	$<$	less than
Predicates	p	$:=$	$(e\ r\ e)$	binary relation
		$\mid$	$(v\ e1\ e2\ \ldots en)$	predicate (or alias) application
		$\mid$	$(p\ \&\&\ p)$	and
		$\mid$	$(p \mid\mid p)$	or
		$\mid$	$(p \implies p)$	implies
		$\mid$	$(p \Leftrightarrow p)$	iff
		$\mid$	$(not\ p)$	negation
		$\mid$	$true \mid True$	
		$\mid$	$false \mid False$	

Table 2.4: Syntax of predicates in Liquid Haskell

- **Constants** are numeric values
- **Variables** are references to binders in the source programs
- **Expressions** are either variables, constants or linear, arithmetic expressions over variables and constants and uninterpreted function applications.
- **Relations** are comparison operators
- **Predicates** are comparisons of two expressions, applications of predicate functions to a list of arguments or any Boolean combination of predicates.

The predicates p are translated into Verification Conditions (VC), which are checked by an SMT solver. The SMT solver tries to find counterexamples such that p is not valid. If it fails to find any counterexample, the VC is considered valid, implying the source program is well-typed. For example, consider Listing 2.3 and the resulting VC in Equation (2.1).

2.4.4 Example

For example, Listing 2.3, shows some Liquid Haskell code, defining a variation of the identity function.

```
1  {-@ type Int4 = {v:Int | v < 4} @-}
2  {-@ type Int5 = {v:Int | v < 5} @-}
3
4  {-@ a :: Int4 @-}
5  a = 2
6
7  a :: Int
8
9  {-@ identity5 :: Int5 -> Int5 @-}
10 identity5    :: Int -> Int
11 identity5 v = v
12
13 b = identity5 a
```

Listing 2.3: Example Liquid Haskell Code with function application

The above code does the following.

- Lines 1-2 define datatypes Int4 and Int5 for integers smaller than 4 and smaller then 5, respectively
- Line 4 declares variable a as liquid type Int4
- Line 5 initialises a with value 2
- Line 7 declares a as Haskell type Int
- Lines 9-11 declare a function identity5 that takes an argument of the Haskell type Int and produces an output as Int. Both, input and output have been refined to liquid type Int5
- Line 13 shows the function application application to the argument a

Since every parameter of type Int4 is also a valid value of type Int5, this version of the identity function allows for a simple kind of polymorphism. A proof that the function call is well typed for the refinement type based identity function is presented below in Figure 2.7, where the context at the time of the function application Γ is $\{a : Int\{a < 4\}\}$.

$$\dfrac{\dfrac{\dfrac{x : Int\{x < 5\} \in \Gamma, x : Int\{x < 5\}}{\Gamma, x : Int\{x < 5\} \vdash x : Int\{x < 5\}}}{\Gamma \vdash \lambda x : Int\{x < 5\}.x : Int\{x < 5\} \rightarrow : Int\{x < 5\}} \quad \dfrac{\dfrac{\text{SmtValid}(\forall x : Int.x < 4 \implies x < 5)}{\{\} \vdash \forall x : Int.x < 4 \implies x < 5}}{\{a : Int\{a < 4\}\} \vdash a : Int\{x < 5\}}}{\Gamma \vdash ((\lambda x.x)\, a) : Int\{v < 5\}}$$

Figure 2.7: Complete Proof Tree for Proofing that the Refinement Type Based Identity Function is Well-Typed

This will result in the VC in Equation (2.1), which is deemed valid by an SMT solver. Therefore, in turn, the program is considered well typed.

$$\forall x : Int.x < 4 \implies x < 5 \tag{2.1}$$

2.5 Dependent Types

Dependent types and programming languages that natively implement them in their type systems blur the line between "conventional" programming and formal verification. It is no coincidence that these programming languages are also partly referred to as "proof assistents" or "theorem prover" [30]. Many such programming languages are based on type systems which are based on **intuitionistic** logic, like Martin-Löf's type theory [89, 90, 95]. These type systems leverage the observed isomorphism between computer programs and the construction of formal proofs, which can be validated by formal verification. This isomorphism is known as **Curry-Howard isomorphism** and will be explained in Section 2.5.1. There is a great deal of literature on (programming languages with) dependent types [52, 126, 127, 54, 59], intuitionistic logic [88, 74, 63] and the Curry-Howard isomorphism [75, 96, 61, 118], including the lectures of Sørensen and Urzyczyn [116] on which we lean heavily when explaining the Curry-Howard isomorphism. We will focus on what we consider to be the essential parts for understanding dependent types, how they are connected to formal verification, and what their benefits and drawbacks are, without diving too deep into logic and mathematics.

2.5.1 Intuitionistic Logic and Type Theories

Intuitionistic (or **constructive**) logic, is a branch of formal logic, which places emphasis on constructive proofs. In contrast to classical logic, proofs in intuitionistic logic not only proof the validity of logical formulas. Instead, they provide a method (an algorithm) to create an "object" that proves the formula. To give a very basic and simplified example, a proof in classical logic might be interpreted as a statement like "prime numbers greater than 1000 exist", without explicitly listing any. On the other hand, a proof in intuitionistic logic would provide a method, an algorithm, to generate prime numbers greater than 1000, proofing that prime numbers greater than 1000 exist. In particular, intuitionistic

logic does not accept the general validity of the **law of excluded middle** and the **double negation elimination**, which are both fundamental inference rules of classical logic [125].

$$.P \vee \neg P \tag{2.2}$$

Equation (2.2) shows the law of excluded middle. In classical logic, a statement is considered either true or false. In intuitionistic logic, neither is true, without giving constructive proof.

$$.\neg\neg P \implies P \tag{2.3}$$

Equation (2.3) on the other hand shows the double negation elimination rule, which is also not considered generally valid in intuitionistic logic.

Intuitionistic logic uses the same alphabet and grammar as classical logic. However, the interpretations are slightly different. For example, consider the Brouwer–Heyting–Kolmogorov interpretation [118] of some of the logical connectives.

- A proof for $P \wedge Q$ consists of two constructive proofs. One for P and one for Q
- A proof for $P \implies Q$ consists of a function that translates any proof for P to a proof of Q
- A proof for $P \implies \bot$ $(\neg P)$ consists of a function that translates any proof for P to a proof of $\bot$. The symbol $\bot$ indicate a contradiction.

Curry-Howard Isomorphism

The Curry-Howard isomorphism, also called the Curry-Howard correspondence or equivalence, is a combination of observations between Haskell B. Curry and William A. Howard, where they notice an equivalence between logical proof theories and computational calculi, like type theory. In particular, dependent types can be translated into predicates in first-order logic. A dependent type for strings of fixed length can be formulated as $\mathrm{string}(\mathrm{n})$ with $\mathrm{n}{:}\mathrm{int}$, which translates to a predicate over int in intuitionistic logic. For example, consider Table 2.5, which informally translates type system terms to their counterpart in intuitionistic logic.

Interpretation

We started Section 2.5 by stating that programming languages with built-in dependent types "blur the line" between what might be considered as "conventional" programming and formal verification. This is because these programming languages are usually built on intuitionistic logic and leverage the observations of the Curry-Howard isomorphism.

Programming	**(Intuitionistic) Logic**
type A	proposition A
dependent type $B(a)$	exist. quantification over predicate B for a
dependent function type $a- > B(a)$	univ. quantification over predicate B for a
a term of type A	a proof of formula A

Table 2.5: The Curry-Howard Isomorphism exemplified by mapping programming terms to logic

Intuitively, this means that a programmer does not need to construct a proof to show that their program is well-typed. The program already **is** the proof.

However, this expressibility does not come without drawbacks. Dependently-typed programming languages might be considered harder to learn since they require some basic understanding of formal informatics, which could help explain the low rate of adoption discussed in Section 2.5. Furthermore, this degree of sophistication in a type system causes the type checking problem to be undecidable in some programming languages [52].

In contrast to refinement types, which allow the specification of a logical constraint alongside base types, dependent types allow the definitions of types that are dependent on values. Dependent types enable the definition of complex types in programs that can be checked for validity by formal verification tools. They tend not to be implemented in "mainstream" programming languages, like Java or JavaScript. In fact, none of the 51 most commonly used programming, scripting, and markup languages in the annual StackOverflow report [26] implements dependent types natively. Programming languages, which natively integrate dependent types into their type systems, include, but are not limited to **Agda** [2], **Coq** [30] **F*** [8] and **Idris** [11].

The syntax and typing rule extensions we are about to introduce are the absolute minimum we need to further understand type evaluation for dependent types. In particular, we omit any definitions of **Kinding** or dependent pairs and limit our discussion to dependent function types. For a much more extensive coverage of this topic, we again refer the reader to the works of Benjamin C. Pierce [100].

2.5.2 Syntax

Again, we extend the syntax of $\lambda_{\rightarrow}$ to formalise dependent types, summarised in Table 2.6.

Types	τ, σ	$::=$	$\ldots$
		$\mid$	$\Pi x : \tau.\sigma$ dependent product type

Table 2.6: Syntax Extensions for τ to Support Dependent Types

The only addition to our simply typed lambda calculus is the dependent product type.

- $\Pi x : \tau.\sigma$ formulates a dependent product (or Π) type and replaces the "arrow type" $\tau \to \sigma$ in $\lambda_{\to}$, generalising function types. Basically, a Π type is a function, mapping elements t of type τ to elements of type $[x \mapsto t]\sigma$. The expression $[x \mapsto \tau]\sigma$ means to replace every occurrence of x in σ with t, assuming σ is parameterised by the variable x of type τ.

2.5.3 Typing Rules

Figure 2.8 shows two typing rules from $\lambda_{\to}$ that have to be adjusted for dependent types.

$$\frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \lambda x : \sigma.e : \Pi x : \sigma.\tau}$$

(a) T-Abs

$$\frac{\Gamma \vdash e_1 : \Pi x : \sigma.\tau \qquad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1\, e_2 : [x \mapsto e_2]\tau}$$

(b) T-App

Figure 2.8: Typing Rule Extensions of $\lambda_{\to}$ to Support Dependent Types

- **T-Abs** replaces **T-Abs** of $\lambda_{\to}$ by generalising it. Given context Γ with variable binding x to type σ and e is being a term of type τ, the return type of a function applying x to e depends on the value of x.

- **T-App**, on the other hand, replaces **T-App** of $\lambda_{\to}$. In context Γ, applying e_1 to e_2 returns a value, whose type is given by evaluating replacing every free occurrence of x in τ with $e2$, if e_1 has Π type $\Pi x : \sigma.\tau$ and e_2 has type σ.

2.5.4 Examples

Showing examples for dependent types and presenting their formal representation becomes increasingly challenging. Especially with dependent types, the examples would have to be so trivial to not let the size of the proof tree explode, that the value and expressiveness of dependent types would be under-represented. However, we will at least take a look at how a dependent function in Idris would be formalised in our calculus.

```
1 isSingleton : (b:Bool) -> Type
2 isSingleton True = Nat
3 isSingleton False = List Nat
```

Listing 2.4: Example for a Dependent Function Type in Idris

Listing 2.4 shows a short code snippet of Idris code, defining a dependently typed function. In Idris, types are "first-class citizens" and can be returned by function. The function `isSingleton` takes a boolean value `b` and returns a type depending on the value of `b`.

$$\frac{\dfrac{b : Bool \in \Gamma, b : Bool}{\Gamma, b : Bool \vdash e : isSingleton(b)}}{\Gamma \vdash \lambda b : Bool.e : \Pi b : Bool.isSingleton(b)}$$

Figure 2.9: Expanded Abstraction for Dependent Function isSingleton

Figure 2.9 shows how the function definition for isSingleton would translate to our calculus, where e is the function body, deciding whether the provided b:Bool is True or False.

Another common example for dependent types that is often used in the literature is the vector, i.e. a list with a fixed length. Consider the definition of the datatype Vect from the dependent type examples for Idris.

```
1 data Vect : (len : Nat) -> (a : Type) -> Type where
2   Nil  : Vect 0 a
3   (::) : (x : a) -> (xs : Vect n a) -> Vect (S n) a
```

Listing 2.5: Example for datatype definition for the dependent type Vect

In Listing 2.5 a new datatype Vect is defined. Datatype Vect depends on two parameters. The second parameter, (a: Type) defines the type of elements in the resulting vector. However, Vect is not only dependent on type a, but also depends on value (len: Nat), which is a natural number and defines the number of elements in the vector. Lines 2 and 3 recursively define Vect via the type constructors Nil and ::. The type constructor Nil constructs the type Vect 0 a, i.e., an empty vector of elements of arbitrary type a. The type constructor :: takes an element x of type a (x:a) and a vector xs for elements of type a with length n (x: Vect n a) and constructs the type Vect (S n) a, that is, the vector a of length n+1 for elements of type a. The definition of the function S is omitted here, but it represents the "successor function " of Peano arithmetic, where the natural numbers $\mathbb{N}$ are defined recursively by providing an element 0 and a successor function that increments a natural number by 1. The next Idris example is more complex but will be more applicable in our problem domain.

2.6 Infrastructure as Code

Infrastructure for software is an umbrella term for all of the underlying systems that need to be in place for software to run properly. This includes, but is not limited to, databases, networks, virtual machines, Kubernetes clusters, load balancers, users, or even complete development stages and isolated tenants. A continuously available infrastructure is therefore crucial for the overall availability of a software product. Cloud computing, virtualisation and the increasing amount [98] of cloud infrastructure have made this issue even more prevalent. Historically, infrastructure deployments have been treated differently than the development and deployment of software, in that it was often manually managed, while testing and automation were rare [111, 72]. Additionally,

organisational teams were often isolated in so-called “silos”, disconnected teams and departments, with little connection with and limited insight into other teams. This led to poor communication and collaboration.

The DevOps [64] movement aims to connect these isolated silos, by establishing “end-to-end” processes and practices, in order to enable a team to build, test, deliver, and run software. One of those practises is Infrastructure as Code (IaC), which aims to bring well-known, established software development techniques and best practices to infrastructure provisioning. Morris et al. [92] define the following three core practises of IaC.

- Define everything as code
- Continuously test and deliver all work in progress
- Build small, simple pieces that you can change independently

Defining infrastructure components as code makes them subject to established, rigorous software practices, like continuous testing, frequent integration, and continuous deployment. This increases reusability, consistency, and transparency, because infrastructure can be deployed multiple times, using the same mechanisms, while being subject to reviews from other developers. One of the most prominent [48] IaC tools at the moment is Terraform [28] by HashiCorp, which defines its own scripting language, HCL, for defining Cloud resources. Other IaC tools include Puppet, Chef, and Ansible. While Ansible defines the target state of the infrastructure with YAML files, the others define the desired infrastructure resources in a custom Domain-Specific Language (DSL).

```
1  resource "aws_db_instance" "changeme_simple_aws_db_instance" {
2    allocated_storage    = 5
3    engine               = "mysql"
4    engine_version       = "5.7"
5    instance_class       = "db.t3.micro"
6    name                 = "changeme_simple_aws_db_instance"
7    username             = "changemeusername"
8    password             = "changeme_password"
9    skip_final_snapshot  = true
10 }
```

Listing 2.6: Terraform Resource Defining a MySQL Database

For example, consider Listing 2.6, which defines an AWS managed MySQL database with parameters, like the engine_version, or the initial provisioning of a database user changeusername. This resource definition is part of a Terraform project, which will create a MySQL database in AWS according to the specification. This process is repeatable, should always yield the same result, and is completely transparent to any developer reviewing the code.

2.6.1 PL-IaC

In contrast to the previous examples, Programming Languages Infrastructure as Code (PL-IaC) [114] describes IaC, where developers define infrastructure resources in general-purpose programming languages. This allows developers to leverage their existing knowledge, tools, and coding practices of a supported language and apply them to IaC. It also allows developers to use the expressiveness of a general-purpose programming language for provisioning infrastructure. Currently, there are three "industrial-grade" PL-IaC solutions available.

- The Cloud Development Kit for AWS (AWS CDK) [7]
- The CDK for Terraform (CDKTF) and [34]
- Pulumi [20]

All three allow developers to define the target state of their Cloud infrastructure to be defined in programming languages, like TypeScript, Java, or Python. However, only Pulumi lets developers use post-deployment state [112]. This means that only in Pulumi programs, developers can use values in their programs, which are only calculated during and available after the deployment of a Cloud resource. Consequently, AWS CDK and CDKTF do not leverage all of PL-IaCs capabilities.

2.7 Pulumi

Pulumi is not a monolithic program. Rather, it is a collection of smaller programs used to generate language-specific software development kits (SDKs), which can, in turn, be used by developers to manage their infrastructure. To understand which parts of the Pulumi ecosystem have to be adjusted in order to facilitate better type safety, we first have to understand how these components interact with each other.

In contrast to IaC tools like Terraform, Pulumi does not rely on a custom configuration language to deploy infrastructure components. Rather, Pulumi provides SDKs for a list of supported programming languages and software development ecosystems. At the time of writing, these are NodeJS, Python, Go, Java,.NET, and YAML. Language-specific SDKs are called **provider packages** or **providers**.

2.7.1 Providers

Providers offer language-specific and idiomatic APIs for developers to deploy infrastructure. Most importantly, for this thesis, they also provide static types for these infrastructure components. Therefore, tinkering with the Pulumi type system means adjusting the types supplied by the language-specific Pulumi Providers. Here is a minimal example of a Pulumi program using the Java Pulumi provider:

```java
package myproject;

import com.pulumi.Pulumi;
import com.pulumi.aws.s3.Bucket;

public class App {
    public static void main(String[] args) {
        Pulumi.run(ctx -> {

            // Create an AWS resource (S3 Bucket)
            var bucket = new Bucket("my-bucket");

            // Export the name of the bucket
            ctx.export("bucketName", bucket.bucket());
        });
    }
}
```

Listing 2.7: Minimal Pulumi example for Java. This programs creates a new AWS S3 bucket called "my-bucket"

Notice in Listing 2.7 that in Pulumi infrastructure components, such as AWS S3 buckets, are formally described as classes in Java. Pulumi establishes static type safety by providing programming language constructs, like Classes or Structs for infrastructure components managed by Pulumi. Parameters to these resources are typically described by a static type, such as an integer or string. In the Java example, the Pulumi.run(...) function creates infrastructure resources corresponding to the objects created in the Lambda expression body.

```java
@ResourceType(
    type = "aws:s3/bucket:Bucket"
)
public class Bucket extends CustomResource {
    @Export(
        name = "arn",
        refs = {String.class},
        tree = "[0]"
    )
    private Output<String> arn;
    ....
    }
```

Listing 2.8: Snippets of the Class definition for com.pulumi.aws.s3.Bucket

Listing 2.8 shows parts of the Class definition for AWS S3 buckets. All of the parameters shown are of type String, wrapped in Pulumi's own generic type **Output**. The types Pulumi offers are generated by the code generation capabilities of a component of the Pulumi ecosystem, named CrossCode [21]. The source of code generation is the **Pulumi Package Schema** or **Pulumi Schema**.

2.7.2 Pulumi Schema

The Pulumi Schema [25] is a formal definition of infrastructure components, functions, and types of resources and their parameters. It also supports language-specific extensions to manipulate code generation for different target languages. A package definition is a JSON file that validates and conforms to this Pulumi Schema.

```
{
  "name": "aws-native",
  "displayName": "AWS native",
  "resources": {
    "aws-native:s3:Bucket": {
      "properties": {
        ...
        "arn": {
          "type": "string",
          "description": "The Amazon Resource Name (ARN) of the specified
              bucket."
        },
        ...
      }
    }
  },
  "types": {
    ...
    "aws-native:s3:BucketAccelerateConfiguration": {
      "properties": {
        ...
        "accelerationStatus": {
          "$ref": "#/types/aws-native:s3:
              BucketAccelerateConfigurationAccelerationStatus",
          "description": "Configures the transfer acceleration state for an
               Amazon S3 bucket."
        }
      },
      "type": "object",
    },
    ...
  }
}
```

Listing 2.9: Excerpt from the Pulumi Schema for the AWS native provider.

Listing 2.9 shows part of the Pulumi Schema from which the AWS native provider packages are generated. Using CrossCode to generate code from this Schema will yield the code from the previous Listing 2.8. With a Schema, package developers can define the following properties about their provider packages:

- Package Information, like name, authors, homepage URL, etc.
- Package Metadata

- Infrastructure resource definitions and
- Type definitions

The type definitions adhere to the Pulumi type system. At the time of writing, this type system supports **Primitive Types**, like boolean, integer, number, string, and array, and **Complex Types** or object types for nested type definitions. Complex Types also allow Schema developers to define a type as enum, as can be seen in Listing 2.10.

```
"types": {
  "aws-native:s3:BucketDefaultRetentionMode": {
    "type": "string",
    "enum": [
      {
        "name": "Compliance",
        "value": "COMPLIANCE"
      },
      {
        "name": "Governance",
        "value": "GOVERNANCE"
      }
    ]
  },
}
```

Listing 2.10: Complex Type definition for a property of an AWS S3 bucket.

Listing 2.11 shows the (abbreviated) central type information specification of the Pulumi Schema. At the core, the Pulumi Schema distinguishes five kinds of types.

- **Primitive Types** are not subject to any further indirection and manifest in the types boolean, integer, number and string.
- **Array Types** represent arrays of items with the same type.
- **Map Types** represents maps, mapping strings to values of arbitrary type.
- **Named Types** are types that use the ref property to reference types declared elsewhere in the specification.
- **Union Types** combine multiple types. Values can be of either type included in the union type.

```
{
    ...
    "$defs": {
        "typeSpec": {
            "title": "Type Reference",
            "type": "object",
```

```
7               ...
8               "oneOf": [
9                   {
10                      "title": "Primitive Type",
11                      ...
12                      "type": "object",
13                      "properties": {
14                          "type": {
15                              "description": "The primitive type, if any",
16                              "type": "string",
17                              "enum": ["boolean", "integer", "number", "
                                    string"]
18                          },
19                          "additionalProperties": false,
20                          "items": false,
21                          "oneOf": false,
22                          "$ref": false
23                      },
24                      "required": ["type"]
25                  },
26                  {
27                      "title": "Array Type",
28                      ...
29                      "type": "object",
30                      "properties": {
31                          "type": {
32                              "const": "array"
33                          },
34                          "items": {
35                              "description": "The element type of the array",
36                              "$ref": "#/$defs/typeSpec"
37                          },
38                          "additionalProperties": false,
39                          "oneOf": false,
40                          "$ref": false
41                      },
42                      "required": ["type", "items"]
43                  },
44                  {
45                      "title": "Map Type",
46                      ...
47                  },
48                  {
49                      "title": "Named Type",
50                      ...
51                  },
52                  {
53                      "title": "Union Type",
54                      ...
55                  }
56              ]
57          },
58      }
```

```
59      ...
60  }
```

Listing 2.11: Type Definition Structure in Pulumi Schema.

Based on the type definitions in the Pulumi Schema, we can already draw some conclusions about the expressibility of the Pulumi type system.

1. If the target language supports Enums, they can be generated from the Schema.
2. Types cannot be further refined by logical constraints, i.e. refinement types can currently not be generated from the Schema.
3. An idiomatic extension of the type system of Pulumi will involve adjusting the Pulumi Schema and the code generation capacities of CrossCode, such that it supports the extensions in the Schema.

To make matters more complicated, there are different flavours for some large providers. Amazon Web Services (AWS), as well as Microsoft Azure and Google Cloud Provider (GCP) providers, are available as "classic" and "native" variants. For a Cloud infrastructure provider $i \in I = \{AWS, Azure, GCP\}$, c_i denotes the classic, and n_i denotes the native variant of a Pulumi provider for i. Table 2.7 gives an overview of the Pulumi providers for AWS, Azure and GCP and their development status.

Provider	**Name**	**Status**	**Initial Commit**
c_{AWS}	pulumi-aws [40]	Productive	17.07.2017
c_{Azure}	pulumi-azure [42]	Deprecated	11.09.2017
c_{GCP}	pulumi-gcp [44]	Productive	17.07.2017
n_{AWS}	pulumi-aws-native [41]	Preview	04.11.2019
n_{Azure}	pulumi-azure-native [43]	Productive	24.02.2019
n_{GCP}	pulumi-google-native [45]	Preview	22.12.2020

Table 2.7: Pulumi Provider Overview

2.7.3 Classic Providers

Historically, the classic provider variants are the older, in part deprecated versions of provider packages for large Cloud providers. Although c_{GCP} and c_{AWS} are still recommended providers for their respective Cloud environments, c_{Azure} is already deprecated in favor of n_{Azure}. What distinguishes the classic provider c_i from the native provider n_i is the source from which the Pulumi Schema and, therefore, the providers are generated. Every classic provider c_i generates its SDKs from Terraform providers. Figure 2.10 shows the high-level architecture of how provider packages and Pulumi Schema are created in classic providers.

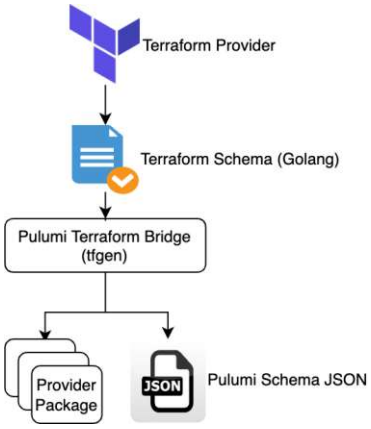


Figure 2.10: High Level Architecture of Pulumi classic provider package generation.

A software component in the Pulumi ecosystem, called the **Pulumi Terraform Bridge** [46] accepts Terraform provider metadata as input and generates a Pulumi Schema file or Pulumi provider packages in supported languages. Consequently, the types provided by c_i depend on the type information of the underlying Terraform schema.

Terraform Schema

The Terraform Schema [19] supports, similar to the Pulumi Schema, complex types called **Aggregate Types**, like maps, lists, and sets, and **Primitive Types**, like `boolean`, `integer` and `string`. A noteworthy difference, however, is the usage of validation functions in Terraform Schema type definitions.

```
1 type SchemaValidateFunc func(interface{}, string) ([]string, []error)
```

Listing 2.12: Go function signature for Terraform Schema validation function.

Listing 2.12 shows the function signature of validation functions. A validation function accepts a generic object and a string and returns a list of warnings and errors. An empty list of errors can be interpreted as a successful validation. By leveraging these validation functions, it is possible to describe complex types. For example, Listing 2.13 shows the definition of the `string` property `expiration`. The value of `expiration` must validate successfully against the `validation.IsRFC3339Time(...)` function.

```
1 "expiration": {
2   Type:         schema.TypeString,
3   ValidateFunc: validation.IsRFC3339Time,
4 },
```

Listing 2.13: Type definition for a String in date-time format.

The primitive types and validation functions from the Terraform Schema effectively form refinement type definitions. However, since the Pulumi Schema does not support logical

constraints on type definitions, the type information of the validation functions is lost. Interestingly, even validation functions that could be represented in Pulumi Schema are ignored.

```
"security_posture_config": {
    Type:         schema.TypeList,
    Optional:     true,
    MaxItems:     1,
    Computed:     true,
    Description: `Defines the config needed to enable/disable features for
        the Security Posture API`,
    Elem: &schema.Resource{
        Schema: map[string]*schema.Schema{
            "mode": {
                Type:             schema.TypeString,
                Optional:         true,
                Computed:         true,
                ValidateFunc:     validation.StringInSlice([]string{"
                    DISABLED", "BASIC", "MODE_UNSPECIFIED"}, false)
            }
        },
    },
},
```

Listing 2.14: Schema for GCP resource with enums properties.

Listing 2.14 shows a type definition in Terraform schema. The property mode is a string and can take one of three possible values, DISABLED, BASIC and MODE_UNSPECIFIED. As discussed in Section 2.7.2, enums can be represented in Pulumi Schema. However, the generated Pulumi Schema does not define mode in terms of enums, as can be seen in Listing 2.15.

```
"gcp:container/ClusterSecurityPostureConfig:ClusterSecurityPostureConfig":
    {
 "properties": {
   "mode": {
    "type": "string",
    "description": "Sets the mode of the Kubernetes security posture API's
        off-cluster features. Available options include `DISABLED` and `
        BASIC`.\n"
    },
  }
}
```

Listing 2.15: Pulumi Schema generated from enums property.

2.7.4 Native Providers

In contrast to their classic variants c_i, native providers n_i are not generated from Terraform providers. Instead, they use Cloud provider APIs and registries directly to generate Pulumi packages, in theory, enabling same-day delivery of resource additions

and changes. Notably, neither c_{AWS}, c_{Azure}, nor c_{GCP} define enums, while their native provider variants offer **2385**, **2601**, **1616** definitions [124] of enum type, respectively. In classic providers, these enums are formalised as simple strings, without any further static validation. However, currently, only the native variant of the Azure Pulumi provider n_{Azure} [43] is recommended for production usage, while native providers n_{GCP} and n_{AWS} are in preview. One reason for this is that especially for AWS and GCP, the native providers tend to lack the support for some resources, which are available in the corresponding classic provider. This will be further discussed in Section 6.2. Figure 2.11 shows which Cloud provider APIs are used to generate Pulumi provider packages and the Pulumi Schema.

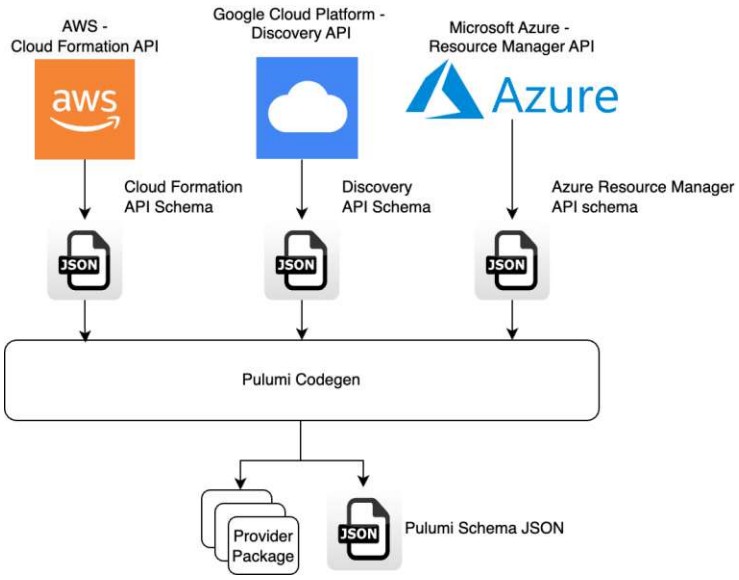


Figure 2.11: High Level Architecture of Pulumi native Provider package generation.

Pulumi's code generation package uses the AWS CloudFormation API [4], the Azure Resource Manager API [5] and the GCP Discovery API [9] to generate the provider packages and the Pulumi Schema. Since native providers n_i generate their Schema and provider packages from these APIs directly, the resources and types are directly dependent on the expressibility of the APIs. Cloud provider APIs are all based on a common specification language.

2.8 Cloud Provider APIs

The APIs on which Pulumi native providers for AWS, Azure and GCP are built share a common original specification language, the JSON Schema.

2.8.1 JSON Schema

The API specifications of AWS, Azure and GCP are all directly, or indirectly, based on the JSON Schema specification [130]. JSON Schema comes in many revisions, and the API specifications of the three Cloud providers are based on different versions of the JSON Schema specification. However, all versions share the following primitive types.

- array
- boolean
- integer
- number
- null
- object
- string

In addition to primitive types, the validation of the JSON Schematic defines two kinds of type refinements, **formats** and **validations**.

Formats

The Format Attribute is used for the semantic validation of values, based on some well-defined, potentially external, specification. For example, the **date−time** format defines that a string must adhere to the **date−time** format, as defined in RFC 3339 [94]. JSON Schema validation defines the following formats **date−time**, **email**, **hostname**, **ipv4**, **ipv6**, **uri** and **Password**. OpenAPI extends this list by formats **int32**, **int64**, **float** , **double**, **byte** and **binary**.

Validations

Validations are additional constraints on datatypes. Examples are listed below.

- multipleOf
- maximum
- exclusiveMaximum

2.8.2 AWS Cloud Formation

The AWS Cloud Formation API [4] is a proprietary, JSON-formatted text file that defines Cloud resources and their types. Because resource availability varies depending on the geographical location, AWS provides API specifications per region. The type definitions and validation properties are based on Draft 7 [14] of the JSON Schema. Therefore, it

inherits its primitive types, such as booleans, integers, numbers, and strings. Additionally, the schema allows one to specify further constraints on the properties, such as maximum, minimum, format or enum.

2.8.3 Azure Resource Manager API

Azure Resource Manager [33] provides OpenAPI (formerly known as Swagger) specification [17] for the description of their REST API. Currently, only the version **v2.0** of the specification is supported. OpenAPI and its types are based on JSON Schema Draft 4 [68]. Therefore, Resource Manager type definitions are limited by the expressiveness of OpenAPI. Resource Manager supports integer, number, string, and boolean types. Furthermore, some types support additional format constraints, like int32, or date−time. Enums are also supported.

2.8.4 GCP Discovery API

GCP's Discovery API [9] is defined as a proprietary JSON format based on JSON Schema Draft 3 [130]. It supports all types and validations defined in the JSON Schema but defines its own set of format attributes, like **google−datetime**, while omitting formats like **regex**. The type support of the Discovery API ranges from primitive types like boolean, integer, number, and string. Enums are also supported, as well as validation keywords like maximum and minimum.

2.8.5 Summary

Table 2.8 summarises details about the Cloud provider APIs and their corresponding underlying API specifications. Note that all Cloud provider APIs are based on some version of the JSON Schema. AWS and GCP provide their own proprietary API specification format, based on JSON Schema versions, while Azure provides API specification in the form of OpenAPI specification files, which are in turn also based on JSON Schema.

Cloud	API	Specification	JSON Schema Version
AWS	Cloud Formation	proprietary	Draft 7
Azure	Resource Manager	OpenAPI version 2	Draft 4
GCP	Discovery	proprietary	Draft 3

Table 2.8: Cloud Provider APIs and their corresponding underlying API specification formats

CHAPTER 3

Finding and Categorising Runtime Errors

In this chapter, we establish error categories based on runtime errors found in GitHub Issues. We show later that all errors in a category can be mitigated by the same kind of static type.

3.1 Motivation and Challenges

To do a systematic analysis of runtime errors that occur in IaC programs, we search through GitHub repositories and related issues to find runtime errors related to typing. Our goal is to find runtime errors in the IaC programs that could have been mitigated with more restrictive types. A rigorous search that would establish ground truth for all runtime errors that occur in IaC programs is a challenging task, even if we reduce the search to include only Pulumi repositories on GitHub. This is because **(1)** finding runtime errors is difficult to automate, **(2)** errors might have been removed from the programs before ever being published on GitHub, and **(3)** the search space is vast due to the following dimensions:

1. The number of repositories to be searched for errors.
2. The number of Pulumi projects within a repository.
   For example, Pulumi has an “examples” repository containing hundreds of Pulumi projects.
3. The number of commits in a repository.
   Some repositories contain thousands of commits. Errors might have been introduced or fixed in earlier commits of the repository.

3.2 Methodology

Executing the IaC programs inside the found repositories to (automatically) find runtime errors is also not feasible. Presumably, many programs would run correctly, leading to the creation of various Cloud resources, resulting in unacceptable monetary costs. Therefore, a manual search is necessary.

Pulumi's own repositories are a good place to start looking for functional examples of Pulumi projects. In addition to the Pulumi examples repository, almost all Pulumi repositories tend to have a folder with examples for the specific Cloud provider or library. In total, we analyse **5** Pulumi repositories and their GitHub Issues to find runtime errors related to types. The GitHub Issues in the analysed Pulumi repositories mostly have common labels that helped to narrow down the search. Most repositories define a **kind/bug** label to mark GitHub Issues as bugs, which helps to narrow down GitHub Issues to only include those related to erroneous behaviour.

3.3 Runtime Error Categories

Using the methodology described previously, we find **7** different type-related runtime errors in **5** repositories, which will act as case studies for the remaining thesis. Table 3.1 shows the GitHub repositories, the label we used for a specific repository to narrow down the search space, the GitHub issue numbers, and the category in which we place the error.

Repository	Issue Labels	Issue Numbers	Category
Pulumi Examples [39]	**kind/bug**	1427	**Enumerations**
		1392	**Refinements**
Pulumi Templates [47]	**kind/bug**	501	**Dependencies**
Pulumi Azure [42]	**kind/bug**	615	**Refinements**
Pulumi Azure Native [41]	**kind/bug**	3414	**Dependencies**
		3074	**Dependencies**
Pulumi GCP [44]	**kind/bug**	813	**Refinements**

Table 3.1: GitHub Issues for Runtime Errors in Pulumi Repositories

The errors can be categorised into three different categories of increasing complexity. Coincidentally, the errors in the three categories naturally correspond to errors that can be mitigated using static type techniques. Errors in category **Enumerations** map naturally to **enums**, runtime errors in the category **Refinements** can be mitigated statically through **refinement types** and errors of category **Dependencies** correspond to **dependent types** in static type systems.

3.3.1 Enumerations

This error class describes issues where the static type of a parameter allows arbitrary string values, despite the fact that only a well-defined, finite set of values is allowed. This finite set of values can commonly be formalised as **enums** in many languages, such as Java or Python. Cloud providers might allow only a certain set of values for a parameter, while the type system in an IaC program allows any string. For example, Listing 3.1 shows (part of) the Pulumi program that causes GitHub issue number 1427 in the Pulumi examples repository. The program tries to configure the cluster arguments for a Kubernetes cluster with version **1.18.14** in Azure. However, at the time of running the program, Azure no longer supported Kubernetes version **1.18.14**.

```
...
managed_cluster = containerservice.ManagedCluster(
    managed_cluster_name,
    resource_group_name=resource_group.name,
    agent_pool_profiles=[{
        "count": 3,
        "max_pods": 110,
        "mode": "System",
        "name": "agentpool",
        "node_labels": {},
        "os_disk_size_gb": 30,
        "os_type": "Linux",
        "type": "VirtualMachineScaleSets",
        "vm_size": "Standard_DS2_v2",
    }],
    enable_rbac=True,
    kubernetes_version="1.18.14",
    ...
```

Listing 3.1: Outdated Kubernetes Version

Issues like this could be mitigated if the type system of the program would define the set of valid Kubernetes versions as enums, instead of arbitrary an string.

3.3.2 Refinements

Some parameters of Cloud resources do not require a certain set of allowed values, but rather have limitations on the format or require additional constraints to be fulfilled. This includes parameters like IP ranges in CIDR notation or Cloud resource names. For example, Listing 3.2 shows the Pulumi code that caused GitHub Issue number 813 in the Pulumi GCP provider. GCP HTTP health checks are Cloud resources that require that their names have a certain format. In this instance, the resource name must match the regular expression (regex) **'(?:[a-z](?:[-a-z0-9]0,61[a-z0-9])?)'**. Since the author of the GitHub issue did not name the resource explicitly, Pulumi automatically generated a name that did not match the regex. However, no static error was produced, since the generated name matched the declared type, string, of HTTP health check names.

```
1  import * as pulumi from "@pulumi/pulumi";
2  import * as gcp from "@pulumi/gcp";
3
4  export = async () => {
5
6      const defaultHttpHealthCheck = new gcp.compute.HttpHealthCheck("
           defaultHttpHealthCheck", {
7          // name: "abs123",
8          requestPath: "/",
9          checkIntervalSec: 1,
10         timeoutSec: 1,
11     });
12     return {
13
14     }
15 }
```

Listing 3.2: Invalid HTTP Health Check Name in GCP

Validating parameters in this category against a known set of valid values would not work since, theoretically, there might be an infinite number of them. Depending on the data type of the parameter, additional conditions must be in place to verify that a value is valid. For string, this might be a regular expression. For integer, this might be an interval that contains all valid values. In general, the types of parameters in this category can be defined by two aspects.

- A base type, like string to integer.
- A predicate, which further restricts the base type.

3.3.3 Dependencies

In some cases, the valid values of the parameters are dependent on the values of other parameters. For example, Listing 3.3 shows part of the Pulumi program that caused GitHub issue number 501 in the Pulumi templates repository. The creation of a Kubernetes cluster in AWS led to a runtime error, even though all parameters had valid values according to the AWS API description and documentation. Although every individual value used to create the Kubernetes cluster was valid, the combination of two parameters was not.

In particular, Cloud providers might offer different machine types for their VM instances depending on the geographical region the VMs should be deployed to. In this case, the developer tried to create a Kubernetes cluster, in which the underlying VMs should be deployed to the region **eu-north-1** with machine types **t2.medium**.

```
1  config:
2    aws:region: eu-north-1
3    ...
```

```
4   hello-eks:eksNodeInstanceType: t2.medium
5   ...
```

Listing 3.3: Invalid parameter combination. The region eu-north-1 does not supported VMs of type t2.

At the time, AWS did not support VMs of type **t2** in region **eu-north-1**, making the combination of the parameter values invalid.

CHAPTER 4

A Pulumi Repository Dataset

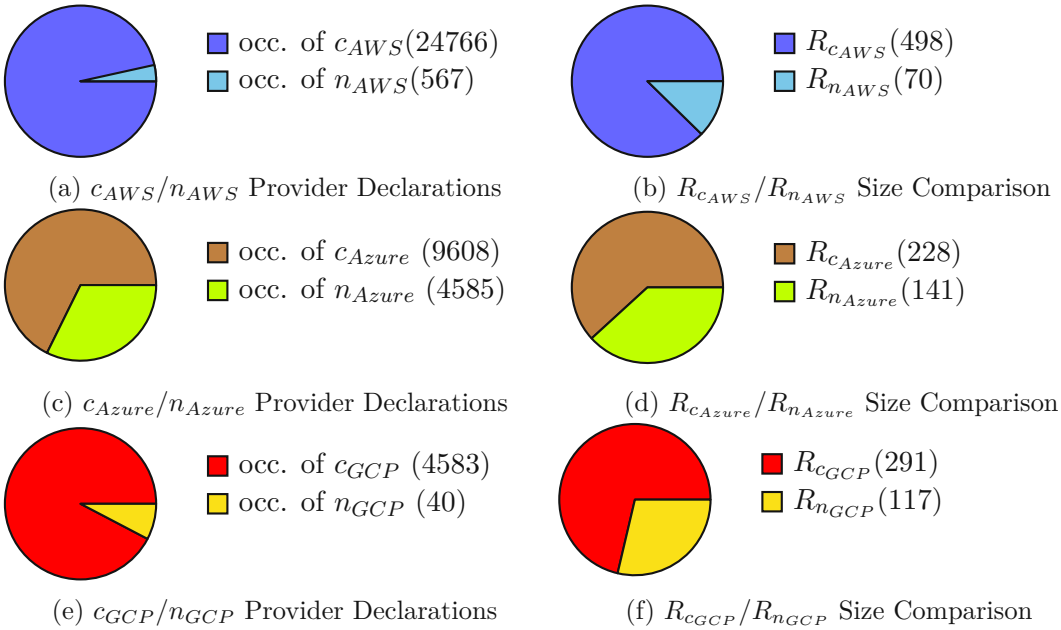
Because the source of initial type information for the Cloud infrastructure provider i depends on whether c_i or n_i is used, the strategies to improve static type safety vary between them. To assess the impact of improving the static types of either variant of provider and to quantify how big the impact of improving static type safety in general would be, we create a dataset of Pulumi repositories from public GitHub repositories.

4.1 Methodology and Dataset

GitHub offers a vast REST API [31] for searching repositories based on various criteria. Every Pulumi project requires a project file, that is, a YAML file containing metadata about the Pulumi project. This project file must be named *Pulumi.[yml/yaml]*. Therefore, Pulumi repositories can be found using the GitHub API by searching repositories containing a *Pulumi.yaml* or *Pulumi.yml* file. Conducting such a search results in a dataset of **1023** Pulumi repositories. All scripts used for our analysis can be found in Table 7.2 in the appendix (see Chapter 7) and the list of repositories is publicly available in Google Sheets [123].

4.2 Provider Usage

Analysing the dataset, we calculate the occurrences of dependency declarations for each n_i and c_i. Let R be our dataset. Then $R_{c_i} \subseteq R$ and $R_{n_i} \subseteq R$ are the subsets of repositories in which the dependency declarations for c_i and n_i can be found, respectively. We found **24766**, **9608** and **4581** occurrences of dependency declarations for c_{AWS}, c_{Azure}, and c_{GCP} in **498**, **228** and **291** repositories, respectively. For native providers, we found **567**, **4585** and **381** occurrences of dependency declarations for n_{AWS}, n_{Azure}, and n_{GCP} in **70**, **141** and **117** repositories, respectively.



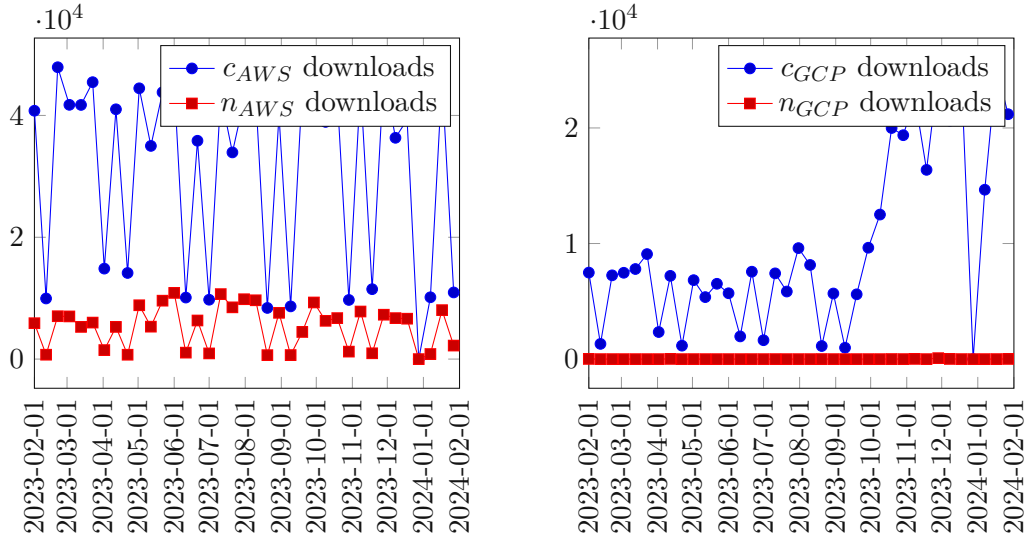
(a) c_{AWS}/n_{AWS} Provider Declarations (b) $R_{c_{AWS}}/R_{n_{AWS}}$ Size Comparison

(c) c_{Azure}/n_{Azure} Provider Declarations (d) $R_{c_{Azure}}/R_{n_{Azure}}$ Size Comparison

(e) c_{GCP}/n_{GCP} Provider Declarations (f) $R_{c_{GCP}}/R_{n_{GCP}}$ Size Comparison

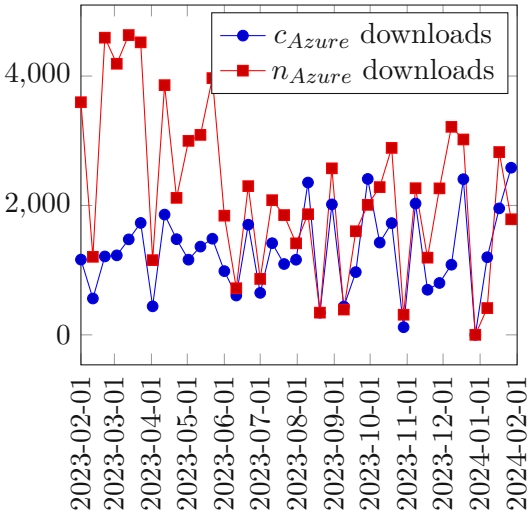
Figure 4.1: Classic Provider Compared to Corresponding Native Provider Usage in our Dataset

While distribution between classic and native providers varies between AWS, GCP, and Azure, the native providers still appear significantly less frequent. Although n_{Azure} is the recommended Azure provider, it is still only used in roughly 32.3% of all repositories using Azure providers. Analysing the use of Pulumi provider NPM [16] packages reveals a similar picture. NPM provides a public API [24] to retrieve the download counts of packages in a certain time range. Figure 4.2 shows a comparison of the number of downloads between classic and the respective native providers. The numbers indicate a much greater adoption of c_{GCP} and c_{AWS} compared to their native variants. This is almost certainly related to the fact that n_{Azure} is still the only native provider recommended for production use.



(a) AWS Provider Downloads

(b) GCP Provider Downloads



(c) Azure Provider Downloads

Figure 4.2: Provider Downloads According to the NPM Public API

When looking at claims of often superior static type safety and same-day delivery of native providers, one could argue that switching to native providers could already solve a lot of type-related runtime errors. However, our analysis indicates that classic providers are significantly more prevalent than native providers. Therefore, enhancing static type safety in classic providers is likely to yield a more substantial impact than exclusively focusing on native providers.

4.3 Enum Usage Analysis

As discussed in Section 2.7.4, native provider variants offer enum definitions that are missing in classic provider variants. The large number of enum definitions shows the potential for improving static type safety. To quantify the impact of using n_i instead of c_i, we will investigate the potential impact within our dataset. We do this by searching for occurrences of string values in c_i, which would likely be replaced by the use of an enum in the corresponding native provider n_i.

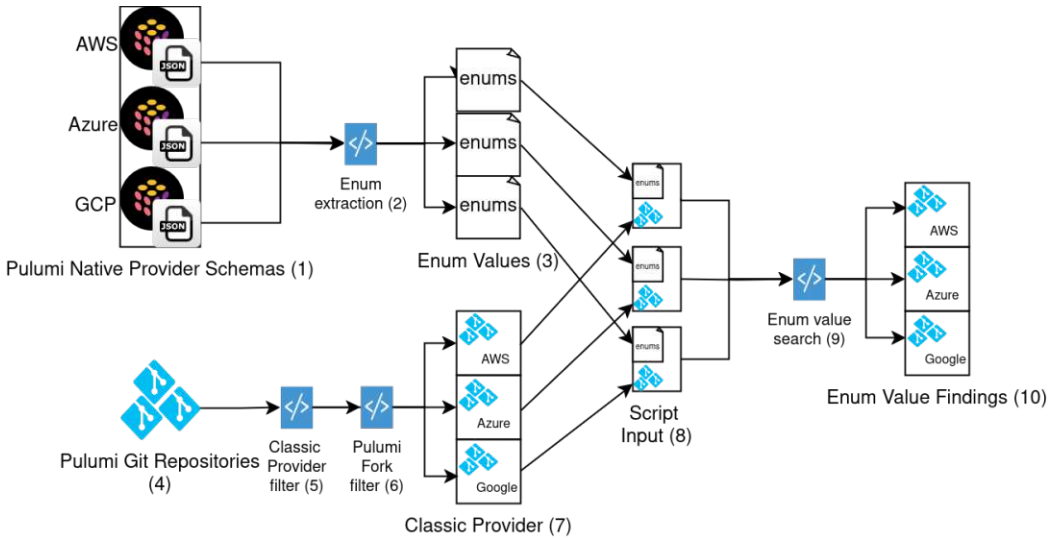


Figure 4.3: Enum Usage Analysis Process

Since our dataset of 1023 Pulumi repositories with thousands of different Pulumi projects is too large for manual migration and analysis, we first have to filter the repositories for potential migration candidates. To accomplish this, we extract all possible enum values from the native providers and perform a full-text search of these values in the repositories. The occurrence of an enum value in a Pulumi project using c_i indicates that after upgrading to n_i, an enum value can be used instead. If, for example, a Pulumi project using c_{AWS} contains the string "POST", then it could be because it uses a property, which is formulated as an enum in n_{AWS}. The process of filtering the Pulumi repositories in this way is visualised in Figure 4.3 and explained below.

1. For every n_i where $i \in I$, the respective Pulumi Schema $schema_{n_i}$ contains all type definitions for n_i, including all possible enum values for n_i: $enum_{n_i}$.

2. For all n_i we extract $enum_{n_i}$ and produce the unique set of all possible enum values for i: $enum_{n_i}^{uniq}$.

3. All $enum_{n_i}^{uniq}$ values are then passed as input to step 8.

4. Our dataset contains 1023 repositories. Let the set of these repositories be R where $|R| = 1023$.

5. This step yields R_{c_i}, the subsets of repositories, containing dependency declarations of classic providers for each i.

6. The dataset contains a lot of forks of Pulumi repositories, including the native provider repositories. Since we are only interested in "real-world" applications using Pulumi, we identify repositories that were forked from Pulumi repositories and remove them from the set. The resulting sets $R'_{c_{AWS}}$, $R'_{c_{Azure}}$ and $R'_{c_{GCP}}$ are smaller with **207**, **31** and **75** repositories respectively.

7. For all $i \in I$ the tuple $(enum^{uniq}_{n_i}, R'_{c_i})$ is used as input to step 8.

8. All possible enum values of n_i are combined with all repositories containing references to c_i for all $i \in I$.

9. The individual combinations of step 8 are used as input for step 9, which searches for occurrences of the enum values in the repositories. This step filters out false positives. For example, it ignores findings in included "node_modules" folders, which are included in some repositories.

10. The resulting sets $R^{enum}_{c_i} \subseteq R'_{c_i} \mid i \in \{AWS, Azure, GCP\}$ are the sets of repositories containing dependency declarations for c_i and at least one enum value of $enum^{uniq}_{n_i}$. There are **13052**, **1912** and **1389** enum findings in **204**, **31** and **69** repositories for AWS, Azure and GCP respectively.

Table 4.1 shows that out of the 1023 repositories 498, 228, and 291 repositories use the classic provider variants of AWS, Azure and GCP respectively. However, only 207(41.56%), 31(13.6%), and 75(23.71%) of those repositories are no forks from Pulumi repositories. Out of these 204, 31, and 69 non-Pulumi repositories almost all (204, 31, and 69) repositories contain at least one string value that corresponds exactly to the value of an enum in the corresponding native provider. In total, of the 1023 repositories 19.94%, 3.03%, and 6.74% of them use classic providers for AWS, Azure and GCP respectively and contain enums values.

Cloud	$\lvert R\rvert$	$\lvert R_{c_i}\rvert$	$\lvert R'_{c_i}\rvert$	$(\lvert R'_{c_i}\rvert/\lvert R_{c_i}\rvert) \times 100$	$\lvert R^{enum}_{c_i}\rvert$	enums in $R^{enum}_{c_i}$	$\lvert R^{enum}_{c_i}\rvert/\lvert R\rvert$
AWS	1023	498	207	41.56	204	13052	19.94
Azure	1023	228	31	13.6	31	1912	3.03
GCP	1023	291	75	23.71	69	1389	6.74

Table 4.1: Overview of the individual results of the enum usage analysis

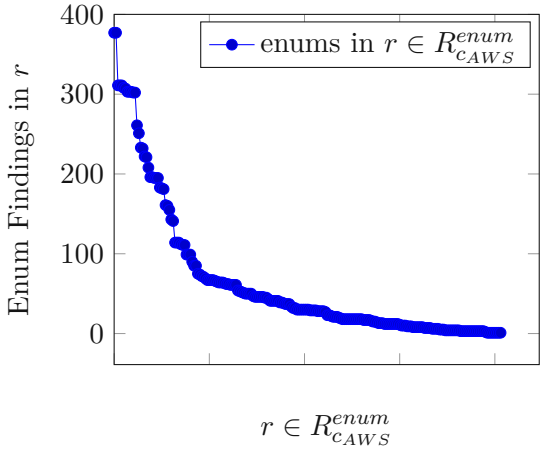
Figure 4.4 shows the distribution of enum values in individual Pulumi repositories. Each data point represents the number of enum value findings in repository $r \in R^{enum}_{c_i}$. Generally our findings reveal the following insight.

1. The number of enum findings is not distributed equally, but rather follows a power law distribution. That is, a relatively small number of repositories contain the majority of enum findings. The rest of the repositories each contain only a "small" amount of enum values, leading to a so-called "long tail", as can be seen in Figure 4.4. Each data point represents the number of enum findings in an individual repository $r \in R_{c_i}^{enum}$.

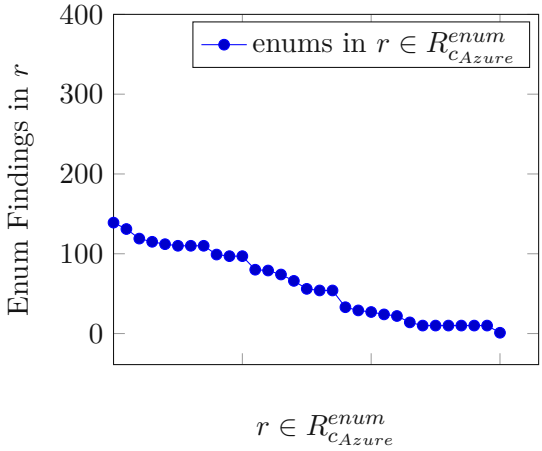
2. 48.68%, 22.29% and 28.45% of the repositories in our dataset use AWS, Azure, and GCP classic providers, respectively.

3. 41.56%, 13.6% and 23.71% of repositories with classic provider dependency declarations from AWS, Azure, and GCP respectively in our dataset were no Pulumi repositories forks.

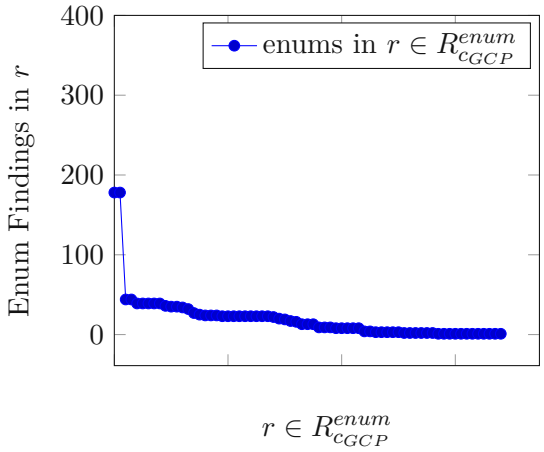
4. 19.94%, 3.03% and 6.74% of repositories use classic providers for AWS, Azure and GCP, respectively, are not forked from Pulumi repositories **and** contain enum values that are equal to predefined enum values in the corresponding native providers.



(a) AWS enum findings



(b) Azure enum findings



(c) GCP enum findings

Figure 4.4: Enum Findings in $R^{enum}_{c_i}$

CHAPTER 5

Mitigating Errors

In order to mitigate runtime errors in all three error categories **Enumerations**, **Refinements**, and **Dependencies** statically, static type information for enums, refinement types, and dependent types must be present at three levels, the Cloud provider, the intermediate type representation of the IaC tool, and the target programming language. For Pulumi, this means that type information has to be present in the following three points in the Pulumi SDK generation pipeline.

1. **The source APIs**: This includes the Cloud providers APIs, on which Pulumi native providers are based on, as well as the Terraform Schema, on which Pulumi classic providers are based on.

2. **The Pulumi Schema**: Since Pulumi SDKs are generated from the Pulumi Schema specification file, static type information has to be present in the Pulumi Schema.

3. **The target Programming Language**: Depending on the error category, enums, refinement types, or dependent types must be supported by the target programming languages, either natively or via third-party extensions.

The following sections each address these three levels. Depending on whether the necessary static type information is present at each level or not, we show how Pulumi programs can leverage the improved static types and discuss the impact they would have.

5.1 Mitigating Enumerations Errors

All three levels in the Pulumi SDK generation pipeline support enums, at least in parts.

5.1.1 Enum Type Information in the Source APIs

As discussed in Section 2.8, AWS CloudFormation, Azure Resource Manager, and GCP Discovery API are based on JSON Schema. Since JSON Schema natively supports the definition of enums, the three Cloud provider APIs do so as well. For example, consider the AWS CloudFormation type specification of an S3 bucket in Listing 5.1. The property AccessControl is defined as an enum.

```
{
  "typeName" : "AWS::S3::Bucket",
  "description" : "Resource Type definition for AWS::S3::Bucket",
  "additionalProperties" : false,
  "properties" : {
    "AccessControl" : {
      "description" : "This is a legacy property, [...]",
      "enum" : [ "AuthenticatedRead", "AwsExecRead", "
          BucketOwnerFullControl", "BucketOwnerRead", "LogDeliveryWrite", "
          Private", "PublicRead", "PublicReadWrite" ],
      "type" : "string"
    },
    }
  }
}
```

Listing 5.1: AWS CloudFormation API Description Snippet of a S3 Bucket

5.1.2 Enum Type Information in the Pulumi Schema

As presented in Section 2.7.2, the Pulumi Schema is also based on JSON Schema and, therefore, also supports enums natively. Taking the definition of S3 bucket AccessControl from the AWS CloudFormation API in Listing 5.1, Listing 5.2 shows the equivalent definition in the Pulumi Schema.

```
{
  "aws-native:s3:BucketAccessControl": {
  "type": "string",
  "enum": [
    {"name": "AuthenticatedRead","value": "AuthenticatedRead"},
    {"name": "AwsExecRead","value": "AwsExecRead"},
    {"name": "BucketOwnerFullControl","value": "BucketOwnerFullControl"},
    {"name": "BucketOwnerRead","value": "BucketOwnerRead"},
    {"name": "LogDeliveryWrite","value": "LogDeliveryWrite"},
    {"name": "Private","value": "Private"},
    {"name": "PublicRead","value": "PublicRead"},
    {"name": "PublicReadWrite","value": "PublicReadWrite"}
    ]
  }
}
```

Listing 5.2: Pulumi Schema Generated from AWS CloudFormation API

5.1.3 Enum Type Support in Pulumi Target Languages

While not all programming languages supported by Pulumi implement enums natively, Pulumi translates enums from the Pulumi Schema into idiomatic enums in the programming languages that do. Finalizing the example of the S3 bucket AccessControl type definition, Listing 5.3 shows the generated enum in the Java provider.

```
1  @EnumType
2  public enum BucketAccessControl {
3      AuthenticatedRead("AuthenticatedRead"),
4      AwsExecRead("AwsExecRead"),
5      BucketOwnerFullControl("BucketOwnerFullControl"),
6      BucketOwnerRead("BucketOwnerRead"),
7      LogDeliveryWrite("LogDeliveryWrite"),
8      Private("Private"),
9      PublicRead("PublicRead"),
10     PublicReadWrite("PublicReadWrite");
11 }
```

Listing 5.3: Java Code Generated Pulumi from AWS CloudFormation API

Table 5.1 summarises which programming languages supported by Pulumi support enums natively. Programming languages like C#, Java, TypeScript, and Python support enums natively, while Go and YAML do not.

Programming Language	**Enum Support**
.NET (C#)	✓
Go	×
Java	✓
NodeJS (TypeScript)	✓
Python	✓
YAML	×

Table 5.1: Enum Support in Programming Languages Supported by Pulumi

5.1.4 Impact

When it comes to mitigating errors in the **Enumerations** category (Section 3.3.1), it is only necessary to focus on classic providers, since native providers already generate enums idiomatically in the target SDKs for supported programming languages. We propose two different mitigation strategies: updating to the respective native provider, or enhancing the Pulumi Terraform Bridge to generate enums, where applicable. Both strategies present challenges.

Upgrading to Native Providers

There is no automated way [15] to migrate Pulumi resources from c_i to n_i. This means that developers must do manual work to change the underlying providers of their Pulumi

programs. Ignoring the additional benefits of native providers, such as same-day delivery and the potential risks of using them, due to their prevalent "preview" status, developers may be reluctant to commit to upgrading to native providers if there is no consequent benefit to static type safety. To benefit from enums, Pulumi programs have to manage resources which use the enum types provided by the native providers. For example, developers who manage AWS Lambdas with the AWS classic provider will benefit from upgrading to the AWS native provider.

```
"aws:lambda/FunctionUrlCors:FunctionUrlCors": {
    "properties": {
      "allowMethods": {
        "type": "array",
        "items": {
          "type": "string"
        },
        ...
    }
  }
}
```

Listing 5.4: Schema for AWS Lamda CORS Settings in the AWS Classic Provider

Listing 5.4 shows the definition of the allowMethods property. According to the type, any string is applicable. However, in practice, only valid HTTP methods [65] are allowed. The corresponding type definition of the AWS native provider in Listing 5.5 shows how this is already encoded directly into the type system.

```
"types": {
    "aws-native:lambda:UrlCors": {
                "properties": {
                    "allowMethods": {
                        "type": "array",
                        "items": {
                            "$ref": "#/types/aws-native:lambda:
                                UrlAllowMethodsItem"
                        },
                    },
                ..
                }
    }
    "aws-native:lambda:UrlAllowMethodsItem": {
        "type": "string",
        "enum": [
            {
                "name": "Get",
                "value": "GET"
            },
            {
                "name": "Put",
                "value": "PUT"
            },
            {
```

```
                    "name": "Head",
                    "value": "HEAD"
                },
                {
                    "name": "Post",
                    "value": "POST"
                },
                {
                    "name": "Patch",
                    "value": "PATCH"
                },
                {
                    "name": "Delete",
                    "value": "DELETE"
                },
                {
                    "value": "*"
                }
            ]
        },
}
```

Listing 5.5: Schema for AWS Lamda CORS Settings in the AWS Native Provider

Users of n_{AWS} can use the enums provided by the respective schema. Invalid values for the allowMethods property will therefore result in a static type error, instead of a runtime error. The results of the enum usage analysis in Section 4.3 quantify how many Pulumi projects would benefit from upgrading from classic to native providers.

Impact on the Dataset The above-presented example shows that upgrading to native providers can have distinct benefits on static type safety. Taking the analysis results from Section 4.3 into account shows how big the impact of upgrading to native providers would be on our dataset.

1. 48.68%, 22.29% and 28.45% of the repositories in our dataset use AWS, Azure, and GCP classic providers, respectively, and could therefore benefit from better static type safety by upgrading to the corresponding native providers.

2. 19.94%, 3.03% and 6.74% of repositories use classic providers for AWS, Azure and GCP, respectively **and** contain enum values that are equal to predefined enum values in the corresponding native providers and are therefore likely to benefit from the static type safety that enums provide.

Adding Enums to Classic Providers As we have shown in Section 2.7.3, Terraform providers, which are the foundation of the Pulumi classic providers, include type definitions using string as the base type and validation functions in the form of validation.StringInSlice() to denote enum types. This type information is, in part, lost during the conversion

from Terraform provider to the Pulumi Schema, because the validation functions are not considered during translation. In theory, it should be possible to enhance the **Pulumi Terraform Bridge** to respect this validation function, to generate enums in Pulumi classic providers, where applicable. The issue [46] has been recognised by Pulumi developers, although there is currently no implementation for this enhancement.

5.2 Mitigating Refinements Errors

In contrast to enums, refinement type information is not available at every point of the Pulumi SDK generation pipeline. However, we argue that on the level of the Pulumi Schema, static refinement type information can be added idiomatically. Although there are no native refinement type implementations in the supported programming languages, there are viable options developed by third parties.

5.2.1 Refinement Type Information in the Source APIs

As presented in Section 2.8, all APIs from the three analysed Cloud providers support the type information necessary for refinement types. Depending on the JSON Schema version the APIs are based on, the support for validation keywords, which can be translated to logical constraints for refinement types, varies. Table 5.2 lists all primitive types available in Cloud provider APIs. In addition to primitive types, properties might define further validation keywords, including the format property.

Type Name	**Supported By** AWS	Azure	GCP	**Description**
any	✓		✓	property can have any of the defined types
array	✓	✓	✓	array values must have one of the defined types
boolean	✓	✓	✓	true|false
file		✓		defined in OpenAPI spec
integer	✓	✓	✓	
null	✓	✓	✓	
number	✓	✓	✓	
object	✓	✓	✓	for nested types
string	✓	✓	✓	arbitrary string

Table 5.2: Primitive Types Supported by the Different Cloud Provider APIs

Table 5.3 shows all format values supported by the Cloud provider APIs. Notice that the AWS CloudFormation API supports the most format values, since it takes them directly from JSON Schema Draft 7, while GCP and OpenAPI version 2, which is the basis of Azure Resource Manager API, provide their own smaller set of allowed format values.

Primitive Type	**Format Value**	**Supported By** AWS	Azure	GCP
integer	int32		✓	
	int64		✓	
	uint32			✓
number	double	✓	✓	✓
	float	✓	✓	✓
string	binary	✓	✓	✓
	byte	✓	✓	✓
	date	✓	✓	✓
	date-time	✓	✓	✓
	email	✓		
	google-datetime			✓
	google-duration			✓
	google-fieldmask			✓
	hostname	✓		
	idn-email	✓		
	idn-hostname	✓		
	iri	✓		
	iri-reference	✓		
	ipv4	✓		
	ipv6	✓		
	int32			✓
	int64			✓
	json-pointer	✓		
	regex	✓		
	relative-json-pointer	✓		
	password	✓	✓	✓
	time	✓		
	uri	✓		
	uri-reference	✓		
	uri-template	✓		

Table 5.3: Format Values Supported by the Different Cloud Provider APIs

Primitive Type	**Validation**	**Supported By** AWS	Azure	GCP
integer|number	exclusiveMaximum	✓	✓	✓
	exclusiveMinimum	✓	✓	✓
	divisibleBy			✓
	maximum	✓	✓	✓
	minimum	✓	✓	✓
	multipleOf	✓	✓	
array	contains	✓		
	maxItems	✓	✓	✓
	minItems	✓	✓	✓
	uniqueItems	✓	✓	✓
string	enum	✓	✓	✓
	maxLength	✓	✓	✓
	minLength	✓	✓	✓
	pattern	✓	✓	✓

Table 5.4: Validations Supported by the Different Cloud Provider APIs

Table 5.4 shows all supported validation keywords and their base primitive types. Validation keyword support is very similar between all three Cloud providers. AWS supports the most validation keywords because it is based on the newest version of the JSON Schema specification. Note that **divisibleBy** has been renamed to **multipleOf** after JSON Schema Draft 3.

Terraform Schema

As presented in Section 2.7.3, Pulumi classic providers do not use the Cloud provider API descriptions directly to generate their SDKs, but rather are built on the Terraform providers for the respective Clouds. Terraform providers contain multiple validation functions [19, 29] that could be used to formulate the logical constraints necessary for refinement type definitions. The validation functions include

- string validations for maxLength and minLength
- string validations for pattern matching for specific formats like date–time, uuid or ipv4 and ipv6
- number validation for maximum, minimum or divisibleBy
- list validations for verifying that a list only has unique items

The examples listed above map well to the validations offered by JSON Schema and the Cloud providers APIs. This is not surprising, since the Pulumi native providers and the Terraform provider target the same Cloud provider APIs. Therefore, it should be possible to apply logical constraints in the form of Terraform validation functions to the Pulumi Schema, in a similar way, as with the validation keywords offered by the Cloud providers.

5.2.2 Refinement Type Information in the Pulumi Schema

As discussed in Section 2.7.2, the Pulumi Package Schema does not support validations and formats as presented in Section 5.2.1, although issues [1] for this improvement have already been raised. To bring the refinement type information to the Pulumi Schema, we propose extending the definitions for **Primitive Types** and **Array Types** in the #/defs /typeSpec property in the Pulumi Schema by validation and format properties. Listing 5.6 shows how primitive type definitions in the Pulumi Schema can be extended to support the validation attributes.

```
1 {
2     "$defs": {
3         "typeSpec": {
4             ...
5             "oneOf": [
6                 {
```

```
                    "title": "Primitive Type",
                    "type": "object",
                    "properties": {
                        "type": {
                            "description": "The primitive type, if any",
                            "type": "string",
                            "enum": ["boolean", "integer", "number", "
                                string"]
                        },
                        ...
                        "maximum": {
                            "description": "Maximum value",
                            "type": "number"
                        },
                        "format: {
                            "description": "Allowed format",
                            "type": "string"
                            "enum": ["date-time", "time", ...]
                        }
                    },
                    "required": ["type"]
                }
            }
        }
    }
}
```

Listing 5.6: Proposed Pulumi Schema Extension for Validations of Primitive Types

Array validation properties such as maxItems will have to be added to the respective array type definitions, as can be seen in Listing 5.7.

```
{
    "$defs": {
        "typeSpec": {
            ...
            "oneOf": [
                {
                    "title": "Array Type",
                    "type": "object",
                    "properties": {
                        "type": {
                            "const": "array"
                        },
                        "items": {
                            "description": "The element type of the array",
                            "$ref": "#/$defs/typeSpec"
                        },
                        ...
                        "maxItems": {
                            "description": "Maximum number of items allowed
                                in the array.",
                            "type": "integer"
```

```
                        }
                    },
                    "required": ["type", "items"]
                }
            }
        }
    }
}
```

Listing 5.7: Proposed Pulumi Schema Extension for Validations of Array Types

The proposed extensions would allow Pulumi Schema specification files to formulate the logical constraints provided by the Cloud provider APIs and make them available for Pulumi Codegen. Codegen could use the constraints to generate refinement types in the supported target languages, for which refinement type implementations exist.

5.2.3 Refinement Type Support in Pulumi Target Languages

Pulumi Codegen currently generates SDKs in five different programming languages (not counting YAML), **(1)** C#, **(2)** Go, **(3)** Java, **(4)** JavaScript, and **(5)** Python, none of which have native refinement type implementations. However, there are a handful of third-party implementations for these programming languages.

Third-Party Implementations

Using the GitHub API we found a set of refinement type implementations for programming languages for which Pulumi generates SDKs. Table 5.5 shows an overview of our findings. Some repository findings turned out to be false positives, because they did not contain implementations for refinement types for the respective language. For completeness' sake, we included them in the table, but marked them by striking through their names.

Language	Implementation	Created At	Latest Commit	Archived
JavaScript	~~intro-refinement-types~~ [50]	11.01.2016	23.06.2017	
	refinement.js [86]	05.06.2018	11.06.2018	
Python	deal [38]	25.01.2018	23.03.2024	
	Phantom Types [51]	07.02.2020	09.02.2024	
	refined [117]	23.09.2021	04.10.2021	
Go	refined [66]	03.07.2022	21.07.2022	
	refinement [81]	04.08.2019	09.08.2019	✓
C#	ConstraintComposite [83]	04.12.2022	08.12.2022	
Java	~~regex-refined~~ [97]	06.11.2018	23.11.2018	

Table 5.5: Refinement Type Implementations for Supported Programming Languages on GitHub

Further research reveals some additional implementations that are not covered by our GitHub API search. The findings are summarised in Table 5.6.

Language	Implementation	Created At	Latest Commit	Archived
JavaScript	ts-refined [76]	16.08.2017	18.08.2017	
	refscript [49, 121]	10.04.2024	14.01.2019	
Java	OpenJML [18, 85, 55, 13]	30.06.2015	15.02.2024	
	Liquid Java [70, 69]	23.04.2020	07.12.2023	

Table 5.6: Further Refinement Type Implementations for supported Programming Languages

Using the base types already supported by the Pulumi Schema and the logical predicates of our proposed solution, Pulumi developers could extend the capabilities of Codegen to generate refinement types in the SDKs. For example, Phantom Types provide a Python implementation for refinement types that map very naturally to the definition of refinement types presented in Section 2.4 and to the types and constraints provided by our proposed addition to the Pulumi Schema.

Phantom Types Creating refinement types with Phantom Types works by defining subclasses for the class Phantom and providing a predicate function. A predicate function is a function that accepts an argument of a base type, such as str and returns a bool value. For example, Listing 5.8 shows a code snippet from the Python SDK, which was generated by the Pulumi native provider for Azure. The snippet shows the definition of a class Origin that accepts an argument of type Optional[pulumi.Input[int]], which represents an http port.

```
1 class Origin(pulumi.CustomResource):
2     @overload
3     def __init__(__self__,
4                  ...
5                  http_port: Optional[pulumi.Input[int]] = None,
6                  ...
```

Listing 5.8: Class Definition from the Azure Native Provider for Python

Listing 5.9 shows the corresponding Azure Resource Manager specification for HTTP ports, which limits the applicable values to integers between 1 and 65535.

```
1         "httpPort": {
2           "description": "The value of the HTTP port. Must be between 1 and
                65535.",
3           "type": "integer",
4           "format": "int32",
5           "maximum": 65535,
6           "exclusiveMaximum": false,
7           "minimum": 1,
8           "exclusiveMinimum": false
9         },
```

Listing 5.9: Azure Resource Manager API Specification for HTTP Ports

A Phantom Type implementation for the type of HTTP ports could look something like in Listing 5.10

```
1  def is_valid_http_port(instance: int) -> bool:
2      return instance >= 1 and instance <= 65535
3
4  class HttpPort(int, Phantom, predicate=is_valid_http_port)
5      ...
6
7  class Origin(pulumi.CustomResource):
8      @overload
9      def __init__(__self__,
10                  ...
11                  http_port: Optional[pulumi.Input[HttpPort]] = None,
12                  ...
```

Listing 5.10: Phantom Types Implementation for HTTP ports

Although implementing refinement types in the Pulumi target languages is possible, a handful of caveats apply.

- Since none of the languages supported by Pulumi implement refinement types natively, they rely on libraries, IDE plugins, or additional software to provide static type checking. Therefore, the responsibility of setting up those tools lies with the developers and users of Pulumi SDKs, even if refinement type information is provided.
- Many refinement type implementations we listed in Table 5.5 and Table 5.6 have not been committed to in years, indicating that the projects have been abandoned.
- Depending on the implementation, adding refinement types to the generated SDKs will most likely break backward compatibility.
- Even though AWS, Azure, and GCP APIs offer validation keywords, not all Cloud resource properties use them where they would be applicable. For example, according to AWS documentation, the S3 bucket names must adhere to a certain pattern [4]. This pattern could be formalized with the `pattern` keyword provided by the AWS CloudFormation API. However, the API description for S3 bucket names only specifies that they can be an arbitrary `string`.

5.2.4 Impact

To measure the impact that refinement types could have on static type safety, if they would be implemented in the Pulumi Schema and subsequently in the SDKs, we analyse the API specification file of AWS CloudFormation, Azure Resource Manager, and Google Discovery. Using the API specification files for Cloud providers included in the git repositories of native providers, we calculate how many resource properties use the logical constraints and formats presented in Section 2.8.5. Let P_c be the set of properties defined

in the API description of Cloud c, where $c \in \{AWS, Azure, GCP\}$. Furthermore, let $P'_c \subseteq P_c$ be the set of properties for Cloud c that use any validation keyword applicable. The findings are summarised in Table 5.7

Cloud c	$\lvert P_c \rvert$	$\lvert P'_c \rvert$	$(\lvert P'_c \rvert / \lvert P_c \rvert) \rvert \times 100$	**Native Provider Version**
AWS	20606	5777	28	8a4a7d82
Azure	972740	123578	12.7	d00db35d
GCP	105335	18672	17.7	4511d6e8

Table 5.7: Cloud API Validation Keyword Usage

Our analysis shows that if the Pulumi Schema gets extended to be able to formulate the same constraints as the Cloud provider APIs, this would affect a significant percentage of all Cloud resource properties. 28%, 12.7% and 17.7% of all Cloud resource properties of AWS CloudFormation, Azure Resource Manager, and GCP Discovery API, respectively, use logical constraints that can be used to generate refinement types. This type information is currently lost in the Pulumi Schema.

5.3 Mitigating Dependencies Errors

As with errors in the **Enumerations** and **Refinements** error categories, the dependent type information must be presented in three stages, **(1)** the source APIs, **(2)** the Pulumi Schema, and **(3)** the target programming languages to leverage the power of dependent types.

5.3.1 Dependent Type Information in the Source APIs

While certainly being available in some form, neither of Amazon's CloudFormation API, Microsoft's Resource Manager API, nor Google's Discovery API provides dependency type information in a machine-readable format. Usually, this kind of information is presented in a human-readable format in the form of documentation. For example, GCP provides a table in their documentation [23], listing all available VM machine types for each compute region.

5.3.2 Dependent Type Information in the Pulumi Schema

As presented in Section 2.7.2, the Pulumi Schema currently does not have a way to present dependent type information.

5.3.3 Dependent Type Support in Pulumi Target Languages

As discussed in Section 2.5, only very few programming languages support dependent types natively. Furthermore, none of these programming languages, like Idris or Coq, is supported by Pulumi.

Third-Party Implementations

For supported languages, we search for third-party implementations on GitHub, which would enable the support of dependent types, Table 5.8 sums up our findings.

Language	Implementation	Created At	Latest Commit	Archived
JavaScript	~~λC+~~ [79]	12.02.2021	13.02.2023	
	~~AutoNomic-pyco~~ [82]	31.01.2019	24.02.2019	
TypeScript	~~cicada-solo~~ [36]	28.03.2021	11.05.2024	
	~~cicada-plct~~ [35]	07.08.2022	17.05.2024	
	nominal [37]	28.03.2021	03.05.2024	
	~~proof-cat~~ [131]	10.01.2024	28.01.2024	
	ts-dependent-types [84]	05.02.2022	14.02.2022	
	~~pie-ts~~ [122]	03.05.2023	17.09.2023	
	~~x-json~~ [128]	11.04.2023	10.09.2023	
Python	~~coq_jupyter~~ [87]	26.12.2018	25.01.2024	
Go	-	-	-	
C#				
Java	~~Aya Prover~~ [129, 32]	09.11.2020	06.06.2024	

Table 5.8: Dependent Type Implementations for Supported Programming Languages on GitHub

All but two of the dependent type implementations on GitHub are false positives. The two remaining are **ts-dependent-types** and **nominal**. Both are implementations in TypeScript.

- **ts-dependent-types** has not seen active development since 2022 and is not available on NPM. According to the documentation, however, the functionality seems to be very limited, since it only provides "wrappers" for TypeScript union types, like type SupportedRange = Range<−24, 24>; // −24 | −23 | ... | 24
- **nominal** actually is a library to support nominal typing in the otherwise structural typed TypeScript. However, it also claims to support a very limited form of what the documentation calls "pseudo-dependent types", in which properties that can be defined on a type level, are propagated over multiple function calls.

To leverage the full expressibility of dependent types, Pulumi would have to support programming languages like Idris, which have native dependent type support. The errors, such as those presented in Section 3.3.3 could then be mitigated by defining the appropriate dependent types. If Pulumi supported a programming language like Idris, these dependencies could be encoded like in the following code snippet.

```
1  -- Step 1: Define the Region record
2  record Region where
3    constructor MkRegion
4    availableMachineTypes : List String
5
6  -- Step 2: Define a type-level function to check if a machine type is in
       the available machine types
7  data Elem : String -> List String -> Type where
8    Here  : Elem x (x :: xs)
9    There : Elem x xs -> Elem x (y :: xs)
10
11 -- Step 3: Define the VirtualMachine datatype
12 data VirtualMachine : (region : Region) -> (machineType : String) -> Type
       where
13   MkVirtualMachine : (region : Region) ->
14                      (machineType : String) ->
15                      {auto prf : Elem machineType (availableMachineTypes
                            region)} ->
16                      VirtualMachine region machineType
17
18 -- Example usage
19 awsRegion = MkRegion ["t2.micro", "m5.large", "c5.xlarge"]
20
21 -- Valid VM
22 awsT2Micro = MkVirtualMachine awsRegion "t2.micro"
23 -- Invalid VM
24 awsInvalid = MkVirtualMachine awsRegion "unknownType"
```

Listing 5.11: Cloud Regions and Virtual Machines with Dependent Types in Idris.

Listing 5.11 shows how Cloud regions and machine types can be defined using dependent types in Idris.

- Lines 2-4 define the new record Region. A Region instance takes a parameter of type List String, which defines the machine types available in that region.
- Lines 7-9 define a new datatype that represents the occurrence of an element of type String inside of a list of Strings.
- Lines 12-16 define the datatype VirtualMachine. It's constructor, MkVirtualMachine, takes a parameter region of type Region and a parameter machineType of type String to automatically construct the proof in line 15 that machineType is contained in availableMachineTypes of region.
- Line 19 defines a new Region with a list of available machine types.
- Finally, lines 22 and 24 show two VirtualMachine instances.

The first instance is deemed valid by the Idris compiler, as t2.micro is contained in availableMachineTypes of awsRegion. However, awsInvalid cannot be instantiated, as unknownType is not contained in availableMachineTypes. Compilation of the program fails with the error message shown in Listing 5.11.

```
1 Error: While processing the right-hand side of awsInvalid. Can not find an
      implementation for Elem "unknownType" ["t2.micro", "m5.large", "c5.
      xlarge"]
```

Listing 5.12: Error Message from the Idris Compiler Trying to Compile the Code in Listing 5.11.

CHAPTER 6

Evaluation and Discussion

The preceding chapters show the viability and impact of adding more static type information in IaC programs and in Pulumi programs in particular, while qualifying the potential impact on the basis of a dataset of Pulumi repositories. The following chapter discusses the varying viability of adding more static type information to IaC programs, the impact of this addition, and caveats that apply, beginning with the quality of our dataset.

6.1 Quality of the Dataset

A significant portion of our empirical analysis is based on a data set from GitHub repositories. However, due to the availability constraints of private or organisation-owned repositories, our dataset is subject to substantial caveats.

- All of the repositories in our dataset are public repositories. That is, there is likely a large "dark figure" of private repositories, which are hypothetically more complex and useful IaC programs.
- A significant portion of the repositories in our dataset are forks of official Pulumi repositories, resulting in some redundancy and limiting the dataset's reflection of real-world scenarios.

6.2 Pulumi Native Providers

As we have shown, Pulumi native providers for AWS, Azure, and GCP tend to offer better static type safety by specifying enums for their respective cloud resources. However, in practise, native providers will not always pose an improvement for Pulumi programs.

There are at least two reasons why upgrading to native providers might not be feasible for developers.

- The feature coverage of native providers tends to be worse than the feature coverage of the respective classic provider. For example, while the AWS classic providers allow managing 27 [40], the AWS native providers only allow managing 10 [41] different resources in the AWS S3 module. This is because, since native provider SDKs are automatically generated from the underlying API descriptions, they can only offer what was provided by the API descriptions in the first place.
- In theory, native providers offer enums, which are not available in classic providers. However, there are still properties that could be enums, but are `string` because of bad type specification. For example, the description [41] of the `outputSchemaVersion` property of the `BucketDataExport` resource in the AWS native provider reads: "Must be V_1". However, the type of `outputSchemaVersion` is `string`, while it could be an enum, only allowing value `V_1`.

6.3 Mitigating Enumerations Errors

Our analysis highlights the role of enums in Pulumi's support for AWS, Azure, and GCP providers. While enums are natively supported by both the underlying Cloud APIs and the Pulumi Schema, they are lost in translation when using classic providers derived from Terraform. The following points detail these issues and the current limitations in enum support.

- Enums are supported by **all** of the underlying APIs of AWS, Azure and GCP for the corresponding Pulumi providers.
- The central API Schema for Pulumi SDKs, the Pulumi Schema, supports enums natively.
- **Native** providers for AWS, Azure and GCP already support enums natively and translate Pulumi Schema enums to idiomatic programming language structures for each SDK programming language.
- **Classic** providers for AWS, Azure and GCP generate the Pulumi Schema from their corresponding Terraform providers. While these Terraform providers support enums, the Pulumi code generation pipeline does not translate enums in the Terraform provider to enums in the Pulumi Schema. Therefore, enum type information is currently lost in this translation step, despite being readily available.
- Developers using classic providers could change to using the corresponding native provider variants, to get better static type safety with enums. However, native providers of AWS and GCP are currently in preview and should not be used in production.

- Classic providers are significantly more prevalent than native providers. In our dataset of 1023 GitHub repositories only 3.46%, 32.3% and 7.68% of repositories that use AWS, Azure and GCP provider, respectively, use the native provider variants.
- In our dataset 48.68%, 22.29% and 28.45% of all repositories use AWS, Azure and GCP classic providers, respectively, and would therefore benefit from upgrading to the corresponding native provider variants, since they offer enum types.
- However, only 19.94%, 3.03% and 6.74% of all repositories use AWS, Azure and GCP classic providers, respectively **and** contain strings that are exactly equal to any enum value of their corresponding native provider, suggesting that in these cases, instead of strings, enums could have been used, either by adding enum support to classic providers, or by migrating to native providers.
- While it is possible for classic providers to provide enums in the generated SDKs, this feature is not currently implemented. An issue has been raised by the developers, but as of the time of writing, no progress has been made.

6.4 Mitigating Refinements Errors

Validation constraints, which can be used for defining refinement types, are expressible in the public Cloud APIs of AWS, Azure, and GCP. However, the Pulumi Schema does not offer this feature. Additionally, there is no native support for refinement types in programming languages compatible with Pulumi. The following points summarise our findings.

- AWS, Azure, and GCP APIs are based on the JSON Schema and therefore support validation constraints for parameters in addition to the `format` parameter, which can be used to formulate refinement types.
- AWS supports the most validations and formats, followed by GCP and Azure. This is because the AWS CloudFormation API is based on the latest version of the JSON Schema.
- The Pulumi Schema currently does not support validation constraints or formats. However, since the Pulumi Schema is also based on JSON Schema, like the underlying public Cloud APIs, we argue that an extension to the Pulumi Schema to support validation constraints and formats is viable.
- 28%, 12.7% and 17.7% of properties in the AWS, Azure, and GCP API descriptions contain at least one kind of validation keyword, respectively, and would therefore benefit from refinement type support in the Pulumi SDKs.
- However, **none** of the programming languages for which Pulumi SDKs are generated natively support refinement types.

- Third-party implementations of refinement types in programming languages supported by Pulumi, such as **Phantom Types**, can be used to mitigate **Refinements** errors. However, most of the available implementations for refinement types have not seen active development in the last year.

6.5 Mitigating Dependencies Errors

Despite the benefits of dependent types, their use in IaC programs is limited by several factors. These include the absence of dependent type information in Cloud provider APIs, the lack of support in the Pulumi Schema, and the fact that Pulumi-compatible languages do not natively support them. Existing third-party implementations are outdated or inadequate, further hindering their practical application. In detail, our thesis demonstrates the following points.

- Dependent type information is not available in the APIs of neither AWS, Azure, nor GCP.
- Dependent type information is currently not provided by AWS, Azure, or GCP in a machine-readable format, but rather distributed informally via human-readable documentation.
- The Pulumi Schema does not support the definition of dependent type information.
- None of the programming languages supported by Pulumi supports dependent types natively.
- Third-party dependent type implementations for programming languages supported by Pulumi have either not seen any active development in years, or they only support a very limit version of dependent types, making them unpractical for general use.
- Provided that dependent types information is made available, programming languages with native dependent type support, like Idris, can be used to mitigate errors in the **Dependencies** error category.

CHAPTER 7

Conclusion

This paper identifies prevalent runtime errors that occur in IaC programs, which can be divided into three categories of increasing complexity. We highlight the correspondence of errors in these error categories with techniques of static type analysis. We analyse a concrete ecosystem of IaC programs, Pulumi, which leverages the power of conventional programming languages and their type systems. We show that errors in the three error categories can be handled statically, depending on the type information provided by the underlying Cloud provider APIs, Pulumi and the type systems of the supported programming languages. In particular, we answer our two research questions.

There are at least three categories of runtime errors in IaC programs that stem from inadequate static type validation. For each category, specific static type validation techniques (enums, refinement types, and dependent types) are available to mitigate these errors, answering **RQ1**.

Finally, we answer **RQ2** by showing that applying these three static types to an existing IaC solution such as Pulumi can offer significant benefits in ensuring type safety.

Error Categories We find **7** type-related runtime errors in 126 Pulumi repositories, which can be divided into three different error categories, with increasing complexity.

1. **Enumerations**: The errors in this category describe parameters in a program that expect a value from a certain set of allowed values, while not restricting them statically. For example, in available machine types for virtual machines of Cloud providers are usually a fixed set of possible values. Nevertheless, these parameters are often represented as arbitrary strings in programming languages.

2. **Refinements**: Errors in this category appear when parameters are expected to have a certain static type, but also adhere to additional constraints. For example,

while port numbers are usually correctly typed as integers within programming languages, typically only port numbers between 1 and 65535 are allowed port number values.

3. **Dependencies**: The final and most complex category of errors occur, when the allowed values of one parameter is dependent on the value of another parameter. For example, Cloud providers usually do not support all machine types of virtual machines in all geographical regions of the world. The set of allowed machine types is therefore dependent on the value of the geographical region, the virtual machine should be deployed in.

Correspondence Between Error Categories and Static Types There is a correspondence between the runtime errors in our three defined categories and the errors that can be prevented statically by three different kinds of static types. This correspondence is summarised in Table 7.1.

Error Category	*Static Types*
Enumerations	Variants (Enums as special case)
Refinements	Refinement Types
Dependencies	Dependent Types

Table 7.1: Correspondence Between Runtime Error Categories of IaC Programs and Static Types

Mitigating Runtime Errors The use of enums, refinement types, and dependent types can theoretically mitigate the errors in the defined error categories. In general, whether or not these static types can be used in IaC programs depends on three aspects.

1. Whether or not static type information is present at the Cloud provider API.
2. Whether or not static type information is expressible by the specification Schema of the IaC tool.
3. Whether or not static type information is expressible in the programming language used by the IaC tool.

For Pulumi, the answer to those three questions varies between enums, refinement types, and dependent types.

- **Enums**: Static type information for enums is present at the APIs of AWS, Azure, and GCP, as well as in Pulumi and in most of the programming languages supported by Pulumi. The native provider variants already leverage this type information and produce idiomatic enums in the SDKs for the supported programming languages.

However, the more prevalent variant, the classic providers, do not. Upgrading to the respective native provider variants would result in better enum support. This poses the challenge that AWS and GCP native providers are currently not recommended for production use. Adding enum support to classic providers is possible, but not currently implemented. From our dataset of 1023 repositories 48.68%, 22.29%, 28.45% use AWS, Azure and GCP classic providers respectively and would therefore benefit from having additional enum type information.

- **Refinement Types**: Static type information for refinement types is present in the APIs of AWS, Azure, and GCP in the form of validation keywords but missing in Pulumi, although an addition would be viable. Adding these validation keywords to Pulumi would affect 28%, 12.7% and 17.7% of properties configurable in AWS, Azure and GCP Pulumi providers. There is no programming language supported by Pulumi that implements refinement types natively. However, third-party implementations exist, which would make it possible to mitigate **Refinements** errors statically.
- **Dependent Types**: Neither the APIs of AWS, Azure, and GCP, nor Pulumi can express the static type information needed for dependent types. Additionally, none of the programming languages supported by Pulumi supports dependent types natively. Third-party implementations for dependent types are rare and unusable for general dependent types. However, provided that dependent type information is available, a programming language that can express dependent types natively is able to mitigate errors in the **Dependencies** category statically.

Appendix

Table 7.2 lists all scripts that have been used in this thesis. All scripts are publicly available on GitHub [3].

Script Name	Description
Enum Analysis	
filter_pulumi_forks.sh	Takes a set of repositories and remove all that are forks of Pulumi repositories.
find_enum_values.sh	Finds all Enums values used in a Pulumi Schema JSON file.
find_classic_provider_repos.sh	Finds Pulumi classic providers for AWS, Azure and GCP by searching for dependency declarations.
find_enums_in_repos.sh	Finds strings in .go, .java, .cs, .js, .ts, .py, .scala and .kt files. Used to find enums values in Pulumi projects.
find_native_provider_repos.sh	Finds Pulumi native providers for AWS, Azure and GCP by searching for dependency declarations.
Refinement Type Analysis	
find_constraints.sh	Finds logical constraints in OpenAPI specs for AWS, Azure and GCP and calculates how many properties use some kind of constraint.
find_on_github.sh	Searching for refinement type projects in GitHub for languages java, javascript, typescript, python, go and c#.
Dependent Type Analysis	
find_on_github.sh	Searching for dependent type projects in GitHub for Java, Javascript, Typescript, Python, Go, and c#.

Table 7.2: Scripts used for Analysis done in this Thesis

List of Figures

List of Tables

Bibliography

[1] Add more options to pulumi config schema · Issue #11547 · pulumi/pulumi. URL: https://github.com/pulumi/pulumi/issues/11547.

[2] The Agda Wiki. URL: https://wiki.portal.chalmers.se/agda/pmwiki.php.

[3] AndreasWeichselbaum/master-thesis-analysis-scripts: This repository contains scripts for analysing data collected for my master thesis. URL: https://github.com/AndreasWeichselbaum/master-thesis-analysis-scripts.

[4] AWS CloudFormation Documentation. URL: https://docs.aws.amazon.com/cloudformation/.

[5] Azure Resource Manager overview - Azure Resource Manager | Microsoft Learn. URL: https://learn.microsoft.com/en-us/azure/azure-resource-manager/management/overview.

[6] Chef Software DevOps Automation Solutions | Chef. URL: https://www.chef.io/.

[7] Cloud Development Framework - AWS Cloud Development Kit - AWS. URL: https://aws.amazon.com/cdk/.

[8] F*: A Proof-Oriented Programming Language. URL: https://www.fstar-lang.org/.

[9] Google API Discovery Service | Google for Developers. URL: https://developers.google.com/discovery.

[10] Homepage | Ansible Collaborative. URL: https://www.ansible.com/.

[11] Idris: A Language for Type-Driven Development. URL: https://www.idris-lang.org/.

[12] Infrastructure as Code Market Size, Share | Growth Report, 2030. URL: https://www.fortunebusinessinsights.com/infrastructure-as-code-market-108777.

[13] The Java Modeling Language (JML) Home Page. URL: https://www.cs.ucf.edu/~leavens/JML/index.shtml.

[14] JSON Schema - Draft-07 Release Notes. URL: https://json-schema.org/draft-07/json-schema-release-notes.

[15] Migrate from classic Azure to Azure-Native | Pulumi Registry. URL: https://www.pulumi.com/registry/packages/azure-native/from-classic/#move-resources-from-classic-azure-to-azure-native.

[16] npm | Home. URL: https://www.npmjs.com/.

[17] OpenAPI Specification v2.0 | Introduction, Definitions, & More. URL: https://spec.openapis.org/oas/v2.0.

[18] OpenJML. URL: https://github.com/OpenJML.

[19] Plugin Development - Framework: Validation | Terraform | HashiCorp Developer. URL: https://developer.hashicorp.com/terraform/plugin/framework/validation.

[20] Pulumi. URL: https://www.pulumi.com.

[21] pulumi/pkg/codegen at master · pulumi/pulumi. URL: https://github.com/pulumi/pulumi/tree/master/pkg/codegen.

[22] Puppet Infrastructure & IT Automation at Scale | Puppet by Perforce. URL: https://www.puppet.com/.

[23] Regions and zones | Compute Engine Documentation | Google Cloud. URL: https://cloud.google.com/compute/docs/regions-zones.

[24] registry/docs/download-counts.md at master · npm/registry. URL: https://github.com/npm/registry/blob/master/docs/download-counts.md.

[25] Schema. URL: https://www.pulumi.com/docs/using-pulumi/pulumi-packages/schema/.

[26] Stack Overflow Developer Survey 2023. URL: https://survey.stackoverflow.co/2023/#technology-most-popular-technologies.

[27] Summary of the Amazon S3 Service Disruption in the Northern Virginia (US-EAST-1) Region. URL: https://aws.amazon.com/message/41926/.

[28] Terraform by HashiCorp. URL: https://www.terraform.io/.

[29] terraform-plugin-sdk/helper/validation at main · hashicorp/terraform-plugin-sdk. URL: https://github.com/hashicorp/terraform-plugin-sdk/tree/main/helper/validation.

[30] Welcome! | The Coq Proof Assistant. URL: https://coq.inria.fr/.

[31] GitHub REST API documentation - GitHub Docs, October 2023. URL: https://docs.github.com/en/rest?apiVersion=2022-11-28.

[32] aya-prover/aya-dev, June 2024. original-date: 2020-11-09T02:16:13Z. URL: https://github.com/aya-prover/aya-dev.

[33] Azure/azure-rest-api-specs, May 2024. original-date: 2015-07-14T18:37:13Z. URL: https://github.com/Azure/azure-rest-api-specs.

[34] CDK for Terraform | Terraform | HashiCorp Developer, October 2024. URL: https://developer.hashicorp.com/terraform/cdktf.

[35] cicada-lang/cicada-plct, May 2024. original-date: 2022-08-07T07:38:02Z. URL: https://github.com/cicada-lang/cicada-plct.

[36] cicada-lang/cicada-solo, May 2024. original-date: 2021-03-28T13:40:18Z. URL: https://github.com/cicada-lang/cicada-solo.

[37] Coder-Spirit/nominal, May 2024. original-date: 2021-03-28T07:57:31Z. URL: https://github.com/Coder-Spirit/nominal.

[38] life4/deal, April 2024. original-date: 2018-01-25T06:38:47Z. URL: https://github.com/life4/deal.

[39] pulumi/examples, September 2024. original-date: 2017-10-27T19:50:31Z. URL: https://github.com/pulumi/examples.

[40] pulumi/pulumi-aws, March 2024. original-date: 2017-07-17T14:20:33Z. URL: https://github.com/pulumi/pulumi-aws.

[41] pulumi/pulumi-aws-native, April 2024. original-date: 2019-11-04T19:00:32Z. URL: https://github.com/pulumi/pulumi-aws-native.

[42] pulumi/pulumi-azure, April 2024. original-date: 2017-09-11T20:19:15Z. URL: https://github.com/pulumi/pulumi-azure.

[43] pulumi/pulumi-azure-native, March 2024. original-date: 2019-02-24T20:30:21Z. URL: https://github.com/pulumi/pulumi-azure-native.

[44] pulumi/pulumi-gcp, August 2024. original-date: 2017-07-17T14:28:37Z. URL: https://github.com/pulumi/pulumi-gcp.

[45] pulumi/pulumi-google-native, March 2024. original-date: 2020-12-22T16:39:01Z. URL: https://github.com/pulumi/pulumi-google-native.

[46] pulumi/pulumi-terraform-bridge, April 2024. original-date: 2019-09-28T20:54:02Z. URL: https://github.com/pulumi/pulumi-terraform-bridge.

[47] pulumi/templates, September 2024. original-date: 2018-03-09T18:21:12Z. URL: https://github.com/pulumi/templates.

[48] terraform, Pulumi, Chef, Ansible, Puppet - Explore - Google Trends, October 2024. URL: https://trends.google.com/trends/explore?cat=5&date=all&q=terraform,Pulumi,Chef,Ansible,Puppet&hl=en-GB.

[49] UCSD-PL/refscript, February 2024. original-date: 2014-04-10T17:51:20Z. URL: https://github.com/UCSD-PL/refscript.

[50] ucsd-progsys/intro-refinement-types, January 2024. original-date: 2016-01-11T18:41:49Z. URL: https://github.com/ucsd-progsys/intro-refinement-types.

[51] Anton Agestam. antonagestam/phantom-types, April 2024. original-date: 2020-02-07T21:28:59Z. URL: https://github.com/antonagestam/phantom-types.

[52] Lennart Augustsson. Cayenne — A Language with Dependent Types. In Gerhard Goos, Juris Hartmanis, Jan Van Leeuwen, S. Doaitse Swierstra, José N. Oliveira, and Pedro R. Henriques, editors, *Advanced Functional Programming*, volume 1608, pages 240–267. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999. Series Title: Lecture Notes in Computer Science. URL: http://link.springer.com/10.1007/10704973_6, doi:10.1007/10704973_6.

[53] Clark Barrett and Aaron Stump. The SMT-LIB Standard – Version 2.0.

[54] Ana Bove, Peter Dybjer, and Ulf Norell. A Brief Overview of Agda – A Functional Language with Dependent Types. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 73–78, Berlin, Heidelberg, 2009. Springer. doi:10.1007/978-3-642-03359-9_6.

[55] Daniel Bruns. Formal Semantics for the Java Modeling Language.

[56] Christophe Cerin, Camille Coti, Pierre Delort, Felipe Diaz, Maurice Gagnaire, Marija Mijic, Quentin Gaumer, Nicolas Guillaume, Jonathan Le Lous, Stephane Lubiarz, Jean-Luc Raffaelli, Kazuhiko Shiozaki, Herve Schauer, Jean-Paul Smets, Laurent Seguin, and Alexandrine Ville. Downtime Statistics of Current Cloud Solutions.

[57] Wei Chen, Guoquan Wu, and Jun Wei. An Approach to Identifying Error Patterns for Infrastructure as Code. In *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 124–129, Memphis, TN, October 2018. IEEE. URL: https://ieeexplore.ieee.org/document/8539175/, doi:10.1109/ISSREW.2018.00-19.

[58] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, June 1940. URL: https://www.cambridge.org/core/product/identifier/S0022481200108187/type/journal_article, doi:10.2307/2266170.

[59] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C. Necula. Dependent Types for Low-Level Programming. In Rocco De Nicola, editor, *Programming Languages and Systems*, pages 520–535, Berlin, Heidelberg, 2007. Springer. doi:10.1007/978-3-540-71316-6_35.

[60] Thierry Coquand. Type Theory. In Edward N. Zalta and Uri Nodelman, editors, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, fall 2022 edition, 2022. URL: https://plato.stanford.edu/archives/fall2022/entries/type-theory/.

[61] H. B. Curry. Functionality in Combinatory Logic. *Proceedings of the National Academy of Sciences*, 20(11):584–590, November 1934. URL: https://pnas.org/doi/full/10.1073/pnas.20.11.584, doi:10.1073/pnas.20.11.584.

[62] Ting Dai, Alexei Karve, Grzegorz Koper, and Sai Zeng. Automatically detecting risky scripts in infrastructure code. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, pages 358–371, New York, NY, USA, October 2020. Association for Computing Machinery. doi:10.1145/3419111.3421303.

[63] Peter Dybjer and Erik Palmgren. Intuitionistic Type Theory. In Edward N. Zalta and Uri Nodelman, editors, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, spring 2023 edition, 2023. URL: https://plato.stanford.edu/archives/spr2023/entries/type-theory-intuitionistic/.

[64] Christof Ebert, Gorka Gallardo, Josune Hernantes, and Nicolas Serrano. DevOps. *IEEE Software*, 33(3):94–100, May 2016. Conference Name: IEEE Software. URL: https://ieeexplore.ieee.org/abstract/document/7458761, doi:10.1109/MS.2016.68.

[65] Roy T. Fielding and Julian Reschke. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. Request for Comments RFC 7231, Internet Engineering Task Force, June 2014. Num Pages: 101. URL: https://datatracker.ietf.org/doc/rfc7231, doi:10.17487/RFC7231.

[66] Firiath. valentinHenry/refined, July 2022. original-date: 2022-07-03T20:48:28Z. URL: https://github.com/valentinHenry/refined.

[67] Tim Freeman and Frank Pfenning. Refinement types for ML. *ACM SIGPLAN Notices*, 26(6):268–277, May 1991. URL: https://dl.acm.org/doi/10.1145/113446.113468, doi:10.1145/113446.113468.

[68] Francis Galiegue, Kris Zyp, and Gary Court. JSON Schema: core definitions and terminology. Internet Draft draft-zyp-json-schema-04, Internet Engineering Task Force, January 2013. Num Pages: 14. URL: https://datatracker.ietf.org/doc/draft-zyp-json-schema-04.

[69] Catarina Gamboa. CatarinaGamboa/liquidjava, November 2023. original-date: 2020-04-23T13:42:59Z. URL: https://github.com/CatarinaGamboa/liquidjava.

[70] Catarina Gamboa, Paulo Alexandre Santos, Christopher S. Timperley, and Alcides Fonseca. User-driven Design and Evaluation of Liquid Types in Java, October 2021. arXiv:2110.05444 [cs]. URL: http://arxiv.org/abs/2110.05444.

[71] Gerhard Gentzen. Untersuchungen ber das logische Schlieen. I. *Mathematische Zeitschrift*, 39(1):176–210, December 1935. URL: http://link.springer.com/10.1007/BF01201353, doi:10.1007/BF01201353.

[72] MIchele Guerriero, Martin Garriga, Damian A. Tamburri, and Fabio Palomba. Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 580–589, September 2019. ISSN: 2576-3148. URL: https://ieeexplore.ieee.org/document/8919181, doi:10.1109/ICSME.2019.00092.

[73] Muhammad Raquibul Hasan and Md Sifuddin Ansary. Cloud Infrastructure Automation Through IaC (Infrastructure as Code). *International Journal of Computer (IJC)*, 46(1):34–40, February 2023. URL: https://ijcjournal.org/index.php/InternationalJournalOfComputer/article/view/2043.

[74] Martin Hofmann. Syntax and semantics of dependent types. In Martin Hofmann, editor, *Extensional Constructs in Intensional Type Theory*, pages 13–54. Springer, London, 1997. doi:10.1007/978-1-4471-0963-1_2.

[75] William Alvin Howard. The Formulae-as-Types Notion of Construction. In Haskell Curry, Hindley B, Seldin J. Roger, and P. Jonathan, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Academic Press, 1980.

[76] Jeremy Hughes. jedahu/ts-refined, February 2024. original-date: 2017-08-16T23:34:22Z. URL: https://github.com/jedahu/ts-refined.

[77] Waldemar Hummer, Florian Rosenberg, Fábio Oliveira, and Tamar Eilam. Testing Idempotence for Infrastructure as Code. In David Eyers and Karsten Schwan, editors, *Middleware 2013*, pages 368–388, Berlin, Heidelberg, 2013. Springer. doi:10.1007/978-3-642-45065-5_19.

[78] Ranjit Jhala and Niki Vazou. Refinement Types: A Tutorial, October 2020. arXiv:2010.07763 [cs]. URL: http://arxiv.org/abs/2010.07763.

[79] joewatt95. joewatt95/lambdacplus, April 2024. original-date: 2021-02-12T15:13:03Z. URL: https://github.com/joewatt95/lambdacplus.

[80] Sebastian Kleinschmager. *Can static type systems speed up programming? An experimental evaluation of static and dynamic type systems*. Grin Publishing, November 2012.

[81] Mazunin Konstantin. mazuninky/refinement, December 2023. original-date: 2019-08-04T02:47:11Z. URL: https://github.com/mazuninky/refinement.

[82] koo5. koo5/AutoNomic-pyco, December 2019. original-date: 2019-01-31T15:22:15Z. URL: https://github.com/koo5/AutoNomic-pyco.

[83] Kuinox. Kuinox/ConstrainedComposite, March 2024. original-date: 2022-12-04T02:28:00Z. URL: https://github.com/Kuinox/ConstrainedComposite.

[84] Yoav Lavi. yoav-lavi/ts-dependent-types, October 2023. original-date: 2022-02-05T15:19:12Z. URL: https://github.com/yoav-lavi/ts-dependent-types.

[85] Gary Leavens, Albert Baker, and Clyde Ruby. JML: a Java Modeling Language. October 1998.

[86] Jinwei Long. overlogged/refinement.js, June 2018. original-date: 2018-06-05T07:31:41Z. URL: https://github.com/overlogged/refinement.js.

[87] Eugene Loy. EugeneLoy/coq_jupyter, May 2024. original-date: 2018-12-26T14:40:15Z. URL: https://github.com/EugeneLoy/coq_jupyter.

[88] Per Martin-Löf. An Intuitionistic Theory of Types: Predicative Part. In *Studies in Logic and the Foundations of Mathematics*, volume 80, pages 73–118. Elsevier, 1975. URL: https://linkinghub.elsevier.com/retrieve/pii/S0049237X08719451, doi:10.1016/S0049-237X(08)71945-1.

[89] Per Martin-Löf. Constructive Mathematics and Computer Programming. In *Studies in Logic and the Foundations of Mathematics*, volume 104, pages 153–175. Elsevier, 1982. URL: https://linkinghub.elsevier.com/retrieve/pii/S0049237X09701892, doi:10.1016/S0049-237X(09)70189-2.

[90] Per Martin-Löf. *Intuitionistic type theory: notes by Giovanni Sambin of a series of lectures given in Padua, June 1980*. Number 1 in Studies in proof theory. Bibliopolis, Napoli, 1984.

[91] Clemens Mayer, View Profile, Stefan Hanenberg, View Profile, Romain Robbes, View Profile, Éric Tanter, View Profile, Andreas Stefik, and View Profile. An empirical study of the influence of static type systems on the usability of undocumented software. *ACM SIGPLAN Notices*, 47(10):683–702, October 2012. Publisher: Association for Computing Machinery. URL: https://dl.acm.org/doi/abs/10.1145/2398857.2384666, doi:10.1145/2398857.2384666.

[92] Kief Morris. *Infrastructure as code: dynamic systems for the cloud age*. O'Reilly, Beijing Boston Farnham Sebastopol Tokyo, second edition edition, 2020.

[93] Charles Gregory Nelson. *Techniques for Program Verification*. Stanford University, 1980. URL: https://people.eecs.berkeley.edu/~necula/Papers/nelson-thesis.pdf.

[94] Chris Newman and Graham Klyne. Date and Time on the Internet: Timestamps. Request for Comments RFC 3339, Internet Engineering Task Force, July 2002. Num Pages: 18. URL: https://datatracker.ietf.org/doc/rfc3339, doi:10.17487/RFC3339.

[95] B. Nördstrom and K. Petersson. Martin-Löf 's type theory. In *Handbook of Logic in Computer Science: Volume 5. Algebraic and Logical Structures*. Oxford University Press, January 2001. URL: https://academic.oup.com/book/40635/chapter/348289679, doi:10.1093/oso/9780198537816.003.0004.

[96] Atsushi Ohori. A Curry-Howard Isomorphism for Compilation and Program Execution. In Jean-Yves Girard, editor, *Typed Lambda Calculi and Applications*, pages 280–294, Berlin, Heidelberg, 1999. Springer. doi:10.1007/3-540-48959-2_20.

[97] Tomohisa Osada. henoc/regex-refined, December 2018. original-date: 2018-11-06T11:35:00Z. URL: https://github.com/henoc/regex-refined.

[98] Sneh Pandya and Riya Guha Thakurta. Introduction to Infrastructure as Code. In Sneh Pandya and Riya Guha Thakurta, editors, *Introduction to Infrastructure as Code: A Brief Guide to the Future of DevOps*, pages 3–17. Apress, Berkeley, CA, 2022. doi:10.1007/978-1-4842-8689-0_1.

[99] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, Mass, 2002.

[100] Benjamin C. Pierce, editor. *Advanced topics in types and programming languages*. MIT Press, Cambridge, Mass, 2005.

[101] Emil L. Post. Introduction to a General Theory of Elementary Propositions. *American Journal of Mathematics*, 43(3):163, July 1921. URL: https://www.jstor.org/stable/2370324?origin=crossref, doi:10.2307/2370324.

[102] Akond Rahman, Effat Farhana, Chris Parnin, and Laurie Williams. Gang of Eight: A Defect Taxonomy for Infrastructure as Code Scripts. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 752–764, October 2020. ISSN: 1558-1225. URL: https://ieeexplore.ieee.org/document/9284113, doi:10.1145/3377811.3380409.

[103] Akond Rahman, Rezvan Mahdavi-Hezaveh, and Laurie Williams. A systematic mapping study of infrastructure as code research. *Information and Software Technology*, 108:65–77, April 2019. URL: https://linkinghub.elsevier.com/retrieve/pii/S0950584918302507, doi:10.1016/j.infsof.2018.12.004.

[104] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 159–169, New York, NY, USA, June 2008. Association for Computing Machinery. doi:10.1145/1375581.1375602.

[105] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. Low-level liquid types. *ACM SIGPLAN Notices*, 45(1):131–144, January 2010. doi:10.1145/1707801.1706316.

[106] J. Rushby, S. Owre, and N. Shankar. Subtypes for specifications: predicate subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, September 1998. URL: http://ieeexplore.ieee.org/document/713327/, doi:10.1109/32.713327.

[107] Bertrand Russell. *Principles of Mathematics*. Routledge, London, 3 edition, March 2020. doi:10.4324/9780203822586.

[108] Julian Schwarz, Andreas Steffens, and Horst Lichter. Code Smells in Infrastructure as Code. In *2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC)*, pages 220–228, Coimbra, September 2018. IEEE. URL: https://ieeexplore.ieee.org/document/8590193/, doi:10.1109/QUATIC.2018.00040.

[109] Rian Shambaugh, Aaron Weiss, and Arjun Guha. Rehearsal: a configuration verification tool for puppet. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 416–430, Santa Barbara CA USA, June 2016. ACM. URL: https://dl.acm.org/doi/10.1145/2908080.2908083, doi:10.1145/2908080.2908083.

[110] Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. Does your configuration code smell? In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 189–200, Austin Texas, May 2016. ACM. URL: https://dl.acm.org/doi/10.1145/2901739.2901761, doi:10.1145/2901739.2901761.

[111] Vijay Kartik Sikha, Dayakar Siramgari, and Satyaveda Somepalli. Infrastructure as Code: Historical Insights and Future Directions. *Open Access*, 12(8), 2022.

[112] Daniel Sokolowski. Reliable Infrastructure as Code for Decentralized Organizations, May 2024.

[113] Daniel Sokolowski and Guido Salvaneschi. Towards Reliable Infrastructure as Code. In *2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C)*, pages 318–321, L'Aquila, Italy, March 2023. IEEE. URL: https://ieeexplore.ieee.org/document/10092598/, doi:10.1109/ICSA-C57050.2023.00072.

[114] Daniel Sokolowski, David Spielmann, and Guido Salvaneschi. Automated Infrastructure as Code Program Testing. *IEEE Transactions on Software Engineering*, 50(6):1585–1599, June 2024. URL: https://ieeexplore.ieee.org/document/10516612/, doi:10.1109/TSE.2024.3393070.

[115] Daniel Sokolowski, David Spielmann, and Guido Salvaneschi. Unleashing the Giants: Enabling Advanced Testing for Infrastructure as Code. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, pages 300–301, Lisbon Portugal, April 2024. ACM. URL: https://dl.acm.org/doi/10.1145/3639478.3643078, doi:10.1145/3639478.3643078.

[116] Morten Heine Sørensen and Paweł Urzyczyn. *Lectures on the Curry-Howard isomorphism*. Number 149 in Studies in logic and the foundations of mathematics. Elsevier, Amsterdam Heidelberg, 1st ed edition, 2006.

[117] Quino Terrasa. espetro/refined, May 2023. original-date: 2021-09-23T21:50:37Z. URL: https://github.com/espetro/refined.

[118] Mark van Atten. The Development of Intuitionistic Logic. In Edward N. Zalta and Uri Nodelman, editors, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, fall 2023 edition, 2023. URL: https://plato.stanford.edu/archives/fall2023/entries/intuitionistic-logic-development/.

[119] Niki Vazou, Patrick M. Rondon, and Ranjit Jhala. Abstract Refinement Types. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems*, pages 209–228, Berlin, Heidelberg, 2013. Springer. doi:10.1007/978-3-642-37036-6_13.

[120] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. Refinement types for Haskell. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*, pages 269–282, Gothenburg Sweden, August 2014. ACM. URL: https://dl.acm.org/doi/10.1145/2628136.2628161, doi:10.1145/2628136.2628161.

[121] Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. Refinement Types for TypeScript.

[122] Job Vonk. stickyPiston/pie-ts, August 2023. original-date: 2023-05-03T10:13:01Z. URL: https://github.com/stickyPiston/pie-ts.

[123] Andreas Weichselbaum. Repositories found by searching GitHub API. URL: https://docs.google.com/spreadsheets/u/0/d/1376SCIlcAYkiChJjso_tYNl021CftyPWJ6Ofb1xLSV4/edit?usp=embed_facebook.

[124] Andreas Weichselbaum. Enum Value Analysis Pulumi Repositories, September 2024. URL: https://docs.google.com/spreadsheets/d/1LHTPaGFJjHV07Nw-KYsAVBnzqUdiNA_yhr8wIXVakkc/edit?usp=sharing.

[125] Alfred North Whitehead and Bertrand Russell. *Principia mathematica*. Cambridge University Press, Cambridge [Eng.] New York, 2d ed edition, 1927.

[126] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 249–257, New York, NY, USA, May 1998. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/277650.277732, doi:10.1145/277650.277732.

[127] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '99, pages 214–227, New York, NY, USA, January 1999. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/292540.292560, doi:10.1145/292540.292560.

[128] / Xie Yuheng. xieyuheng/x-json, September 2023. original-date: 2023-04-11T00:18:52Z. URL: https://github.com/xieyuheng/x-json.

[129] Tesla Zhang. A simpler encoding of indexed types, 2021. Version Number: 4. URL: https://arxiv.org/abs/2103.15408, doi:10.48550/ARXIV.2103.15408.

[130] Kris Zyp and Gary Court. A JSON Media Type for Describing the Structure and Meaning of JSON Documents. Internet Draft draft-zyp-json-schema-03, Internet Engineering Task Force. Num Pages: 28. URL: https://datatracker.ietf.org/doc/draft-zyp-json-schema-03.

[131] . 5eqn/proof-cat, January 2024. original-date: 2024-01-10T09:57:47Z. URL: https://github.com/5eqn/proof-cat.