UNIVERSIDADE FEDERAL DE MINAS GERAIS
Instituto de Ciências Exatas
Programa de Pós-Graduação em Ciência da Computação

Guilherme de Oliveira Silva

Certification of programs via Gödel numbering

Belo Horizonte
2025

Guilherme de Oliveira Silva

Certification of programs via Gödel numbering

Final Version

Thesis proposal presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Fernando Magno Quintão Pereira

Belo Horizonte
2025

Silva, Guilherme de Oliveira.

S586c Certification of programs via Gödel numbering [recurso eletrônico] /Guilherme de Oliveira Silva. Belo Horizonte – 2025.
1 recurso online (68 f. il., color.) : pdf.

Orientador: Fernando Magno Quintão Pereira.

Dissertação (Mestrado) - Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação.
Referências: f. 63-66.

1. Computação – Teses. 2. Compiladores (Programas de Computador) – Certificados e licenças – Teses. 3. Programas de Computador – Validação – Teses. I. Pereira, Fernando Magno Quintão. II. Universidade Federal de Minas Gerais Instituto de Ciências Exatas, Departamento de Ciência da Computação. III. Título.

CDU 519.6*33(043)

Ficha catalográfica elaborada pela bibliotecária Irénquer Vismeg Lucas Cruz
CRB 6/819 - Universidade Federal de Minas Gerais - ICEx

INSTITUTO DE CIÊNCIAS EXATAS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

CERTIFICATION OF PROGRAMS VIA GÖDEL NUMBERING

GUILHERME DE OLIVEIRA SILVA

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores(a):

Prof. Fernando Magno Quintão Pereira - Orientador
Departamento de Ciência da Computação - UFMG

Prof. Arthur Azevedo de Amorim
Department of Computer Science - Rochester Institute of Technology

Prof. Xavier Rival
Department of Computer Science - INRIA Paris

Belo Horizonte, 12 de setembro de 2025.

To my wife, Laura.

Acknowledgments

This work was only made possible thanks to many people. First, I would like my family—especially my wife, Laura; my parents, Adriana and Wellyngton; and my brother, Vinícius—for their unwavering support throughout this journey. I'm thankful for your everlasting love, your words of encouragement, and for the selfless understanding of all the long weekends we've relinquished our time together so I could work on this thesis. I love you.

I'd like to also thank my advisor, Prof. Fernando Pereira, for his guidance. Thank you for teaching me how to be a better researcher, computer scientist, and citizen—this work is for the good of the Brazilian people, after all. You're not only a great mentor, but also a good friend.

To all the great folks I've met at the Compilers Lab, I also appreciate and treasure every moment we've spent together. To Caio Raposo, João Amorim, Lucas Victor, Michael Canesche, Rafael Sumitani, and Thaís Damásio: thank you very much!

I also would like to extend my appreciation to my friends and colleagues at Cadence Design Systems—in particular my manager and friend, Vanderson Rosário—, to UFMG, and to the Department of Computer Science.

"*The road to wisdom? Well, it's plain. And simple to express: Err and err and err again, but less and less and less.*"

(Piet Hein)

Resumo

Em sua palestra ao receber o Prêmio Turing em 1984, Ken Thompson demonstrou que um compilador poderia ser sistematicamente adulterado para que insira *backdoors* em programas que compila e para que perpetue este comportamento ao modificar qualquer compilador que construa posteriormente. A brecha apontada por Thompson já foi reproduzida em sistemas do mundo real para fins de demonstração. Diversas contramedidas foram propostas para se defender de *backdoors* desta natureza, incluindo a famosa técnica de Compilação Dupla Diversa (*Diverse Double-Compiling*, ou DDC), e as abordagens de validação de tradução e de compilação no estilo CompCert. Estas solução, entretanto, sempre retornam à pergunta fundamental: "*Como podemos confiar no compilador que compilou as ferramentas das quais dependemos?*"

Esta dissertação propõe uma nova abordagem para gerar certificados que garantam que um binário represente fielmente o código fonte. Estes certificados asseguram que o binário contém todas — e somente isto — as operações do código fonte, que preserve sua ordem, e que mantenha as mesmas relações de dependência entre definição e uso. O certificado é representado como um inteiro que possa ser derivado a partir de ambos o código fonte e o binário, utilizando um conjunto conciso de regras de derivação aplicáveis em tempo constante. Para demonstrar a praticidade do método proposto, apresentamos Charon, um compilador desenhado para lidar com um subconjunto da linguagem C expressivo o suficiente para compilar programas escritos na linguagem de programação criptográfica FaCT (*the Flexible and Constant Time*).

Palavras-chave: certificação de programas; validação de tradução; numeração de Gödel.

Abstract

In his 1984 Turing Award lecture, Ken Thompson showed that a compiler could be maliciously altered to insert backdoors into programs it compiles and perpetuate this behavior by modifying any compiler it subsequently builds. Thompson's hack has been reproduced in real-world systems for demonstration purposes. Several countermeasures have been proposed to defend against Thompson-style backdoors, including the well-known *Diverse Double-Compiling* (DDC) technique, as well as methods like translation validation and CompCert-style compilation. However, these approaches ultimately circle back to the fundamental question: "*How can we trust the compiler used to compile the tools we rely on?*"

This thesis proposes a novel approach to generating certificates to guarantee that a binary image faithfully represents the source code. These certificates ensure that the binary contains all and only the statements from the source code, preserves their order, and maintains equivalent def-use dependencies. The certificate is represented as an integer derivable from both the source code and the binary using a concise set of derivation rules, each applied in constant time. To demonstrate the practicality of our method, we present Charon, a compiler designed to handle a subset of C expressive enough to compile FaCT, the Flexible and Constant Time cryptographic programming language.

Keywords: program certification; translation validation; Gödel numbering.

List of Figures

Contents

Chapter 1

Introduction

In his Turing Award lecture, "*Reflections on Trusting Trust*" [39], Ken Thompson introduced an influential concept in cybersecurity: he described an attack in which a compiler is modified to insert a backdoor into the UNIX login command. The brilliance of this attack lies in its ability to self-propagate: the compromised compiler not only injects the backdoor but also recognizes and reinserts it into new compiler versions, even if the malicious code is removed from the source. The concrete implementation of the attach is relatively simple [11], something noticeable, given how hard is to prevent it. Thompson's work highlights two critical truths: software tools can be compromised in ways that evade traditional inspection, and trust must extend beyond source code to include the entire toolchain. By exposing this hidden vulnerability, Thompson laid the groundwork for understanding and addressing *software supply chain attacks* [8, 22, 30, 31], a concern that remains relevant today.

1.1 The Contribution of This Work

This thesis proposes a technique to certify that a target (low-level) program was produced as the result of translating a source (high-level) program. Said technique is based on the famous encoding Kurt Gödel used in the proof of his incompleteness theorems [18]. As explained in Chapter 3, the certificate of faithful compilation is embedded within the structure of the high-level program and in the structure of the low-level program, rather than existing as a separate artifact, as is the case with proof-carrying code. Figure 1.1 illustrates the proposed methodology.

In Figure 1.1, p_h is a program written in a high-level source language $Lang_H$, and p_ℓ is the program written in a low-level target language $Lang_L$, produced by a compiler $Comp$, i.e., $p_\ell = Comp(p_h)$. Figure 1.1 defines two mapping algorithms, $H : Lang_H \mapsto \mathbb{N}$ and $L : Lang_L \mapsto \mathbb{N}$. If $Cert_H(p_h) = Cert_L(Comp(p_h))$, then $Comp$ has not modified the compilation contract, meaning that $Comp$ has generated only the instructions necessary

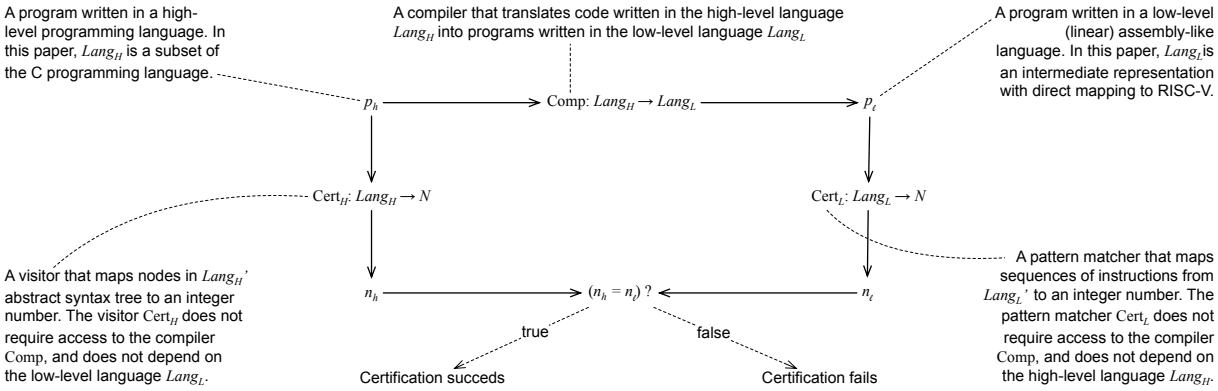


Figure 1.1: Overview of the proposed certification methodology.

to implement in p_ℓ the semantics of p_h. Example 1 will clarify these ideas.

Example 1. *Figure 1.2 shows how we produce a certificate for a piece of code that is often considered the shortest valid C program. This example illustrates some basic principles behind our approach. First, a certificate is a product of powers of prime numbers. The base of such exponents, e.g., numbers such as 2, 3 and 5 in Figure 1.2 mark the position where each syntactic construct appears (first, second, third, etc) in the program's AST, or in the low-level intermediate representation (IR) that we adopt in this work. The exponents are associated with particular programming constructs in the high-level language, or with particular sequences of instructions in the low-level language. For instance, a* `return` *statement in the high-level language is associated with the exponent 19. The code that corresponds to a* `return` *statement in the low-level language uses two opcodes:* `MOV` *and* `JR`*. The certificate of a program is the product of the certificate of all the constructs that make up that program. This sequence is also associated with exponent 19. In this example, this certificate is* $2^2 \times 3^{19} \times 5^{113}$*. In this work we adopt as the low-level language a linear intermediate representation (IR) that has one-to-one correspondents with RISC-V instructions, as Figure 1.2 shows. Thus, by certifying the intermediate representation, we also certify its corresponding RISC-V image.*

Example 1 illustrates how the proposed methodology works on a very simple program. This example misses two capacities of this approach: the ability to certify high-level constructs that are translated to non-contiguous sequences of instructions (like loops and branches); and the ability to encode def-use relations, as we would have once the high-level program manipulates variables. Chapter 3 will explain how we deal with such constructs.

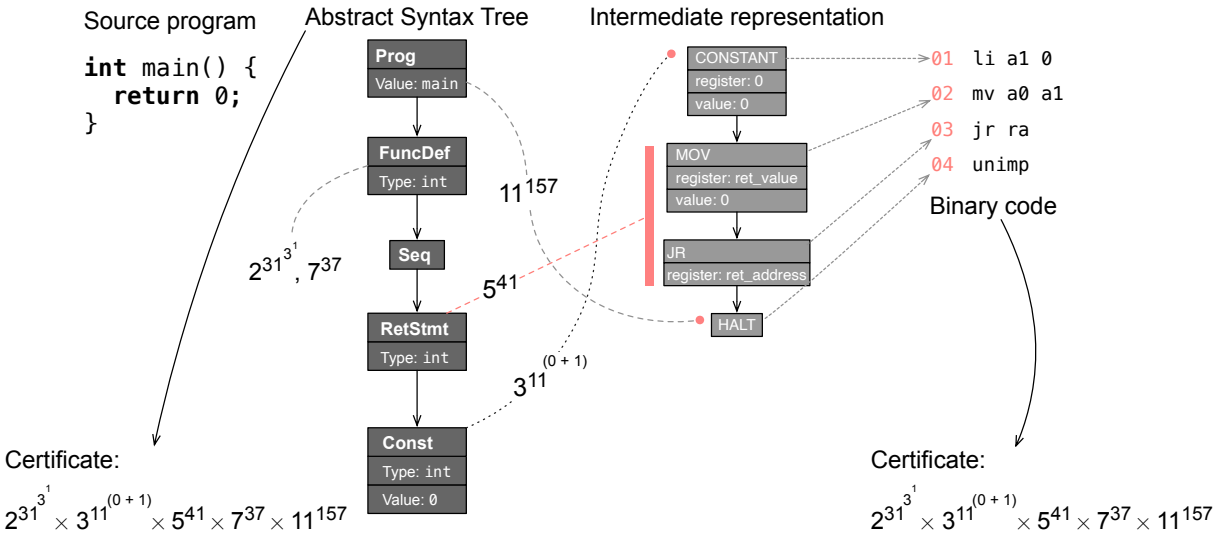


Figure 1.2: How CHARON generates certificates for a simple C program.

1.2 Summary of Results

We have implemented the ideas proposed in this work in a compilation framework called CHARON. This compiler is capable of certifying a subset of C that is expressive enough to support FACT [7] programs. FACT is a restricted version of C designed to facilitate constant-time programming. It provides users with three control-flow constructs: `if-then-else`, `while`, and `return`, along with a type system that allows for the specification of public and classified data. `Charon` supports all the control-flow constructs in FACT, three data types (`int16`, `int32`, and `float`), and implicit type coercions. Henceforth, we call this version of FACT (the $Lang_H$ in Figure 1.1), CHARONLANG. Similarly, we name CHARONIR our target low-level intermediate representation (the $Lang_L$ in Figure 1.1).

Our certification methodology provides the guarantees shown in Figure 1.1, along with the property of *invertibility*. Invertibility means that there exists a function $Canon : \mathbb{N} \mapsto Lang_H$, such that if $Cert_H(P_H) = n$, then $Canon(n) = P'_H$, where P'_H is the *Canonical Representation* of program P_H. As we explained in Section 3.6, P_H and P'_H need not to be syntactically identical, but they are guaranteed to be semantically equivalent. This result mimic Gödel's usage of of the *Fundamental Theorem of Arithmetics* (FTA), which states that every integer greater than one can be expressed uniquely (up to the order of factors) as a product of prime numbers. Building on this fact, Section 4 shows that certificates are invertible at both the high-level and low-level representations of programs.

The absolute value of a certificate, e.g., measured as the number of bits necessary to represent it, grows exponentially with the size of programs. This asymptotic behavior characterizes $Cert_H$ and $Cert_L$. In the former case, because a certificate contains a multiplicative factor for each construct in the program's AST. In the latter, because the

certificate contains a multiplicative factor for each instruction in the program's low-level representation. Nevertheless, as we show in Chapter 5, the symbolic representation of a certificate, as a string that encodes a series of multiplications, is still linear on the size of the program, be it written in CharonLang or in CharonIR.

1.2.1 Software and publications

The main product of this work is the Charon project. It consists of the compiler and a suite of benchmarks, as discussed in Chapter 5. It is publicly available under the GPL 3.0 license and can be obtained at https://github.com/guilhermeolivsilva/project-charon.

Additionally, this work inspired the publication of a paper, that is under submission at the time this thesis is being prepared. A version of this work can be retrieved on arXiv [14].

Chapter 2

Background

In this chapter, we examine the fundamental challenge of establishing trust in software, especially in the presence of Ken Thompson's "Trusting Trust"-style attacks. As such, we discuss methods that aim to strengthen confidence in software correctness and safety, such as Diverse Double-Compiling, Proof-Carrying Code, Translation Validation, Certified Compilation, and Symbol Certification.

2.1 Revisiting "Reflections on Trusting Trust"

A seminal illustration of this issue of establishing trust in software is found in Ken Thompson's "Reflections on Trusting Trust" Turing Award Lecture [39]. In this context, Thompson demonstrates how a carefully crafted compiler backdoor can perpetuate itself undetected, even in the presence of complete source code transparency. This example underscores the inherent difficulty of verifying the integrity of the very software toolchains and the dependencies they produce. In this section, we discuss the Thompson hack in more detail (Subsection 2.1.1), and then explore the question of how trust can be established in software in adversarial contexts (Subsection 2.1.2).

2.1.1 The Thompson Hack

Ken Thompson's famous "Trusting Trust" hack demonstrates a self-replicating backdoor embedded into a compiler – the "Trojan Compiler". Thompson's exploit starts with a clean compiler, which does not contain any backdoors. This compiler is trusted by the user and produces backdoor-free binaries. The attack leverages two backdoors to propagate malicious behavior, even when the source code appears clean. The process can

be described in four stages, as illustrated in Figure 2.1.

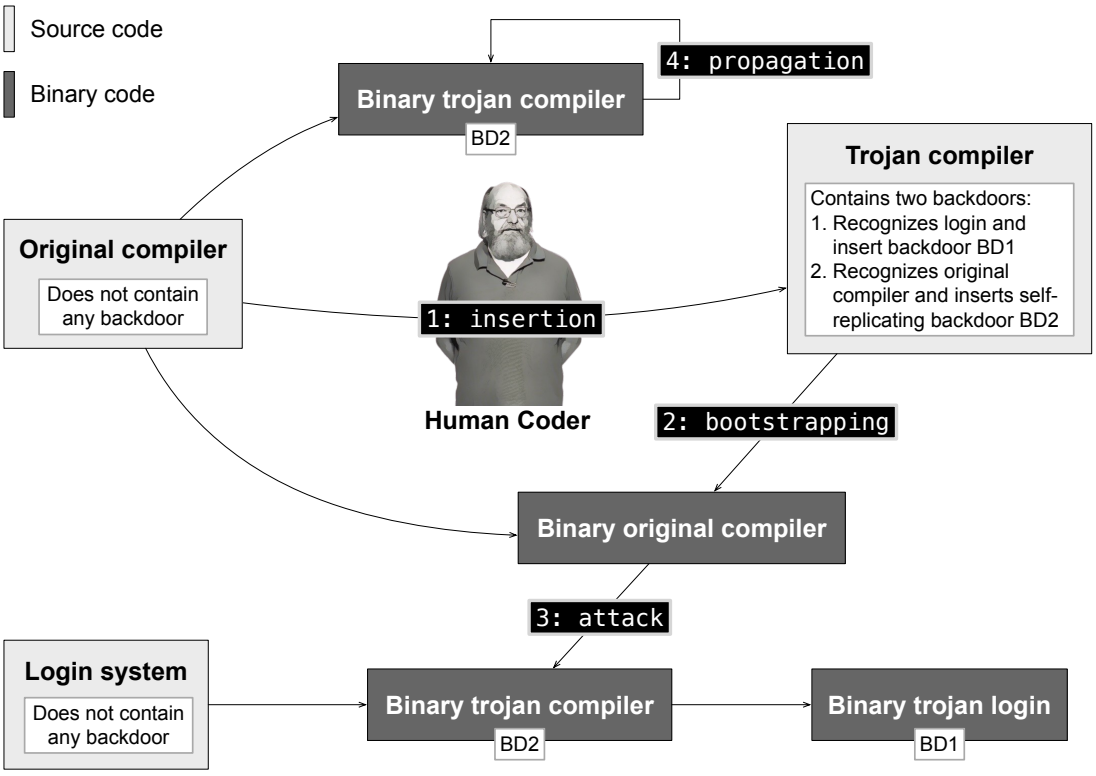


Figure 2.1: A schematic view of Thompson's "Trusting Trust" attack.

1. **Insertion:** In Thompson's hack, a (malicious) user modifies the compiler source to produce a "trojan compiler", which contains two backdoors:

   - **BD1:** This backdoor recognizes when the trojan compiler is compiling the `login` program and inserts a malicious backdoor (e.g., a hardcoded password) into the compiled binary.
   - **BD2:** This self-replicating backdoor recognizes when the trojan compiler is compiling its own source code and reinserts both BD1 and BD2 into the new compiler binary.

2. **Bootstrapping:** Once the trojan compiler's source is compiled with the original compiler, it produces a binary version of the trojan compiler. This binary now contains both backdoors, BD1 and BD2. At this point, the source code of the trojan compiler is no longer necessary, and can be removed. Therefore, the source code of the compiler appears clean, as BD2 exists only at the binary level.

3. **Attack:** When the binary trojan compiler compiles the `login` system, it inserts BD1 into the resulting binary, creating a `login` login system.

4. **Propagation:** When the binary trojan compiler compiles a clean version of its own source code, it reinserts BD1 and BD2 into the new binary compiler. This ensures

> the trojan behavior propagates indefinitely, even if the source code is reviewed and found to be clean.

The genius of this attack lies in its self-propagating nature: the malicious behavior becomes embedded at the binary level and is undetectable through inspection of the source code alone. Trusting the compiler binary is therefore essential, as the hack demonstrates how source code auditing alone cannot guarantee the absence of malicious behavior.

2.1.2 The Challenge of Building Trust

In Thompson's word, instead of trusting that a program is free of Trojan horses, "*Perhaps it is more important to trust the people who wrote the software*" [39]. Nevertheless, various techniques have been developed to enhance trust in software and mitigate the risks of Thompson-style backdoors. One prominent approach specifically addressing this issue is the Diverse Double-Compiling technique, proposed by David A. Wheeler in his PhD dissertation [41, 42]. As Section 2.2.1 further explains, this method compares binaries produced by different compilers to identify malicious behavior in a potentially compromised compiler.

Going beyond Wheeler's certification methodology, programming language techniques provide broader mechanisms to build trust in software. Translation validation [33, 28] verifies that the output of a compiler faithfully preserves the semantics of the input program by validating each individual compilation rather than the entire compiler. Proof-carrying code (PCC) [27] allows a program consumer to ensure that code provided by an untrusted producer satisfies specific safety properties without needing to trust the producer. Ultimately, compilers can be fully verified. For instance, CompCert [23] is a compiler whose correctness is formally guaranteed via the Coq proof assistant. Together, these techniques strengthen the foundations of trust in the software ecosystem. And yet, the fundamental issue of "*who we trust*" still plagues them all.

The aforementioned techniques — Diverse Double-Compiling (DDC), Proof-Carrying Code (PCC), Translation Validation, and certified compilation (as in CompCert) — each have limitations in addressing the fundamental challenge raised by Thompson's "Trusting Trust" attack. These limitations arise primarily because demonstrating that a binary faithfully implements the semantics of arbitrary source code requires solving the problem of semantic equivalence, which is undecidable in general due to its relationship to the Halting Problem [9, 40].

2.2 Countering Thompson-style backdoors

The challenge posed by Thompson-style backdoors is a persistent concern for any system that relies on compiled code. Once such a backdoor is embedded in the toolchain, it can invisibly propagate across builds, surviving even complete source code audits. Therefore, proposed countermeasures will try to establish a verifiable correspondence between trusted specifications and generated binaries. In this section, we discuss how the most prominent techniques—Diverse Double-Compiling, Proof-Carrying Code, Translation Validation, Certified Compilers, and Symbolic Certification—address threats of this nature.

2.2.1 Diverse Double-Compiling

Diverse Double-Compiling (DDC) [41, 42], along with its many adaptations [20, 36, 38, 15], is a defense technique proposed by David Wheeler in his PhD dissertation. It seeks to detect malicious behavior introduced by a compromised compiler by comparing the outputs of independently built compiler binaries. The method proceeds in four steps:

1. Compile the source code of the compiler C under test using a trusted compiler C_t, producing an executable compiler $X_t = C_t(C)$.
2. Compile the same source using the suspect compiler C_s, producing an executable compiler $X_s = C_s(C)$.
3. Use both resulting compiler executables to compile a test program p.
4. Compare the resulting binaries, i.e., $b_t = X_t(p)$ and $b_s = X_s(p)$.

The key insight is that a trusted compiler C_t, assumed to be free of malicious logic, should produce a clean version X_t of the compiler C. If the executables X_t and X_s produce different binaries in Step 3, this discrepancy can be detected through syntactic comparison in Step 4.

DDC serves as a practical defense against self-propagating attacks such as Thompson's "Trusting Trust" backdoor. However, it relies on two assumptions [6]: (i) the existence of at least one trusted compiler C_t, and (ii) that both X_t and X_s produce deterministic outputs for the same input. Furthermore, any differences observed in Step 4, such as $b_t \neq b_s$, require additional investigation to determine whether they stem from benign implementation differences or intentional tampering. While this technique reduces

the likelihood of an undetected backdoor, it cannot eliminate the possibility entirely. If both compilers used in the DDC process are compromised, then Thompson's backdoor may still persist.

This work and Diverse Double-Compiling (DDC) share the same goal: increasing confidence that a compiler is not inserting malicious code into its output. However, they belong to different categories. DDC is a comparative technique: it detects discrepancies by building the same compiler with multiple toolchains and comparing outputs. The work's approach, instead, establishes a structural correspondence (a morphism) between two languages: the high-level source language and the low-level target language. In this sense, the two techniques are complementary: the compiler described in this thesis could be used as one of the compilers in a DDC setup.

2.2.2 Proof-Carrying Code

The proof-carrying code (PCC) framework enables software developers to generate safety proofs to attest a program complies to a safety policy, and allows software consumers to independently verify the proof. If the proof succeeds, then the software must be compliant. As this mechanism is executed on the low level, assembly representation of the program, a maliciously modified compiler would not be able to generate the same proof as a regular compiler. Thus, code injection attacks would be exposed. [27].

Necula's framework operates in three steps:

1. Code consumers specify a safety policy under which a program is considered safe to run, such as the absence of out-of-bounds memory accesses (e.g., ensuring all array accesses `arr[i]` satisfy $0 \leq$ `i` $<$ `size`). The policy has two components: a set of safety rules and an interface. The safety rules list all the allowed operations and its associated preconditions, while the interface contains calling conventions to intermediate the communication between the code consumer and the program.

2. A certifying compiler produces low level machine code simultaneously to a formal proof that the program complies to the provided set of rules. It outputs an executable file for the program, and its safety certificate.

3. Code consumers check the certificate with a general use validator (i.e., a piece of software that is able to validate the safety proof of any program compiled with this framework). If is succeeds, the program can be safely executed.

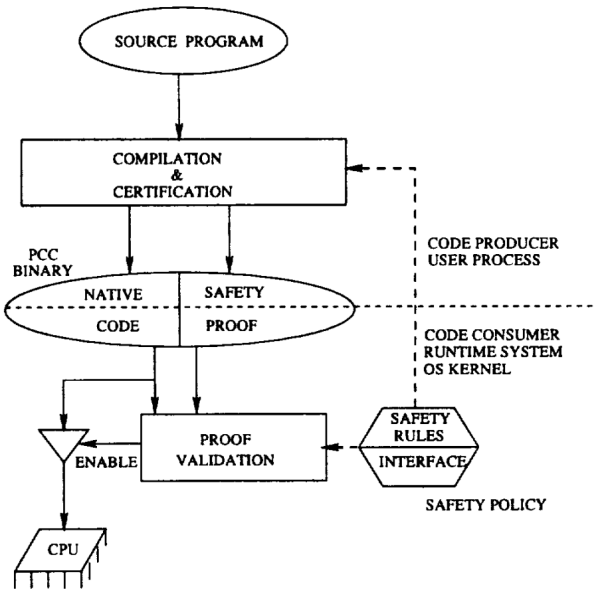


Figure 2.2: Necula's PCC flow diagram. [27].

This framework has been implemented using multiple levels of abstraction or specification languages [16]. The different strategies include the development of certifying compilers for Java [10], for subsets of the C language [29, 26], and a universal type framework that proves the adherence to safety rules as lemmas in higher-order logic [2]. While effective at verifying certain well-defined properties, PCC does not address the presence of malicious behavior, such as Trojan horses, in the binary.

2.2.3 Translation Validation

Another approach to increase the user's confidence in third party provided software is to validate the translation from high level programming language source code to the low level machine code. If the semantics are preserved and the source code is trustworthy, then the compiled program can be trusted as well. This is achieved by passing both the source code and the compiled program to an analyzer, that will test whether the latter correctly implements the former. If it does, it will generate a formal proof; if not, it will present a counter example. This technique was first introduced in Pnueli et al. [33] "Translation Validation".

Translation validation verifies the correctness of individual compiler transformations, such as register allocation [34] or constant propagation [32], rather than the entire compiler — it checks the correctness of each individual compilation. By focusing on specific transformations, translation validation avoids undecidability. As such, it may be able to validate

programs transformed by optimization flags: one could verify the semantics preservation in the intermediate representation of a program before and after each optimization pass applied by the compiler. This enables the practical use of translation validation together with industrial strength compilation infrastructures, such as GNU C. [28].

The translation validation process requires several modules to operate. Pnueli's abstraction requires a common semantic framework that simultaneously supports the semantics of both the source code and the compiled code; a formal standard of implementation correctness; an automatic proof method that allows one to determine whether the semantics of a source code correctly implements the semantics in some machine code (the *analyzer*), and generates a certificate to be further verified (*proof script*); and a verification method able to check the generated proof and validate the translation. These steps combined produce the translation validation flow presented in Figure 2.3

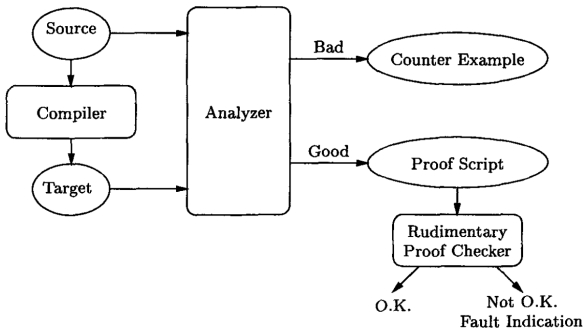


Figure 2.3: Pnueli's translation validation flow diagram. [33].

Translation validation, however, is inherently limited to the scope of the program and transformations being validated. If validation fails, it may indicate a limitation in the validation technique rather than an actual error in the compiler.

2.2.4 Certified Compilers

Proof-carrying code and translation validation systems must implement semantics for the language of types and the machine language for each particular machine architecture. For each type system, it must also use a verification software able to derive logical formulae that guarantees the safety or the semantic preservation of the analyzed code (*verification condition*). This, however, raises the *quis custodiat ipsos custodes* ("who verifies the verifier itself?") problem, as the verification software itself is not constructed to be mechanically-checkable correct — as doing so is a complex task. Thus, a bug in the validator could lead to a falsely checked proof. [1].

A more robust strategy is to use a formally verified compiler. Such a compiler is bundled with a formal, machine-checked proof that it generates low level machine code with exact same semantics specified by the high level programming language it compiles. This verification can be achieved by using a set of compilation passes that are formally proven to preserve semantics, together with a source, intermediate and a target languages with formally defined, deterministic semantics. [25].

CompCert, a compiler that generates PowerPC assembly code from a large subset of the C language (*Clight*), is formally verified and guarantees that a source code deemed safe will generate an equally safe executable compiled code. The compiler uses Coq to prove a program specification is correct, and is itself written in Coq — its executable Caml code is extracted from Coq's functional specifications. As Clight supports pointer arithmetic, struct and union types, C loops and structured switch statements, CompCert is able to compile a broad arrange of programs, and benchmark against GCC. On average, the code it generates is on par with unoptimized GCC programs, but 7-12% slower than code optimized with `-O1` or `-O2`. [23, 5].

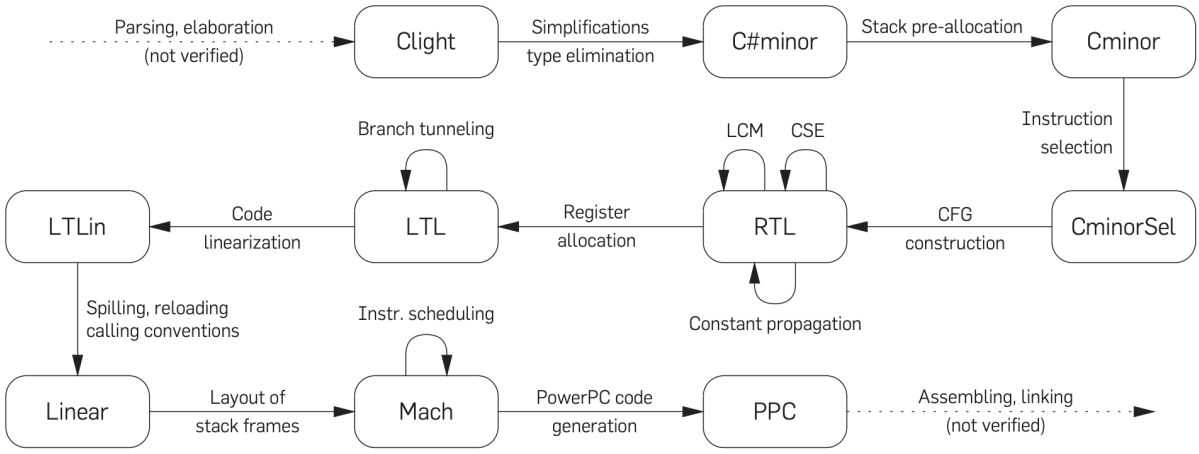


Figure 2.4: CompCert compilation workflow. [23].

Implementing a formally verified compiler, nonetheless, is a challenging endeavor. One of the challenges it imposes is to produce an executable file for the compiler from Coq, as its extraction facility is not formally verified. If the toolchain itself has a bug or is inconsistent, it could invalidate the guarantees obtained from the correctness proof. Moreover, to add other compilation targets to such compilers may require significant changes to its back-end, and to add optimizing transformations to such compiler one must formally prove it correct as well — further increasing the required work for it to be practical. [24].

2.2.5 Symbolic Certification

Symbolic certification [4, 35, 19, 43, 21] refers to techniques that verify the correctness of compiled code by constructing and comparing symbolic representations of program semantics at both the source and binary levels, ensuring that they exhibit behaviorally equivalent execution.

In Rival's [35]'s framework, the correctness of compilation is established through the construction of *symbolic transfer functions* (STFs), which encode the semantics of source and binary code in a compositional manner by "precisely describing the store transformation between two program points". Correctness is then verified by demonstrating that these symbolic descriptions are behaviorally equivalent: it checks whether corresponding program points in the source and assembly programs assign the same values to the same memory locations. This approach not only supports optimizing compiler backends, but it also enables the verification of semantic properties such as the absence of runtime errors or the preservation of invariants.

Example 2. *To illustrate this framework, consider the source and assembly programs displayed in Figure 2.5. The source program points l_0^s, l_1^s, l_2^s, l_3^s, l_4^s, l_5^s, l_6^s correspond to the assembly program points l_0^a, l_2^a, l_4^a, l_8^a, l_{11}^a, l_{15}^a, l_{16}^a, respectively. One of the requirements for the compilation to be considered correct is the value of the variable i in the source program point l_0^s to be the same the assembly program stores in point l_0^a.*

This can be checked by simultaneously evaluating Symbolic Transfer Functions regarding the source and assembly programs. The STF for the assignment operation $l : lv := e'; l' : \ldots$ in the source program is expressed as $\delta_{l,l'} = C_{lv}(lv, C_e(e, \lfloor lv \leftarrow e \rfloor); l' : \ldots$, where lv is an l-value, e is an expression, and C evaluates the first value prior to computing the STF in the second value. As for the assembly language, the STF for the load integer instruction $l : \texttt{li} : r_0, n; l' : \ldots$ is $\delta_{l,l'} = \lfloor r_0 \leftarrow n \rfloor; l' : \ldots$; and the STF for the store instruction $l : \texttt{store} : r_0, \underline{x}(v); l' : \ldots$ is $\delta_{l,l'} = \lfloor \texttt{isaddr}(\underline{x} + v)? \lfloor \underline{x} + v \leftarrow r_0 \rfloor \mid \square \rfloor$. As such, this requirement will be satisfied if both STFs hold.

$$\delta_{l,l'} = C_{lv}(i, C_e(e, \lfloor i \leftarrow -1 \rfloor)$$

$$\delta_{l,l'} = \lfloor r_0 \leftarrow -1 \rfloor; \lfloor \texttt{isaddr}(\underline{i} + 0)? \lfloor \underline{i} + 0 \leftarrow r_0 \rfloor \mid \square \rfloor$$

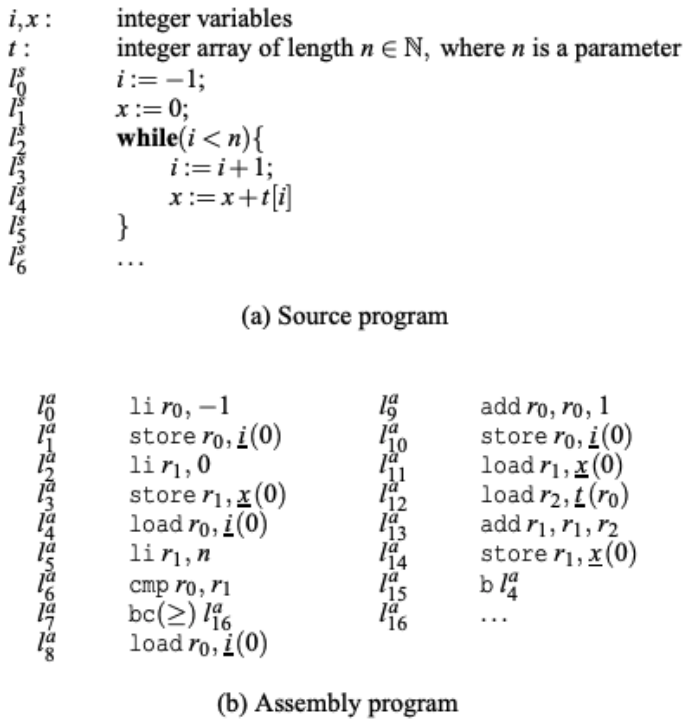


Figure 2.5: Source and assembly programs. [35].

Zaks' and Pnueli's [43] approach also supports optimizing backends. In particular, it is preserved by most of intraprocedural optimizations, such as constant folding or dead code elimination. Their work is based on the analysis of the cross-product of two input programs, S and T, without relying on compiler annotations. Instead of checking whether S and T are equivalent, this technique produces another program, C, from pairs of states of the input programs, and checks if it satisfies a safety property. The safety property establishes that state pairs must hold the same values for identical inputs at certain program points, such as function exits or loop boundaries, and that outputs must match at the end.

Although symbolic techniques like that of Rival [35] and Zaks and Pnueli [43] offer broad expressiveness and can validate aggressive compiler optimizations, they still ask for trust in the verification infrastructure. As such, it still falls short on fully mitigating Thompson-style backdoors.

2.3 Final Remarks

The techniques reviewed in this chapter—Diverse Double-Compiling, Proof-Carrying Code, Translation Validation, Certified Compilation, and Symbol Certification—represent

significant milestones in the ongoing effort to establish trust in software systems. Each provides valuable tools for detecting or preventing certain classes of malicious or erroneous behavior. Therefore, each of these techniques represents a step toward building trust in software, but none can completely eliminate the risks highlighted by Thompson's seminal work.

This recognition motivates the need for new approaches that either reduce the trust assumptions we must make, or make them more transparent and verifiable. In the next chapter, we introduce what we call "Structural Certification", a certification methodology in which the layout of the binary itself encodes evidence of its derivation from the source code.

Chapter 3

Structural Certification

This chapter introduces our solution to the "Trusting Trust" problem, using what we call "Structural Certification". This technique assigns unique identifiers to each construct in both the high- and low-level representations of a program. These identifiers are integers computed from a numbering system based on *Gödel Numbers*. This chapter is structured as it follows:

- Section 3.1 introduces Gödel Numbers, and discusses how it can be used to encode constructs such as an arithmetic expression.
- Section 3.2 introduces the CHARON compiler, that implements our solution to the "Trusting Trust" problem. It discusses its scope and capabilities.
- Section 3.3 presents the Numbering Schema we've designed to encode each construct the CHARON compiler supports.
- Section 3.4 presents how high-level programs, written in the CHARONLANG language, are certificated.
- Section 3.5 discusses the certification of low-level programs generated by the CHARON compiler.
- Section 3.6 introduces the canonical form and how certificates can be inverted back into the original, high-level program.

3.1 Gödel Numbering

We aim to certificate programs by assigning unique identifiers to each instruction that encodes the operation, its operands, and its relative position in the program simultaneously. These identifiers are then combined to produce a single number that uniquely represents a program in such manner that any changes made to the order of operations

or operands will produce a different certificate. This is accomplished with a numbering system based on *Gödel Numbers*. This numbering system works according to the following sequence of steps, which Example 3 will illustrate:

1. **Assign Unique Numbers to Symbols:** Create an *encoding table* where each symbol in the logical system (e.g., variables, operators, parentheses) is assigned a unique integer.

2. **Map Each Position to a Prime Number:** Use the sequence of prime numbers $(2, 3, 5, 7, \ldots)$ to represent the positions of symbols in the expression. The n-th symbol in the expression corresponds to the n-th prime number.

3. **Calculate the Contribution of Each Symbol:** For each symbol s_i at position i:

   - Take the i-th prime p_i as the *base*.
   - Use the integer assigned to s_i from the encoding table as the *exponent*.
   - Compute $p_i^{e_i}$, where e_i is the encoded value of s_i.

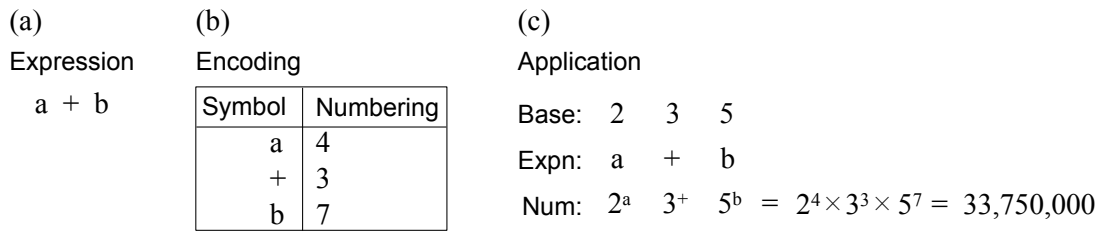
4. **Multiply All Contributions:** Compute the Gödel number by multiplying the contributions of all symbols:

$$G = \prod_{i=1}^{n} p_i^{e_i}$$

   where n is the total number of symbols in the expression.

Example 3. *Figure 3.1 shows how we can encode the expression $a + b$ using a Gödel Numbering System. Let us assume that our example system assigns the following numbers to each potential symbol in our arithmetic system: $a \mapsto 4, + \mapsto 3, b \mapsto 7$. Similarly, let us assume that the order in which symbols appear in the logic expression are associated with the primes in ascending order, e.g.: $1^{st} \mapsto 2, 2^{nd} \mapsto 3, 3^{rd} \mapsto 5, \ldots$. Given these assumptions, we compute the contribution of each one of the three symbols in the expression as follows: $2^4, 3^3$ and 5^7. The final encoding is the product of all these parcels, e.g.,* $16 \times 27 \times 78,125 = 33,750,000$.

Gödel's numbering relies on the Fundamental Theorem of Arithmetic [17], which states that every integer has a unique prime factorization. This ensures that each Gödel number maps uniquely to one expression and that the original expression can be reconstructed by factorizing the number. Thus, the Gödel Numbering System achieves two properties: *uniqueness* and *reversibility*. The numbering system proposed in this work also meets these two properties.



(a) The Gödel Numbering System assigns a unique number to each expression that can be written in a logical system. In this example, we have the expression "a + b", which encodes the addition of variables a and b.

(b) The system consists of an encoding, a table that associates symbols with numbers. These numbers will encode how many times different prime numbers appear in the factorization of the number that represents the expression. In this example, variable "a" is associated with the exponent 4; the symbol "+" with three, and so on.

(c) To assign a number to an expression, we associate each symbol of that expression with a prime number. This prime number is given by the position of the symbol within the expression. For instance, the first symbol will be associated with 2, the second with 3, the third with 5, and so on. Then, we let this prime be the base of an exponentiation, where the exponent is given by the encoding of the symbol in the encoding table. And finally, we multiply all these terms to have the final Gödel number associated with an expression.

Figure 3.1: An example of encoding based on Gödel Numbers.

3.2 The Charon Compiler

Our solution to the *trusting trust* problem is implemented by the Charon compiler. It includes a toy language, CharonLang, that covers a large subset of the C language, and a compiler that targets an intermediate representation (CharonIR) inspired by the RISC-V architecture [3].

CharonLang The Charon toy language supports all the control-flow constructs present in FaCT, the Flexible and Constant Time cryptographic programming language [7], namely forward and backward branches (loops) plus non-recursive function calls. In this regard, we adopt the implementation of FaCT available in the work of **(author?)** [37]. Additionally, this subset of the C language supports integers (16-bit `short`, 32-bit `int`) and 32-bit floating point (`float`) types, plus (implicit) type casts[1] and all the ANSI C unary and binary mathematical, logical, and bit-wise operations. CharonLang also supports static arrays and user-defined structures that combine built-in types. Similarly to FaCT, we chose not to add support to pointer arithmetic to CharonLang. Figure 3.2 presents its grammar specification.

[1]The compiler emits adequate type cast instructions to compatibilize operands. Users can not explicitly make type coercion operations.

Procedure definitions

func. def.	::=		
	|	$\theta\ f\ (\theta\ x)\{S\}$	internal procedure

Statements

S	::=		
	|	$S; S$	sequence
	|	$\theta_p\ x$	variable declaration
	|	$f(e)$	procedure call
	|	$x = e$	assignment
	|	**if** $(e)\ \{S\}$	if conditional
	|	**if** $(e)\ \{S\}$ **else** $\{S'\}$	if...else conditional
	|	**while** $(e)\ \{S\}$	while loop
	|	**return** e	procedure return

Expressions

e	::=		
	|	n	numeric literal
	|	x	variable
	|	$\Theta\ e$	unary operation
	|	$e\ \Theta\ e$	binary operation
	|	$x[e]$	array element
	|	$x.y$	struct element

Types

θ	::=		
	|	**short** | **int** | **float**	built-in types
	|	$\theta\ x[n]$	arrays
	|	**struct** $\{\ \theta\ x; \theta\ y;\ \ldots\ \}$	user-defined structures

Figure 3.2: The CHARON Toy Language grammar.

Example 4. *Figure 3.3 (a) shows the* CHARONLANG *implementation of a program that computes the maximum common divisor of two integers. Part (b) of that figure shows the program's abstract syntax tree. The compiler (Comp in Fig. 1.1) visits each node of the AST to produce the low-level program representation seen in Figure 3.3 (c). Similarly, the high-level certification algorithm ($Cert_H$ in Fig. 1.1) visits the AST to produce the program's certificate, as Section 3.4 will explain.*

Intermediate Representation The CHARON Intermediate Representation, CHARONIR, uses the set of instructions presented in Figure 3.4. Notice that the IR features floating-point variations for all the operations, except the bit-wise ones. These variations have been omitted for brevity. Most instruction parameters are registers, except for `imm`, which takes a constant literal.

Example 5. *Figure 3.3 (c) shows the three-address code representation of the implementation of maximum common divisor seen in Figure 3.3 (a). This example highlights that any value, variable or constant must be loaded into a temporary register before use.*

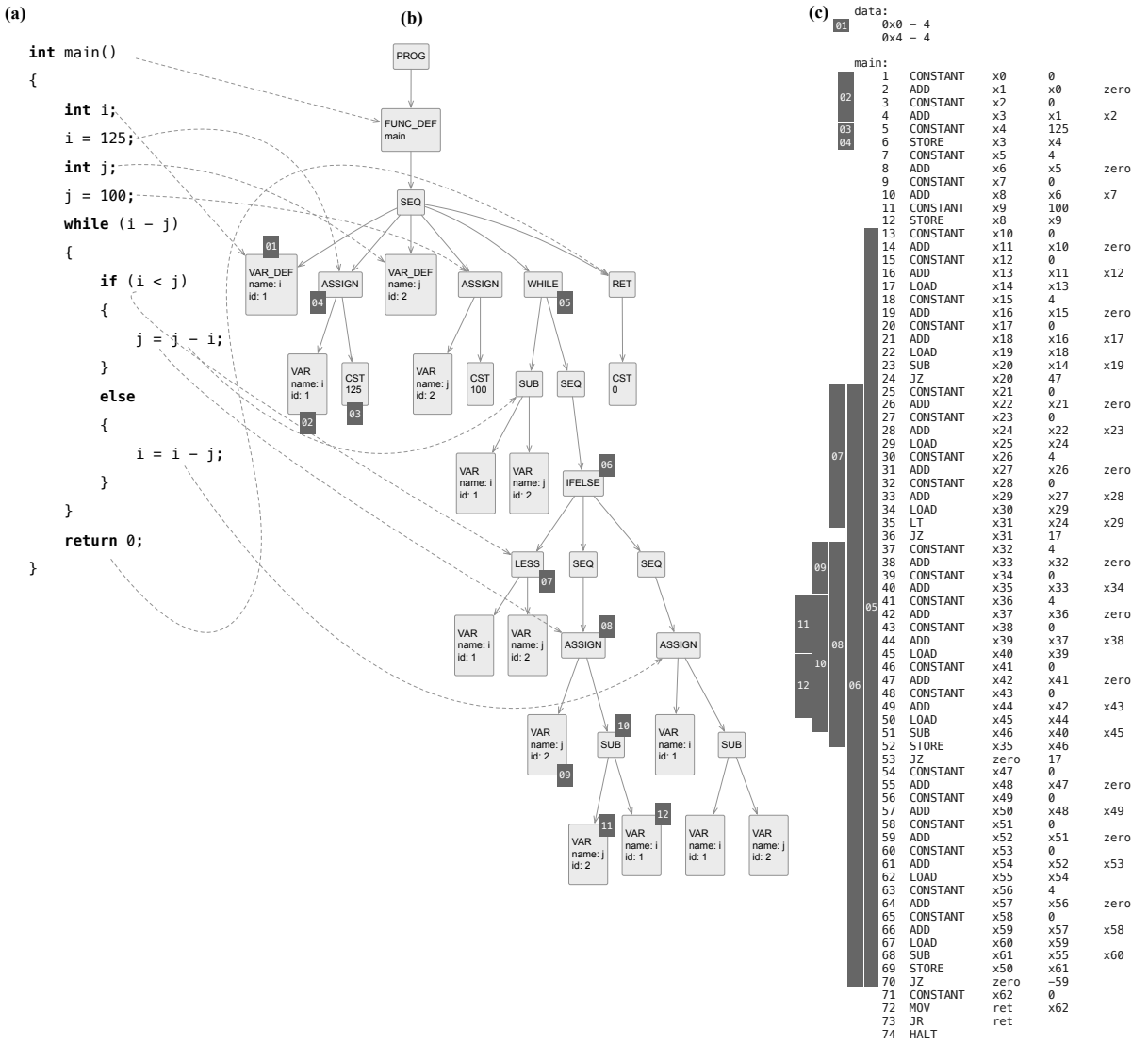


Figure 3.3: (a) Maximum common divisor implemented in CharonLang (b) Abstract syntax tree. (c) Low-level representation of the algorithm written in CharonIR.

CharonIr *provides the* `CONSTANT` *instruction to load constants, because this explicit handling of literals simplifies the construction of their certificates, as Section 3.5 will explain. Retrieving the contents of a variable with* `LOAD` *requires the compiler to save its address in a temporary register first. This is achieved in two steps: first, it will emit a* `CONSTANT` *with its temporary address; then, it will compute the offset, if any, and store the final address in the register to load from. Following RISC-V, an object file is exported with minimal metadata; namely, a* `data` *section that describes the lengths of variables identified by their base addresses.*

As Example 5 illustrates, Charon's intermediate representation follows the RISC-V syntax. The primary difference from RISC-V is that CharonIR has access to an infinite number of register names. Thus, CharonIR registers should be understood as memory locations instead of typical registers. Despite this, the IR includes some predefined

Operations

ADD RD R1 R2	Addition	AND RD R1 R2	Logical and
BITAND RD R1 R2	Bit-wise and	BITOR RD R1 R2	Bit-wise or
DIV RD R1 R2	Division	EQ RD R1 R2	Equality
GT RD R1 R2	Greater than	LSHIFT RD R1 R2	Left bit shift
LT RD R1 R2	Less than	MOD RD R1 R2	Modulo
MULT RD R1 R2	Multiplication	NEQ RD R1 R2	Not equal
NOT RD R1	Logical negation	OR RD R1 R2	Logical or
RSHIFT RD R1 R2	Right bit shift	SUB RD R1 R2	Subtraction

Variables and constants

CONSTANT RD IMM$_{value}$	Load constant value
LOAD RD R1	Load variable value to register
STORE RD R1	Store value in address
MOV R1 R2	Move between registers

Type casts

FPTOSI RD R1	`float` → `int`
SIGNEXT RD R1	`short` → `int`
SITOFP RD R1	`int` → `float`
TRUNC RD R1	`int` → `short`

Control Flow

JAL LABEL	Jump-and-link to label
JR R1	Jump to register
JZ R1 IMM$_{size}$	Branch if zero
HALT	Terminate execution

Figure 3.4: The CHARON intermediate representation. We let RD be destination and R1/R2 be source registers.

registers, adhering to the typical RISC-V/Linux Application Binary Interface. These predefined registers include one for storing the return address of functions, others for holding function arguments and return values, and a dedicated register that always contains zero for convenience. Variables in CHARONIR can be defined at multiple allocation sites, meaning the IR does not conform to the static single-assignment (SSA) format [12], so pervasive in modern compiler design.

Translation Specification Intermediate Representation code is generated by parsing CHARONLANG following the translation rules in Figure 3.5. Figure 3.5 uses a runtime environment T that maps variables to memory addresses. The n-th declared variable x will be mapped to its memory address, e.g., such as $T(x) \rightarrow address(n)$. Whenever a new variable or function is declared, the environment T is updated to a new T' that includes this definition. This mapping is necessary to distinguish variables that have the same name, when generating code (as seen in Figure 3.5), and when generating certificates.

$$\frac{n' = n + \text{len}(\theta) \quad T_{n'} = T_n + (v, n)}{T_n \vdash \mathsf{Comp}(\theta\ v) \xrightarrow{d} T_{n'}}$$

$$\frac{T \vdash \mathsf{Comp}(e) \xrightarrow{e} I_e, \mathtt{R_e} \quad \mathit{fresh}(\mathtt{R})}{T \vdash \mathsf{Comp}(\Theta\ e) \xrightarrow{e} I_e; \mathtt{UnOp}_\Theta\ \mathtt{R}\ \mathtt{R_e}, \mathtt{R}}$$

$$\frac{i' = i + \text{len}(f) \quad T_{i'} = T_i + (f, i)}{T_i \vdash \mathsf{Comp}(\theta_f\ f) \xrightarrow{d} T_{i'}}$$

$$\frac{T \vdash \mathsf{Comp}(e_0) \xrightarrow{e} I_0, R_0 \quad T \vdash \mathsf{Comp}(e_1) \xrightarrow{e} I_1, R_1 \quad \mathit{fresh}(\mathtt{R})}{T \vdash \mathsf{Comp}(e_0\ \Theta\ e_1) \xrightarrow{e} I_0; I_1; \mathtt{BinOp}_\Theta\ \mathtt{R}\ \mathtt{R}_0\ \mathtt{R}_1, \mathtt{R}}$$

$$\frac{T \vdash \mathsf{Comp}(S) \xrightarrow{s} I, T' \quad T' \vdash \mathsf{Comp}(S') \xrightarrow{s} I', T''}{T \vdash \mathsf{Comp}(S; S') \xrightarrow{s} I; I', T''}$$

$$\frac{T \vdash \mathsf{Comp}(e) \xrightarrow{e} I_e, \mathtt{R_e} \quad T \vdash \mathsf{Comp}(S) \xrightarrow{s} I_t, T'}{T \vdash \mathsf{Comp}(\mathit{if}(e)\ \{S\}) \xrightarrow{s} I_e; \mathtt{JZ}\ \mathtt{R}_e\ \mathit{len}(I_t); I, T'}$$

$$\frac{\mathit{fresh}(\mathtt{R}) \quad \mathit{fresh}(\mathtt{R}_{base}) \quad \mathit{fresh}(\mathtt{R}_{offset}) \quad \mathit{fresh}(\mathtt{R}_v)}{\begin{array}{c}T \vdash \mathsf{Comp}(v) \xrightarrow{a} \mathtt{CONSTANT\ R}\ T(v); \mathtt{ADD\ R}_{base}\ \mathtt{R\ R}_{zero}; \\ \mathtt{CONSTANT\ R}_{offset}\ 0;\ \mathtt{ADD\ R}_v\ \mathtt{R}_{base}\ \mathtt{R}_{offset}, \mathtt{R}_v\end{array}}$$

$$\frac{T \vdash \mathsf{Comp}(\mathit{if}(e)\ \{S\}) \xrightarrow{s} I_t, T' \quad T' \vdash \mathsf{Comp}(S') \xrightarrow{s} I_f, T''}{T \vdash \mathsf{Comp}(\mathit{if}(e)\ \{S\}\ \mathit{else}\ \{S'\}) \xrightarrow{s} I_t; \mathtt{JZ\ zero}\ \mathit{len}(I_f); I_f, T''}$$

$$\frac{T \vdash \mathsf{Comp}(v) \xrightarrow{a} I_a, \mathtt{R}_v \quad T \vdash \mathsf{Comp}(e) \xrightarrow{e} I_e, \mathtt{R}_e \quad \mathit{fresh}(\mathtt{R})}{T \vdash \mathsf{Comp}(v[e]) \xrightarrow{a} I_a; I_e; \mathtt{ADD\ R\ R}_v\ \mathtt{R}_e, \mathtt{R}}$$

$$\frac{T \vdash \mathsf{Comp}(e) \xrightarrow{e} I_e, \mathtt{R}_e}{T \vdash \mathsf{Comp}(\mathit{return}\ e) \xrightarrow{e} I_e; \mathtt{MOV\ R}_{ret}\ \mathtt{R}_e; \mathtt{JR\ R}_{addr}, \mathtt{R_e}}$$

$$\frac{\mathit{fresh}(\mathtt{R}) \quad T \vdash \mathsf{Comp}(e) \xrightarrow{a} I, \mathtt{R}_a}{T \vdash \mathsf{Comp}(e) \xrightarrow{e} I; \mathtt{LOAD\ R\ R}_a, \mathtt{R}}$$

$$\frac{\mathit{fresh}(\mathtt{R}) \quad \mathit{fresh}(\mathtt{R}_{base}) \quad \mathit{fresh}(\mathtt{R}_{offset}) \quad \mathit{fresh}(\mathtt{R}_v)}{\begin{array}{c}T \vdash \mathsf{Comp}(v.x) \xrightarrow{a} \mathtt{CONSTANT\ R}\ T(v); \mathtt{ADD\ R}_{base}\ \mathtt{R\ R}_{zero}; \\ \mathtt{CONSTANT\ R}_{offset}\ T(v.x);\ \mathtt{ADD\ R}_v\ \mathtt{R}_{base}\ \mathtt{R}_{offset}, \mathtt{R}_v\end{array}}$$

$$\frac{\mathit{fresh}(\mathtt{R})}{T \vdash \mathsf{Comp}(n) \xrightarrow{e} \mathtt{CONSTANT\ R}\ n, \mathtt{R}}$$

$$\frac{T \vdash \mathsf{Comp}(e_0) \xrightarrow{a} I_0, R_0 \quad T \vdash \mathit{Comp}(e_1) \xrightarrow{e} I_1, \mathtt{R}_1 \quad \mathit{fresh}(\mathtt{R})}{T \vdash \mathsf{Comp}(e_0 = e_1) \xrightarrow{s} I_0; I_1; \mathtt{STORE\ R}_0\ \mathtt{R}_1, T}$$

$$\frac{T \vdash \mathsf{Comp}(e) \xrightarrow{e} I_e, \mathtt{R_e} \quad T \vdash \mathsf{Comp}(S) \xrightarrow{s} I_t, T'}{T \vdash \mathsf{Comp}(\mathit{while}(e)\ \{S\}) \xrightarrow{s} I_e; \mathtt{JZ\ R_e}\ \mathit{len}(I_t); I_t; \mathtt{JZ\ zero}\ -(\mathit{len}(I_t) + \mathit{len}(I_e) + 1), T'}$$

$$\frac{T_n \vdash \mathsf{Comp}(\theta_p\ p) \xrightarrow{d} T_{n'} \quad T_{n'} \vdash \mathsf{Comp}(S) \xrightarrow{s} I_b, T_{n''} \quad T_i \vdash \mathsf{Comp}(\theta_f\ f) \xrightarrow{d} T_{i'} \quad \mathit{fresh}(\mathtt{R})}{T_{n,i} \vdash \mathsf{Comp}(\theta_f\ f(\theta_p\ p)\{S\}] \xrightarrow{s} \mathtt{f}: \mathtt{CONSTANT\ R}\ n; \mathtt{STORE\ R\ R}_{\mathtt{arg}}; I_b, T_{i',n''}}$$

$$\frac{T \vdash \mathit{Comp}(e) \xrightarrow{e} I_e, \mathtt{R}_e \quad \mathit{fresh}(\mathtt{R}) \qquad \mathit{addr} = \mathit{num\_inst} + \mathit{len}(I) + 2}{T \vdash \mathsf{Comp}(f(e)) \xrightarrow{e} I_e; \mathtt{MOV\ R}_{arg}\ \mathtt{R}_e; \mathtt{CONSTANT\ R}_{addr}\ \mathit{addr}; \mathtt{JAL\ f}; \mathtt{MOV\ R\ R}_{ret}, \mathtt{R}}$$

Figure 3.5: Specification of the translation rules.

3.3 The Numbering Schema

Given a CharonLang program P written with n symbols, its certificate $Cert(P)$ is the product of the first n primes, each raised to a particular exponent. Each symbol uses a different exponent, as we have already illustrated in examples 1 and 3. Figure 3.6 shows the mapping of symbols to exponents.

Example 6. *Figure 3.6 shows that an* If-Then *construct will be always associated with a triple of exponents* $(41, 43, 47)$. *These exponents indicate, respectively, the condition the if-then-else command will evaluate, the start of the conditional code block, and the end of the conditional code block. In regards to if-then-else statements, we have chosen to assign explicit markers to the beginning and to the end of these blocks so the certificate can be inverted back into the original program. Section 3.6 discusses inversion in more detail.*

Types		**Variables and constants**	
short	2	Constant	$11^{c'}$
int	3	Var. def.	$13^{var.\ def.\ exp.}$
float	5	Var. usage	$17^{var.\ usage.\ exp.}$
__unknown_type__	7		
Functions			
Func. param.	$19^{var.\ def.\ exp.}$	Func. (start)	$31^{func.\ def.\ exp.}$
Func. arg.	23	Func. (end)	37
Func. call	29^{fp}	Func. return	41
Control Flow			
Conditional	43	Else (end)	59
If (start)	47	While (start)	61
If (end)	53	While (end)	67
Operations			
Assign (=)	71	Equal (==)	109
Negation (!)	73	Not equal (!=)	113
Addition (+)	79	Logical and (&&)	127
Subtraction (−)	83	Logical or (||)	131
Multiplication (∗)	89	Left shift (<<)	137
Division (/)	97	Right shift (>>)	139
Modulo (%)	101	Bit-wise and (&)	149
Less than (<)	103	Bit-wise or (|)	151
Greater than (>)	107		
Miscellaneous			
HALT	157		

Figure 3.6: Mapping of programming constructs to the exponents used in the numbering schema.

In contrast to the encoding of control-flow statements, the encoding of constants,

variables, and functions vary depending on which symbols are being referenced. These cases are discussed below:

Constant: A constant c is associated with the exponent $11^{c'}$, where c' is $c+1$ if $c \geq 0$, or c otherwise. Therefore, each constant is represented by a unique value. Adding one is necessary to avoid the identity case of the exponentiation.

Variable definition: Variables definitions encoded with the base number 13 to the power of *var. def. exp.*, following the formula $var.\ def.\ exp. = type\ symbol_1^{type\ symbol_2^{\cdots}}$, where $type\ symbol_n$ is the symbol associated with the type of the n-th element of the variable. Scalars will have a single *type symbol*. Function parameters are encoded similarly, but use 23 as the base number.

Variable usage: Variable usage cases employ the base number 17, to the power of *var. usage exp.*, which considers the variable prime *vp* and the memory offset of this element from the variable's base address. This exponent comes from the formula $var.\ usage\ exp. = vp^{mem.\ offset}$. If the memory offset is static (i.e., indexing an array with a constant, or accessing an element from a *struct*), then $mem.\ offset = 2^{os+1}$, where *os* is the offset in bytes. Scalar variables are covered by this case and have $os = 0$. If the memory offset is dynamically determined (as in indexing an array with a variable), then $mem.\ offset = 3^{vp_{id}}$, where vp_{id} is the variable prime of the index.

Function definition: Function definitions are associated with a pair of symbols: FUNC. (START) and FUNC. (END). These symbols explicitly delimit the beginning and the end of the function scope. While the latter uses 37 as the base number, the former is represented by $31^{func.\ def.\ exp.}$, following the formula $func.\ def.\ exp. = type\ symbol(\theta)^{(num.\ params.(f)+1)}$ where $type\ symbol(\theta)$ is the symbol associated with the function type, and *num. params.* is the number of parameters the function takes. *num. params.* is incremented by 1 to avoid the identity case of exponentiation for the case when a function takes no parameters.

Function call: Function calls are encoded with a base number, 29, to the power of the function prime fp_m that corresponds to the called function.

3.3.1 Identification of variables and functions

The Numbering Schema uses unique prime numbers as a means to identify variables and functions. These numbers are determined by the variable's or function's declaration site, in a way that resembles De Bruijn's indexation [13].

We collect all the variables definitions — including function parameters —, in the order they are defined, and map the n-th active variable in this set to the n-th prime number vp_n. A variable is active if it has at least one use case, and we treat function parameters as always active. The prime of a variable is emitted as soon as an use case of it is found.

Function primes fp_m are produced by mapping the m-th function label to the m-th prime number fp_m. These numbers are emitted as soon as the function definition is parsed by the compilation algorithm *Comp*. Unlike variable primes, that only consider active variables, function primes are emitted regardless of whether the function is ever called.

Example 7. *Consider the set of variables definitions* $\{var_1, var_2, var_3\}$. *Assume that* var_1 *and* var_3 *both have use cases, but* var_2 *does not. In this case,* var_1 *is mapped to the first variable prime* $vp_1 = 2$, *while* var_3 *is mapped to the second variable prime* $vp_2 = 3$.

Variable primes vp and function primes fp are managed by a certification environment C. The high-level certification algorithm uses the C_H environment, which will be discussed in Section 3.4. C_H maps variables' aliases to variable primes, and function names to function primes. The low-level certification algorithm, on the other hand, is described in Section 3.5 and uses C_L. C_L maps variables' base addresses to variable primes, and the function labels to function primes. While these environments are fully independent, Figure 3.7 presents how they produce variable and function primes from the compilation process.

$$\frac{vp' = \mathsf{next}(vp) \text{ if } v \notin C_H \quad {C_H}' = C_H \cup \{v \mapsto vp'\} \quad {C_L}' = C_L \cup \{T(v) \mapsto vp'\} \quad fresh(\mathtt{R}) \quad fresh(\mathtt{R}_{base}) \quad fresh(\mathtt{R}_{offset}) \quad fresh(\mathtt{R}_v)}{(T, C_H, C_L, vp) \vdash \mathsf{Comp}(v) \xrightarrow{a} \mathtt{CONSTANT\ R}\ T(v); \mathtt{ADD\ R}_{base}\ \mathtt{R}\ \mathtt{R}_{zero}; \mathtt{CONSTANT\ R}_{offset}\ 0;\ \mathtt{ADD\ R}_v\ \mathtt{R}_{base}\ \mathtt{R}_{offset}, \mathtt{R}_v,\ {C_H}', {C_L}', vp'}$$

$$\frac{(T, C_H, C_L, vp) \vdash \mathsf{Comp}(v) \xrightarrow{a} I_a, \mathtt{R}_v, {C_H}', {C_L}', vp' \quad T \vdash \mathsf{Comp}(e) \xrightarrow{e} I_e, \mathtt{R}_e \quad fresh(\mathtt{R})}{(T, C_H, C_L, vp) \vdash \mathsf{Comp}(v[e]) \xrightarrow{a} I_a; I_e; \mathtt{ADD\ R\ R}_v\ \mathtt{R}_e, \mathtt{R}, {C_H}', {C_L}', vp'}$$

$$\frac{vp' = \mathsf{next}(vp) \text{ if } v \notin C_H \quad {C_H}' = C_H \cup \{v \mapsto vp'\} \quad {C_L}' = C_L \cup \{T(v) \mapsto vp'\} \quad fresh(\mathtt{R}) \quad fresh(\mathtt{R}_{base}) \quad fresh(\mathtt{R}_{offset}) \quad fresh(\mathtt{R}_v)}{(T, C_H, C_L, vp) \vdash \mathsf{Comp}(v) \xrightarrow{a} \mathtt{CONSTANT\ R}\ T(v); \mathtt{ADD\ R}_{base}\ \mathtt{R}\ \mathtt{R}_{zero}; \mathtt{CONSTANT\ R}_{offset}\ T(v.x);\ \mathtt{ADD\ R}_v\ \mathtt{R}_{base}\ \mathtt{R}_{offset}, \mathtt{R}_v,\ {C_H}', {C_L}', vp'}$$

$$\frac{fp' = \mathsf{next}(fp) \text{ if } f \notin C_H \quad {C_H}' = C_H \cup \{f \mapsto fp\} \quad {C_L}' = C_L \cup \{f \mapsto fp\}}{(T_i, C_H, C_L, fp) \vdash \mathsf{Comp}(\theta_f\ f) \xrightarrow{d} \mathtt{f}, T_i, {C_H}', {C_L}', fp'}$$

Figure 3.7: Emission of variable and function primes vp_n and fp_m.

Example 8. *Figure 3.8 shows how the Numbering Schema handles functions and variables. As* `func_1` *is the first function declared in this program, it will be identified by the first function prime,* $fp_1 = 2$*. The next declared functions (*`func_2` *and* `main`*) are identified by* $fp_2 = 3$ *and* $fp_3 = 5$*, respectively. A similar logic applies to variables definitions, but, in this case, their types are also taken into consideration. The first variable to be declared is* `int param_1`*, from* `func_1`*. It has type symbol* $= 3$ *and will be identified by the first variable prime* $vp_1 = 2$*. The next variables are* `float param_2` *(*$vp_2 = 3$*, type symbol* $= 5$*), and* `int some_var[3]` *(*$vp_3 = 5$*, type symbol* $= 7^3$*), and* `int var_2` *(*$vp_5 = 11$*, type symbol* $= 3$*).* `some_var` *is an array. Only its third position is used in the program, and it is not indexed via variable indices; hence, only the third position consumes a prime in the certification process. The last line of the example shows an expression that takes a variable and the result of a function call as operands. To encode it, the Numbering Algorithm will use the addition base symbol (79), and will produce symbols to represent the variable usage and the function call. The variable usage symbol will be the base* 17 *to the power of* $vp^{mem.offset}$*, where* $mem.offset = 2^{0+1} = 2$ *and* $vp_5 = 11$*. Rearranging, the symbol will be* $17^{11^2} = 17^{121}$*. The function call symbol, in turn, will be* $29^{fp_m} = 29^3$*. The positional primes of each component have been omitted for brevity.*

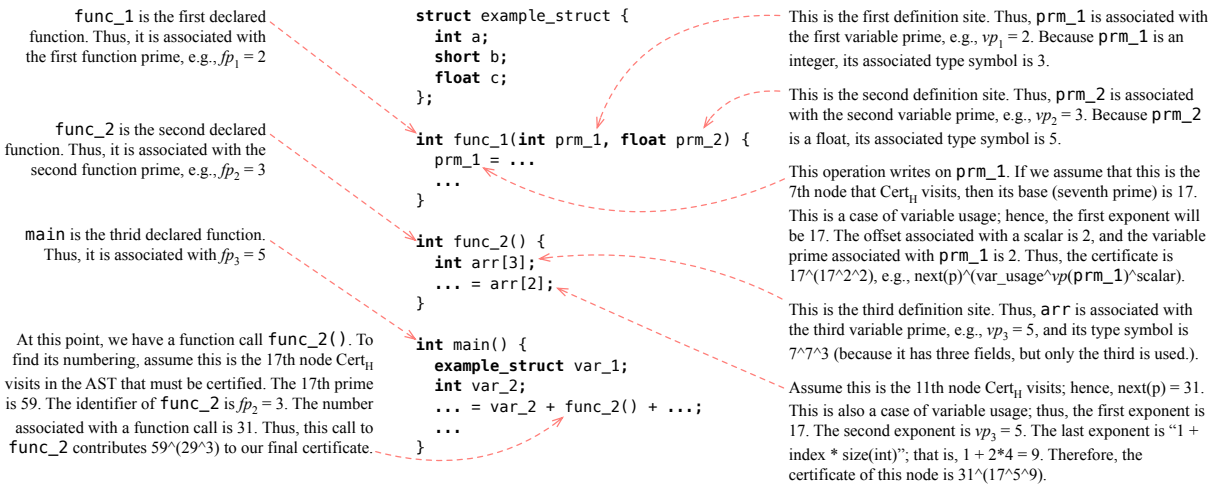


Figure 3.8: Example illustrating the certification of variables and functions.

By associating a different prime to each variable, the certification schema is able to distinguish programs that access the same variables, but in different order. We notice that this schema does not take commutativity into consideration. Thus, $x + y$ and $y + x$ will produce different certificates. This property applies to both, the high-level certifier and the low-level certifier, for, according to Figure 3.8, both associate user-defined symbols with primes via the environments C_H and C_L.

3.4 Source Code Certification

The certification of programs written in the high-level language (CharonLang) happens in two phases. First, an environment C_H is created following the rules in Figure 3.7. Then, the certificates are produced by a function $Cert_H$, following the rules in Figure 3.9. Notice that these rules are parameterized by the environment C_H, and that we do not pass C_H as a parameter to $Cert_H$, because it never changes; rather, we treat C_H as an immutable global table.

In Figure 3.9, p represents the current positional prime, and the other values follow the Numbering Schema seen in Section 3.3. The certifier $Cert_H$ traverses the program's abstract syntax tree, assigning numbers to nodes in post-order. This is motivated by the fact that the compiler will generate code for the operands before emitting the instructions of the operations themselves. By certifying expressions in post-order, we conciliate the sequence of numbers produced by $Cert_H$ with the sequence produced by $Cert_L$.

After traversing the AST and computing the encoding numbers for all of its nodes, $Cert_H$ will move all the variable and parameter definition symbols to the beginning of the certificate in the order they appear in the original program, and recompute all of the positional primes. The relative position of the remaining numbers is also preserved. This is a key design choice that enables the inversion of certificates[2]. Inversion is discussed in Section 3.6.

Example 9. *Figure 3.10 shows the certificate produced for a* While *statement. This example assumes that x has $vp_1 = 2$, y has $vp_2 = 3$, and* `func` *has $fp_1 = 2$. Following the rules in Figure 3.9, $Cert_H$ first certificates the conditional in the loop, which is the binary operation $x < 10$. In this case, $Cert_H$ will first emit the* Cond *symbol, and then compute the certificate of the left-hand side, the reading of variable x; the right-hand side, the constant 10; and finally the binary* `LESS` *operation itself. After that, $Cert_H$ emits the number associated with the start of a* While *construct, which ends up as 11^{61}, i.e., the fifth prime raised to 61. In this case, 61 is the exponent associated with the beginning of a* While *block. Next, $Cert_H$ proceeds to build a certificate to the body of the while operation, similarly to what we saw in Example 8. Finally, it computes the number that represents the end of the conditional code, 59^{67}. The exponent 67 is associated with the end of a* While *block.*

[2]Adding symbols such as Cond, If (start), and If (end) to explicitly mark the boundaries of each part of conditionals when handling them follows the same motivation.

Procedure definitions

$$Cert_H(\theta\ f(\theta_p\ x, ...)\{S\}) ::= p^{31^{func.\ def.\ exp.}} \times Cert_H(\theta_p x) \times Cert_H(\ldots) \times Cert_H(S) \times next(p)^{37}; \quad func.\ def.\ exp. = type\ symbol(\theta)^{(num.\ params.(f)+1)}$$

Statements

$$Cert_H(S; S') ::= Cert_H(S) \times Cert_H(S')$$
$$Cert_H(\theta\ x) ::= p^{13^{type\ \ symbol(\theta)}}$$
$$Cert_H(\theta_p\ x) ::= p^{19^{type\ \ symbol(\theta_p)}}$$
$$Cert_H(f(e)) ::= Cert_H(e) \times next(p)^{23} \times next(p)^{29^{C_H(f)}}$$
$$Cert_H(x = e) ::= Cert_H(x) \times Cert_H(e) \times next(p)^{71}$$

Control flow

$$Cert_H(\textbf{if}\ (e)\ \{S\}) ::= p^{43} \times Cert_H(e) \times next(p)^{47} \times Cert_H(S) \times next(p)^{53}$$
$$Cert_H(\textbf{if}\ (e)\ \{S\}\ \textbf{else}\ \{S'\}) ::= Cert_H(\textbf{if}\ (e)\ \{S\}) \times Cert_H(S') \times next(p)^{59}$$
$$Cert_H(\textbf{while}(e)\ \{S\}) ::= p^{43} \times Cert_H(e) \times next(p)^{61} \times Cert_H(S) \times next(p)^{67}$$
$$Cert_H(\textbf{return}\ e) ::= Cert_H(e) \times next(p)^{41}$$

Expressions

$$Cert_H(n) ::= p^{11^{(n+1)\ \ if\ \ n\ \geq\ 0\ \ else\ \ n}}$$
$$Cert_H(x) ::= p^{17^{var.\ usage\ exp.}};\ var.\ usage\ exp. = C_H(x)^2$$
$$Cert_H(x[n]) ::= p^{17^{var.\ usage\ exp.}};\ var.\ usage\ exp. = C_H(x)^{2^{1+n\times size(type(x))}}$$
$$Cert_H(x.y) ::= p^{17^{var.\ usage\ exp.}};\ var.\ usage\ exp. = C_H(x)^{2^{1+offset(y)}}$$
$$Cert_H(x[y]) ::= p^{17^{var.\ usage\ exp.}};\ var.\ usage\ exp. = C_H(x)^{3^{C_H(y)}}$$
$$Cert_H(\Theta\ e) ::= Cert_H(e) \times next(p)^{symbol(\Theta)}$$
$$Cert_H(e_1\ \Theta\ e_2) ::= Cert_H(e_1) \times Cert_H(e_2) \times next(p)^{symbol(\Theta)}$$

Figure 3.9: Rules for the certification of programs in the high-level language CharonLang.

3.5 Machine Code Certification

The Machine Code Certification algorithm $Cert_L$ produces a certificate for the compiled program, which exists in the low-level CharonIR. The compiled program consists of a list of instructions for the Virtual Machine to execute, and a set of metadata — the `data` section, as previously introduced. As in the high-level case, the certification of low-level code happens in two phases. The first creates the environment C_L (Figure 3.7); whereas the second builds the certificate itself (Figure 3.11).

As already mentioned, Figure 3.7 shows the first phase of the certification process. The rules in Figure 3.7 produce a map between the base memory address of each variable and a unique variable prime *vp*, and add it to the runtime environment C_L. This environment also contains mappings between each function label to an unique function prime *fp*.

The second phase of the certification process, described in Figure 3.11, performs five actions:

1. **Map Dependencies Between Instructions And Temporary Registers:** Produce a mapping between instructions and the temporary registers they depend upon.

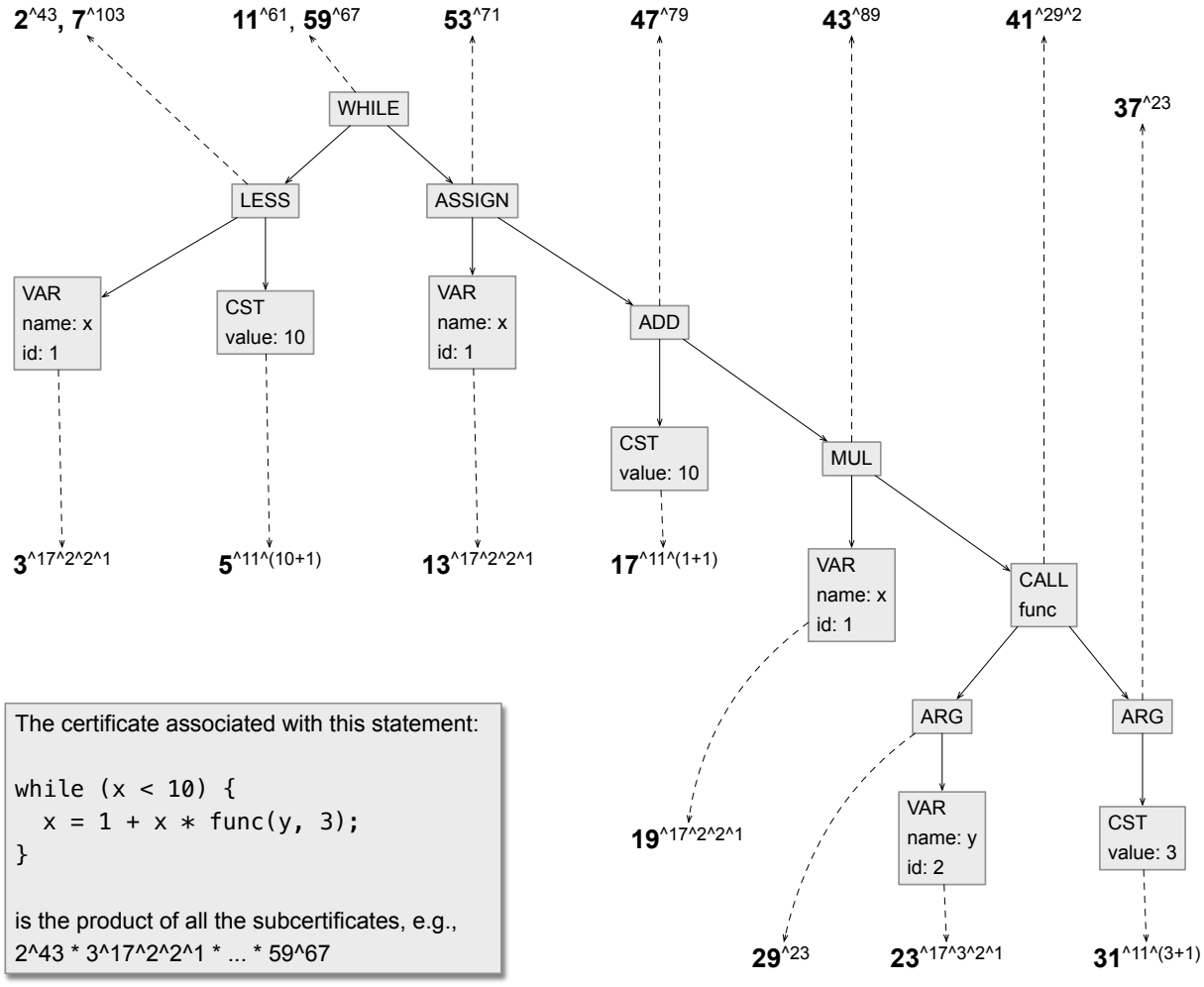


Figure 3.10: Certification of a WHILE statement, i.e., $Cert_H(\textbf{while}(e)\ \{S\}) ::= p^{43} \times Cert_H(e) \times next(p)^{61} \times Cert_H(S) \times next(p)^{67}$.

For instance, it will map the instruction of a binary operation to the temporary registers that contains its operands. This process is recursive. As such, if one of the operands of a binary operation Op_1 is the result of another binary operation Op_2, Op_1 dependency mapping will also include the registers Op_2 depends upon.

2. **Infer Variables Types:** Produce a mapping between variables addresses and their inferred types. Types are inferred based on the sequence of instructions used to read or write a variable. Reading from and writing to `float` variables will use the `LOADF/STOREF` instructions, respectively. The integer types, `int` and `short`, will use `LOAD` and `STORE` — the latter will also be truncated, with the `TRUNC` instruction, immediately after being read or before being written.

3. **Analyze Functions:** Produce a mapping between functions labels and their inferred return types and number of parameters. Function type inference is computed from inferring the type of the value they return, following the same logic applied to infer variables types. The number of parameters it takes, on the other hand, is obtained

from the counting the occurrences of the parameter passing pattern[3] within the function scope. This pass will also set up the insertion points of the FUNC. (START) and FUNC. (END) symbols in the program certificate, based on the scope each function label delimits.

4. **Analyze Conditional Jumps:** Search for conditional jumps (i.e., the `JZ` instruction) and determine the first instruction that is used in its condition. This pass is based on the previously computed dependency mapping, and it sets up the insertion point of the COND symbol in the program certificate.

5. **Certificate Instructions:** Visit each instruction of the machine code to compute the adequate number following the rules if Figure 3.11. This figure presents the patterns the low-level certifier $Cert_H$ will attempt to match. The certifier will then proceed to the first instruction after the instructions in the pattern it just matched until it reaches the end of the program. This follows the design of certificating constructs, rather than individual instructions.

Example 10. *Figure 3.12 (a) displays the a simple program implemented in* CHARONLANG *that contains a* WHILE *loop. Part (b) of that figure presents its low-level representation in* CHARONIR*. The figure groups sequences of instructions into blocks to highlight the patterns that the certifier has matched to produce the final certificate. These blocks are mapped to part (d), which presents the certificate that encodes this program. Part (c) presents the environment C_L after it is populated by the rules in Figure 3.7, and the resulting maps of passes (1) through (3).*

3.6 The Canonical Format

Certificates are invertible. Thus, if $n = Cert_H(P) = Cert_L(Comp(P))$, then it is possible to reconstruct a program P_c from n. These two programs, P and P_c are semantically equivalent[4]; however, they are not syntactically equivalent. The program P_c is called the *canonical* representation of P. As such, the canonical form P_c has the same certificate as the program P it was reconstructed from. Notice that different programs might have the same canonical representation. If two programs have the same canonical

[3] `CONSTANT` R_c $var_{address}$; `STORE` R_c R_{arg}

[4] In this work, we do not show semantic equivalence between high- and low-level constructs: we only show proof that the compilation of one will yield the other. In our work, the only difference between programs that lead to the same canonical form is the declaration and storage of variables. Thus, it is simple to conclude that they would implement the same semantics.

Statements

$Cert_L(S; S') ::= Cert_L(S) \times Cert_L(S')$

$Cert_L(\texttt{data}[var_{address}]) ::= p^{13^{type\ \ symbol(var_{address})}}$

$Cert_L(\texttt{label}; S) ::= p^{31^{func.\ \ def.\ \ exp.}} \times Cert_L(S) \times next(p)^{37}$;
$func.\ def.\ exp. = type\ symbol(type(\texttt{label}))^{(num.\ \ params.(\texttt{label})+1)}$

$Cert_L(\texttt{CONSTANT R}_c\ \texttt{var}_{address}; \texttt{STORE R}_c\ \texttt{R}_{arg}) ::= p^{19^{type\ \ symbol(var_{address})}}$

$Cert_L(\texttt{S;MOV R}_{arg}\ \texttt{R}) ::= Cert_L(S) \times next(p)^{23}$

$Cert_L(\texttt{CONSTANT R}_{addr}\ addr; \texttt{JAL label; MOV R R}_{ret}) ::= p^{29^{C_L(\texttt{label})}}$

$Cert_L(S; S'; \texttt{STORE R}_S\ \texttt{R}_{S'}) ::= Cert_L(S) \times Cert_L(S') \times next(p)^{71}$

Control flow

$Cert_L(S; \texttt{JZ R}_S\ len(S'); S') ::= p^{43} \times Cert_L(S) \times next(p)^{47} \times Cert_L(S') \times next(p)^{53}$

$Cert_L(S; \texttt{JZ R}_S\ len(S'); S'; \texttt{JZ zero}\ len(S''); S'') ::= p^{43} \times Cert_L(S) \times next(p)^{47} \times Cert_L(S') \times next(p)^{53} \times Cert_L(S'') \times next(p)^{59}$

$Cert_L(S; \texttt{JZ R}_S\ len(S'); S'; \texttt{JZ zero}\ -(len(S)+len(S')+1)) ::= p^{43} \times Cert_L(S) \times next(p)^{61} \times Cert_L(S') \times next(p)^{67}$

$Cert_L(S; \texttt{MOV R}_{ret}\ \texttt{R}_S; \texttt{JR R}_{addr}) ::= Cert_L(S) \times next(p)^{41}$

Expressions

$Cert_L(\texttt{CONSTANT R}_c\ \texttt{c}) ::= p^{11^c}$

$Cert_L(\texttt{CONSTANT R}_c\ \texttt{x; ADD R}_{base}\ \texttt{R}_c\ \texttt{zero; CONSTANT R}_{offset}\ offset; \texttt{ADD R}_x\ \texttt{R}_{base}\ \texttt{R}_{offset}) ::= p^{17^{var.\ \ usage\ \ exp.}}$;
$var.\ usage\ exp. = C_L(x)^{2^{1+offset}}$

$Cert_L(\texttt{CONSTANT R}_c\ \texttt{x; ADD R}_{base\ \ x}\ \texttt{R}_c\ \texttt{zero;}$
$\texttt{CONSTANT R}_{offset\ \ x}\ \texttt{offset(x);}$
$\texttt{ADD R}_x\ \texttt{R}_{base\ \ x}\ \texttt{R}_{offset\ \ x}\texttt{;}$
$\texttt{CONSTANT R}_{c'}\ \texttt{y; ADD R}_y\ \texttt{R}_{c'}\ \texttt{zero;}$
$\texttt{CONSTANT R}_{offset\ \ y}\ \texttt{offset(y);}$
$\texttt{ADD R}_y\ \texttt{R}_{base\ \ y}\ \texttt{R}_{offset\ \ y}\texttt{;}$
$\texttt{LOAD R}_{val(y)}\ \texttt{R}_y\texttt{;}$
$\texttt{CONSTANT R}_{c''}\ \texttt{size(x);}$
$\texttt{MUL R}_{offset}\ \texttt{R}_{val(y)}\ \texttt{R}_{c''}\texttt{;}$
$\texttt{ADD R}_{x+offset}\ \texttt{R}_x\ \texttt{R}_{offset}) ::= p^{17^{var.\ \ usage\ \ exp.}}$;
$var.\ usage\ exp. = C_L(x)^{3^{C_L(y)}}$

$Cert_L(S; \texttt{UnOp R}_{op}\ \texttt{R}_S) ::= Cert_L(S) \times next(p)^{symbol(\texttt{UnOp})}$

$Cert_L(S; S'; \texttt{BinOp R}_{op}\ \texttt{R}_S\ \texttt{R}_{S'}) ::= Cert_L(S) \times Cert_L(S') \times next(p)^{symbol(\texttt{BinOp})}$

Figure 3.11: Machine Code Certification rules.

representation, then they implement the same semantics. However, these programs might differ in how data is allocated. The following differences are possible:

Variables: All the variables are global, except for function parameters. The canonical form preserves the order in which variables are declared in the original program.

Structures: The canonical representation only considers data structures elements that are used in the program. However, it preserves the number of elements and their order in the original program. Thus, unused elements in the original program are still defined in the canonical representation, but they are typed as `__unknown_type__`.

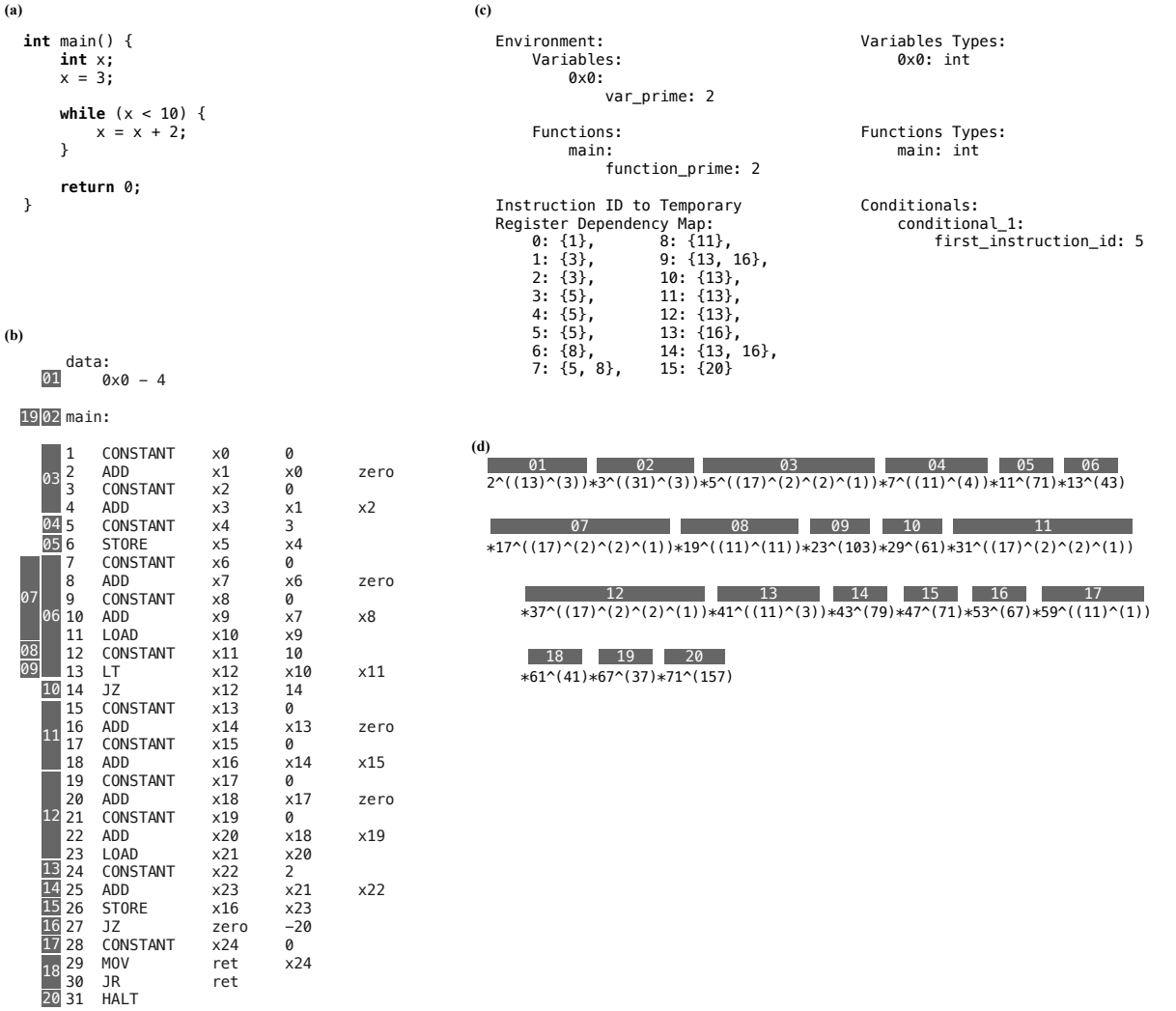


Figure 3.12: Certification of a simple program with a WHILE loop.

Arrays that are only indexed by constants in the original program are represented as structs in the canonical format. Arrays that are indexed with a variable, on the other hand, are still declared as arrays in the canonical format.

Example 11. *Figure 3.13 presents two high-level programs, P1 and P2, such that* $Cert_H(P1) = Cert_H(P2)$. *While these programs are not syntactically identical, they possess the same semantics: they set the values of the first and third elements of a data structure — both integers — using constant offsets and return their sum. These programs differ only on how variables are stored. The second program stores variables as fields of a struct. The first does the same, albeit using an array. The canonical representation uses a struct to represent both situations. However, one of the fields of this struct is left undefined. Notice that this field still occupies 4 bytes, as this is the default alignment that we use in* CHARONLANG. *Given that this field is never used, it is also not declared, meaning that it could be represented as any type that fits into 4 bytes. In this case, this element's type is*

Figure 3.13: Inversion of the certificate of two semantically equivalent high-level programs.

set to __unknown_type__ in the canonical representation, which occupies one word in the memory layout.

3.7 The Canonical Reconstruction

Given a certificate n, it is possible to obtain a canonical program P_c, such that $Cert_H(P_c) = n$. This reconstruction process is defined by a function $Canon(n)$, which Algorithm A shows. This reconstruction process happens in four stages:

1. **Factorization:** Factorize the certificate n into a product of primes in the form p^{exp}, where p is the positional prime and exp is the construct-encoding exponent. This exponent is expected to follow the symbols and rules from Numbering Schema.

2. **Analysis:** This step decides whether a data structure — i.e., a variable whose definition has more than one type symbol in exp — should be defined as an array or as a struct. This is achieved by analyzing all the use cases of these variables. If the variable's attributes are only accessed with static offsets, it will be defined as a struct; *Canon* defines it as an array otherwise.

3. **Generation:** Iterate once more over the factorized certificate to produce the canonical program by matching the pattern from the Numbering Schema, and applying the rules discussed above.

4. **Finalization:** As there is no explicit symbol for the `main` function in the certificate, and CharonLang assumes all the programs will follow a def-use relation, *Canon* will set the last function it defines to be the `main` function.

Algorithm A shows the pseudo-code of the reconstruction process for expressions, and it is implemented by the procedure *Canon*. As a simplification, it omits the handling of functions and control-flow constructs. The rest of this section illustrates how the reconstruction process in Algorithm A works through a series of examples, following Figure 3.14. This figure shows the canonical reconstruction of the certificate n from Figure 3.13.

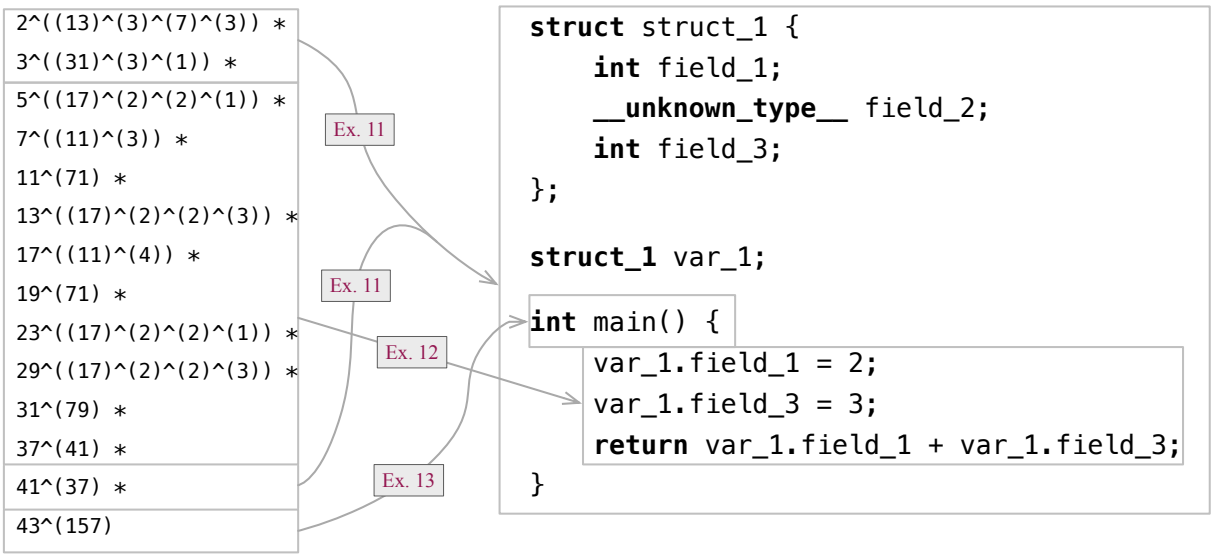


Figure 3.14: The canonical reconstruction of a simple program.

Canon **Parses Certificates Left-to-Right.** The canonical program generation begins by parsing variable definitions ($exp. = 13^{var.\ def.\ exp.}$) and function parameters ($exp. =$

$19^{var.\ def.\ exp.}$). These are the first constructs encoded in the certificate, if any, as discussed in section 3.4. Variable definitions will be added to the program body immediately. Function parameters, on the other hand, are first added to the list of function parameters, and then passed to the program when *Canon* parses a FUNC. (START) symbol ($exp. = 31^{type\ symbol(type(f))^{(num.params.(f)+1)}}$). In this case, it will take the first n elements from this list and add to the function definition, where n is the number of parameters this function takes obtained from the symbol. *Canon* will name the n-th declared variable or parameter `var_n`, and the m-th defined function `func_m`, and will retrieve its type from *exp*. It will use these names to refer to the variable or parameter with the n-th variable prime, or the function with the m-th function prime in the certificate factors that follow.

Example 12. *The certificate in Figure 3.14 encodes a single variable definition,* ***`internal_var`****. Its factor is* 2^((13)^(3)^(7)^(3))*, and it tells this variable is a data structure as there are three type symbols in it (*3*,* 7*, and* 3*). Canon will search for variable use cases with* 2 *(the first prime) as the variable prime because this is the first variable to be declared. There are four instances:* 5^((17)^(2)^(2)^(1))*,* 13^((17)^(2)^(2)^(3))*,* 23^((17)^(2)^(2)^(1))*, and* 29^((17)^(2)^(2)^(3))*. The fields of the variable are accessed with a constant for index in all of the instances; as such, this variable will be set to be declared as a struct with an integer, an element of unknown type, and another integer — in this order. The next factor,* 3^((31)^(3))*, marks the beginning of a function, which is first defined as* ***`func_1`****. This function scope will be closed when Canon reaches factor* 41^37*.*

Expressions in Reverse Polish Notation. Expressions are parsed following a simple Reverse Polish Notation (RPN) inversion algorithm. Operands — variables ($exp. = 17^{var.\ usage\ exp.}$), constants ($exp. = 11^{c'}$), and function calls ($exp. = 29^{fp}$) — are always added to the expression stack. Function calls are also parsed following this notation: the algorithm will first parse the arguments ($exp. = 23$), and then the function call itself. When *Canon* finds a certificate factor that represents an operator, it will take the top elements (one if it is an unary operator, or two if it is a binary operator) from the stack, write the parenthesized expression, and add it back to the top of the stack. The expression is only added to the program body when the algorithm finds a terminator. Any factor that does not encode an operand or an operator is a terminator. The assignment operator is an exception for this rule, for it terminates an expression in RPN.

Example 13. *Following,* 5^((17)^(2)^(2)^(1)) *encodes a variable usage and causes Canon to start analyzing an expression. This certificate factor has* 2 *as the variable prime; as such, it refers to* ***`var_1`****. Given that this variable is a struct, and that the certificate factor encodes the access of the first element, Canon will add* ***`.field_1`*** *as element access suffix to the variable name it will push to the expression stack.* 7^((11)^(3)) *comes right after it, and it encodes a constant. As the associated constant is positive, Canon will subtract 1 from*

it before pushing it to the expression stack. The next factor, 11ˆ(71)*, is the assignment operator, and the algorithm will write* `var_1.field_1 = 2;`*. As it is a terminator, this expression will be immediately added to* `program`*. This iterative process continues until Canon reaches the last factor,* 43ˆ(157)*.*

Reconstruction of Control Flow. Control-flow constructs are computed in *Canon* by first parsing the expression that follows a COND factor (*exp.* = 43), until it finds the pattern that marks the beginning of a conditional code block (eg., IF (START), which has *exp* = 47). Then, it will add the control-flow construct followed by the conditional expression to the program body. Function and control-flow scopes are closed as soon as the algorithm finds the corresponding ending marker (eg., IF (END), which has *exp.* = 53).

Example 14. *As Canon finds the program-ending factor (exp.* = 157*), it will add the* `main` *function to the program it just created. It will rename the last function it defined — in this case,* `func_1` *— to* `main`*. Finally, it returns the canonical program* P_c*.*

This chapter presented "Structural Certification", a technique based on Gödel's Numbers that encodes programs both in the high-level CHARONLANG and the low-level CHARONIR representations — both part of the CHARON compiler it introduced. The certificate it produces is an integer that can be inverted back into a canonical form, following the Canonical Reconstruction Algorithm, displayed at Appendix A. In the next chapter, we demonstrate the correctness of the "Structural Certification" technique.

Chapter 4

Correctness

This chapter provides sketches of proofs for the two main results discussed in this work. In Section 4.1 we show that compilation preserves certificates. Then, in Section 4.2, we show that certificates are invertible. These proof sketches highlight a benefit from Gödel's Numbering System. By relying on this encoding, the certification framework inherits the Fundamental Theorem of Arithmetic, which guarantees the unique factorization of every integer into primes. This fact will play a role in results such as Theorems 4.1.2 and 4.2.1, and Lemma 4.1.3.

4.1 Correctness of Equivalence of Certificates

To ensure that the certification process is correct, we establish that our compiler preserves the certification result. Specifically, for any high-level program P_H compiled to a low-level program P_L, the outputs of the certification algorithms $Cert_H$ and $Cert_L$ coincide. This property requires the n-th variable prime vp_n produced by both certification algorithms $Cert_H$ and $Cert_L$ to be equal. This is proven in Lemma 4.1.1. It shows that the certification environments C_H and C_L produced for the high- and low-level certification algorithms, respectively, map the high- and low-level representations of a left-hand side term a to the same variable prime.

Lemma 4.1.1 (Equivalence of Variable Primes)**.** *If a is a left-hand side term, and $Comp(a) = \ldots T(a)$, then $C_H(a) = C_L(T(a))$.*

Proof. The proof goes by case analysis on the rules shown in Figure 3.7. Consider the case where $a = v$, i.e., a variable. The corresponding rule Figure 3.7 shows that if $v \notin C_H$, then a fresh variable prime $vp' = \mathsf{next}(vp)$ is generated and C_H is updated with $v \mapsto vp'$, while C_L is updated with $T(v) \mapsto vp'$. Thus, both environments remain equivalent, assigning the same prime to v and to its corresponding address in the low-level domain.

The array access case, $a = v[e]$, happens in two steps. First, v is compiled under the same rule as above, assigning it a prime if it is not already mapped to one in C_H and C_L. Then, the index e is compiled separately. If e happens to be a variable, then it will also be subject to the *variable* rule, and the certification environments will be updated accordingly. If not, then the compilation of e will not change C_H and C_L. As such, any changes to the certification environment will be consistent across C_H and C_L.

For $a = v.x$, i.e., a field access in a struct, the rule shows that the base variable v is compiled following the first rule, and then the offset $T(v.x)$ is added to it. Since C_H and C_L are only updated when v is compiled (and not for the field access itself), consistency is preserved for the assigned variable prime.

Finally, in the case of function definitions, $\theta_f\ f$, the rule specifies that if $f \notin C_H$, a fresh function prime $fp = \mathsf{next}(fp)$ is generated. The environments C_H and C_L are then updated with $f \mapsto fp$ and $T(f) \mapsto fp$, respectively. Since the same prime is used in both updates, the environments remain equivalent.

In all cases, when a fresh prime is introduced, it is added to both environments simultaneously. Thus, we have that for all a, the relation $C_H(a) = C_L(T(a))$ is true. □

Theorem 4.1.2 establishes that certification is preserved through compilation: the certificate computed from a high-level program matches exactly the one computed from its compiled low-level form. This result ensures that certificates abstract program behavior in a way that is invariant under compilation. The proof relies on structural induction over program syntax, plus the fact that the Fundamental Theorem of Arithmetic ensures that the multiplicative encoding preserves the structure during compilation. For instance, a `while` loop's certificate (e.g., $p^{43} \times Cert_H \times \ldots$) must match its compiled low-level counterpart because the prime bases and exponents are fixed by the schema seen in Sections 3.4 and 3.5.

Theorem 4.1.2 (Preservation of Certificates)**.** *If P_H is a program in the high-level* CharonLang *language, then $Cert_H(P_H) = Cert_L(Comp(P_H))$.*

Proof. The proof proceeds by structural induction on the syntax of the source program P_H.

Base cases: the certification of constants, variables, array and struct accesses does not involve the recursive certification of subexpressions; thus, the theorem follows without induction:

Constant: If c is a constant, then $Cert_H(c) = Cert_L(\texttt{CONSTANT R}_\texttt{c}\ \texttt{c}) = 2^{11^c}$

Variable: If x is a variable name, then $Cert_H(x) = Cert_L(\texttt{CONSTANT R}_\texttt{c}\ \texttt{x};\ \texttt{ADD R}_\texttt{x}\ \texttt{R}_\texttt{c}\ \texttt{zero}) = p^{17^{var.\ usage\ exp.}}$. The exponent *var. usage exp.* tells whether this is a simple variable or an element in a data structure — and whether it is accessed with a static [1] or a

[1] As an array with constant for index or element in a struct

dynamic offset[2]. If it is a data structure, the high- and low-level programs P_H and P_L programs will also contain expressions or instructions to calculate the element offset from the base variable.

In either case, $Cert_H$ and $Cert_L$ will have the same environments C_H and C_L, as a direct result from Lemma 4.1.1. As *var. usage exp.* depends solely on the expressions and instructions that implement the variable and the certification environments C_H and C_L, we can conclude that the high- and low-level certification of variables coincide.

Inductive step: We apply induction to prove preservation of certificates on constructs whose syntax is defined recursively in Figure 3.2. Thus, if we have a construct defined as $S_0 ::= \ldots S_1 \ldots$, then we assume preservation over S_1 to demonstrate that it holds for S_0. As an example, we shall consider the compound program construct $P_H = S_0 = \texttt{if}(e)\ \{S\}$. According to the compilation rules (Fig. 3.5), the compiler emits the following instruction sequence for such conditionals:

$$Comp(\texttt{if}(e)\ \{S\}) = I_e;\ \texttt{JZ}\ R_e\ \ell;\ I_t,\ T'$$
$$Comp(e) \rightarrow I_e,\ R_e$$
$$Comp(S) \rightarrow I_t,\ T'$$

From the certificate construction rules (Fig. 3.10), we know:

$$Cert_H(\texttt{if}(e)\ \{S\}) = p^{43} \times Cert_H(e) \times next(p)^{47} \times Cert_H(S) \times next(p)^{53}$$

Similarly, from the low-level certificate construction rules (Fig. 3.11):

$$Cert_L(I_e;\ \texttt{JZ}\ R_e\ \ell;\ I_t) = p^{43} \times Cert_L(I_e) \times next(p)^{47} \times Cert_L(I_t) \times next(p)^{53}$$

By the inductive hypothesis, we assume:

$$Comp(e) = I_e \Rightarrow Cert_H(e) = Cert_L(I_e)$$
$$Comp(S) = I_t \Rightarrow Cert_H(S) = Cert_L(I_t)$$

We now observe that the same structure appears in both $Cert_H$ and $Cert_L$. From this, we can conclude that the conditional symbol p^{43} and the the control flow symbols $next(p)^{47}$ and $next(p)^{53}$ will be placed identically in both constructions, as they depend exclusively on the structure of the programs. Therefore,

$$Cert_H(\texttt{if}(e)\ \{S\}) = Cert_L(Comp(\texttt{if}(e)\ \{S\}))$$

The proof of equivalence between $Cert_H$ and $Cert_L$ for other constructs is analogous. □

[2]As an array with a variable for index

Unlike the certification of high-level programs, where $Cert_H(P_{H_1})$ might be equal to $Cert_H(P_{H_2})$ even if $P_{H_1} \neq P_{H_2}$, $Cert_L(P_{L_1})$ will be equal to $Cert_L(P_{L_2})$ if, and only if $P_{L_1} = P_{L_2}$. This property is demonstrated in Theorem 4.1.4. To prove it, we will use an auxiliary result: Lemma 4.1.3. This lemma demonstrates that each product of certificate factors can only be obtained from unique sequences of instructions. Intuitively, the Fundamental Theorem of Arithmetic guarantees that every integer greater than one has a unique prime factorization (up to ordering). This result ensures that the Gödel-style certificates are injective, because:

- Each certificate is a product of primes raised to construct-specific exponents (Section **??**).
- By the FTA, two distinct programs cannot share the same certificate, as their factorizations would differ.

Lemma 4.1.3 (Uniqueness of Low-level Certificate Factors)**.** *The low-level certification function $Cert_L$ is injective.*

Proof. The proof proceeds by analyzing the cases presented in Figure 3.11.

Most cases are trivial because each construct-encoding exponent (the exponent of p on the right side of Figure 3.11) is unique and strictly positive, and only appears in a single rule. Given that both p and every exponent are prime numbers, the Fundamental Theorem of Arithmetic establishes that these powers produce a unique factorization[3].

There are two exceptions for this rule: the rules that map to $p^{17^{var.\ usage\ exp.}}$, and the rules that feature $p^{43} \times Cert_L(S) \times \ldots$: these exponents, $17^{var.\ usage\ exp.}$ and 43 appear multiple times on the right side of the rules in Figure 3.11.

The argument for the first exception, $p^{17^{var.\ usage\ exp.}}$, relies on the fact that the exponent *var. usage exp.* is also a power of primes that will produce different numbers depending on the applied rule, such that $C_L(x)^{2^{1+offset}} \neq C_L(x)^{3^{C_L(y)}}$. $C_L(x)$ (and, similarly, $C_L(y)$) is the variable prime that uniquely represents the variable x (or y), and $1 + offset$ is also strictly positive. The result follows from the observation that there are no values for which $2^{1+offset} = 3^{C_L(y)}$.

As for the second exception, there are three rules where p^{43} is featured:

1. S; `JZ` $\mathtt{R}_S$ $len(S')$; S'
2. S; `JZ` $\mathtt{R}_S$ $len(S')$; S'; `JZ zero` $len(S'')$; S''
3. S; `JZ` $\mathtt{R}_S$ $len(S')$; S'; `JZ zero` $-(len(S) + len(S') + 1)$

[3]For distinct primes $p \neq q$ and positive integers m, n, the equality $p^m = q^n$ is impossible, since by the Fundamental Theorem of Arithmetic the prime factorization of an integer is unique.

Even though all of the factors produced by (1) are also produced by (2), the latter is the only rule that can add a factor with exponent 59. Similarly, (3) is the only rule that can add factors with exponents 61 and 67. □

Theorem 4.1.4 (Uniqueness of Low-level Programs)**.** *Let P_1 and P_2 be two programs in the low-level* CharonIR *intermediate representation. Then $Cert_L(P_1) = Cert_L(P_2)$ if and only if $P_1 = P_2$.*

Proof. The *if* case, $Cert_L(P_1) = Cert_L(P_2)$ if $P_1 = P_2$, is trivial because $Cert_L$ is deterministic. Therefore, passing the same certificate to *Cert* yields the same result.

To prove $Cert_L(P_1) = Cert_L(P_2)$ *only if* $P_1 = P_2$, we will use structural induction on n, where $n = Cert_L(P_1) = Cert_L(P_2)$. We proceed by analyzing the right side of each rule in Figure 3.11.

Base case: The certification of constants, variables definitions, and variable usage do not cause the recursive invocation of $Cert_L$ on subprograms. Thus, the equality of these constructs in programs P_1 and P_2 and can be demonstrated by applying the result of Lemma 4.1.3 without induction.

Constant: From Lemma 4.1.3, $p^{11^{c+1 \text{ if } c\geq 0 \text{ else } c}}$ is the only factor that encode the instruction `CONSTANT R_c c`. As this formula is injective, the equality of certificates implies the same `CONSTANT` instruction appears in the same position in both programs.

Variable definition: A variable definition is encoded from an address in `data` section of a program. From Lemma 4.1.3, it is solely certificated by $p^{13^{type\ \ symbol(var_{address})}}$, where *type symbol* encodes the inferred type of this variable. If both programs have the same certificate, then the same type has been inferred for the same address of `data`. This implies programs P_1 and P_2 declare the exact same variables, defined at the same relative position.

Variable usage: The variable usage certificate follows the formula $p^{17^{var.\ usage.exp.}}$. The exponent, *var. usage exp.*, will vary depending on the variable usage construct it encodes:

Variable usage with static offset: This case is implemented in low-level from a sequence of instructions that will load the variable base address into a temporary register, compute the static offset, and add it to the base address. Lemma 4.1.3 establishes that this is the only case that produces $var.\ usage\ exp. = C_L(x)^{2^{1+offset}}$, where x is the variable's base memory address, $C_L(x)$ is its variable prime in the certification environment, and *offset* is its static memory offset.

Variable usage with dynamic offset: The other case for the variable usage construct also starts by loading the base memory address into a temporary register.

Unlike the previous case, it will load the contents of another variable into another temporary register, and have memory address arithmetic instructions to compute the address of interest in runtime. Lemma 4.1.3 tells the exponent can only be *var. usage exp.* $= C_L(x)^{3^{C_L(y)}}$, where $C_L(x)$ and $C_L(y)$ are the variable primes of the variable being used and of the variable to dynamically compute the memory offset with. x is the base memory address.

If the certificates are identical, then the same sequence of instructions must be present in the same position in both programs, the variables must use the same addresses, and any offsets from the base memory addresses must be identical, in either variable construct the programs implement. Otherwise, there would be at least one differing variable usage certificate.

Inductive step: We shall analyze one case, as the others will be similar. We consider the compound programs $P_1 = S_1; S_1'$. We have that $Cert_L(P_1) = Cert_L(S_1) \times Cert_L(S_1')$. We have that $Cert_L(S_1) < Cert_L(P_1)$ and $Cert_L(S_1') < Cert_L(P_1)$; thus, we can apply induction. By induction, we assume that the programs S_1 and S_2 are unique. Therefore, it follows that if there exists P_2 such that $Cert_L(P_1) = Cert_L(P_2)$, then $P_2 = S_1; S_1'$. A similar result can be derived from the other recursive rules.

Therefore:

$$Cert_L(P_1) = Cert_L(P_2) \iff P_1 = P_2$$

□

4.2 Correctness of Invertibility

The canonical form is constructed from the certificate produced by either $Cert_H$ or $Cert_L$, as discussed in Section 3.6. The program produced by *Canon* captures the logical structure implied by the certificate, independent of the specific syntactic choices made in the source or compiled program. If two programs yield the same certificate, then they must share the same canonical form. This property is demonstrated in Theorem 4.2.1. Intuitively, the Fundamental Theorem of the Arithmetic enables the canonical reconstruction of programs from certificates (the *Canon* function), because since factorizations are unique, the inversion process (Algorithm A) can deterministically map a certificate back to a program structure by decoding the exponents. As an example, a certificate $2^{e_1} \times 3^{e_2} \times \ldots$ is invertible because the exponents $e_1, e_2, \ldots$ unambiguously represent program constructs.

Theorem 4.2.1 (Uniqueness of Canonical Form)**.** *Let P_1 and P_2 be two programs in the high-level* CharonLang *representation. Then $Canon(Cert_H(P_1)) = Canon(Cert_H(P_2))$ if and only if $Cert_H(P_1) = Cert_H(P_2)$.*

Proof. The proof for both directions is presented below:

$\Leftarrow$ If $Cert_H(P_1) = Cert_H(P_2)$, then $Canon(Cert(P_1)) = Canon(Cert(P_2))$.

The proof follows directly from the determinism of the *Canon* algorithm (Algorithm A). As the inputs are identical, then it will produce the same result in both cases.

$\Rightarrow$ If $Canon(Cert(P_1)) = Canon(Cert(P_2))$, then $Cert_H(P_1) = Cert_H(P_2)$.

Let $Cert_H(P_1) = n_1$ and $Cert_H(P_2) = n_2$, and $n_1 = f_1 \times f_2 \times \ldots$ and $n_2 = f'_1 \times f'_2 \times \ldots$. We have that $F = [f_1, f_2, \ldots,]$ and $F' = [f'_1, f'_2, \ldots,]$, where F (and F') is the list of factors that Canon (Algorithm A) passes to CanonRec as the first argument. The proof follows by induction on the length of F. We shall demonstrate that $F = F'$ so that these two invocations of CanonRec produce the same program. On the base case, we have that $F = [] = F'$. Thus, CanonRec($[]$, *empty program*, $[], 0$) = CanonRec($[]$, *empty program*, $[], 0$), and it follows that *empty program* = *empty program*.

On the inductive case, we assume that the theorem holds up to a list of factors of length k. Thus, we need to show that

$$\text{CanonRec}([f_k, \ldots], P_k, S_k, v_k) = \text{CanonRec}([f'_k, \ldots], P'_k, S'_k, v'_k)$$

We assume, by induction, that $P_k = P'_k$, $S_k = S'_k$ and $v_k = v'_k$. In what follows, we analyze some of the cases that Algorithm A handles.

Constant: Assume CanonRec($[f, \ldots], P, S, v$) = CanonRec($[f', \ldots], P, S, v$). If the factor f is the certificate of a constant, then it has the exponent $e = 11^c$. The only way the equality can hold is if the factor f' has the exponent $e' = 11^{c'}$, where $c = c'$, following **else if** IsConstant.

Variable definition: Assume CanonRec($[f, \ldots], P, S, v$) = CanonRec($[f', \ldots], P, S, v$). If f encodes the definition of a variable, it has an exponent in the form $e = 13^{var.\ def.\ exp.}$. The equality will only hold if f' also encodes a variable definition with an exponent $e' = 13^{var.\ def.\ exp.'}$, so CanonRec will parse f and f' with the case **if** IsVarDef. Additionally, it is necessary that $var.\ def.\ exp. = var.\ def.\ exp.'$ for the types τ and τ' the algorithm extracts with GetVarType to be identical. The equality between τ and τ' is required for CanonRec to add the same global variable, or struct with the same fields, or array with the same length to the canonical program being reconstructed.

Variable usage: Assume CanonRec($[f, \ldots], P, S, v$) = CanonRec($[f', \ldots], P, S, v$). If f is a factor that encodes the usage of a variable, then its exponent must be $e = 17^{var.\ usage\ exp.}$. For the equality to hold, f' must also have a variable usage encoding exponent, $e' = 17^{var.\ usage\ exp.'}$, so it will fall into the same case — **else if** IsVarUsage. In particular, f and f' must encode the usage of the same variable, such that $var.\ usage\ exp. = var.\ usage\ exp.'$, and the same variable prime vp can be retrieved from it. This is mandatory for CanonRec to obtain the same variable name from GetVarNameFromVarUsage — this method will return the name `var_n`, where n is the index of the variable prime vp in the ordered set of prime numbers.

Operation: Assume CanonRec($[f, \ldots], P, S, v$) = CanonRec($[f', \ldots], P, S, v$). If f is the certificate of an operation, then CanonRec will parse it with the **else if** IsOperation case. It is required that f and f' encode the same operation for the equality to hold, so GetOperationFromExp will return the same symbol in both cases.

Thus, we have that if

$$\text{CanonRec}(F, \textit{empty program}, [], 0) = \text{CanonRec}(F', \textit{empty program}, [], 0)$$

then $F = F'$, and therefore,

$$Cert_H(P_1) = Cert_H(P_2)$$

□

The canonical form is also preserved by the compilation process. In other words, if two programs P_1 and P_2 share the same canonical form, then they compile to the same low-level program P_L. Theorem 4.2.2 proves this property.

Theorem 4.2.2 (Preservation of Canonical Forms)**.** *Let P_1 and P_2 be two programs such that $Canon(Cert_H(P_1)) = Canon(Cert_H(P_2))$. In this case, $Comp(P_1) = Comp(P_2)$.*

Proof. Theorem 4.2.1 demonstrates that if two programs P_1 and P_2 share the same canonical form, then they must have the same certificate. From this property, we conclude that $Cert_H(P_1) = Cert_H(P_2)$ because $Canon(Cert_H(P_1)) = Canon(Cert_H(P_2))$, from the hypothesis.

Theorem 4.1.4, on the other hand, shows that two programs with the same certificate will compile to the same low-level program. As we've learned that $Cert_H(P_1) = Cert_H(P_2)$ from the previous step, then $Comp(P_1) = Comp(P_2)$. □

In this chapter, we established the correctness of our approach by demonstrating the properties of the certification algorithms. In this context, we demonstrated that the variable primes the high- and low-level certification algorithms use are identical for the same program, and, as such, certificates are preserved by the compilation process — low-level programs, in particular, have been shown to be unique. Additionally, we formalized the canonical reconstruction algorithm: the canonical form is unique for each certificate, and it is also preserved by the compilation process.

4.3 Limitations

The proofs presented in this chapter are primarily sketches of the full demonstrations. They are intended to reason about the core logical structure and key arguments, rather than to serve as rigorously verified artifacts. While this approach is sufficient for illustrating the correctness principles the certification method was built upon, a natural direction for future work would be to mechanize these proofs using a proof assistant such as Agda or Coq.

We now turn to the next chapter, where we empirically evaluate the certification algorithms behavior when applied to increasingly larger benchmarks.

Chapter 5

Evaluation

This chapter explores the following research questions:

1. **RQ1:** How does the certificate length scale with the size of the high- and low-level source programs (AST nodes and instructions)?
2. **RQ2:** How does the running time of the high- and low-level certification algorithms grow with program size?

Hardware/Software Experiments evaluated in this chapter were performed on an Apple M1 CPU, featuring 8 GB of integrated RAM, clock of 3.2 GHz, and banks of 192 KB and 128 KB L1 caches — four of each. The experimental setup runs on macOS Sequoia 15.4.1. The CHARON project is implemented in Python version 3.12.4 (Jun'24).

Benchmarks The GitHub repository of the CHARON project[1] contains 20 programs written in CHARONLANG. Each program exercises different features of the high-level language. These programs vary in size, which can be measured in terms of the number of statements, AST nodes, and instructions. This diversity allows us to evaluate the asymptotic behavior of the certification technique proposed in this thesis. The corresponding results are summarized in Figures 5.1 and 5.2, and will be discussed in the remainder of this chapter.

5.1 RQ1: On the Size of Certificates

The compilation certificate is represented as a number whose magnitude grows exponentially with the quantity of constructs in the target program, since each additional construct introduces a new multiplicative factor. However, the certification process does not require computing or storing the exact value of this number. Instead, the certificate

[1] https://github.com/guilhermeolivsilva/project-charon

is maintained in its symbolic form; that is, as a string representing the sequence of multiplicative expressions used to construct it. Therefore, we measure the size of a certificate based on its symbolic representation, defined as the character length of its string form, rather than by the numeric magnitude of the final value. In this section, we show that although the absolute value of a certificate grows exponentially, its symbolic representation grows linearly with the size of the program.

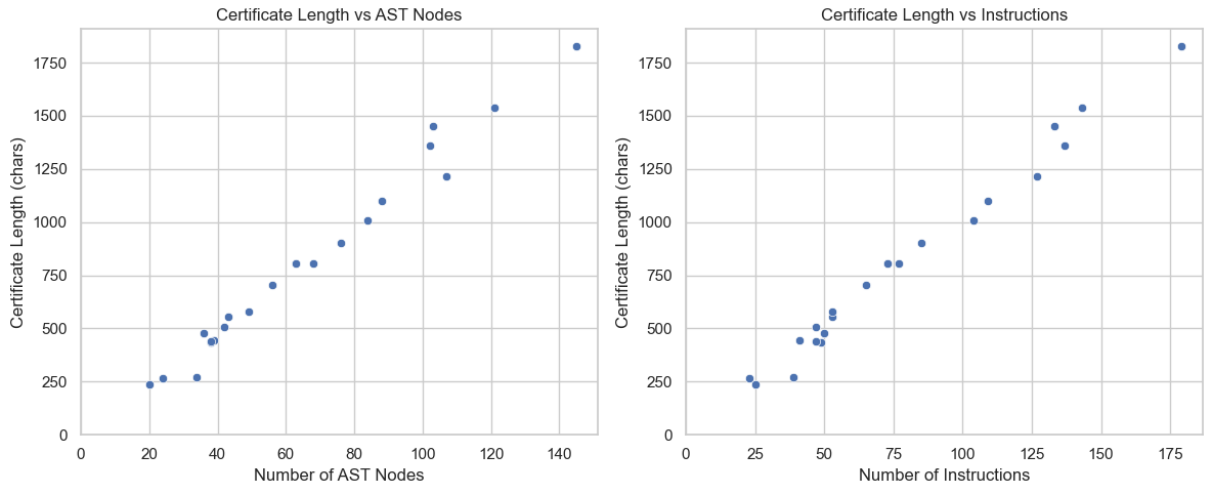


Figure 5.1: Certificate expression length (in characters) vs program size.

Discussion Figure 5.1 illustrates the relationship between program size—measured by the number of AST nodes—and certificate length. The size of the certificates grows linearly with the size of the input program, both in terms of high-level AST nodes and low-level instructions.

In both cases, the linear correlation is very strong: the Pearson correlation coefficient (R^2) between the number of AST nodes and the certificate length is 0.98. Similarly, the R^2 value between the length of certificates and the number of instructions in the CharonIR representation is 0.97. This linear trend reflects the design of the certification rules, where each program construct contributes a constant-size component to the overall certificate expression.

5.2 RQ2: On the Performance of the Certifier

We evaluate algorithmic performance by measuring the average running time of the certification procedures across a suite of test programs, using Python's `%%timeit` for high-resolution timing. Our current implementation, written in Python, is not optimized

for performance and inherits the inefficiencies associated with the language. However, in this section, we demonstrate that the asymptotic complexity of the certification process grows linearly with the size of the input program. This suggests that a reimplementation in a systems-oriented language such as C, C++ or Rust, would likely result in significantly faster certification times.

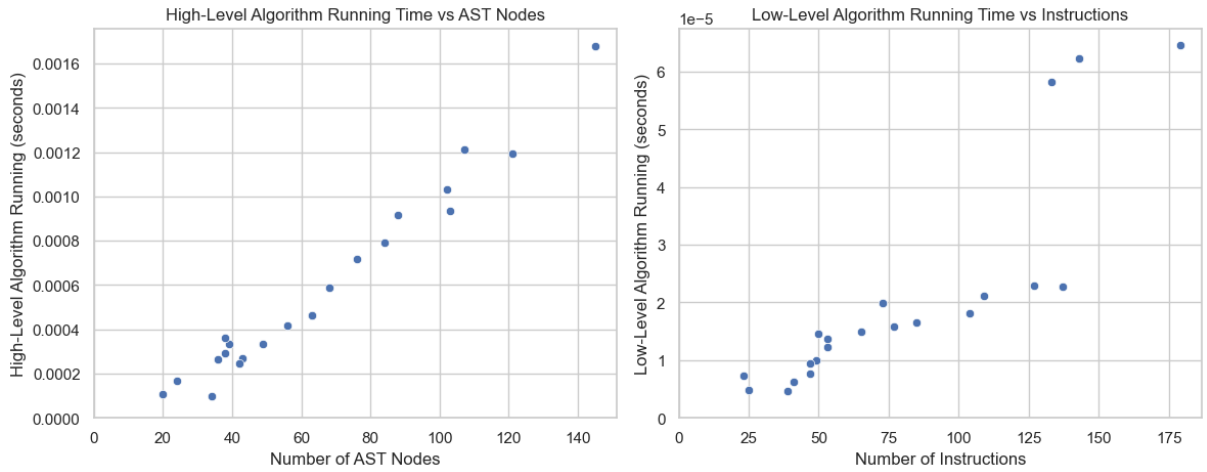


Figure 5.2: Running time (in seconds) vs program size.

Discussion Figure 5.2 presents the correlation between the running time of the certification process and the size of the certified programs. The left-hand plot shows the performance of the high-level algorithm (as a function of AST nodes), and the right-hand plot analyzes the low-level algorithm (as a function of instruction count). The high-level certification algorithm runs in sub-millisecond time across the entire benchmark suite. As shown in Figure 5.2 (left), the runtime scales linearly with the number of AST nodes, reaching approximately 1.6 ms for the largest inputs, which contains 145 nodes. The coefficient of correlation between running time and size is 0.96; hence, very close to a perfectly linear correlation. This behavior confirms the expected complexity of the algorithm, which performs a recursive descent over the abstract syntax tree with constant-time processing at each node.

The low-level certification algorithm, shown in Figure 5.2 (right), exhibits even faster execution times, typically in the tens of microseconds. It processes the instruction stream through a sequential scan based on pattern matching. Although the low-level certification procedure demonstrates near-linear behavior in practice ($R^2 = 0.93$), it is not guaranteed to run in linear time. In particular, the algorithm may exhibit quadratic behavior in the presence of deeply nested branches. This super-linear behavior is observable in our results: the three largest programs in the benchmark suite deviate from the linear trend. These outliers contain several control-flow constructs (e.g., `if` and `while` statements), which trigger multiple executions of the *Analyze-Conditional-Jumps* pass.

The empirical results corroborate the theoretical claims established in the previous chapter: both certificate size and certification time scale linearly with program size under typical conditions, with minor deviations arising in low-level certification for programs containing highly nested control-flow constructs. The next chapter concludes this thesis by discussing its main results, limitations, and directions for extending it in future work.

5.3 Threats to Validity

While the previously discussed results demonstrate the scalability and practicality of the proposed approach, several directions remain open for exploration. In this section, we highlight potential research questions that could extend the impact and applicability of this work.

The current system stores certificates as integers with a very large bit-width, preserving their full structure for verification. As such, an alternative design could represent certificates as fixed-size hash codes, potentially reducing the certificates memory requirements. This raises the question: what would be the collision rate in realistic scenarios, and how would such collisions impact the certification algorithms' properties? Investigating this trade-off could reveal whether hashing offers a practical path for optimizing certificate handling without compromising reliability.

Additionally, our implementation targets an Intermediate Representation inspired by the RISC-V instruction set, rather than the standard architecture itself. This design choice facilitated rapid development and experimentation, but leaves open the question of how the approach would perform against an industrial hardware target. Applying the system to an actual RISC-V backend would provide a more rigorous test of its correctness and efficiency, and could yield insights into its adoption in production compiler toolchains.

Chapter 6

Conclusion

This work has introduced a new certification methodology, which we call *structural certification*. In this approach, the certificate of correct compilation is inherently embedded in the structure of the binary code itself. Unlike previous techniques, such as proof-carrying code, translation validation, or verified compilation, which ask for trust in external artifacts, our approach enables certification through the form and layout of the compiled program. Inspired by Gödel's numbering system, our method encodes syntactic and semantic properties of the source program directly into the binary using arithmetic encodings derived from prime factorizations. This design enables lightweight verification that requires neither theorem provers nor trusted third-party verifiers. We have demonstrated the feasibility of this idea through the development of Charon, a compiler for a subset of C capable of certifying programs written in FaCT. By embedding certification directly into the binary representation, our method opens a new path for certifying compilers—one in which the evidence of correct compilation is inseparable from the code it certifies.

6.1 Limitations

While structural certification offers a novel and lightweight approach to trusted compilation, it comes with limitations. Most notably, the current method assumes a non-optimizing, syntax-directed compiler, where each source construct maps deterministically to a corresponding binary pattern. This limitation restricts support for aggressive low-level optimizations, such as instruction reordering or register allocation. These transformations, if applied on the low-level code would invalidate the numeric correspondence required by our encoding. Thus, programs certified via our approach must be optimized at the source-code level. Moreover, because the certification relies on the preservation of positional and syntactic structure, it may not scale naturally to full-featured programming languages or to target architectures with highly complex instruction sets. Finally, our method

verifies structural fidelity between source and binary, but does not presently capture deeper semantic properties, such as memory safety or functional correctness.

We also acknowledge that the certification infrastructure implemented in Project Charon is not immune to Thompson-like attacks—a more sophisticated adversary could tamper both the compiler and the certification algorithms in such manner so the latter won't explain the malicious behavior injected into the former. However, adding a backdoor to it is a much more difficult challenge, as the *attack surface* is considerably small. It takes only approximately 314 lines of code (LOC) to implement $Cert_H$, but about 4,500 LOC to implement our compiler. Because of this reduced surface, gaining trust in software through auditing, fuzzing, or formal verification of the certification algorithms is a more attainable goal.

6.2 Future Work

These limitations suggest several directions for future research. One avenue is to extend the structural encoding to tolerate limited classes of compiler optimizations, perhaps by generalizing the encoding to sets of acceptable patterns or by embedding structured metadata alongside numeric codes. Another possibility is to enrich the certification mechanism to express and verify semantic properties, such as safety or constant-time guarantees, while retaining the lightweight, arithmetic-checkable format. Additionally, it would be valuable to explore how structural certification interacts with techniques like diverse double-compiling or translation validation, potentially combining their strengths to build higher-assurance compilation pipelines. Finally, further work is needed to assess the robustness of structural certification in adversarial settings, where attackers may attempt to tamper with or forge encodings in compiled binaries.

Software The certifying infrastructure described in this work is available at https://github.com/guilhermeolivsilva/project-charon.

Bibliography

[1] Andrew W Appel. Foundational proof-carrying code. In Proceedings 16th Annual IEEE Symposium on Logic in Computer Science, pages 247–256. IEEE, 2001.

[2] Andrew W Appel and Amy P Felty. A semantic model of types and machine instructions for proof-carrying code. In Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 243–253, 2000.

[3] Krste Asanović and David A. Patterson. Instruction sets should be free: The case for risc-v. Technical Report UCB/EECS-2014-146, University of California at Berkeley, Aug 2014.

[4] Gilles Barthe, Tamara Rezk, and Ando Saabas. Proof obligations preserving compilation. In International Workshop on Formal Aspects in Security and Trust, pages 112–126. Springer, 2005.

[5] Sandrine Blazy and Xavier Leroy. Mechanized semantics for the clight subset of the c language. Journal of Automated Reasoning, 43(3):263–288, 2009.

[6] Xavier Carnavalet and Mohammad Mannan. Challenges and implications of verifiable builds for security-critical open-source software. In Computer Security Applications Conference, pages 16–25, 2014.

[7] Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S. Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. Fact: a dsl for timing-sensitive computation. In PLDI, page 174–189, New York, NY, USA, 2019. Association for Computing Machinery.

[8] Carmine Cesarano, Vivi Andersson, Roberto Natella, and Martin Monperrus. Gosurf: Identifying software supply chain attack vectors in go. In SCORED, page 33–42, New York, NY, USA, 2024. Association for Computing Machinery.

[9] Alonzo Church. An unsolvable problem of elementary number theory. American Journal of Mathematics, 58(2):345–363, 1936.

[10] Christopher Colby, Peter Lee, and George C Necula. A proof-carrying code architecture for java. In Computer Aided Verification: 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000. Proceedings 12, pages 557–560. Springer, 2000.

[11] Russ Cox. Running the "reflections on trusting trust" compiler. https://research.swtch.com/nih, October 2023. Accessed: 2024-12-30.

[12] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst., 13(4):451–490, October 1991.

[13] N.G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem**reprinted from: Indagationes math, 34, 5, p. 381-392, by courtesy of the koninklijke nederlandse akademie van wetenschappen, amsterdam. In R.P. Nederpelt, J.H. Geuvers, and R.C. de Vrijer, editors, Selected Papers on Automath, volume 133 of Studies in Logic and the Foundations of Mathematics, pages 375–388. Elsevier, 1994.

[14] Guilherme de Oliveira Silva and Fernando Magno Quintão Pereira. Certified compilation based on gödel numbers, 2025.

[15] Joshua Drexel, Esther Hänggi, and Iyán Méndez Veiga. Reproducible builds and insights from an independent verifier for arch linux. arXiv preprint arXiv:2505.21642, 2025.

[16] Xinyu Feng, Zhaozhong Ni, Zhong Shao, and Yu Guo. An open framework for foundational proof-carrying code. In Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation, pages 67–78, 2007.

[17] Carl Friedrich Gauss. Disquisitiones arithmeticae. Yale University Press, 1966.

[18] Kurt Gödel. über formal unentscheidbare sätze der principia mathematica und verwandter systeme I. Monatshefte für Mathematik und Physik, 38(1):173–198, 1931. Translated title: "On Formally Undecidable Propositions of Principia Mathematica and Related Systems I".

[19] Paul Govereau. Denotational Translation Validation. PhD thesis, 2012.

[20] Andrea Höller, Nermin Kajtazovic, Tobias Rauter, Kay Römer, and Christian Kreiner. Evaluation of diverse compiling for software-fault detection. In DATE, pages 531–536. IEEE, 2015.

[21] Claude Kirchner, Pierre-Etienne Moreau, and Antoine Reilles. Formal validation of pattern matching code. In PPDP, pages 187–197, New York, NY, USA, 2005. Association for Computing Machinery.

[22] Piergiorgio Ladisa, Serena Elisa Ponta, Antonino Sabetta, Matias Martinez, and Olivier Barais. Journey to the center of software supply chain attacks. IEEE Security and Privacy, 21(6):34–49, August 2023.

[23] Xavier Leroy. Formal verification of a realistic compiler. Commun. ACM, 52(7):107–115, July 2009.

[24] Xavier Leroy. A formally verified compiler back-end. Journal of Automated Reasoning, 43:363–446, 2009.

[25] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. Compcert-a formally verified optimizing compiler. In ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress, 2016.

[26] Zhaopeng Li, Zhong Zhuang, Yiyun Chen, Simin Yang, Zhenting Zhang, and Dawei Fan. A certifying compiler for clike subset of c language. In 2010 4th IEEE International Symposium on Theoretical Aspects of Software Engineering, pages 47–56. IEEE, 2010.

[27] George C. Necula. Proof-carrying code. In POPL, page 106–119, New York, NY, USA, 1997. Association for Computing Machinery.

[28] George C. Necula. Translation validation for an optimizing compiler. In PLDI, page 83–94, New York, NY, USA, 2000. Association for Computing Machinery.

[29] George C Necula and Peter Lee. The design and implementation of a certifying compiler. ACM SIGPLAN Notices, 33(5):333–344, 1998.

[30] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. Backstabber's knife collection: A review of open source software supply chain attacks. In DIMVA, page 23–43, Berlin, Heidelberg, 2020. Springer-Verlag.

[31] Chinenye Okafor, Taylor R. Schorlemmer, Santiago Torres-Arias, and James C. Davis. Sok: Analysis of software supply chain security by establishing secure design properties, 2024.

[32] Amir Pnueli, Ofer Shtrichman, and Michael Siegel. Translation validation: From signal to c. In Correct System Design, Recent Insight and Advances, (to Hans Langmaack on the Occasion of His Retirement from His Professorship at the University of Kiel), page 231–255. Springer-Verlag, 1999.

[33] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In TACAS, page 151–166, Berlin, Heidelberg, 1998. Springer-Verlag.

[34] Silvain Rideau and Xavier Leroy. Validating register allocation and spilling. In ETAPS, page 224–243, Berlin, Heidelberg, 2010. Springer-Verlag.

[35] Xavier Rival. Symbolic transfer function-based approaches to certified compilation. In POPL, page 1–13, New York, NY, USA, 2004. Association for Computing Machinery.

[36] Yrjan Skrimstad. Improving trust in software through diverse double-compiling and reproducible builds. Master's thesis, 2018.

[37] Luigi Soares, Michael Canesche, and Fernando Magno Quintão Pereira. Side-channel elimination via partial control-flow linearization. ACM Trans. Program. Lang. Syst., 45(2), June 2023.

[38] Gabriel L Somlo. Toward a trustable, self-hosting computer system. In SPW, pages 136–143. IEEE, 2020.

[39] Ken Thompson. Reflections on trusting trust. Commun. ACM, 27(8):761–763, August 1984.

[40] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. Proceedings of the London Mathematical Society, 2(42):230–265, 1936.

[41] David Wheeler. Countering trusting trust through diverse double-compiling. In ACSAC, page 33–48, USA, 2005. IEEE Computer Society.

[42] David A. Wheeler. Fully countering trusting trust through diverse double-compiling, 2010.

[43] Anna Zaks and Amir Pnueli. Covac: Compiler validation by program analysis of the cross-product. In International Symposium on Formal Methods, pages 35–51. Springer, 2008.

Appendix A

The Canonical Reconstruction Algorithm

Algorithm 1 CANON — The Reconstruction of Canonical Forms

1: **Input:** Certificate n
2: **Output:** Canonical program P_c
3:
4: **function** CANON(n)
5: **return** CANONREC(F ← FACTORIZE(n), P ← empty program, S ← empty stack, v ← 0)
6:

Algorithm 2 CanonRec — The Recursive Canonical Reconstructor

```
1: Input: Program P, Expression stack S, Variable name counter v
2: Output: Canonical program P_c
3:
4: function CanonRec(F, P, S, v)
5:   if F is empty then
6:     P.AddMain()
7:     return P
8:   end if
9:   f, rest ← F[0], F[1 :]
10:  e ← GetExpFromFactor(f)
11:  if IsVarDef(e) then
12:    v ← v + 1
13:    name ← ``var_'' + v
14:    τ ← GetVarType(e)
15:    if IsSingleType(τ) then
16:      P.AddGlobalVariable(name, τ)
17:    else if IsStruct(e) then
18:      fields ← GetFieldsFromVarType(τ)
19:      P.AddStruct(name, fields)
20:    else
21:      size ← GetSizeFromVarType(τ)
22:      P.AddArray(name, τ, size)
23:    end if
24:  else if IsVarUsage(e) then
25:    name ← GetVarNameFromVarUsage(e)
26:    S.Push(name)
27:  else if IsConstant(e) then
28:    S.Push((c − 1) if c ≥ 0 else c)
29:  else if IsOperation(e) then
30:    op ← GetOperationSymbolFromExp(e)
31:    rhs ← S.Pop()
32:    if IsUnary(e) then
33:      expr ← (op + rhs)
34:    else
35:      lhs ← S.Pop()
36:      expr ← (lhs + op + rhs)
37:    end if
38:    S.Push(expr)
39:  end if
40:  if IsTerminator(e) then
41:    stmt ← S.Pop()
42:    P.AddExpr(stmt)
43:  end if
44:  return CanonRec(rest, P, S, v)
```