μSystems Research Group

School of Engineering

Newcastle University

Formal Methods for Spacecraft Control Programs

Georgy Lukyanov

PhD Thesis

December 2022

Abstract

Software programs that govern various systems often contain subtle errors that slip through even the most rigorous testing and validation routines. One integer overflow error can cause a crash of a spacecraft or a devastating loss of funds in a financial application. Formal methods bring higher levels of correctness guarantees than traditional testing. The aerospace domain requires adherence to high quality standards for both hardware and software system components. Mission requirements motivate development of tailored hardware and software that needs to be rigorously tested to comply with industry quality standards. In this thesis, we apply formal methods and programming languages techniques to design a generic semantics-based verification framework for instruction-set architecture level programs. We instantiate the framework for a custom instruction-set architecture designed for space satellite subsystems and create a formal and executable semantics for the ISA. On top of the semantics, we build a tool set that facilitates simulation, testing, static analysis and formal verification of spacecraft control programs. Our tool aims to shift the main verification effort to an earlier stage in the project timeline, and reduce the costly and time-consuming setbacks caused by bugs discovered on the later stages of system development. We argue that our approach is ISA-generic, and can be applied to other instruction sets and bytecode-style languages.

Acknowledgments

I would like to express gratitude to my first PhD supervisor *Andrey Mokhov*. Andrey's has taught me what does it mean to be a researcher. His curiosity, energy and passion, his attention to detail and, most importantly, his kindness and empathy will always be a standard I will strive to uphold.

I am grateful to *Jakob Lechner*, who has been my host at TU Wien and Ruag Space Austria. Jakob has offered an industry practitioner's perspective on my research. When I write about a "space engineer" in the thesis, Jakob is the person I mean. Without Jakob's input and help, this thesis would end up to be a bunch of castles in the sky.

I would also like to thank *Alex Yakovlev*, who has always been an inspiring leader and provided me with valuable guidance and support in the final stages of my studies, when I needed that most.

Throughout my studies, I interacted with many amazing people, both at Newcastle and while on visits, to whom I am grateful for the opportunity to learn from them. *Danil Sokolov*, who sat right across from my desk, was always ready to discuss research ideas, or to give practical advice on various aspects of life. *Steven Keuchel* and *Dominique Devriese* have taught how to prove soundness of a program logic while also staying sound yourself. *Jakub Hulas*, *Sergey Mileiko* and *Joe Scott* were the best housemates ever: always ready to cheer me up with a joke or cook something together. *Ghaith Tarawneh* helped me to stay fit by dragging me to the university gym.

Finally, I want to thank my wife, *Veronika Lukyanova*. She has always believed in me. The last year of my postgraduate studies was difficult, and if it wasn't for Veronika, I don't think I would have finished my thesis.

For Larisa Ternovskaya

Contents

List of Figures

List of Tables

Glossary

ad-hoc polymorphism a mechanism of overloading a function name with different context-dependent meanings. 46

ADL Architectural Description Language. 17, 18

ADT algebraic data type. 42, 75

ALU arithmetic logic unit. 26, 31, 103

anonymous function A function that can be defined "on-the-fly", at use site, without naming it. 40

ASIC application-specific integrated circuit. 1

ASL Arm Architecture Specification Language. 18

basic block a node in a control-flow graph. Intuitively, a basic block is a sequence of program statements with no jumps in the middle. 102

CFG control-flow graph. 102

concolic execution (from *conc*rete and symb*olic*) an approach to program verification that performs concrete and symbolic execution simultaneously, with the former guiding the latter. 32

data constructor A data constructor, or value constructor, is a function with zero or more arguments that, when fully applied, return a value of the algebraic data type it belongs to. 42

do-notation syntactic sugar for monadic computations. 75

DSL domain-specific language. 126

ECSS European Cooperation for Space Standardization. 3

EDSL embedded domain-specific language. viii, 9, 17, 20, 60, 76, 78, 82, 121, 122, 124–127, 139

ESA European Space Agency. 3, 8, 9

FPGA field-programmable gate array. 1, 4

GHC Glasgow Haskell Compiler. 46–48

HDL Hardware Description Language. 2, 17

higher-order function A function that receives another function as an argument, or returns another function as a result. 39

IDE Integrated Development Environment. 8, 9, 121, 140

ISA instruction-set architecture. 2, 3, 26, 31, 34, 127

LIFO last in, first out. 127

list comprehension a notation for manipulating lists. Widely used in Python, Haskell and other programming languages. 45

loop unrolling A program transformation that translates a program with an arithmetic loop into a one without the loop by concatenating the loop body as much times as there are loop iterations. 88

metalanguage A language used to describe another language. For example, if one programming language is used to implement an interpreter for another programming language, the former is called “metalanguage”, and the latter “object language”. 27

parametric polymorphism A form of generic programming that allows writing functions operating on values of any type. 40

partial correctness A program p is called partially correct with respect to a specification s if it is functionally correct (with respect to s). As opposed to total correctness, the program may diverge on some inputs. 84

proof assistant A specialised software system that is based on formal logic or type theory and is designed to support formal reasoning by mechanically checking correctness of proofs. 50

property-based testing A type of testing which includes generating a number of pseudo-random test cases which satisfy a certain property. 50

REDFIN REDFIN stands for 'REDuced instruction set for Fixed-point & INteger arithmetic'. This instruction set and the corresponding processing core were developed by RUAG Space Austria GmbH for space missions. i, vii–ix, 4–6, 8, 9, 11, 12, 22, 24–26, 28, 30–34, 55, 59–67, 72–80, 82–84, 89, 92, 94, 96–99, 101–104, 106, 112, 113, 115, 117, 118, 121, 122, 124, 125, 127, 128, 131, 132, 137, 139–143

referential transparency An expression is referential transparent if its value only depends on its textual context, and not on some notion of computational history. 38

RISC reduced instruction-set computer. 30, 31

SBV SMT Based Verification. 80, 90

side effect A function is said to have side effects, or to be effectful, if it performs some sort of an additional action besides computing a value from it's arguments. For example, printing something to standard output, or writing to a file. 38

TDD Test-Driven Development. 15

total correctness A program p is called totally correct with respect to a specification s if it is functionally correct (with respect to s) and it terminates for every input. 84

totality A computation is called total if it terminates and produces a value for every value in its domain. 39

Turing-complete A computational system that has a property of Turing completeness. 38

type constructor A type constructor can be though of as a type-level function, that receives zero or more type arguments and constructs the algebraic data type it belongs to. 42, 96

type signature The first line of a function definition in Haskell that states the type of the function.. 38, 46

UVVM Universal VHDL Verification Methodology. 3

Chapter 1

Introduction

Space engineering is famous for the rigour of design, testing and verification of both hardware and software components of systems. Such rigour is motivated by the fact that space electronics is exposed to stress factors that never occur on Earth: radiation, extremely low or high temperatures and absence of gravity. To achieve high fault tolerance, mission-critical circuits are replicated with double or even triple redundancy [1], and are implemented in space-qualified one-time programmable field-programmable gate arrays (FPGAs) or custom application-specific integrated circuits (ASICs). Prohibitive expensiveness of both space-grade FPGAs and small-batch manufactured ASICs is the primary motivator for extensive pre-production design and functionality verification: once an ASIC has been manufactured or an FPGA programmed, a design error discovered at a later stage would have an immense cost.

Software bugs, as opposed to hardware ones, are not set in stone. However, last-resort mitigation techniques such as in-mission software updates can become mission-ending [2]. Therefore, validation and verification of control software is essential to avoid potential critical errors like integer overflows [3] and incorrect unit conversion [4]. Unfortunately, even a formally verified system can expose unwanted behaviour when deployed in space. In [5], the authors analyse undesired triggering of safety-preserving freezes-in-place behaviour of Robonaut2 — a humanoid robot that deployed at International Space Station. Levenson [6] analyses the Lockheed Martin Astronautics (LMA) Centaur's failure to deliver the Milstar

satellite to the intended geostationary orbit of 22,300 miles. The satellite was instead delivered to a different orbit, due to miscoordination of Centaur's Inertial Measurement System (IMS) software and Flight Control Software.

All these systems were developed following strict reliability and dependability guidelines, with great attention paid to the tiniest detail of subsystem interaction. However, software bugs and hardware design flaws still occurred.

In this thesis, we focus specifically on formal specification and verification of software targeting one specific subsystem of a space satellite. We therefore restrict the scope to that one subsystem, and do not consider the satellite as a whole. However, preventing the mishaps referenced earlier in this section would most likely require a system-wide effort.

The subsystem in question is controlled by a custom processing core, and the control programs target this core's instruction-set architecture (ISA). Further in this chapter we summarise the techniques related to formal specification and verification of instruction-set architectures and machine code programs.

The research project that has sparked most of the work behind this thesis has grown form the collaboration of the industry and academia. While our research is academic in nature, we have been closely collaborating with space engineers at RUAG Space Austria to learn about their outlook on formal verification and to guide our research efforts towards the needs of practitioners.

1.1 Microprocessor formal specification and verification

Formal methods have brought a generous harvest to microprocessor designers. Formal verification techniques and, specifically, model checking, are being widely used in industry for specification and verification of correctness of hardware designs. These designs are supplied to verification tools as specifications in an Hardware Description Language (HDL). The most widely accepted industry standard HDLs are VHDL [7] and System Verilog [8]. Another prominent HDL is Bluespec [9] — a modern Hardware Description Language which is an extension of the Haskell programming language. Bluespec also provides a System Verilog frontend for interoperability for other verification tools. Major vendors in the microprocessor industry employ specialised languages to specify both the design of their microarchitectures

and the semantics of ISAs. For example, ARM uses ISA-Formal [10] and SAIL [11]; Intel and AMD use ACL2 [12], and the RISC-V International uses Alloy for the specification of the memory model [13]. These languages were designed to serve as tools for specifying ISAs in a way that allows deriving interpreters and formal verification tools, and are now used as the primary source for the corresponding ISA manuals, substituting prose and pseudocode with formal and executable specifications.

In this thesis, we develop a framework for ISA specification and program verification, which leverages existing and novel functional programming techniques to reduce code repetition and improve the structure of the implementation. We expand the review of previous work in the next chapter 2.

1.2 Formal methods in space industry

Space engineering has a high degree of adoption of hardware formal verification techniques. For example, the European Space Agency (ESA) promotes Universal VHDL Verification Methodology (UVVM) — an open-source methodology that governs the architecture of VHDL testbenches that adhere to the European Cooperation for Space Standardization (ECSS) standards such as "ECSS-Q-ST-60-02C — ASIC and FPGA development" [14].

When it comes to ensuring correctness of software system components, current practice in space engineering has very limited uses of formal verification techniques. Mostly, the verification is performed by means of conventional simulation and testing in the form of unit and integration tests.

The ECSS standard "ECSS-Q-ST-80C Rev.1 — Software product assurance" [15] requires the supplier to define, justify and apply measures to assure dependability and safety of critical software. According to the standard, these measures can include:

1. use of software design or methods that have performed successfully in a similar application;

2. insertion of features for failure isolation and handling (ref. ECSS-Q-HB-80-03, software failure modes and effects analysis);

3. defensive programming techniques, such as input verification and consistency checks;

4. use of a “safe subset” of programming language;
5. use of formal design language for formal proof;
6. 100% code branch coverage at unit testing level;
7. full inspection of source code;
8. witnessed or independent testing;
9. gathering and analysis of failure statistics;
10. removing deactivated code or showing through a combination of analysis and testing that the means by which such code can be inadvertently executed are prevented, isolated, or eliminated.

Some of these measures require direct human intervention, whereas others may be handled by an automated formal verification tool. Of particular interest are measures 3, 4, 5, 6, 9 and 10. In this thesis, we focus on building a formal verification framework that uses formal methods and programming languages techniques to facilitate assurance that these measures are applied.

To provide a more detailed account on how we are going to address these measures, we need to be able to speak more concretely about a particular space system that the software targets. We there for introduce the subject of the case-study of this thesis: the REDFIN instruction-set architecture.

1.3 Motivating case-study: the REDFIN instruction-set architecture

Many spacecraft subsystems rely on integrated circuits to perform control tasks or simple data processing. Typically, these integrated circuits are realised with FPGAs, benefiting from their flexibility and comparably low cost. Modern space-qualified FPGAs that can withstand radiation in Earth orbit or deep space have a limited amount of programmable resources, and it is often not feasible to implement a fully-fledged processor system in such

an FPGA next to the mission-specific circuitry. The REDFIN[1] instruction set was developed to address this issue and meet the following goals: (i) simple instruction set with a small hardware footprint, (ii) reduced complexity to support formal verification of programs, and (iii) deterministic real-time behaviour.

1.3.1 REDFIN Instruction Set and Microarchitecture

REDFIN instructions have a fixed width of 16 bits. The instruction set is based on a register-memory architecture, i.e. instructions can fetch their operands from registers as well as directly from the memory. This architecture favours a small register set, which minimises the hardware footprint of the processing core. Furthermore, the number of instructions in a program is typically smaller in comparison to traditional load/store architectures where all operands have to be transferred to registers before any operations can be performed. There are 47 instructions of the following types:

- Load/store instructions for moving data between registers and memory, and loading of immediate values.
- Integer and fixed-point arithmetic operations.
- Bitwise logical and shift operations.
- Control flow instructions and comparison operations.
- Bus access instructions for read & write operations on an AMBA AHB bus.

The REDFIN processing core fetches instruction and data words from a small and fast on-chip SRAM. This only allows for execution of simple programs, however, it also eliminates the need to implement caches and thus removes a source of non-determinism of conventional processors. High performance is not one of the main goals, hence the core is not pipelined and does not need to resolve data/control hazards or perform any form of speculative execution. These properties greatly simplify worst-case execution time analysis.

[1] REDFIN stands for 'REDuced instruction set for Fixed-point & INteger arithmetic'. This instruction set and the corresponding processing core were developed by RUAG Space Austria GmbH for space missions

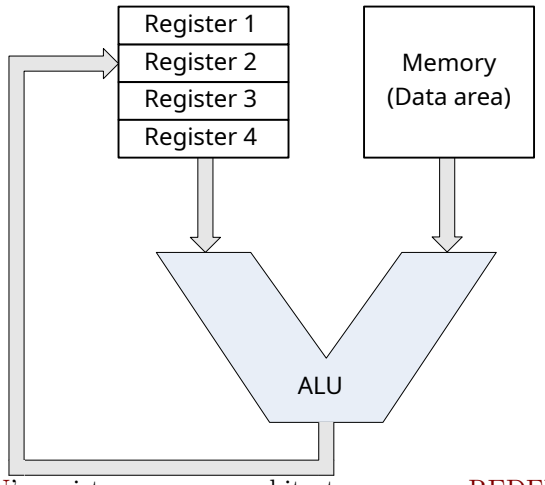


Figure 1.1: REDFIN's register-memory architecture, as per REDFIN V2 data sheet[2].

1.3.2 Intended use-cases for REDFIN

As we have noted before, the REDFIN core is intended to be deployed into highly-specialised subsystems of space satellites. When designing a satellite system, the engineering team must determine whether a certain system should be driven by REDFIN or by another logical unit (for example an ASIC or a conventional processing core). REDFIN is designed to act as a substitute for highly specialised ASICs, and therefore should carry only a very limited functionality. The informal guidelines on program design that we have received from engineers as RUAG Space are:

- subroutines should be very short, rarely exceeding 100 lines of hand-written assembly
- the subroutines should be triggered by the system's central unit

Note also that REDFIN ISA does not have a C compiler, as the effort of its implementation has been deemed unnecessary. If an algorithm is long enough to require a high-level language, it should not target REDFIN.

These considerations make REDFIN the perfect case-study for ISA-level formal verification.

[2]The data sheet is proprietary and could be obtained on request after signing a non-disclosure agreement.

1.3.3 Requirements for Formal Verification

Verification of *functional correctness* of REDFIN programs, as defined by a requirement specification, clearly is an essential task for the development of space electronics. There are also important *non-functional requirements*, such as worst-case execution time and energy consumption, which rely on the implementation guarantees provided by the processing core.

To reduce verification complexity, the REDFIN core only allows to execute a single subroutine whose execution is triggered by a higher-level controller in the system. The implementation guarantees that concurrent bus accesses to the processor registers or memory do not affect the subroutine execution time. Furthermore, the processor does not implement interrupt handling. All these measures are taken to provide real-time subroutine execution guarantees and make the verification of non-functional properties feasible.

Despite these restrictions the REDFIN core has already proved its effectiveness for simple control tasks and arithmetic computations as part of an antenna pointing unit for satellites. Nevertheless, verification can be difficult and time-consuming, even for small and simple programs. Verification activities, following engineering standards for space electronics, typically outweigh programming and design tasks by a factor of two[3] in terms of development hours. Usually verification is performed via program execution on an instruction set simulator or a hardware model of the processor. Manually deriving test cases from the specification is cumbersome and error-prone and simulation times can become prohibitively long with a large number of tests that are often needed to reach the desired functional and code coverage. Formal verification methods can prove that a program satisfies certain properties for all possible test cases and are therefore immensely valuable for completing the verification with superior efficiency and quality.

1.4 Methodology and contributions

1.4.1 A generic semantics-based verification framework

In this thesis, we develop a semantics-based methodology for constructing a generic verification framework for ISA-level programs. The framework has a modular structure and makes

[3] Based on experience of RUAG Space engineers.

a clear separation between the ISA-specific and system-independent modules.

ISA-independent modules include:

- symbolic execution engine
- property specification language
- the core functionality of the Integrated Development Environment (IDE)

ISA-specific modules include:

- the ISA state: execution environment for programs
- the semantics of the particular ISA: how the state is transformed by instructions
- the syntax of the assembly language to write programs in

The framework can be instantiated for any instruction-set architecture by implementing the ISA-specific modules. The ISA-independent modules can be reused as-is.

1.4.2 Methodology of REDFIN program verification

We promote usage of formal verification and programming languages theory-inspired techniques for designing and verifying spacecraft software. By using these techniques, we implement the following measures required by ESA standard "ECSS-Q-ST-80C Rev.1 — Software product assurance":

- We use first-order logic and dependent types as formal specification techniques to specify behaviour of REDFIN programs;
- We use symbolic execution to achieve 100% code coverage: every execution path in the program is checked for compliance with the formal specification;
- The REDFIN core uses "defensive program techniques" to check for integer overflow, memory safety and handling other failures. We employ symbolic execution to prove that these failures never occur in programs that comply with the specification;
- We use control-flow analysis and symbolic execution to detect dead code.

Using the verification framework developed in this thesis, we achieve high compliance with the ESA's Software product assurance standard. By providing high degree of automation and user-friendly interfaces, we enable formal verification techniques to be integrated into everyday workflow of space engineers.

1.4.3 Contributions

We instantiate the generic verification framework for the REDFIN ISA by providing the following:

- Semantics of REDFIN instruction set architecture implemented as a EDSL in Haskell;
- A tool chain for developing REDFIN programs comprising an assembler and a set of command-line tools for program simulation and testing;
- A specification language for functional properties of REDFIN programs that compiles to REDFIN assembly;
- A symbolic execution engine for REDFIN programs that supports verification of program equivalence, safety and liveness properties of programs and Worst-Case Execution Time analysis;
- An IDE that provides a single point of entry to the developed tools and an interactive explorer for symbolic execution traces and verification results.

Alongside that, the thesis contributes two novel functional programming techniques:

- Selective Applicative Functors [16] provide an abstraction for effectful computations with limited dynamic dependencies. The Haskell implementation of the verification framework for REDFIN uses Selective Applicative Functors in its symbolic execution engine;
- Fine-grained store abstraction is used as the metalanguage for defining the semantics of instructions in a way that allows multiple interpretations of the same semantics: efficient simulation, symbolic execution and static analysis.

1.4.4 Further applicability

We believe that the approaches developed in this thesis can be applied to a wide range of instruction-set architectures and bytecode languages. Of particular interest are the low-level languages of blockchain platforms, such as EVM (Ethereum[17]) and TEAL (Algorand[18]), and also the WebAssembly[19]. These languages are lightweight and emphasise correctness over efficiency, therefore are amenable to formal modelling and verification. We expand on this direction in the Future Work (chapter 7).

1.4.5 Publications

1. Steven Keuchel, Sander Huyghebaert, **Georgy Lukyanov**, and Dominique Devriese. **"Verified symbolic execution with Kripke specification monads (and no meta-programming)"**. In: Proceedings of the ACM on Programming Languages, Issue ICFP, 2022. Full text (Open Access).

2. **Georgy Lukyanov**, Andrey Mokhov, and Jakob Lechner. **"Formal Verification of Spacecraft Control Programs"**. In: ACM Transactions on Embedded Computing Systems (2020). Full text.

3. Steven Keuchel, **Georgy Lukyanov**, and Dominique Devriese. **"Katamaran: semi-automated verification of ISA specifications"**. In: REMS-DeepSpec 2020. Extended abstract.

4. Andrey Mokhov, **Georgy Lukyanov**, Simon Marlow, and Jeremie Dimino. **"Selective Applicative Functors"**. In: Proceedings of the ACM on Programming Languages, Issue ICFP, 2019. Full text (Open Access).

5. Andrey Mokhov, **Georgy Lukyanov**, and Jakob Lechner. **"Formal Verification of Spacecraft Control Programs (Experience Report)"**. In: Haskell Symposium 2019. Full text (Open Access).

6. **Georgy Lukyanov** and Andrey Mokhov. **"Concurrency Oracles for Free"**. In: Proceedings of the International Workshop on Algorithms & Theories for the Analysis of Event Data 2018. Full text (Open Access).

1.5 Structure of the thesis

Chapter 1 — Introduction

This chapter introduces the space engineering domain and motivates building a formal specification of the REDFIN ISA and a verification toolchain based on the semantics.

Chapter 2 — Background

In the first half of this chapter, section 2.1, we provide background on known techniques and methods which comprise the state of the art in ISA specification and verification of low-level programs. We discuss the approaches to building ISA specification languages and frameworks. Additionally, we outline the support for formal specification and verification of programs targeting the ISAs which these languages and frameworks provide.

In the second half of this chapter, section 2.2, we provide a short introduction in functional programming in Haskell, since knowledge of certain functional programming techniques is essential for understanding the theoretical contributions of this thesis.

Chapter 3 — Instruction-set architecture semantics

In this chapter we present two approaches to defining ISA semantics we use in this thesis. We build the foundation for the chapters 4 and 5, which discuss in detail the implementation of verification frameworks for REDFIN based on each of the two approaches.

First, we present the concrete and abstract syntax of the REDFIN ISA.

Following the syntax, we present the coarse-grained monadic state transformers and a semantics based on them in section 3.3.1.

As a refinement of the first approach, we build the second one which we base on the novel concept of fine-grained state transformers. This approach enables us to build a dataflow-aware semantics for REDFIN, which we present in the section 3.3.2.

Chapter 4 — REDFIN semantics and program verification with coarse-grained monadic state transformers

We expand on section 3.3.1 and present a verification framework based on the coarse-grained semantics, together with a case-study of spacecraft control program verification.

Chapter 5 — REDFIN semantics and program verification with fine-grained state transformers

We expand on section 3.3.2 and present a verification framework based on the fine-

grained dataflow-aware semantics, together with a case-study of spacecraft control program verification.

Chapter 6 — Tool support

We present the user-facing tools of the verification framework for REDFIN. To present two verification case studies of REDFIN programs that were carried out in collaboration with the authors industrial supervisor, Dr Jakob Lechner, who has been employed as an FPGA and ASIC engineer with Ruag Space Austria Gmbh.

Chapter 7 — Conclusion and future work

We outline the contributions of the thesis and point out possible opportunities for future work on build verification tools for spacecraft control programs.

The flow of this thesis is mostly linear, and the chapters can be read in the order suggested by the figure 1.2. Chapters 4 and 5 can be read independently of each other, and the background chapter may be skipped by an expert reader.

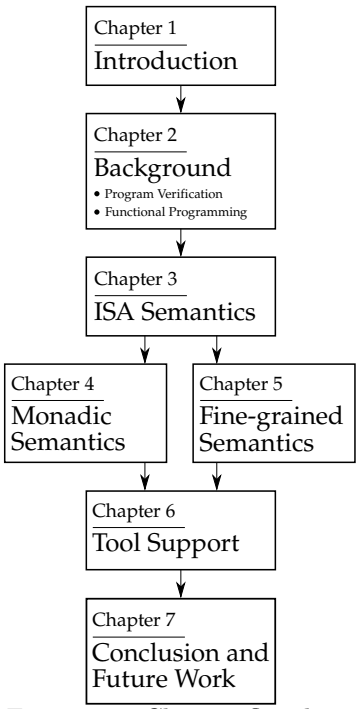


Figure 1.2: Chapter flowchart

Chapter 2

Background

We provide background on techniques and methods used in this thesis. The purpose of this chapter is both to introduce the concepts of formal software verification and functional programming which this thesis builds on, and to highlight the gaps in the state of the art that we aim to cover with our contributions.

This chapter contains two parts. First, we discuss formal verification of software: its purpose and current state, focusing on low-level languages such as assembly and machine code. Second, we introduce the necessary background on functional programming.

2.1 Software verification

Software verification is a big research and practice field that develops methods that aim to find errors in software programs. These methods range from simple manual functionality testing that could be carried out by the program developer's friend over a cup of coffee to multiple person-years efforts aimed at formally verifying computer operating systems and compilers. One pair of axis that software verification methods could be fitted on is *completeness* and *setup effort*. Completeness refers to method's ability to check all possible inputs and execution scenarios, and setup effort indicates how much time must be spent on preparation: specification development and method understanding. Additionally, we depict how much automation the method provides, i.e. the amount of human intervention required

for the testing process. Let us put several data points on the plane, fig. 2.1, equipped with these axes.

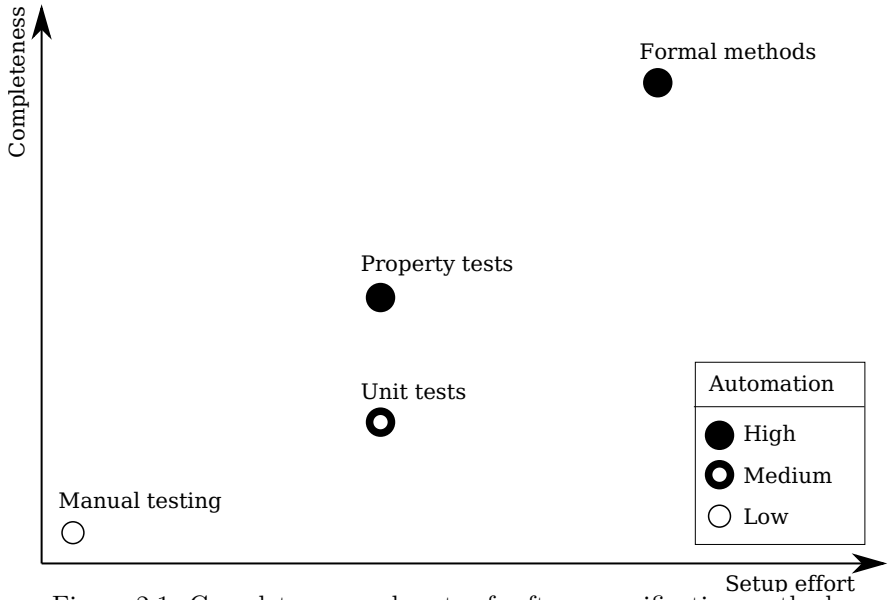


Figure 2.1: Completeness and costs of software verification methods

Manual Testing

Manual testing, while usually having low setup costs, can be time-consuming and is difficult to reproduce and, therefore, is rarely complete. Sometimes, however, manual testing is a perfectly viable option: there is no need to formally verify that a PhD student's web page accurately shows all their published papers, since it is enough to open it in a web browser.

Unit Testing

Most software programs require more serious correctness guarantees. A command-line tool or a web application that have millions of users worldwide need to have their core functionality thoroughly tested. Moreover, the testing process needs to be automated to avoid hours of manual labour on release of every new program version. The current software engineering practice suggests using unit-tests to verify the functionality of application components. Unit-tests provide a simple and intuitive paradigm: the developer needs to identify test inputs for a unit of their program and the results the unit is supposed to produce for these particular inputs. The unit-testing framework will run the program with these inputs and verify that the expected results are produced. Wide adoption of unit test-

ing in its modern form has started with JUnit [20] for the Java object-oriented programming language, giving rise to the paradigm of Test-Driven Development (TDD). In this paradigm, the software engineer first specifies the intended behaviour of the program by a collection of test for every program's unit. This approach is considered one of the best-practises of today's software engineering.

Unit-tests are automated. They could be run both locally by the developer and on a remote server by a build system, facilitating *continuous integration*. However, the completeness of unit tests is restricted to the amount of testing inputs the program's developers are willing to design and maintain.

Property Testing

Completeness of unit tests is restricted: every test represents one point in the program's input domain. Therefore, creating and maintaining a test suite for a large system is a burden, and sometimes makes introducing changes very difficult, since large test suites may become "overfitted" to the current implementation of units. A more progressive approach is *property-based testing*, where single-input tests evolve into *properties* — equality invariants that tie a unit's inputs to its output. Property tests allow generating large sets of input-output pairs and check that the program under test satisfies the property with high assurance.

Property testing frameworks emerged in the Haskell programming language ecosystem, the most popular being QuickCheck [21], SmallCheck [22] and Hedgehog [23]. Other popular programming languages now provide similar tools too.

Property tests provide an improvement over unit tests in terms of automation, since the derivation of test cases is performed automatically. Completeness is improved too: instead of one specific input, the program can be easily tested with hundreds of inputs derived from the property, making corner cases much harder to miss. However, coming up with a good property to test can be difficult for complex program units, thus higher start up human effort is required. Property-based testing requires adjusting the software engineer's mindset to architect the programs in a more modular and compositional way.

Software testing methodologies provide low-overhead ways of verifying correctness of programs. The program's components are enveloped in test suites — collections of unit test cases or properties that specify the behaviour. Having a comprehensive test suite for a software program makes refactoring the codebase and introducing new features safer: an

unintended regression will likely be caught by the test suite. However, test suites remain incomplete, and most likely implicitly so. The process of software *verification* by testing is rarely performed according to a comprehensive *specification*; most often the intended behaviour of the system is described in prose and informal diagrams.

Formal Methods

The term *formal methods* has become a buzzword in the wider software development community. Colloquially, if a piece of software is "formally verified", it is believed to be *correct*, where the definition of correctness is seldom given any attention. However, the very essence of formal methods is the ability to *formally*, mathematically, define what does it mean for a software or hardware system to be correct.

Definition (Formal Software Specification):

A *formal specification* of a software system is a mathematical model of the system which considers a specific collection of the system's behaviours and their invariants.

A formal specification can be a transition system, a Petri Net, a collection of statements in a logic, type theory or a different formal reasoning framework. The crucial thing though that the specification must be *formal*. i.e. it could be reasoned about, mathematically, within a well-founded theory.

Definition (Formal Software Verification):

Formal verification of a software system is a process of establishing, though formal mathematical reasoning, that the implementation of the system conforms to the system's formal specification.

The process of formal verification ensures that a formal mathematical model of the system conforms to the system's formal specification. It is vital that both the specification and the model are expressed in compatible mathematical frameworks.

Software testing can be viewed though this lens too. In the case of testing, the specification is the test-suite, and the model is the implementation of the system itself. Very often both the test suite and the program will be implemented in the same programming

language, but sometimes testing may be handled by an external program.

2.1.1 Formal verification of ISA programs: a related work overview

In this section, we overview the research performed to date on how to formally specify and verify an ISA program. By an ISA program we mean a program in an assembly language for a specific ISA, or in machine codes of the ISA itself. It is not important whether the program has been manually implemented or compiled from a high-level language. Verification of such programs involves building a model of the underlying ISA; therefore, the projects we will discuss often include work on both program verification and ISA formal modelling.

We organise this section as follows:

First, we overview instruction-set architecture modelling languages, which are specifically created to build formal models of ISAs. These languages aim to provide built-in automated verification of essential properties of individual instructions and the ISA as a whole, but have limited or no support for program verification.

Second, we overview models of specific ISAs in more general purpose formal frameworks. Since these models consider a fixed ISA, they are able to focus on the semantics of several instructions executed in sequence, thus achieving whole program verification.

Finally, we position the contributions of this thesis in the presented body of related work.

2.1.1.1 ISA specification languages and frameworks

The design and implementation of ISA and processor description languages and frameworks is a flourishing research sub-field on the edge of Programming Languages and Systems. These languages tend to become more specialised than general HDLs, focusing specifically on processor design, ISA specification, or both. There is a number of standalone domain-specific languages that provide several backends for program simulation, formal verification and building documentation. Other languages follow the alternative methodology of embedding into an existing metalanguage, often of a proof-assistant such as Coq. We overview both the standalone languages and the EDSLs in this section.

2.1.1.1.1 Standalone DSLs for ISA specification In their book "Processor Description Languages" [24], Mishra and Dutt coin the term Architectural Description Languages

(ADLs). The book was published in 2008, and described 11 languages developed across academia and industry to support design and implementation of application-specific instruction-set processors (ASIPs). The ADLs support all stages of processor development: from design to layout and fabrication. However, they were rarely used for general-purpose processors. Since the book was published, there was a significant push from the research and industrial communities to create languages that would be able to accommodate the complexity of general-purpose instruction-set architectures — ISA specification languages, which, in Mishra and Dutt's taxonomy are called "Behavioural Architecture Description Languages".

In contracts to ADLs, ISA specification languages consider only the ISA itself, and are not intended to get involved in any way with microarchitectures implementing the ISA. Therefore, they focus on providing facilities to describe the semantics of an ISA in an implementation-agnostic way. In a way, ISA specification languages are an evolution of ISA manuals, with the informal prose and pseudocode substituted with a formal specification of the instruction semantics.

Most major processor vendors have internal ISA specification languages or frameworks. However, to our knowledge, only Arm Ltd. have made their formal ISA specification public [25]. At Arm, the instruction-set architectures are specified in Arm Architecture Specification Language (ASL) — an imperative language with a specialised static type system that tracks bitvector length, thus eliminating off-by-one errors in specification. The specification written in ASL are the source for both the ISA manuals and the backend simulation and formal verification tool-chains. An overview of Arm's internal use for ASL and the story of its design, implementation and inter-company adoption can be found in Alastair Reid's PhD thesis [26].

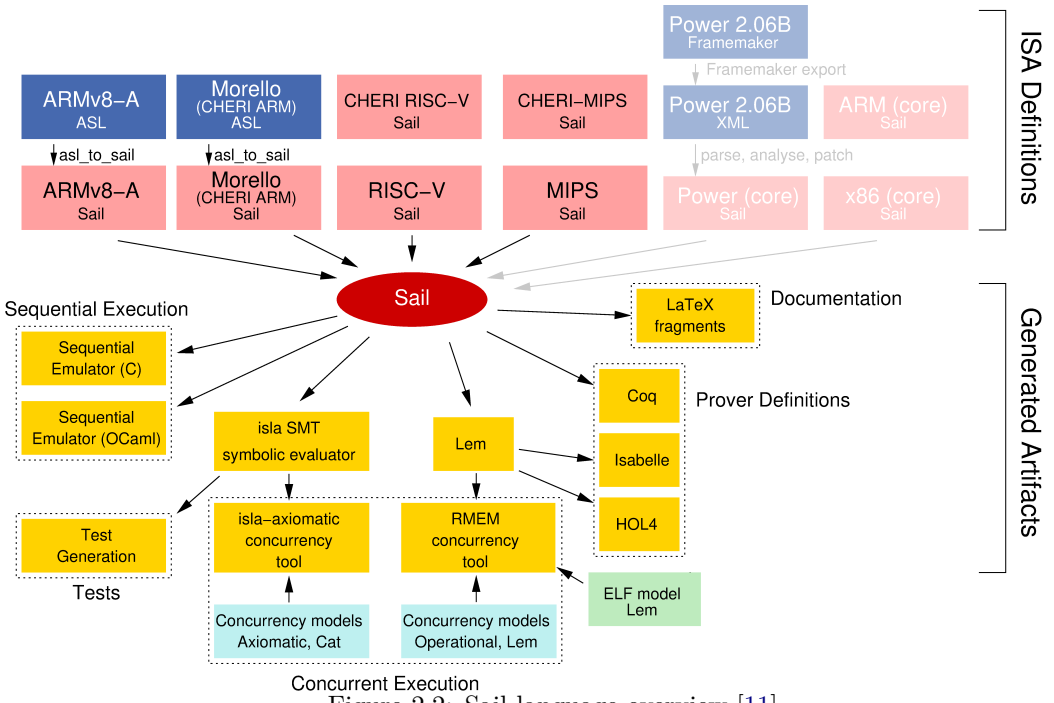


Figure 2.2: Sail language overview [11]

The REMS group at the Cambridge University Computer Laboratory has contributed extensively to the body of research on formal ISA specification. Anthony Fox developed the L3 ISA specification language [27][28], which was used to specify the ARMv7 [28] and ARMv8 [29] instruction-set architectures. The L3 specifications are executable and exportable as HOL4 interactive theorem prover definitions for deductive proof. Later, another ISA specification language, called Sail [11], has been developed by members of the REMS group and their collaborators in other universities. Sail is inspired by L3, and improves over it in terms of language design, providing syntactic and semantics features specifically tailored for writing and maintaining large, complete ISA specifications. Another inspiration for Sail for the ASL language, and the ARMv8.3 ISA specification has been automatically translated to Sail from the Arm's public and internal ASL specs. As other languages, Sail provides derived simulators for the specifications, generates theorem proved definitions, and, additionally, provides a symbolic execution engine for verifying machine code programs, which is a unique feature for an ISA specification language. To our knowledge, Sail is the most advanced ISA specification language to date, and the corpus of the ISA specification implemented in it is the largest and the most complete.

Lyrebird (NICTA and UNSW) [30] — a ISA specification DSL used to support the development of seL4 [31] formally verified operating system microkernel.

Probably one of the earliest ISA specification languages was Barbacci's Instruction set processor specification (ISPS) [32] language. ISPS has been used in a vast body of research on compiler-related techniques.

We conclude this section with a reminder, that the referenced projects develop standalone domain-specific languages. Standalone, as apposed to embedded, languages sacrifice the flexibility of using a powerful metalanguage for the user-experience and performance benefits. Getting the design and implementation of a DSL right is very important for it to be successful. The alternative approach is to embed the language into a flexible metalanguage, thus getting a possibility to adapt the metalanguage's type-system, runtime and tool-chain for the purposes of the EDSL.

2.1.1.1.2 ISA specification EDSLs In the latest years, there has been a surge of project that leverage functional programming languages and proof assistants to build ISA specification EDSLs.

The Kami [33] project is not exactly an ISA specification language, but rather an EDSL for specifying Bluespec-style hardware components. Kami is embedded in the Coq proof assistant, and uses dependent types and proof automation to specify, verify and synthesise hardware implementations that could be directly deployed onto FPGAs. The authors report scaling their approach up to building a formally verified processor with cache-coherent shared memory and pipelined cores implementing (the base integer subset of) the RISC-V instruction set. This work could be used as a bottom layer of a formal specification approach.

The Bedrock project [34] is a Coq framework that uses computational separation logic to achieve close to fully automatic verification of low-level programs. The project is long-standing and has produced numerous advances in Coq-embedded separation logic-based proof systems.

Another Coq-embedded EDSL is Katamaran [35], which makes an emphasis on building a *verified* semi-automatic verification engine for ISA-wide properties. The main goal of the project is specification and verification of hardware capability-enabled ISAs, such as

CHERI [36]. Naturally, this goal requires modelling the target ISA, and Katamaran aims to leverage Coq's type system and proof automation to make the models rigorous, but, at the same moment, easy to reason about.

ISA-level formal modelling is requires by many research and industrial projects in various application domains such as formally verified software and hardware, compilers, high-performance computing and others. Many of these projects only require a part of the ISA to be formally specified, and tend to use the metalanguage of the wider project for the ISA specification. In the following section we overview these models.

2.1.1.2 Specific ISA models

The sel4 project [31], in addition to the Lyrebird language mentioned in the previous section, also employs an Isabelle/HOL specification of the ARMv7 ISA [37] to support verification of the project's C implementation.

In [38] and [39] the authors develop a Proof Carrying Code [40]-related technique to verify complex threading libraries and other examples involving first-class code pointers.

MIT CSAIL aim to create a multi-purpose model [41] of RISC-V ISA that can be reused by other researches for their specific purposes so they do not have spend time on their own formalisations.

The Vale project models the x86-64 ISA [42] to support verification of cryptographic primitives implemented as highly-optimised assembly programs.

Dam at al. developed in HOL4 a verified translation of ARMv7 programs into BAP [43], a multi-ISA binary analysis platform developed at CMU. The authors build on the model to facilitate verification of an ARM-based hypervisor.

Degenbaev has developed a formal Coq model of the x86-64 ISA, with focus on making the specification compact and readable [44].

Kaufmann at al. [45] have built special support into the ACL2 [46] theorem prover to specify microprocessor semantics.

In [47], the authors discuss integrating a symbolic execution-based reasoning into the ACL2 theorem prover to verify x86 programs at scale.

One of the most recent complete models of x86-64 ISA had been developed by Dasgupta at al. [48] using the K Framework. The authors report finding bugs both in the ISA manual

itself and other formal models.

2.1.1.3 Conclusion

These projects and papers constitute only a fraction of the body of research on ISA specification and low-level program verification. We do not aim for this review to be comprehensive, but rather would like to highlight a general trend that has emerged over the past years. It is now considered very important to aim for a complete and comprehensive model of the ISA in question. However, building a complete and faithful ISA specification from scratch is a huge effort, requiring experts with diverse backgrounds and a lot of time. It can be view as an instance of the Pareto principle: it's relatively easy to have 80% of the work done in 20% of time budget, but then the rest takes the remaining 80% 2.3.

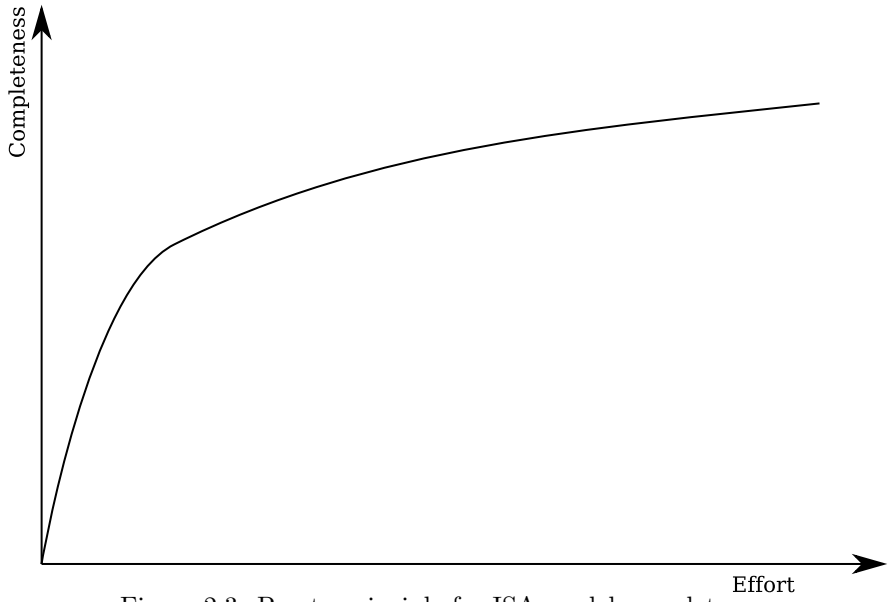


Figure 2.3: Pareto principle for ISA model completeness

Most of the time these remaining 20% are unimportant. As we saw in section 2.1.1.2, many ISA models are tailored for purposes of a specific research project, therefore it is only natural for them to be incomplete. We have experienced this issue ourselves, since our model of the REDFIN ISA does not consider every aspect of it, even though this ISA is relatively small. We, instead, chose to focus on researching the programming paradigms that allow building symbolic execution-based verifiers in functional programming languages.

The languages and tools discussed in this section 2.1.1.1 support building ISA specifications in a more productive and structured way. In future, we would probably see a number of standard, comprehensive models of ISAs implemented in a language like SAIL. These models will be community driven, but also will have support of the microprocessor vendors like Arm, Intel and AMD. Having such model will allow researchers to focus on the applications, building on the solid foundations and not needing to reinvent the wheel of the ISA specification every time.

2.1.2 Caveats of ISA-level formal specification

In this section, we discuss the major point that need to be kept in mind when developing a specification of an ISA, and when building on top of this specification a program verification tool.

2.1.2.1 Numbers are not what they seem

Many bugs arise from the discrepancy between the mathematical ideas that are used to describe what computer programs are supposed to do and the implementations of these ideas as actual computer hardware and software. In mathematics, we think about numbers as mathematical objects, starting from the set of natural numbers $\mathbb{N}$ and gradually building up to the set of real numbers $\mathbb{R}$ and complex numbers $\mathbb{C}$. We carefully design the formal system for the definitions of numbers to be recourse and prove many useful theorems about them. We calculate derivatives of functions and solve differential equations — all that being possible and provably correct thanks to the solid mathematical foundation we build upon. However, when we aim to employ computers to help us in our calculations, we have to abandon the beauty of pure continuous mathematics and succumb to the restrictions of the digital world. However, we sometimes fail to remember that the machine integers we have to work with are only a finite subset of the set of integer numbers $\mathbb{Z}$, and the double precision floating-point numbers are not uncountable, like the *real* real numbers are. This discrepancy can be considered a *leaking abstraction*, meaning that thinking of the machine numbers corresponding to mathematical numbers often is convenient, but there are corner cases when such thinking lets terrible bugs slip into programs. Let us consider some common

bugs that are products of the number representation discrepancy.

Two's complement machine integers

The two's complement representation is one of the most common ways to represent signed integers in modern computers [49], and is the one used in REDFIN too. The numbers are represented as bit vectors of a specific length. The two's complement of a bit vector is calculated by inverting the bits and adding one. For example, consider bit vectors length 3, then 010 represent the decimal integer number 2, while its two's complement, 110, represent -2. Two's complement representation has many convenience advantages over, for example, sign bit representation, such as having unique representation for 0 and fundamental arithmetic operations of addition, subtraction, and multiplication being identical to those for unsigned binary numbers. However, taking a look a table 2.1 reveals that two's complement representation has more space for negative integers than for the positive! In general, two's complement bit vector of length N represents integers from -2^N to $2^N - 1$. This subtle fact may lead to integer overflow and must be accounted for when verifying programs.

Decimal value	Two's complement representation
0	000
1	001
2	010
3	011
-4	100
-3	101
-2	110
-1	111

Table 2.1: Integers represented as two's complement bit vectors of length 3

Integer overflow

To demonstrate the possibility of integer overflow in programs working with two's complement integers, consider the absolute value functions and its implementation in Haskell:

$$|x| = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{if } x < 0 \end{cases}$$

```
abs :: Num a => a -> a -> a
abs x = if x >= 0 then x else -x
```

The type signature of the `abs` function uses the `Num` type class to restrict the type variable `a` to numeric types. The `abs` function can thus be used with the arbitrary precision `Integer` type which denotes the mathematical set $\mathbb{Z}$ of integer numbers. However, will

also be used with, for example, the `Int8` type, which is an instance of the `Bounded` type class, and is implemented via two's complement bit vectors of length 8. Now, according to the mathematical specification of the absolute value function, `abs` should return a positive value for a negative argument. However, explicitly specifying the argument to be of type `Int8` and thus forcing the result to be of the same type reveals the issue:

```
> abs (minBound :: Int8)
-128
```

The fact that the two's complement representation is asymmetrical comes into play and causes a negative result. Let us take a closer look at the evaluation of this program:

```
    abs (minBound :: Int8)
=     { Bounded instance for Int8 }
    abs (-128)
=     { Definition of abs }
    if (-128) >= 0 then (-128) else -(-128))
=     { Semantics of the conditional operator }
    -(-128))
=     { Semantics of unary minus }
    128
=     { 128 overflows the upper bound of Int8 by 1}
    -128
```

The last reduction step is artificially created to illustrate where the integer overflow occurs.

Overflow in integer and fixed-precision arithmetic has led to a number of mission-ending bugs in aerospace systems [2] [3]. Verifying the absence of integer overflow is one of the priorities that we had in mind while designing and implementing the verification framework for REDFIN that is the main contribution of this thesis. In later sections we will give an

account on how REDFIN core itself tracks integer overflow, and how we verify the absence of overflow in REDFIN programs by symbolic execution.

Detecting integer overflow

There are three methods to detect if an arithmetic operation on signed integers overflows [50, p. 170]:

1. Postcondition test using instruction-set architecture status flags
2. Precondition test
3. Extension to a larger signed integer type

The three methods have their advantages and disadvantages. We will now describe them and discuss their applicability to REDFIN.

2.1.2.1.1 Postcondition test Most instruction-set architectures, including REDFIN, have a status register that contains a number of status bits, commonly called *status flags*. If the arithmetic logic unit (ALU), while performing, for example, addition, detects overflow, the relevant flag in the status register is set. When a programmer writes a program targeting a instruction-set architecture, they can examine the overflow status flag after performing addition and see if the result had overflown.

Postcondition testing does not suite us, since we aim to implement the semantics of the REDFIN ISA itself, i.e. to define when the overflow flag should be set.

2.1.2.1.2 Precondition test Precondition testing is a portable way to test if an arithmetic operation will overflow based on the values of the arguments. It involves calculating a specially designed expression prior to performing the operation. Importantly, the intermediate results of the precondition expression will always stay within the bounds of the type in question. Consider an example precondition for addition of signed integers of width 8:

```
addWillOverflow :: Int8 -> Int8 -> Bool
addWillOverflow x y = y > 0 && x > maxBound - y
                    || y < 0 && x < minBound - y
```

The precondition analyses the sign of the second argument and, depending on that, checks if adding the first argument will cause the result to overflow or underflow. Precondi-

tions for subtraction and multiplication could be derived too or can be obtained from the book [50, p. 170] or from the CERT C Coding Standard [51].

2.1.2.1.3 Using a larger signed integer type The third approach is to detect overflow by promoting the bounded integers to an arbitrary-precision type that denote the mathematical integers, and comparing the result of the operation to the upper bound of the original bounded type:

```
addWillOverflow :: Int8 -> Int8 -> Bool
addWillOvervlow x y = toInteger x + toInteger y > toInteger (maxBound :: Int8)
```

Haskell provides the type `Integer` which has arbitrary precision, and the function `toInteger` which can be used to convert a value of any integral type to an `Integer`.

If the metalanguage does not provide an arbitrary-precision integral type, and implementing it is not worth the effort, one can sometimes get away with just a "larger" type. The sum of two numbers represented as bit vectors of length N will always fit into a bit vector of length $N + 1$, while a product will require up to $2N$ bits. For example, a product of two values of `Int8` can be tested for overflow via promotion to `Int16`:

```
addWillOverflow :: Int8 -> Int8 -> Bool
addWillOvervlow x y = fromIntegral x + fromIntegral y > fromIntegral Int8 Int16 maxBound
```

Here we use Haskell's `fromIntegral` function that allows to convert a value of any type that is an instance of `Integral` to a type that is an instance of `Num`. We also use GHC's `TypeApplications` language extension that allows specifying concrete types for type variables.

Recall that symbolic execution provides a way to explore all possible execution paths in a program by treating the inputs as symbolic variables, rather than concrete values. To verify that a program that performs arithmetic operations with symbolic bitvectors never causes integer overflow, the symbolic execution engine must generate verification conditions for every execution paths. These constraints will be determined by the choice of the method to detect overflow. If we choose to detect overflow by promoting the bit vectors to unbounded integers, the overflow constrains will be formulated in terms of the logic that has a notion of integers; alternatively, if we choose to detect overflow by checking

argument preconditions, the constrains can be formulated in terms of the theory of bit vectors. In the verification engine for REDFIN we use the precondition test to determine if an arithmetic operation will overflow, thus keeping the constrains in symbolic variables in the bit vector theory, like the rest of the other types of constraints generated by the framework. The paper [52] provides evidence that using the theory of integer arithmetic for symbolic execution constraints rarely provides significant performance benefits comparing to the theory of bit vectors. Additionally, promoting bit vectors to integers would complicate the definition of ISA semantics; thus, we have made the design decision to check for overflow by evaluating argument preconditions. We will take a closer look at this topic in the chapters 4 and 5.

2.1.3 Symbolic execution

Symbolic execution is an umbrella term for a set of program analysis techniques based on s simple idea: evaluating programs with *symbolic* variables as inputs, allowing to explore the tree of every possible execution of a program instead of just one concrete execution path. Symbolic execution is employed in program verification and automatic test generation at a range of scales and for a wide variety of programming languages. In this thesis, we use symbolic execution for formal verification of assembly language programs targeting REDFIN, aiming at *exhaustive* checking of the programs' state space, thus yielding a sound and complete analysis. In this section, we will provide an introduction to symbolic execution with an accent on assembly code. For a more general introduction we refer to the reader to a Communications of the ACM review article [53] and the more in-depth survey paper [54].

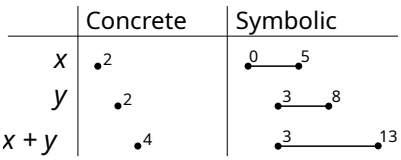


Figure 2.4: Addition of two variables interpreted as concrete values and symbolic ranges approach.

To execute programs symbolically, we first need to define precisely what are these symbolic values we are intending to evaluate the programs with. We have said already that we indent to execute programs with symbolic variables instead of concrete input values. Ultimately, we will be interested in the *ranges* of values that these variables may take: con-

straints on the values will play the key role for verification and test generation. Figure 2.4 shows an example of adding two variables with concrete values and ranges representing symbolic values constrained to be in a certain small interval.

Before going deeper into symbolic executions, let us zoom out and think more generally about what is symbolic execution from a more theoretical point of view.

2.1.3.1 An interlude on programming language semantics

An interpreter of a programming language will evaluate the language's *statements* and *expressions* into a concrete semantic domain. For example, arithmetic expressions can be given semantics in terms of the set of integer $\mathbb{Z}$ or real $\mathbb{R}$ numbers. Expressions are usually pure, i.e. do not have any side effects, and thus can be given such a simple *denotational semantics*, that is, every expression can be evaluated into a single value of the domain it denotes. Now, besides expressions, programming languages will have *statements*, which are the building blocks of programs' control-flow and interactions with the outside world: conditionals, loops, assignment, IO, exceptions, etc. The statements will usually have side effects: assignment can introduce new variables or alter old ones, IO primitives interact with the file system or network and exceptions may cause the program to halt. The most well-established approach to semantics of programming languages is *operational* semantics [55], which formulates the execution of a program as a transition system, where each statement transforms the state of the system in some way. In programming languages theory, these transition systems are traditionally formulated as inductive relations, which makes them convenient to reason about and carry out formal proofs, either on paper or in a proof assistant like Coq [56] or Agda [57]. Denotational semantics of effectful computations, such as statements of an imperative programming language, is an area of active research and there is no consensus what approach is the best. Moggi has introduced *categorical* semantics [58], which is sometimes considered a variant of denotational semantics, and employs the category theoretic notion of a monad to formulate the semantics of effectful computations. Subsequently, monads were popularised by Wadler [59] for functional programming and now constitute one of the corner stones of the Haskell programming language.

Symbolic execution engines are not that much different from the usual interpreters: they too give semantics to expressions and statements of a programming language. However, the

difference lies in the domain which is used for interpretation of programs. Interpreters perform concrete execution and thus evaluate expressions into concrete values and statements into concrete state alterations, thus resulting in a single concrete execution which is determined by the values of the program inputs and the environment. Symbolic executions operates over symbolic inputs and thus must somehow consider all possible executions that may arise from these inputs. The result of symbolic execution is then a, possibly infinite, *symbolic execution tree* which can be thought of as an unfolding of the transition system that constitutes the operational semantics of a programming language, for a particular program.

The symbolic execution tree of a program is a precise representation of the program's all possible executions, and that makes it a great basis for formal verification and analysis of programs. Each node in the tree corresponds to a state in one of the executions and is uniquely determined by an ordered sequence of *path constraints* — logical formulas that depend on the program's inputs. By using a constraint solver, we can check these constraints for satisfiability and thus determine if the state specified by them is reachable. If the state is reachable, we can ask the solver for concrete values of the input variables that steer the execution to follow the path to a particular program state, thus essentially producing a unit test. Even more interestingly, we can use some sort of logic to formulate properties about programs that can be checked for every reachable state in the tree, thus giving us basis for formal verification. We can also check a property as we perform symbolic execution, thus essentially performing model-checking.

The way programs are evaluated to produce a symbolic execution tree, and what is done during and afterwards, is the subject of symbolic execution as a research area, and, consecutively, of this thesis. More specifically, we apply symbolic execution to verification of machine code programs, while often drawing inspiration from programming languages theory to structure the ways we do it.

2.1.4 Symbolic execution of machine code

In this thesis, we discuss the design and implementation of symbolic execution engines targeting REDFIN programs. As we have noted before, REDFIN is a control unit for subsystems of space satellites and is designed with formal verification in mind. REDFIN is a reduced instruction-set computer (RISC) architecture, and its instruction set is optimised

for simple control tasks and arithmetic with integer and fixed-point numbers.

In the verification frameworks we have developed, we perform symbolic execution of programs in the same form that the REDFIN core itself does: as a list of binary instruction codes. However, writing programs in such form is unthinkable today, thus we provide both a low-level assembly language for programming REDFIN and a high-level language of arithmetic expressions that can be compiled into machine codes and used as a specification language for equivalence checking. An assembly language adds only a very thin layer of abstraction on top of an ISA. In the case of REDFIN, the assembly language provides named labels for program locations and conditional `goto` macros for forward and backward jumps to labels. A simple process translates assembly programs into machine code by resolving the labels into instruction counter values and expanding the `goto` macros into jump instructions.

In the previous section, we discussed that in programming languages there is often a distinction between expressions and statements. The former are pure computations that can be given semantics simply by evaluating them, and the latter are computations that have side-effects and need to be treated differently. In the specific case of machine code, we do not have expressions anymore: since our "programming language" is the ISA itself, we only can program with the instructions of this ISA. We therefore need to give semantics to every instruction of the ISA, i.e. specify the transition system that has ISA configurations for states and instructions for transitions. However, since we are interested in execution of programs, and not just single instructions, we also need to include the semantics of fetching and decoding an instruction into the transition system. We will discuss in detail the semantics of REDFIN instructions in chapters 4 and 5.

We proceed with presentation of symbolic execution of machine code by discussing the entities a symbolic execution engine should consist of and how it operates them.

2.1.4.1 Symbolic representation of REDFIN ISA data

The REDFIN ISA is a RISC architecture and its ALU can load arguments of the arithmetic and boolean logical instructions both from its 4 registers and memory, as the figure 2.5 shows. Notably, REDFIN provides separate data and program memory areas.

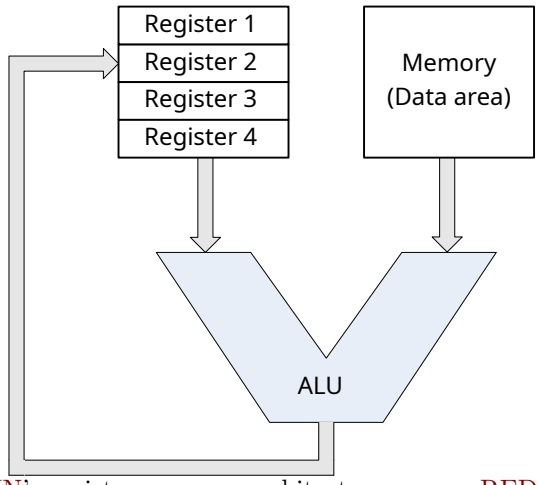


Figure 2.5: REDFIN's register-memory architecture, as per REDFIN V2 data sheet.

When designing a symbolic execution engine, one important decision the engineer has to make is to which parts of the system model completely symbolically and what should be left concrete. This design decision will affect the balance between the engine's scalability, i.e. the ability to handle long and complex programs, and its verification power, i.e what kind of program properties it can verify. For the framework to be applicable to large programs in general-purpose programming languages like C or Java, many tools opt out for so called concolic execution, which combines concrete and symbolic execution and is mainly aimed to automatically generate test cases completely covering all execution paths. In this thesis, we are, on the contrary, aiming to stick as much as possible to symbolic representation, since the simplicity of the REDFIN ISA and the fact that the longest REDFIN programs fit into only hundreds of lines of assembly. This enables us to provide a verification framework that allows sound and complete verification of safety properties and programs equivalence checking.

In our models, REDFIN registers, memory and flags store symbolic expressions. The registers and memory will store machine integers represented with symbolic bit vectors of a particular with, while the flags will store symbolic booleans, i.e. the results of equality or inequality comparisons of symbolic bit vectors. As we have said before, the ultimate goal of symbolically executing REDFIN programs is to verify their properties by generating logical expressions that will be checked for satisfiability with a constraint solver. In this thesis, we use Z3 [60] SMT-solver as the constraint solver and communicate with it using

the SMT-LIB2 [61] language. The SMT-LIB2 language is solver-agnostic and thus can be used with a number of other solvers. We use SMT-LIB2's QF_BV, a quantifier-free logic of fixed-width bit vectors, to represent constraints over symbolic variables. The logic supports arithmetic and logic operations, and bit-precise manipulation of bit vectors, which we will use for conversion between immediate arguments and the usual values. The formulas of the logic can have variables, but can not have quantifiers.

Type	Name	Purpose	Representation	Bit width
Register	R0	General-purpose	Symbolic	16
	R1	General-purpose	Symbolic	16
	R2	General-purpose	Symbolic	16
	R3	General-purpose	Symbolic	16
	IC	Instruction counter	Concrete	8
	IR	Instruction register	Concrete	16
Flag	Overflow	Track integer overflow	Symbolic	1
	Halted	Track program termination	Concrete	1
	...	Omitted here		
Data memory	0		Symbolic	8
	...		Symbolic	8
	255		Symbolic	8
Program memory	0		Concrete	8
	...		Concrete	8
	255		Concrete	8

Table 2.2: REDFIN's data locations

The table 2.2 presents the different data locations of the REDFIN ISA, as they are modelled in the verification framework described in the chapter 5. In that implementation, the registers and data memory store symbolic expressions over bit vectors of width 16. We force some entities to be concrete. The instruction counter, for example, stores the index of the next instruction to be fetched from the program memory, and needs to be concrete in order to represent the program memory as a concrete array.

Instruction counter can only be forced to become symbolic by a conditional jump instruction. However, the jump instruction in REDFIN can only have immediate argument offsets, which are inherently concrete. We choose to fork the execution into two branches of the symbolic execution tree, one where the jump condition is true, and the other where it is false: this allows us to keep the instruction counter concrete by incrementing it in the non-jumped state and adding the concrete offset in the jumped state. Ultimately, this approach leads to the Halted flag to be concrete too, since it can only be set by REDFIN's

`halt` instruction which will be reachable by a sequence of *concrete* alterations of the instruction counter. However, the verification condition of program termination will include the symbolic path constraints associated with these concrete alterations, therefore it still will be sound and complete. We will discuss these matters in more details in chapters 4 and 5.

The `Overflow` flag is altered by arithmetic instructions and after their execution will store a symbolic boolean that encodes the constraint on the instruction's arguments which causes the result to overflow. We discus the overflow conditions in more detail in section 2.1.2.1.

The REDFIN ISA has a number of other status flag that track validity of data and program memory access, division by zero and input-output bus. We will discuss the other flags, their representation and use in chapter 5 when talking in detail about semantics of instructions. We do not consider the IO-related functionality in this thesis at all.

2.1.4.2 Symbolic execution strategies

In the previous section, we have glanced over the symbolic execution strategy we employ in the symbolic execution engine that will be discussed in chapter 5. We now present a more general outlook at symbolic execution strategies, their features and how they can be used for formal verification of machine code programs. We both present the background knowledge and simultaneously discuss the design decisions we have made in the implementation of our symbolic execution framework for REDFIN.

Why at all should we be concerned about how we execute programs symbolically? The answer to this question lies in the challenges that symbolic execution of programs poses, the main one being the *path explosion* problem. This problem shows the most if we consider a naive forward symbolic execution algorithm that forks the state at every conditional jump, effectively causing the symbolic execution tree to have the number of branches exponential in the number of conditional jump instructions in the program's *unfolding*. That may be fine if the program is short and does not have loops, thus making the program's unfolding the program itself. However, if the program does contain a loop, the number of paths will grow quickly and may be infinite.

There is a number of ways to deal with the path explosion problem, and we will overview the mitigation strategies that we have used in this thesis. For a more comprehensive presentation, we refer the reader to the survey paper [54, section 5]

2.1.4.2.1 Pruning unreachable branches on-the-fly Once again, the naive symbolic execution strategy will fork at every conditional jump, creating two new states: the "jumped" state in which the jump's condition is true and the jump is performed, and a "fall-through" state, where the condition is false and the jump is skipped.

Recall now, that at these two states the path condition will be a conjunction of the path condition at the parent (pre-jump) state and either the jump condition or its negation. When symbolically executing the jump instruction, the engine can query a constraint solver and check if the path conditions of the children states are satisfiable. If the solver reports a path condition as *unsatisfiable*, then it is safe to say that the corresponding branch is *unreachable*, i.e. there are no assignments of program variables that can lead the execution into this branch.

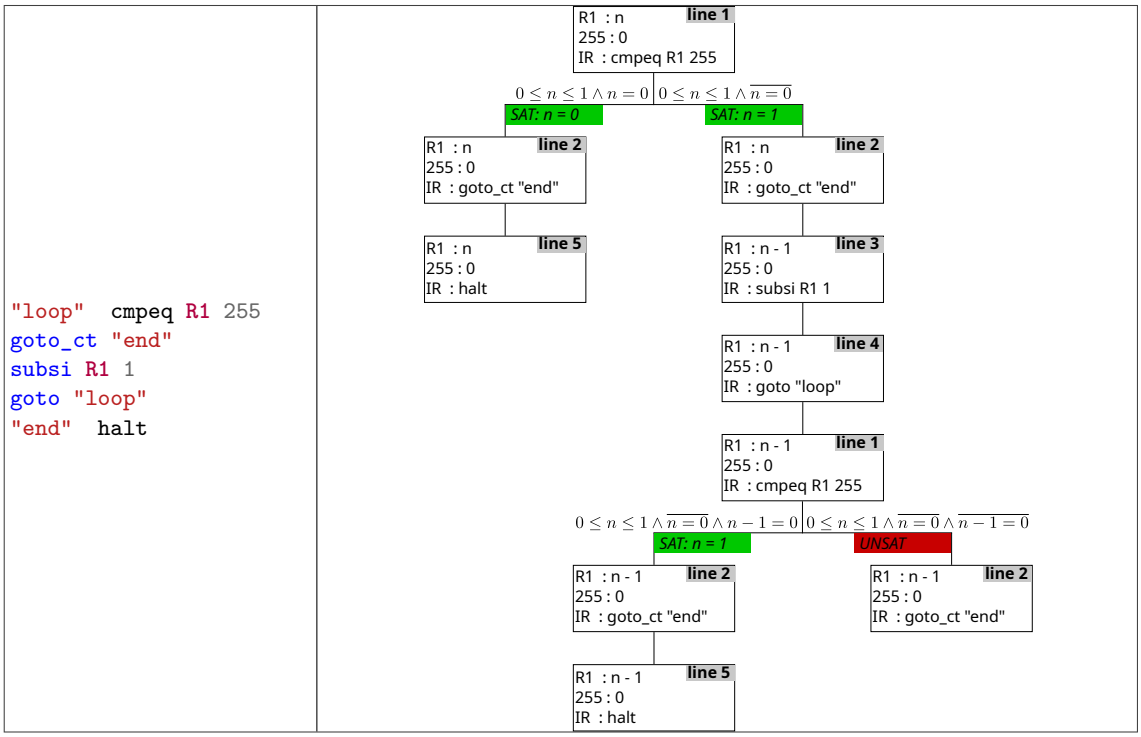


Figure 2.6: Pruning of an unreachable branch

The figure 2.6 displays a simple program that decrements the contents of the register `R1` is a loop. Initially, the registers `R1` contains the symbolic variable n, which we constraint with a precondition $0 \leq n \leq 1$. The line 1 can be though of as a *while*-loop header, which checks if the value of `R1` is equal to the value in the memory cell 255 (containing the

concrete value 0). If `R1` contains a non-zero value, the loop will be executed and the value in `R1` decremented. Every time the conditional jump instruction at line 1 is executed, the symbolic execution engine queries the constraint solver to check the current path condition for satisfiability. As the symbolic execution tree on the right-hand side shows, after one iteration the path condition will become unsatisfiable, thus making all further loop iterations unreachable. Relaxing the precondition on n can force the engine to explore more iterations off the loop, but the same idea applies: there is no need to explore the potentially infinite symbolic execution tree if some paths are guarded by unsatisfiable path conditions.

Pruning of unsatisfiable states has become one of the standard techniques to combat the symbolic path explosion problem. Today, most symbolic execution frameworks such as KLEE [62], EXE [63], PathFinder [64] and others employ off-the-shelf or custom constraint solvers to prune unreachable paths.

In our work, pruning of paths is especially important, because we aim for completely symbolic execution of programs, and therefore other techniques, such is concretisation, are not applicable.

2.1.4.2.2 State merging Another, orthogonal way to mitigate path explosion is to identify similarities in certain execution states and merge them into one, thus reducing the amount of states that need to be analysed. Symbolic state merging has been deeply researched by Kuznetsov at al. [65] as a technique to speed up test generation and bug-finding. Although state merging is a powerful technique, it needs to be used with great care, since it causes the path conditions to grow significantly, and effectively movers the computational load from the symbolic execution engine to the constraint solver.

We use state merging in the part of our symbolic execution framework that handles the specification and verification of arithmetic expressions. Since these expressions only have a finite number of execution paths, i.e. no loops, merging them into a single state transition effectively allows us to create their symbolic summaries.

2.1.4.2.3 Loop summaries Loops and recursive function calls present, perhaps, the most difficult challenge for traditional forward symbolic execution. As we noted before, symbolic execution works by effectively unrolling the state transition of the program's operational semantics into a symbolic execution tree. If the transition system contains a loop, or a

recursive function call, guarded by a symbolic variable, then the unfolding depth becomes bounded by the variable's domain. Even though computers operate with finite machine integers, for all intents and peruses such unfolding can be considered infinite, since it is infeasible to explore them exhaustively.

However, often it is only necessary to explore the body of a symbolically guarded loop once (or a small number of times) to generate a *loop summary*. A loop summary is a collection of symbolic state transformers that alter the state of the variables the loop touches. The ease of generating such summaries depends on the structure of the loop, i.e. is it nested or not, and on how the variables are updates. In their work Godfefroid at al. [66] summarise loops by detecting *induction variables*, i.e. the variables that are only changed by a constant value on every iteration.

2.1.4.2.4 Incremental solving As the engine explores the symbolic execution tree, it issues queries to the solver to check satisfiability of path conditions. Although it is possible to spawn a fresh solver process for every query, it is wise to take advantage of the modern SMT-solvers' mechanisms to keep track of the previously solved constraints. Constraints produced over the course of symbolic execution are often similar since every newly generated constraint will share a prefix with the constraint associated to the parent node. The paper [67] provides a comprehensive overview and benchmarks on using constraints caching and incremental solving to speed up symbolic execution. In our work, we use the Z3 SMT solver's intrinsic capability to cache solutions of sub-formulas to speed-up constraint solving. An interesting opportunity for future work would be to identify domain-specific constraints and implement native caching for those in our system itself.

2.2 Functional Programming

Functional programming is a programming paradigm that models programs as functions, in mathematics sense, i.e. as mapping from inputs to outputs. More formally, execution of programs is represented as *reduction of expressions*, usually in some flavour of Church's λ-calculus [68]. Picking the exact flavour of λ-calculus to represent one's functional programs is no simple task. On the contrary, this question constitutes one of the major directions in

programming languages research.

In this thesis, we will use the programming language Haskell [69], which is based on a *typed* λ-calculus, and puts special emphasis on careful handling of programs with side effects.

This section presents, with examples in Haskell, the basic concepts of functional programming which are necessary for understanding the contributions of this thesis. We do not aim for a comprehensive introduction to functional programming in Haskell; rather, we would like to provide the reader with a quick reference which they may use while reading this thesis.

2.2.1 Pure functions, totality and side effects

The key concept of functional programming is the idea that all programs can be represented as functions, i.e. mappings between types. For example, checking if a number is even could be done with the following simple function:

```
isEven :: Integer -> Bool
isEven x = x % 2 == 0
```

This definition comprises two lines. The first line is called a *type signature*. This type signature declares the type of the function named `isEven` to be a mapping from the type if unbounded integers to booleans. The second line is an *equation*, which gives the function its definition: to determine if the argument `x` is even or odd, the function must compare the reminder of dividing `x` by `2` to `0`. As we will see later, function definitions may comprise several equations.

The function `isEven` is called *pure*, according to Haskell's classification, because it behaves like a mathematical function, i.e. when given an argument, it always produces the same value corresponding to that argument. Indeed, for every integer value the `isEven` function will determine if the value is even or odd. This property is also sometimes referred as referential transparency [70]. However, we must point out here that even though Haskell's functions are indeed very much like mathematical functions, there is a caveat that makes them different. Since Haskell is Turing-complete, there are valid Haskell programs that never terminate. Every Haskell type has an implicit value $\bot$ (pronounced "bottom"),

which represents an infinitely-looping program. Coincidentally, the presence of $\bot$ prevents Hask, the collection of all Haskell types, from being a category in the category-theoretic sense. Speaking more formally, Haskell's type system does not track totality of functions, and consider functions to be pure even if they are not guaranteed to terminate. In other words, non-termination is not a side effect in Haskell.

But what is then a side effect, and how functions with side effects are expressed in Haskell?

The `isEven` function that we wrote earlier is in fact useless to us because we currently do not know of a way to actually run it. Fortunately, Haskell provides primitives that allow us to bridge our programs with the outside world:

```
isEvenTest :: IO Bool
isEvenTest = do
  x <- readLn
  return (isEven x)
```

Here, the `isEvenTest` function is an *impure*, or effectful, counterpart of the pure `isEven` function. Its type signature declares that `isEvenTest` does not have any input arguments, and its result is a boolean annotated with a special marker `IO`, which indicates that the said boolean bears a mark of the input-output side effect. Indeed, the integer value to be checked for evenness is requested from the user input via the `readLn` function, and then supplied as an argument to the pure `isEven` function. In such a manner, Haskell provides ways to clearly separate the code which deals with input-output from the "business logic" of the program, which may remain pure for better maintainability.

2.2.2 Higher-order functions and recursion

In functional programming, functions can be operated over just the same as values of primitive types such as numbers or characters. Functions can be stored in data structures, passed to other functions as arguments and returned from other functions as results. Non-trivial functional programs are written using higher-order functions — functions operating over functions.

2.2.2.1 `map` — structure-preserving transformations of lists

For example, consider a function that traverses a linked-list and applies a function to its every element:

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f x
```

The type signature of `map` states that it receives two arguments: a function from type `a` to type `b`, and a list of type `a`; as a result, it produces a list of type `b`. The function's definition comprises two equations: first, if the input list is empty, an empty list is returned. Otherwise, the list should consist of a value `x` of type `a` (the head), and a possibly empty list `xs` (the tail); then the definition in to apply the argument function `f` to the value `x` and to append the resulting value of type `b` to the recursive call of the `map` function on the tail.

The `map` function is often used with anonymous functions as arguments, for example to append a prefix to every character string in a list:

```
> map (\s -> "edited_" ++ s) ["photo1.jpg", "photo2.jpg", "photo3.jpg"]
["edited_photo1.jpg", "edited_photo2.jpg", "edited_photo3.jpg"]
```

Here, `(\s -> "edited_" ++ s)` is an anonymous function, or lambda-expression, that is applied to every element of the list. Anonymous functions provide a way to quickly define short functions right at use-sites, avoiding the burden of naming them and overpopulating the module's name space.

Note that the type of the `map` function does not refer to concrete types, but rather is abstracted over type variables `a` and `b`. We say that `map` is *parametrically polymorphic*, i.e. its arguments are type parameters, rather than concrete types. parametric polymorphism is a form of *generic programming*.

2.2.2.2 `foldr` — computing summaries of lists

The `map` function provides a way to apply a function to every element of a linked list, independently. In fact, there is no way `map` can be used to consider, say, previous elements while modifying the current one. To perform more general operations with lists, another higher-order function is used:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

The function `foldr` (often called "right fold") provides a way to compute a "summary" of a list using a function `f` that combines an argument of type `a` (the type of the list's elements) and the so-far accumulated summary of type `b`. The result of right-folding an empty list is equal to the initial summary of type `b`, provided as the second argument. If the list is not empty, the result will be the combination of `x`, the head of the list, and the recursive call which produces the summary of the tail `xs`.

For example, the sum of a list of integers can be calculated using `foldr` in the following way:

```
sum :: [Integer] -> Integer
sum xs = foldr (+) 0 xs
```

Here, the `foldr` function is provided with the addition as the way to compute the summary, and with `0` as the initial summary.

The result of folding a list does not have to be a primitive type, but it can also be another list. For example, the `map` higher-order function can be implemented via `foldr`:

```
mapViaFoldr :: (a -> b) -> [a] -> [b]
mapViaFoldr f = foldr (\x summary -> f x : summary) []
```

If we look at the definition of `mapViaFoldr` carefully, we can see two facts:

1. It is not recursive,

2. It resembles the recursive definition of `map` we saw earlier.

These two facts are consequences of a deep insight. In fact, as pointed out by Hutton [71], `foldr` captures a simple pattern of recursion for processing list. In his paper, Hutton explores the limits of expressiveness of `foldr` in a lazy functional language with tuples and first-class functions (Haskell matches these requirements), and describes how `foldr` can be used to *systematically* construct functional programs and proofs about them. In the next section, we will revisit the definition of `foldr` to see that it is a generic *structural* transformation that is guided by the shape of the list algebraic data type.

2.2.3 Algebraic data types

As we have seen, the key concept of functional programming is to think about functions as first-class values, manipulate them freely and combine them in various ways. However, *typed* functional programming languages, and Haskell in particular, are putting emphasis not only on functions, but on *data types* the functions operate with. The concept of algebraic data types (ADTs) is in fact the one that makes Haskell the language I, personally, and many researchers and software developers love so much.

According to the History of Haskell [72], the concept of algebraic data types as we know it today has been first introduced in the Hope programming language [70]. Algebraic data types were subsequently introduced into Haskell and constitute one of its fundamental features up to this day.

In this section, we present algebraic data types, and outline their role in structuring functional programs.

2.2.3.1 Why "algebraic"

As we will see in this section, there are two kinds of algebraic data types: sum-types, which roughly correspond to enumerations in imperative programming languages, and product-types, which correspond to records or structures. Data types can be freely combined using these sum and product operations to form new data types, and that intuition motivates the name "algebraic".

2.2.3.2 Sum types

First, let us consider an example:

```
data Bool = True | False
```

The `Bool` data type is a sum-type of two possible values, `True` and `False`, called data constructors, while the `Bool` itself is called a type constructor. A value of type `Bool` is either `True` of `False`, but never both.

Typed functional programming languages, including Haskell, have a mechanism of pattern-matching for algebraic data types, which provides a way to break them down into their data

constructors. Earlier in this chapter we were defining higher-order functions with equations, where in fact every equation was an analysis case describing what to do with every data constructor of the argument type. For `Bool`, we can use pattern-matching to define conjunction, i.e. the logical "and" operation:

```
and :: Bool -> Bool -> Bool
and False _ = False
and True x  = x
```

Here we perform pattern-matching on the first argument to analyse its value. If it is `False`, we are free to ignore the value of the second argument since the result of the function will be `False` anyway. We do so by specifying the `_` (pronounced "wildcard") pattern for the second argument, effectively saying that we do not care what it is. Now, the first argument can only take one other value, the data constructor `True`, in which case the result of the `and` function will be equal to whatever the second argument is.

In general, a sum-type is a type constructor, possibly with type arguments, and an enumeration of *disjoint* data constructors that may value arguments. For example, another ubiquitous data type is:

```
data Maybe a = Just a | Nothing
```

The `Maybe` data type has a type-variable argument that may be instantiated with any algebraic data type. The data constructor `Just` represents the presence of a value, and `Nothing` represents its absence. `Maybe` is often used as a result of a function that may fail to produce a value, but the reason of failure is of no interest.

When the reason of failure, like in most cases, is important, another type is often used:

```
data Either a b = Left a | Right b
```

The `Either` data type has two type-variable arguments, and its `Left` data constructor is conventionally used to represent the failed result of a computation, while the `Right` data constructor is used to represent success. `Either` is used in Haskell for implementing functions with exceptions, with its first type parameter instantiated with a domain-specific exception data type.

Sum types roughly correspond to set-theoretic disjoint union. The types corresponding to set-theoretic Cartesian product are called product types and are considered in the next section.

2.2.3.3 Product types

In imperative programming languages, records, or structures, serve the purpose of "bunching" several types into one. In typed functional programming, this role is fulfilled by product types. For example, values could be paired together:

```
data Pair a b = Pair a b
```

The `Pair` type has a type constructor with two type-variable arguments, and a single data constructor with the same name which holds values of both types. In general, product types will always have a single data constructor with multiple arguments. Haskell has built-in syntactic support for tuples, a generalisation of pairs to more than two types, which correspond to the set-theoretic notion of Cartesian product, thus giving the name to product types in general.

Moreover, Haskell provides syntactic support for record types, a special case of product types which also have accessor-functions for the fields:

```
data Person = Person { name :: String
                     , age  :: Integer }
```

This syntax creates a product type `Person` with two fields of types `String` and `Integer`, and also two functions of types `Person -> String` and `Person -> Integer`, which provide access to the fields.

2.2.3.4 Recursive types

The conventional data structures such as linked lists and trees are expressed in Haskell with recursive algebraic data types. For example, consider a type for lists:

```
data List a = Nil | Cons a (List a)
```

A `List` is either an empty list, or a pair of a value of type `a`, the head, and a list, the tail. The Haskell `base` library defines a similar type which we have already used in

the examples above, with the conventional higher-order functions such as `map` and `foldr`. However, nothing stops us from implementing these functions for our own list data type:

```
mapList :: (a -> b) -> List a -> List b
mapList f Nil = Nil
mapList f (Cons x xs) = Cons (f x) (mapList f xs)
```

The `map` implementation in section 2.2.2, which closely mirrors the implementation in `base`, features the Haskell's special syntactic support for the built-in list type `[]`. Our own `mapList` function, however, must use the data constructors of the `List` data type. The built-in list data type also provide support for list comprehensions, another special language construction that provides a set-theoretic-like syntax for manipulating lists:

```
> [(i,j) | i <- [1,2],
           j <- [3,4] ]
[(1,3),(1,4),(2,3),(2,4)]
```

The code in the snippet uses a list comprehension to generate the Cartesian product of sets $\{1, 2\}$ and $\{3, 4\}$, represented as lists. It turns out, that lists are a special instance of a more general notion of a monad which we will introduce later in this chapter.

2.2.3.5 `newtype` — introducing a type isomorphic to an existing one

The Haskell programming language provides a construction that allows to define wrap an existing type into a zero-cost envelope. The language standard requires all implementations to guarantee that this wrapper type, called a **`newtype`**, will have the same runtime representation that the wrapped type. Newtypes are especially useful when the type class mechanism comes into play. In short, declaring a **`newtype`** allows to redefine type class instances which govern the behaviour of many standard Hasskell functions. We will discuss this techniques when covering Haskell type classes in section 2.2.4.

The Haskell programming language provides a much larger set of features related to algebraic data types that we have presented in this section. The further sections will touch over a number of more, but we still not aiming for an exhaustive presentation.

2.2.4 Type classes

Type classes enrich the Haskell programming language with ad-hoc polymorphism, providing a way to abstract over a set of operations. They are somewhat similar to interfaces from object-oriented programming, but have a number of important distinctions. Type classes are regarded as one of Haskell's most distinctive characteristic [72], and were consecutively added in a number of other functional programming languages.

As noted in History of Haskell [72], type classes have been designed to solve the problem of overloading of numeric operations and equality. In today's Haskell, type classes are ubiquitous, and solve not only this problem, but many more: from serialisation of data types into strings to abstracting over collection types, to providing a uniform interface for effectful computations.

We now introduce a number of standard type classes, discuss functional programming techniques that employ type classes and then outline important features of the type class mechanism as implemented in Glasgow Haskell Compiler (GHC).

2.2.4.1 The `Eq` class — equality

A `class` declaration in Haskell specifies the name of the class, with optional type variable arguments, and the type signatures for the classe's methods. Type classes can be instantiated at concrete types with an `instance` declaration. Consider the equality type class and its instance for the primitive `Int` type:

```
class  Eq a  where
  (==), (/=) :: a -> a -> Bool

instance Eq Int where
  i1 == i2 = eqInt i1 i2
  i1 /= i2 = not (i1 == i2)
```

Here, the `Eq` type class which abstracts equality has a single parameter `a` and two methods with the same type signature. The instance specialises the type variable `a` to the concrete type `Int` and defines implementations of the classe's methods by referring to the built-in function that implements equality for `Int`.

In GHC [73], the de-facto standard implementation of Haskell, many type classes, including `Eq`, can be derived automatically for user-defined algebraic data types, right at the

point of declaration of the type.

2.2.4.2 The `Ord` class — total order

Another ubiquitous type class is the `Ord` class, which abstracts over types with have a total order relation defined on them.

```
class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<), (<=), (>), (>=) :: a -> a -> Bool
  max, min :: a -> a -> a

data Ordering = LT | EQ | GT
```

The `Ord` type class inherits the methods of the `Eq` class, i.e. a totally ordered data type is required to have equality. The `compare` function demonstrates an important pattern of typed functional programming. While in a language like `C` a comparison function would return an integer and implicitly assign the sign of the integer to the result of the comparison, in Haskell it is customary to create custom data types with more explicit dedication. The `Ordering` data type's dedication is to serve as a return type of `compare`. This pattern is an instance of type-driven development, and is very useful to enforce program correctness through clear type-defined interfaces.

2.2.4.3 Defining custom type classes

Haskell allows users to define their own typeclasses and to declare types as their instances. Conceptually, there is no real difference between standard and user-defined type classes. Even though for standard type classes the GHC compiler provides build-in instance derivation mechanism, there are ways to use GHC's generic programming mechanisms for user-defined type-classes too [74][75].

For example, consider a type class that provides an interface for boolean algebra:

```
class Boolean a where
    true, false :: a
    false = not true

    not :: a -> a

    (|||), (&&&) :: a -> a -> a
```

```
    toBool :: a -> Bool
    fromBool :: Bool -> a
```

The `Boolean` type class is an instance of another pattern often found in Haskell programs, a shallowly-embedded (as opposed to deeply-embedded) syntax, in this case of boolean algebra. This embedding is called shallow because it reuses constructions of the metalanguage (here, Haskell)[1]. Specifically, this embedding does not have its own binders, and reuses Haskell's `let`-bindings and variables.

A natural instance of the `Boolean` type class is the `Bool` type:

```
instance Boolean Bool where
    true = True
    not = Prelude.not
    toBool = id
    fromBool = id

    x ||| y = x || y
    x &&& y = x && y
```

Here we reuse the standard functions defined in GHC's `base` package. Note that we do not have to define `false` explicitly since we have declared a default implementation for it as `not true`. The functions that defined the isomorphism to `Bool` are, naturally, identities.

A much more interesting instance of the `Boolean` type class will be declared in chapter 5 when discussing symbolic execution. It turns out to be useful to treat concrete boolean values and symbolic boolean expressions uniformly, and the `Boolean` type class gives a way to do that.

2.2.5 Programs with side-effects

In this section we will overview the Haskell's approach and machinery for writing *impure* functions, that is, functions that have side effects.

2.2.5.1 Revisiting `IO`

The Haskell programming language's type system distinguishes between pure and effectful computations. Originally, the "effectful" term referred specifically to input/output, i.e.

[1] Embedding of syntax is one of the cornerstones of programming languages research. See, for example [76] for more information and references.

interaction with the operating system to read and modify files. The initial Haskell programming language design, as presented in the Haskell 1.0 Report, presented two mechanisms for I/O, based on streams and continuations. Even though both fit well the purity requirement, they had their own disadvantages which spurred the research into better approaches to integrating I/O into a pure functional language. Motivated by Moggi's research on programming language semantics [58], which suggested using the category theoretic notion of monad to give denotational semantics to effectful computations, Wadler has introduced monads into Haskell to handle I/O and other effects [59]. The History of Haskell paper [72] describes this historical design decision: "Wadler used monads to *express* the same programming language features that Moggi used monads to *describe*". Using monads to structure functional programs with I/O, and side-effects in general, has proved to be a very expressive and convenient approach.

Since the days when monads were introduced into Haskell, many new ways to structure computations with effects have been developed. Today's Haskell has a hierarchy of interfaces for effectful computations with monads situated on top as the most restrictive one. We proceed by introducing several of these interfaces, formulated as Haskell type classes, starting from the most basic one, and building up the hierarchy up to selective functors and monads.

2.2.5.2 The `Functor` Class — independent effects

One way to organise Haskell's effectful computation type class hierarchy is by considering the dependencies between the computation's effects. Informally, two effects are considered independent if one cannot control execution of the other, i.e. cannot stop, force or in any other way affect its execution.

The base of the hierarchy lies at the **`Functor`** type class, which represents computations that can be "mapped over", i.e. have an action performed over their value without altering the structure of the computation.

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

The best way to understand an abstraction is to develop an intuition through looking at concrete examples of types that can be given an instance of **`Functor`**. For example, the

`List` algebraic data type, which we introduced in section 2.2.3.4, is a functor:

```
instance Functor List where
    fmap = mapList
```

Indeed, the function `mapList` "maps over" a list, applying its argument function to every element of the list. The intuition of functors being computations with an effect applies to the list functor too: list represent non-deterministic computations with multiple possible results, and the `fmap` function applies its argument to every result.

As we noted in section 2.2.4, it is conventional to formulate *laws* that the instances of a specific type class must obey. In Haskell, that is indeed a convention, since the type system is not powerful enough to check, enforce or even encode their validity for instances. However, it is considered a good practice to check laws either by hand, or using property-based testing frameworks such as QuickCheck [21]. Other, more powerful type systems, such as the ones of proof assistants like Coq and Agda, can be used to formalise type classes together with their laws and enable formal proof of compliance for instances.

The laws of the `Functor` type class are motivated by its origin in category theory. Categorically, a functor is a structure-preserving transformation between two categories. Functors are only allowed to change the objects of the category, but must preserve the structure of the morphisms: identities and composition. These laws can be written down as the following two equations:

$$fmap\ id = id \tag{2.1}$$

$$\forall\ (a\ b : Type)(g : a \to b)(f : b \to c).\ fmap\ (f \circ g) = fmap\ f \circ fmap\ g \tag{2.2}$$

Here id is the identity morphism which, in Haskell, is represented by the function `id :: a -> a`. When the `fmap` function is given the identity function as an argument, it will map it into the identity function too. The second law says that composing functions in the source category and then applying the functor results in the target category composition of transformed functions.

When talking about functors in Haskell, we usually consider *endofunctors* — functors that have the same source and target category, Hask, the category of Haskell types and

total Haskell functions. For the purposes of presentation, we forget that every Haskell type implicitly contains the value $\bot$, which prevents the collection of all Haskell types form being a category.

Let us now prove the `List` type constructor is an endofunctor. We will have to apply the axiom of functional extensionality to transform the equality of functions into pointwise equality.

Proposition 1. *The* `List` *type constructor preserves the identity morphism, i.e. the equation 2.1 holds.*

Proof. We prove the proposition by induction on the structure of the `List` algebraic data type. We first consider the base case of the `Nil` data constructor:

$$
\begin{aligned}
& fmap\ id\ Nil \\
= \quad & \{\ \texttt{Functor}\ \text{instance for}\ \texttt{List}\ \} \\
& mapList\ id\ Nil \\
= \quad & \{\ \text{definition of}\ \texttt{mapList}\ \} \\
& \text{id Nil} \\
= \quad & \{\ \text{definition of}\ \texttt{id}\ \} \\
& \text{Nil}
\end{aligned}
$$

We now consider the inductive case of the `Cons` data constructor:

$$
\begin{aligned}
& fmap\ id\ (Cons\ y\ ys) \\
= & \quad \{\ \texttt{Functor}\ \text{instance for}\ \texttt{List}\ \} \\
& mapList\ id\ (Cons\ y\ ys)) \\
= & \quad \{\ \text{definition of}\ \texttt{mapList}\ \} \\
& \text{Cons (id y) (mapList id ys))} \\
= & \quad \{\ \text{definition of}\ \texttt{id}\ \} \\
& \text{Cons y (mapList id ys))} \\
= & \quad \{\ \text{inductive hypothesis}\ \} \\
& \text{Cons y ys}
\end{aligned}
$$

□

Similarly, the proposition about the `List` type constructor preserving composition (equation 2.2) can be formulated and proved.

Most Haskell data types can be given a `Functor` instance, including the ones we have discussed earlier in this chapter. However, the functor interface is very limited: it only provides a way to transform the resulting value of an effectful computation.

2.2.5.3 The `Applicative` class — statically defined effects

Applicative functors extend the functor class interface with additional methods and laws. Their purpose is to provide a mechanism to *lift* a pure function of multiple arguments to an effectful computational context and *apply* it, hence the name. Applicative functors have been discovered by McBride and Paterson [77], and since then have become an important abstraction for effectful computations in Haskell and other languages.

```
class Functor f => Applicative f where
  -- | Lift a value.
  pure :: a -> f a
  -- | Lift a binary function to actions.
  liftA2 :: (a -> b -> c) -> f a -> f b -> f c
```

```
  -- | Sequential application.
  (<*>) :: f (a -> b) -> f a -> f b
```

Let us consider the `Applicative` instance for the Haskell's datatype of linked-lists:

```
instance Applicative [] where
    pure x         = [x]
    liftA2 f xs ys = [f x y | x <- xs, y <- ys]
    fs <*> xs      = [f x   | f <- fs, x <- xs]
```

We see that the `pure` function is just a singleton list — it embeds a value into the *list* computational context. The `liftA2` function *lifts* a binary function onto lists. Using a list comprehension it picks non-deterministically a value from each list and applies the function to them, effectively generating a list of all possible combinations. One can implement the *Cartesian product* of two lists via `liftA2` by lifting the pair constructor onto lists:

```
> liftA2 (\x y -> (x,y)) [1,2] [3,4]
[(1,3),(1,4),(2,3),(2,4)]
```

The *apply* operator `<*>` can be considered a generalisation of `liftA2`. It takes two effectful computations, which — independently — compute values of types `a -> b` and `a`, and returns their composition that performs both computations, and then applies the obtained function to the obtained value producing the result of type `b`. Crucially, both arguments and associated effects are known statically, which, for example, allows us to pre-allocate all necessary computation resources upfront and execute all computations in parallel. This operator is often used to lift n-ary functions.

Applicative functors provide an interface for composing independent effectful computations. However, the effects can only be executed together — without any dynamic conditions. One may wonder, if it is possible to retain the benefits of applicatives, while also gaining the possibility to *skip* some effects depending on runtime conditions.

2.2.5.4 `Selective` class — statically defined, dynamically dispatched effects

The research on ISA semantics summarised in this thesis has inspired a very interesting abstraction, which we call *Selective applicative functors* [16], or just *selective functors*, for brevity. The basic idea of selective functors is enriching the `Applicative` interface with an

operator that allows *conditional* execution of effects. We say that selective functors allow *declaring* effects statically, and *choosing* which to execute dynamically. In this section, we give a brief introduction to selective functors. For a more in-depth presentation, the interested reader may consult our paper [16].

Like applicative functors [77], selective functors provide a way to embed pure values into an effectful context `f` using the function `pure`, and give meaning to composition of two *independent* effectful computations using the operator `<*>`. Selective functors enrich the applicative interface with the `select` function, which gives meaning to the composition of two effectful computations, where, in contrast to `<*>`, the second computation *depends* on the first one:

```
class Applicative f => Selective f where
    select :: f (Either a b) -> f (a -> b) -> f b
```

One can think of `select` as a selective function application: parametricity [78] dictates that, when given a `Left a`, we *must execute the effects* in `f (a -> b)`, apply the obtained function to `a`, and return the resulting `b`; on the other hand, when given a `Right b`, we *may skip the effects* associated with the function, and return the given `b`[2].

Following the notational convention for applicative operators, we also define the infix operator alias `<*?` for `select`: the angle bracket pointing to the left means we always use the corresponding value; the value on the right, however, may be skipped, hence the question mark.

One can implement `select` using monads in a straightforward manner: examine the value produced by `f (Either a b)` with the bind operator, and then, in the `Left a` case, execute the subsequent effect `f (a -> b)`, passing the `a` to it using the `Functor`'s map operator, as shown below.

```
selectM :: Monad f => f (Either a b) -> f (a -> b) -> f b
selectM x y = x >>= \e -> case e of Left  a -> ($a) <$> y -- Execute y
                                    Right b -> pure b     -- Skip y
```

One can also implement a function with the type signature of `select` using applicative functors, but it will always execute the effects associated with the second argument, rendering any conditional execution of effects impossible:

[2]Note, however, that if `f a` holds no values of type `a`, i.e. `a` is a *phantom type variable* [79], then the effects in `f (a -> b)` can be skipped unconditionally.

```
selectA :: Applicative f => f (Either a b) -> f (a -> b) -> f b
selectA x y = (\e f -> either f id e) <$> x <*> y -- Execute x and y
```

While `selectM` is useful for *conditional execution* of effects, `selectA` is useful for *static analysis*. As we will see in §2.2.5.4.2, selective functors used for static analysis need to collect information about all possible effects instead of skipping some of them, hence they directly use `select = selectA` in their `Selective` instance definitions.

Any `Applicative` instance can thus be given a `Selective` instance. The opposite is also true in the sense that one can recover the operator `<*>` from `select` as follows:

```
apS :: Selective f => f (a -> b) -> f a -> f b
apS f x = select (Left <$> f) (flip ($) <$> x)
```

Here we tag a given function `a -> b` with `Left` and turn a value of type `a` into the reverse application function `\a f -> f a`, which yields `b` when given `a -> b`, as desired. Since the `Right` case is impossible, the effect `f a` is executed unconditionally. Note however, that the equality `(<*>) = apS` does not always hold. Selective functors that satisfy the law `(<*>) = apS` will be called *rigid*.

It is worth emphasising that the subclass relationships `Applicative f => Selective f` and `Applicative f => Monad f` are different. *Some applicative functors are not monads*, e.g. the `Const` functor, but *every applicative functor is also a selective functor*, as witnessed by the function `selectA`. The subclass relationship `Applicative f => Selective f` is justified only by the extra method `select` in `Selective`. While `select = selectA` is a valid implementation of `select`, *it is not the only useful implementation*. The applicative-selective-monad hierarchy therefore reflects method set inclusion: {`<*>`} ⊂ {`<*>`, `select`} ⊂ {`<*>`, `select`, `>>=`}.

Table 2.3 compares the three methods in terms their expressive power.

2.2.5.4.1 Selective combinators In the fine-grained semantics of the REDFIN ISA, which will be described in chapter 5, we actively use selective functors, but, in fact, we never use the `select` function directly. Instead, we find that the semantics of some instructions, such as conditional jumps, can be naturally expressed with combinators derived from `select`.

Notions that can be expressed using an operator	apply (<*>)	select (<*?)	bind (>>=)
Independent effects and parallelism	✓		
Static visibility and analysis of effects	✓	✓	
Speculative execution of effects		✓	
Conditional execution of effects		✓	✓
Arbitrary dynamic effects			✓

Table 2.3: Comparison of apply, select and bind operators in terms of their expressive power. Note that each operator has one unique ability that the two others lack.

Consider the implementation of `whenS`, which executes an effectful computation (for example, updates the instruction counter) depending on a boolean value:

```
whenS :: Selective f => f Bool -> f () -> f ()
whenS x y = selector <*? effect
  where
    selector = bool (Right ()) (Left ()) <$> x -- NB: convert True to Left ()
    effect   = const                     <$> y
```

We first bring the given effectful computations into the right shape by using the `Functor`'s map operator. Specifically, `x :: f Bool` is converted into the `selector :: f (Either () ())`, and `y :: f ()` is converted into the `effect :: f (() -> ())`. The results are composed using the select operator `<*?`, and the meaning of this composition is determined by the supplied `Selective f` instance. For example, an instance like `f = IO` would skip `y` if `x` yields `False`.

It is worth noting that unlike the select operator, whose implementation is almost completely determined by parametricity (i.e., the only real question is: *"To skip, or not to skip?"*), `whenS` admits a variety of (incorrect) implementations. In particular, due to *Boolean blindness*[3], it is easy to inadvertently implement `unlessS`, which has the same type but flips the meaning of the Boolean value. The ability to reason parametrically was one of the guiding principles we used when looking for a good abstraction for selective functors: `select` provides this ability, whereas `whenS` does not.

A strong contender for playing the leading role in selective functors is the function `branch` that, given an effectful computation `x :: f (Either a b)`, selects which subsequent computation, namely `l :: f (a -> c)` or `r :: f (b -> c)`, to execute:

```
branch :: Selective f => f (Either a b) -> f (a -> c) -> f (b -> c) -> f c
branch x l r = fmap (fmap Left) x <*? fmap (fmap Right) l <*? r
```

[3]The term refers to the fact that the `True` and `False` values are not distinguished at the type level, see [80].

The select operator allows to eliminate one of the cases in a sum type, namely the `Left a` case in `Either a b`, leaving the other case intact. To implement `branch`, we will need to apply `<*?` twice, eliminating `a` and `b` one after another. The first application is tricky because `f (Either a b)` and `f (a -> c)` do not match the type signature of `<*?`. To fix the mismatch, we convert them to `f (Either a (Either b c))` and `f (a -> Either b c)`, respectively. The second application of `<*?` is then straightforward.

By instantiating `select` with `a = b = ()` we have earlier obtained `whenS`. Below we repeat the same trick but with `branch`, obtaining another familiar conditional combinator `ifS`:

```
ifS :: Selective f => f Bool -> f a -> f a -> f a
ifS x t e = branch selector (const <$> t) (const <$> e)
  where
    selector = bool (Right ()) (Left ()) <$> x -- NB: convert True to Left ()
```

Many conditional combinators, which are typically associated with the `Monad` type class, can be expressed using selective functors. To emphasise the monadic flavour of selective functors, we can use `ifS` to implement the bind operator specialised to `Bool`:

```
bindBool :: Selective f => f Bool -> (Bool -> f a) -> f a
bindBool x f = ifS x (f False) (f True)
```

2.2.5.4.2 Examples of selective functors Having explored various useful combinators that can be implemented on top of the minimalistic selective interface, in this section we look at several examples of selective functors.

The `Const m a` functor is an interesting instance of the `Applicative` type class, which stores no values of type `a`, but keeps track of the applicative structure in the monoid value of type `m`:

```
newtype Const m a = Const { getConst :: m }
```

```
instance Functor (Const m) where
    fmap _ (Const x) = Const x
```

```
-- 'mempty' and '<>' are the identity and the binary operation of the Monoid m
instance Monoid m => Applicative (Const m) where
    pure _              = Const mempty   -- Pure values have no effects
    Const x <*> Const y = Const (x <> y) -- Collect effects in x and y
```

It turns out there are two useful selective instances for `Const`. To disambiguate between them, we will call them `Over` and `Under`, reusing[4] the above `Functor` and `Applicative` instances:

```
newtype Over  m a = Over  { getOver  :: m }
newtype Under m a = Under { getUnder :: m }

instance Monoid m => Selective (Over m) where
    select (Over x) (Over y) = Over (x <> y) -- Collect effects in x and y
```

```
instance Monoid m => Selective (Under m) where
    select (Under x) _ = Under x             -- Discard conditional effects
```

The selective functor `Over` can be used for computing a list of all possible effects embedded in a computation, i.e. an *over-approximation* of the effects that will actually occur. This is achieved by keeping track of effects in both arguments of `select`. The selective functor `Under`, on the other hand, discards the second argument of `select`, and therefore computes an *under-approximation*, i.e. a list of effects that are guaranteed to occur. Let us give these two instances a try:

```
λ> ifS (Over "a") (Over "b") (Over "c") *> Over "d" *> whenS (Over "e") (Over "f")
Over "abcdef"
```

```
λ> ifS (Under "a") (Under "b") (Under "c") *> Under "d" *> whenS (Under "e") (Under "f")
Under "ade"
```

As expected, `Over` collects all effects, whereas `Under` does not look beyond "opaque" conditions. A deeper difference between them is that `Over` is a rigid selective functor, i.e. `(<*>) = apS`, but `Under` is not: indeed, `Under "a" <*> Under "b"` records both `"a"` and `"b"`, but `apS (Under "a") (Under "b")` records just `"a"` since `apS` is implemented via `select`. Intuitively, non-rigid selective functors have a much richer structure, because `<*>` is not expressible via `select`.

At that point, we conclude the presentation of selective functors. We refer the interesting reader to our paper [16], which provides more details on selective functors themselves and on their applications beyond ISA semantics.

[4] Fortunately, thanks to the new GHC extension DerivingVia [81], we can reuse `Const` instances without duplicating any code, simply by adding `deriving (Functor, Applicative) via (Const m)` to the `newtype` definitions.

2.2.5.5 The `Monad` class — fully dynamic effects

Monads, introduced to functional programming by Wadler [82], are a powerful and general approach for describing effectful (or impure) computations using pure functions. The key ingredient of the monad abstraction is the *bind* operator, denoted by `>>=` in Haskell:

```
(>>=) :: Monad f => f a -> (a -> f b) -> f b
```

The operator takes two arguments: an effectful computation `f a`, which yields a value of type `a` when executed, and a recipe, i.e. a pure function of type `a -> f b`, for turning `a` into a subsequent computation of type `f b`. This approach to composing effectful computations is inherently sequential: until we execute the effects in `f a`, there is no way of obtaining the computation `f b`, i.e. these computations can only be performed in sequence.

The monadic interface can be naturally used to describe ISA semantics. In chapter 4, we will develop a coarse-grained semantics of the REDFIN ISA based on monadic state-transformers. As we will see, the semantics will syntactically look very natural, mirroring the pseudocode samples usually found in ISA manuals, while also being executable. However, as we will see, the semantics will have a number of shortcomings, which we will mitigate by developing a novel approach, based on selective functors, in chapter 5.

Chapter 3

Instruction-set architecture semantics

In section 2.1.3.1, we have outlined the major approaches for describing the semantics of programming languages. These approaches constitute the *theory* of programming languages. In this thesis, however, we are more concerned with *practice*: how can these theoretical approaches be put to use by software and computer engineers to make programs, programming languages and computer architectures better?

The theoretical approaches to semantics are employed in practice to structure the implementations of interpreters and compilers. There is a wide variety of techniques of programming language implementation, ranging in abstraction from high-level *semantic frameworks* such as Ott [83], Lem [84], $\mathbb{K}$ [85] and others, to concrete implementations using another programming language as a *metalanguage*. The latter group has an important subspace of EDSLs — domain-specific languages embedded into the metalanguage and thus reusing its syntax and semantics. In this thesis, we present approaches for describing semantics of instruction-set architectures as EDSLs embedded in Haskell, and taking advantage of Haskell's powerful type system. As a case-study, we apply the presented approaches to REDFIN.

3.1 Instruction syntax

In this section, we consider the data types and techniques used to represent REDFIN's instructions. The section 1.3 describes the motivation for the design of REDFIN, and outlines the classes of instructions that the ISA provides. In this thesis, and the associated verification frameworks, we do not consider all REDFIN instructions. We leave out the bus access instructions and fixed-point arithmetic. The former lies outside of the scope of this thesis[1], and the latter can be handled similarly to their integer analogues, and are not required for the verification case-studies.

The representation, and even the very concept, of the instruction syntax is one of the major influences of the programming languages techniques in this thesis. If this thesis was written from a purely computer engineering perspective, we would not have such a section, but rather just a diagram of the binary instruction codes. However, since we are at the edge of programming languages and computer engineering, we will have both the machine codes diagram and a generalised algebraic data type of instructions' abstract syntax. We first take a look at the machine codes.

3.1.1 Concrete syntax: instruction codes

	15					10	9 8	7 0
TYPE A	0	0	0	0	0	0	-	
TYPE B	0	X	X	X	X	X	REG2	MEM8
TYPE C	1	0	0	X	X	X	REG2	SIMM8
TYPE D	1	0	1	0	X	X	REG2	SIMM8
TYPE E	1	0	1	1	X	X	REG2	UIMM8
TYPE F	1	1	0	X	X	X	UIMM10/SIMM10	
TYPE G	1	1	1	0	X	X	REG2	-
TYPE H	1	1	1	1	1	1	UIMM5	UIMM5

Table 3.1: REDFIN ISA machine code formats

The table 3.1 is an excerpt from REDFIN ISA manual, and contains the types of instruction codes REDFIN supports. Every code is a 16-bit word, where the 6 most significant bits correspond to the opcode, and the rest encodes the instruction's arguments. The symbols "X" in the table can take either 0 or 1. "REG2" is a 2-bit register code, "MEM8" is a data memory address, and the "*IMM*" mnemonics are immediate arguments of various widths.

[1] Modelling the system bus is a promising avenue of future work, see section 7.3.1.

After this table, the REDFIN ISA manual enumerates every instruction code and describes its semantics in prose and pseudo code. However, such bare-bones representation of instruction syntax is not convenient for the purposes of building a verification framework. We therefore present a representation of instruction syntax augmented with additional information about the structure of instructions' data dependencies.

3.1.2 Abstract syntax

An important categorisation criterion that constitutes the corner stone of our semantics of the REDFIN ISA is the structure of the instructions' data dependencies. Most instructions have static dataflow: they read and write data locations, and do so consistently, independently of the data contents of these locations. Other instructions have their dataflow defined by the contents of the locations they read: based on it they may write into different locations. For example, the equality comparison instruction will set the `Condition` flag if the values in its argument locations are equal. In their turn, the conditional jump instructions will change the instruction counter based in the value of the `Condition` flag. This dichotomy of static versus dynamic dataflow can be captured by assigning the instructions with the correct dataflow type class. Instruction with static data flow belong to either `Functor` or `Applicative`, and the dynamic dataflow can we expressed with `Selective` or `Monad`. We will discuss the dataflow classes in more detail further.

We represent the instructions as a generalised algebraic data type (figure 3.1), with individual instructions being its constructors. This type includes a type-level value which represents the class of the dataflow effects of instructions. We will talk about the classes in more detail when formulating the semantics of instructions in the next section.

```
data InstructionImpl
    (control :: (Type -> Type) -> Constraint) (value :: Type -> Constraint) a
    where
    Load   ::            Register -> CAddress -> InstructionImpl Functor     value a
    Set    :: value a => Register -> Imm a    -> InstructionImpl Functor     value a
    Store  ::            Register -> CAddress -> InstructionImpl Functor     value a
    Halt   ::                                    InstructionImpl Applicative value a
    Add    ::            Register -> CAddress -> InstructionImpl Applicative value a
    AddI   :: value a => Register -> Imm a    -> InstructionImpl Applicative value a
    Sub    ::            Register -> CAddress -> InstructionImpl Applicative value a
    SubI   :: value a => Register -> Imm a    -> InstructionImpl Applicative value a
    Mul    ::            Register -> CAddress -> InstructionImpl Applicative value a
    Div    ::            Register -> CAddress -> InstructionImpl Applicative value a
    Mod    ::            Register -> CAddress -> InstructionImpl Applicative value a
    Abs    ::            Register             -> InstructionImpl Applicative value a
    Jump   :: value a => Imm a                 -> InstructionImpl Applicative value a
    CmpEq  ::            Register -> CAddress -> InstructionImpl Selective   value a
    CmpGt  ::            Register -> CAddress -> InstructionImpl Selective   value a
    CmpLt  ::            Register -> CAddress -> InstructionImpl Selective   value a
    JumpCt :: value a => Imm a                 -> InstructionImpl Selective   value a
    JumpCf :: value a => Imm a                 -> InstructionImpl Selective   value a
    LoadMI ::            Register -> CAddress -> InstructionImpl Monad       value a

data Instruction a = forall c v. v a => Instruction (InstructionImpl c v a)
```

Figure 3.1: Instruction syntax

3.2 ISA state

To create an executable model of REDFIN, we should formulate the components of the instruction-set architecture as a collection of algebraic data types. These data types will represent the possible states of the ISA, and the semantics of the instruction will describe transitions between the states.

3.2.1 Data types

The REDFIN ISA has six types of data locations, represented by the six data constructors of the `Key` data type (figure 3.2):

- `Reg` is a general-purpose registers of REDFIN;
- `Addr` represents a potentially symbolic memory location;
- `F` refers to a field of the flag register;

- `IC` is the instruction counter — during program execution it contains the address of the next instruction to be fetched from program memory;
- `IR` is the instruction register — during program execution it contains the binary instruction code of the fetched instruction;
- `Prog` is an address in program memory.

The constructors of the `Key` data type are essentially just tags, which enclose the values of the data types on the right-hand-side of the figure 3.2. The `Key` data type will be essential for the construction of fine-grained store abstraction which we will use to describe the semantics of instructions.

Note that the REDFIN ISA has separate memory regions for data and instructions. We, however, model them similarly and reuse the data type of addresses, even though the program addresses will never be accessed symbolically.

```
data Key where
    -- | data register
    Reg :: Register -> Key
    -- | memory cell
    Addr :: Address -> Key
    -- | flag, a special boolean register
    F :: Flag -> Key
    -- | instruction counter
    IC :: Key
    -- | instruction register
    IR :: Key
    -- | program address
    Prog :: Address -> Key
```

```
-- | Registers
data Register = R0 | R1 | R2 | R3

-- | Concrete memory address
newtype CAddress = CAddress Word8

-- | Immediate argument
newtype Imm a = Imm a

-- | Flags
data Flag
    = Halted
    | Condition
    | Overflow
    | DivisionByZero

-- | Binary instruction code
newtype InstructionCode =
    InstructionCode Word16
```

Figure 3.2: REDFIN ISA keys

The data type of keys in figure 3.2 account for potentially symbolic memory addresses, but we have not yet introduced how they are represented. Symbolic values will become relevant in the next chapter, which will describe verification of REDFIN programs by symbolic execution.

3.3 Instruction semantics

Informally, the semantics of an instruction is a sequence of steps that change the state of the ISA. Every instruction will alter relevant parts of the state: registers, memory or flags. Before executing an instruction, the processor will perform a fetch-decode-execute process, which includes fetching an instruction code from program memory into the instruction register, decoding the instruction into a value of the abstract syntax data type, and then executing its semantics and advancing the instruction counter. This means that we do not include the fetch-decode-execute process into the semantics of every instruction, but rather keep it separate. The reason to do that is that the dataflow of the fetch-decode-execute process is inherently *monadic*, i.e. the values of instruction register and instruction counter dynamically depend on program memory. Besides the fetch-decode-execute process, the only monadic operation is the semantics of memory-indirect load instruction. The semantics of all other instructions can be formulated in terms of more lax dataflow classes, thus enabling us to perform static analysis of non-monadic program fragments. We will discuss that in more detail in the next chapter.

3.3.1 Coarse-grained operation semantics

We will now define a small-step operational semantics of REDFIN ISA which we further refine with a Haskell implementation. The semantics is a transition system with its states being the states of REDFIN ISA and the transitions being instructions.

Definition (ISA state):

The set of states of the REDFIN instruction-set architecture is the Cartesian product of the states of its seven components:

We can now define the formal semantics of REDFIN instructions and programs as a *state transformer* $T : S \to S$, i.e. a function that maps states to states. We distinguish instructions and programs by using Haskell's list notation, e.g. $T_{\texttt{nop}}$ is the semantics of the instruction `nop` $\in I$, whereas $T_{\texttt{[nop]}}$ is the semantics of the single-instruction program `[nop]` $\in P$. [2]

[2] REDFIN does not have a dedicated `nop` instruction, but it can be expressed as a jump to the next instruction, i.e. `jmpi 0`.

Definition (program semantics): The semantics of a program $p \in P$ is inductively defined as follows:

The semantics of the *empty program* `[]` $\in P$ coincides with the semantics of the instruction `nop` and is the identity state transformer: $T_{\texttt{[]}} = T_{\texttt{nop}} = \texttt{id}$.

The semantics of a *single-instruction program* `[i]` $\in P$ is a composition of (i) fetching the instruction from the program memory T_{fetch}, (ii) incrementing the instruction counter T_{inc}, and (iii) the state transformer of the instruction itself $T_{\texttt{i}}$:

$$T_{fetch} \quad = \quad , f, c{+}1)T_{inc} = (r, m, ic, ir, p, f, c) \mapsto (r, m, ic{+}1, ir, p, f, c)T_{\texttt{[i]}} = T_{\texttt{i}} \circ T_{inc} \circ T_{fetch}$$

The semantics of a *composite program* `i:p` $\in P$, where the operator `:` prepends an instruction `i` $\in I$ to a program `p` $\in P$, is defined as $T_{\texttt{i:p}} = T_{\texttt{p}} \circ T_{\texttt{[i]}}$.

As an example, let us consider the semantics of some instructions as implemented in the coarse-grained semantics of REDFIN:

```
load :: Register -> MemoryAddress -> Redfin ()
load rX dmemaddr = writeRegister rX =<< readMemory dmemaddr
```

The `load` instruction loads a value from a memory location to the specified register. The `(=<<) :: Monad f => (a -> m b) -> m a -> m b` function is the monadic bind with its arguments swapped. Here we can see why we call this approach *coarse-grained*. The use of the monadic bind implicitly introduces the data dependency on the whole memory space, even though we only need a single address.

The next function implements the semantics of an unconditional jump and is formulated as an explicit state transformer:

```
jmpi :: SImm10 -> Redfin ()
jmpi simm = transformState $
    \ (State rs  ic                    ir fs m p c)
    -> State rs (ic + fromSImm10 simm) ir fs m p c
```

To perform a jump means to advance the instruction counter `ic`. Even though we only need to alter this particular item in the state, we still need to depend on the state as a

whole. Even though we see the subject of the alteration in the source code, we would not be able to determine it by automated static ananlysis

A conditional jump instruction semantics reuses the unconditional one, but predicates the state update on the value of the `Condition` flag:

```
jmpi_ct :: SImm10 -> Redfin ()
jmpi_ct simm = do
    condition <- readFlag Condition
    state <- readState
    let jumpState = snd $ transform (jmpi simm) state
    writeState $ ite condition jumpState state
```

In this semantics, we explicitly *merge* the two possible future states via the symbolic conditional `ite`: the "jumped" with the instruction counter altered, and the "fall-though".

The resulting program semantics is a *sequence* of state updates comprising the resulting state transformer. The presented formal semantics is *coarse-grained* since the state transformers are inherently monadic and operate on the whole ISA state. While this approach has the benefit of being very intuitive, it could be refined by specifying more precise state alterations. This leads to a semantics based on *fine-grained state transformers*. This semantics will be dataflow-aware, and can be though of as a *tree*, rather than as a sequence.

We give an extended account of the coarse-grained semantics and the verification framework based on it in the chapter 4.

3.3.2 Fine-grained dataflow-aware semantics

The semantics based on coarse-grained state transformers presented in the previous section has a number of limitations in terms of what kind of programs can be verified and how. We do not discuss these limitation in detail here, and refer the reader to the following chapters 4 and 5. We, however, present a different approach, based on the novel formalism of fine-grained state transformer.

In the fine-grained semantics, the state transformers do not operate on the ISA state as a whole, but rather impose constraints on the state to contain specific *keys*, with every key being a constructor of the `Key` data type (fig. 3.2).

We will have an extensive discussion of fine-grained state transformers, and the verification framework for REDFIN that employs them, in chapter 5. For now, we limit ourselves to

a concise demonstration of the approach by presenting the semantics of several instruction. The dataflow-awareness that is brought by the `InstructionImpl` datatype of instruction's abstract syntax allows us to derive several distinct useful interpretations of the abstract semantics of instructions.

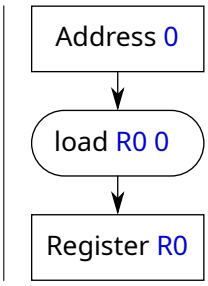
3.3.2.1 Linear dataflow

The instructions marked with `Functor` have linear dataflow, meaning that the instruction has no ability to replicate inputs: if one inputs flows into the instruction, there will be exactly one flowing out. For example, the `Load` instruction fetches the contents of a memory cell and writes it into a register:

```
Load :: Register -> CAddress -> InstructionImpl Functor value a
```

```
load :: Register -> Address -> FS Key Functor '[] a
load reg addr read write = write (Reg reg) (read (Addr addr))
```

```
load R0 0
```

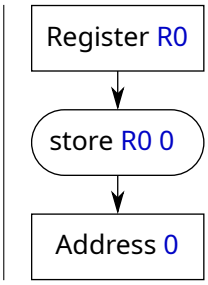


The `Store` instruction does the reverse, i.e. stores the contents of a register into a memory cell:

```
Store :: Register -> CAddress -> InstructionImpl Functor value a
```

```
store :: Register -> Address -> FS Key Functor '[] a
store reg addr read write = write (Addr addr) (read (Reg reg))
```

```
store R0 0
```



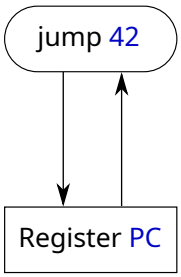
The semantics of `Load` and `Store` constitute a degenerate form of the linear fine-grained state transformer. One may notice, that they, in fact, do not need the interface of the `Functor` type class at all. We, however, assign them with it for simplicity of treatment.

A non-degenerate linear fine-grained state transformer implements the semantics of an unconditional jump, the `Jump` instruction:

```
Jump :: value a => Imm a -> InstructionImpl Functor value a
```

```
jump :: Imm a -> FS Key Functor '[Num] a
jump (Imm offset) read write = write IC (fmap (+ offset) (read IC))
```

```
jump 42
```



The `Jump` instruction fetches the current instruction counter using the `read` base transformer, transforms the fetched value by using the `fmap` method of the `Functor` type class and writes the result with the offset applied back to the instruction counter. There are several notable features of this computations that we need to break down:

1. Note that the first argument of the `jump` function of type `Imm a`, the immediate jump offset, is not displayed in the dataflow diagram on the right. It is omitted because it is a *pure* value, and is not a part of the ISA state, therefore can not directly effect the overall semantics. However, we still see it as the immediate of the instruction in the dataflow diagram.

2. The instruction syntax, the `Jump` constructor of the `InstructionImpl` data type, carries the `value` constraint on the existential type variable `a` of the result. In the semantics, the `jump` function, we instantiate `value` to be the singleton list `'[Num]`. This allows us to use arithmetic in the semantics, specifically the `(+)` function to offset the instruction counter. This indicate the second difference from the semantics of `Load` and `Store`, which were parametric in the type variable `a`.

3. Again, the semantics of `Jump` uses the interface provided by the `Functor` type class and thus does indeed require this constraint.

The unconditional jump is the most complicated semantics we can express with the very limited capabilities of a linear fine-grained state transformer. To express more intricate semantics, like that of arithmetical instructions, we need additional power to be able to read and write values of several data locations.

3.3.2.2 Static Tree Dataflow

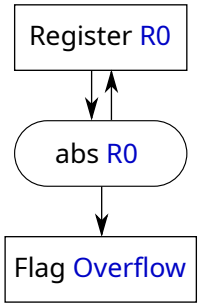
The linear semantics is only allowed to have one input and one output data dependency. The next step is to allows several *static* input and output dependencies by given the state transformer an access to the `Applicative` type class. Consider the semantics of the absolute value, the `Abs` instruction:

```
Abs :: Register -> InstructionImpl Applicative value a
```

```
abs :: Register
    -> FS Key Applicative '[Monoid, Num, Bounded, Boolean, BEq] a
abs reg read write =
    let arg = read (Reg reg)
        o = absOverflows <$> arg
        result = (Prelude.abs) <$> arg
    in write (F Overflow) o *> write (Reg reg) result

absOverflows :: (Bounded a, Boolean a, BEq a) => a -> a
absOverflows arg = arg === minBound
```

```
abs R0
```



The semantics of `Abs` reads the value of a register and computes two things based off it: (1) the absolute value of the argument and (2) the overflow condition. The computation of absolute value on two's complement integers will overflow if called on the minimal value that could be represented at the given bit width. The semantics employs the `Bounded` class to be aware of the `minBound` and the `BEq` class to compare values for equality. As for the dataflow, the semantics uses the `Applicative`'s sequencing operator `(*>)` to perform two write operations: to the `Overflow` flag and to the target register.

Note that the linear semantics of `Load`, `Store` and `Jump` may need to be promoted to the static tree one if memory safety considerations are brought into scope. For example, the instruction semantics can raise an `OutOfMemory` flag if the address requested lies outside of the valid address space. However, in this thesis we do not implement memory safety checks in the semantics. In order to keep the semantics itself as simple as possible, it is wise to offload some of the checks to the implementation backend. We do just that, and validate data and program memory access by imposing constraints on variables in concrete and symbolic execution. In concrete execution, an invalid memory access would cause an

exception, while in symbolic execution it would result in a constraint system pin-pointing the erroneous instruction. More on that in the next chapter.

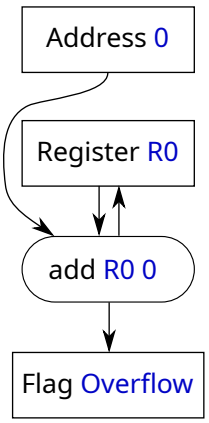
A little more complicated example of static tree semantics can be given to binary arithmetic operations, for example, addition. The semantics reads two data locations instead of just one in `Abs` and features a more involved overflow check:

```
Add :: Register -> CAddress -> InstructionImpl Applicative value a
```

```
add :: Register -> Address
    -> FS Key Applicative '[Monoid, Num, Bounded, Boolean, BOrd] a
add reg addr read write =
    let arg1 = read (Reg reg)
        arg2 = read (Addr addr)
        o = addOverflows <$> arg1 <*> arg2
        result = (+) <$> arg1 <*> arg2
     in write (F Overflow) o *> write (Reg reg) result

addOverflows :: (Num a, Bounded a, Boolean a, BOrd a)
             => a -> a -> a
addOverflows x y =
    let o1 = gt y 0
        o2 = gt x (maxBound - y)
        o3 = lt y 0
        o4 = lt x (minBound - y)
     in o1 &&& o2 ||| o3 &&& o4
```

```
add R0 0
```



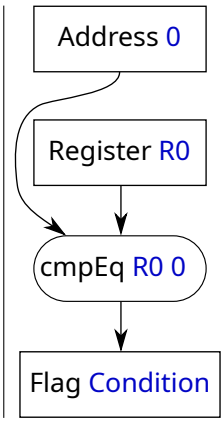
The `addOverflows` is a pure function that, when applied to two numbers, computes a boolean condition that indicates if the sum will overflow the `Bounded` type `a`. The result of this function will be written into the `Overflow` flag as a concrete boolean by the concrete backend and as a symbolic constraint by the symbolic execution backend. The polymorphism of fine-grained state transformers allows us to express both as a single computation and give it different interpretations by supplying the relevant `read` and `write` base transformers.

In the next section we will be considering conditional jumps, which can not be expressed with static tree transformers. However, conditional jumps need conditions, and calculation of those can be done with the applicative interface of static tree transformers:

```
CmpEq :: Register -> CAddress -> InstructionImpl Applicative value a
```

```
cmpEq :: Register -> Address
      -> FS Key Applicative '[Boolean, BEq, Monoid] a
cmpEq reg addr read write =
    write (F Condition)
          ((===) <$> read (Reg reg) <*> read (Addr addr))
```

```
cmpEq R0 0
```



Here, we apply the equality operator `(===)` to the data in the register and the memory location, and put the resulting boolean into the `Condition` flag.

So far we have been looking at linear and static tree fine-grained state transformers, therefore the dataflow diagrams on the right hand side were *precise*, i.e. we know, statically, from the source code of the semantics, the exact locations it will read and write. Let us go further and consider *selective* tree transformers, that still can be analysed statically for their data dependencies, but do not have to access or modify them all.

3.3.2.3 Selective Tree Dataflow

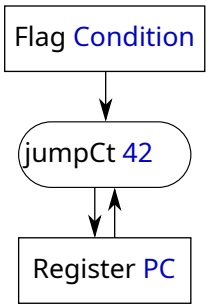
Selective tree fine-grained state transformers are governed by the new, recently introduced interface of selective applicative functors [16]. Selective applicative functors extend the `Applicative` type class of applicative functors with conditional execution, without losing amenability to static analysis.

In fact, the very idea of selective applicative functors has been inspired by the semantics of conditional jumps in instruction-set architectures. Without further delay, let us consider the selective transformer that gives the semantics to conditional jumps in REDFIN:

```
JumpCt :: value a => Imm a -> InstructionImpl Selective value a
```

```
jumpCt :: Imm a -> FS Key Selective '[Boolean, Num] a
jumpCt (Imm offset) read write =
    ifS (toBool <$> read (F Condition))
        (write IC ((+) <$> pure offset <*> read IC))
        (pure 0)
```

```
jumpCt 42
```



The semantics of `JumpCt` reads the value of the `Condition` flag and performs the jump if that value evaluates to true. This is done via the effectful conditional operator `ifS` of type `Selective f => f Bool -> f a -> f a -> f a`. Note that we need the `toBool` function to be applied to the value of the flag to force the evaluation of the boolean in the concrete execution backend. The symbolic execution backend, in turn, uses a different instance of `Selective` and evaluates both branches, updating their path conditions accordingly, thus making a conservative approximation of the jump's behaviour. The same is done by the static analysis backend to produce the dataflow diagram shown on the right. The selective semantics allows us to declare the effects of an instruction statically, and choose which of them to actually execute at runtime.

The selective fine-grained transformers provide enough flexibility to express the semantics of most of the instructions in REDFIN. This allows to enjoy the benefits of the delicate balance between static analysis and efficient execution that the `Selective` interface provides for many real-world programs. We will have a closer look at the examples in Chapter 5.

However, there is one instruction that requires even more expressive power, since its semantics involves transferring values between unpredictable, dynamic locations.

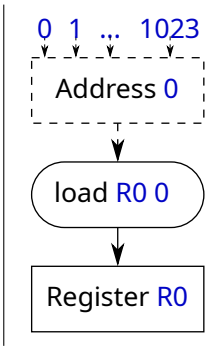
3.3.2.4 Dynamic Tree Dataflow

The instruction in question is `LoadMI`, which implements indirect memory access. Effectively, it enables using integer values stored in memory as pointers to access other addresses. The semantics of indirect memory access cannot be naturally expressed with the interface of the `Selective` type class, since it requires "unwrapping" of the integer value read from memory and interpreting it as an address for the consecutive memory access. Such behaviour can be naturally implemented with the monadic bind operation `(>>=)`:

```
LoadMI :: Register -> CAddress -> InstructionImpl Monad value a
```

```
loadMI :: Register -> Address -> FS Key Monad '[Addressable, Show] a
loadMI reg pointer read write =
    read (Addr pointer) >>= \x ->
        case toAddress x of
            Nothing -> error $
                "ISA.Semantics.loadMI: invalid address " <> show x
            Just addr -> write (Reg reg) (read (Addr addr))
```

```
loadMI R0 0
```



Here, the `(>>=)` operation allows treating the result of an effectful computation as a pure value, and thus perform case analysis on it. The downside of this is that we cannot anymore predict the target address to be accesssed, and the dataflow diagram on the right becomes too general to be useful, effectively treating any memory location as potentailly accessed. We therefore conclude, that the semantics of `LoadMI` can only be expressed as a coarse-grained state transformer.

Thankfully, indirect memory access in the only instruction in the REDFIN ISA that cannot be effectively implemented as a fine-grained state transformer. This setback will not hinder the benefits of the semantics as a whole, since we can give special treatment to this particular instruction in the backends, as we will see in the chapter 5.

3.4 Conclusion

In this chapter, we have presented the syntax of the REDFIN ISA, the representation of its state and two approaches to formulating its semantics. In the following two chapters 4 and 5, we will discuss these approaches in more detail and how to use them to build executable specification and verification frameworks for REDFIN.

Chapter 4

REDFIN semantics and program verification with coarse-grained monadic state transformers

In this chapter we describe the design of a model of the REDFIN ISA in terms of monadic state transformers; the model allows for both simulating REDFIN programs with concrete values for debugging and testing and for symbolically executing them with abstract symbolic variables for verifying their functional correctness, as well as some non-functional properties.

We represent the state of the REDFIN ISA as an ADT, and the semantics of instructions then are denoted as state transformers, i.e. pure functions that encode the changes of ISA state. We then use the established [86] fact that state transformers form a monad, and thus can exploit the support for monads in the Haskell programming language, giving both the benefit of the mathematical rigour, since all `Monad` instances have to obey monadic laws, and the ease of describing the semantics of instructions with do-notation.

In order to construct a fully usable ISA model which could be employed to simulate and verify REDFIN programs, we need to implement a number of components. First, since the machine and the human programmer work on different level of abstractions, it is not enough to implement an interpreter for the machine codes, i.e. the binary representation of

REDFIN instructions; we also need to implement an assembly language providing human-readable mnemonics for REDFIN instructions. Although, the assembly commands are more comprehensible than raw machine code, it is still very difficult to tell, even for a short program, what exactly is its meaning; thus, for the purpose of verifying the functional properties of REDFIN programs implemented in assembly, it would be good to have a high-level language that could be compiled down to REDFIN machine code and could be used as a concise *specification* language. To implement these components, we will follow the language-oriented programming paradigm and design a number of EDSLs.

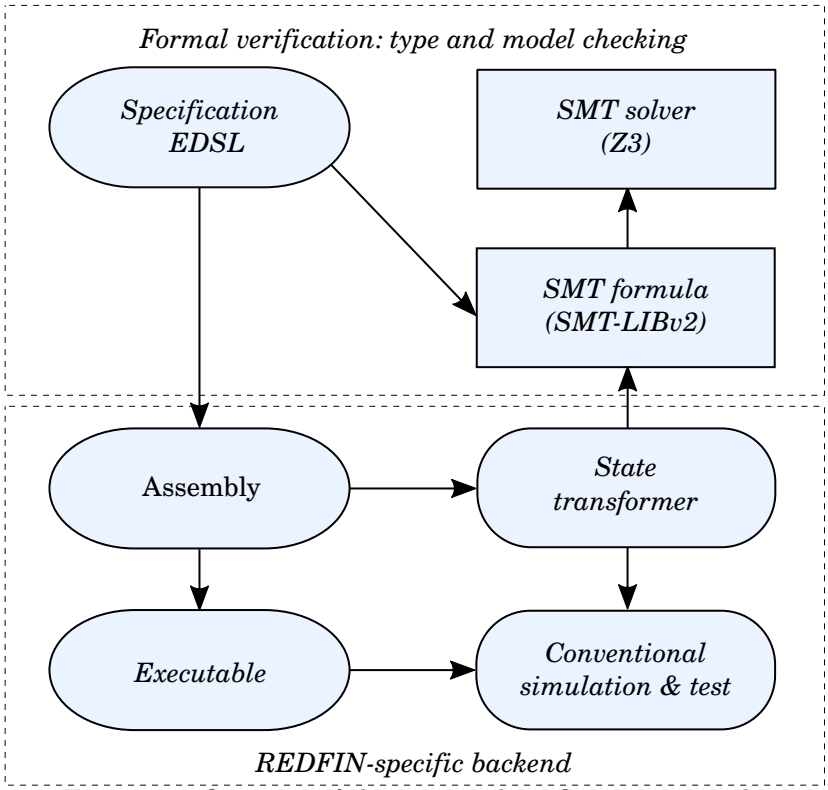


Figure 4.1: Overview of the presented verification approach.

Fig. 4.1 shows an overview of the model. The bottom part corresponds to conventional code generation and test, where REDFIN assembly language is executed by simulating the effect of each instruction on the state of the processor and memory. The corresponding *state transformer* is typically implicit and intertwined with the rest of the simulation infrastructure. The main idea of our approach is to represent the state transformer explicitly so that it can be symbolically manipulated and used not only for simulation but also for

formal verification. The latter is achieved by compiling state transformers to SMT formulas and using an SMT solver, e.g. Z3 [60], to verify that certain correctness properties hold, for example, that integer overflow cannot occur regardless of input parameters and that the program always terminates within stated time.

By using Haskell as the host language we can readily implement embedded compilers from higher-level *typed* languages to untyped assembly, eradicating incorrect number and unit conversion bugs. As shown at the top of Fig. 4.1, engineers can write high-level control programs for the REDFIN architecture directly in a small subset of Haskell. These high-level programs can be used for type-safe code generation and as executable specifications of intended functionality.

To facilitate whole-program verification, we use a specific flavour of symbolic execution known as *symbolic execution with merging*. In this approach, whenever a branch in the program is encountered, the symbolic execution engine performs merging of the two resulting disjunctive states into one, thus producing linear traces which could be translated into singular SMT-formulas representing whole programs. An overview of symbolic execution techniques can be found in the background chapter, section 2.1.3.

4.1 The REDFIN ISA state

In this section we define the state of the REDFIN instruction-set architecture, outlining all the components of the state and how they are represented as Haskell data types which support symbolic execution.

```
data State = State
  { registers           :: RegisterBank
  , memory              :: Memory
  , instructionCounter  :: InstructionAddress
  , instructionRegister :: InstructionCode
  , program             :: Program
  , flags               :: Flags
  , clock               :: Clock }

type Value = SymbolicValue (IntN 64)

type Register           = SymbolicValue (WordN 2)
type RegisterBank       = SymbolicArray (WordN 2) (IntN 64)

type MemoryAddress      = SymbolicValue (WordN 8)
type Memory             = SymbolicArray (WordN 8) (IntN 64)

type InstructionAddress = SymbolicValue (WordN 10)
type InstructionCode    = SymbolicValue (WordN 16)
type Opcode             = SymbolicValue (WordN 6)
type Program            = SymbolicArray (WordN 10) (WordN 16)

data Flag               = Condition | Overflow | Halt ...
type Flags              = SymbolicArray Flag Bool
type Clock              = SymbolicValue (WordN 64)
```

Figure 4.2: Basic types for modelling REDFIN.

The `State` of the entire processing core is a product of states of every component, see Fig. 4.2. We define `SymbolicValue` and `SymbolicArray` on top of the SBV library [87] that we use as a frontend for SMT translation and verification.

There are 4 registers (addressed by `WordN 2`) and 256 memory cells (addressed by `WordN 8`) that store 64-bit values (`IntN 64`). The register bank and memory are represented by symbolic arrays that can be accessed via SBV's functions `readArray` and `writeArray`. REDFIN uses 16-bit `InstructionCode`s, whose 6 leading bits contain the opcode, and the remaining 10 bits hold instruction arguments. A `Program` maps 8-bit instruction addresses to instruction codes.

Note that we use *indexed types* to represent bit-vectors (i.e. `WordN 8`, `IntN 64` etc.) as a lightweight type-based verification mechanism for making sure that the especially error-prone components of the model, for example the instruction decoder and the assembler, will benefit from Haskell's type-checker ensuring that the bit-vector sizes always are compatible. We will see the benefits when we discuss the assembly EDSL.

The status `Flags` support conditional branching, track integer overflow, and terminate the program (we omit a few other flags for brevity. The `Clock` is a 64-bit counter incremented on each clock cycle. Status flags and the clock are used for diagnostics, formal verification, and worst-case execution time analysis.

4.2 Instruction and Program Semantics

In this section, we instantiate the abstract operational semantics presented in chapter 3 with monadic state transformers. This concrete semantics is written in idiomatic Haskell and has an advantage of closely resembling a simple instruction interpreter. However simple it may look, this semantics is in fact powerful enough to facilitate formal verification of terminating programs by symbolic execution.

We represent state transformers in Haskell using the *state monad*, a classic approach to emulating mutable state in a purely functional programming language [86]. We call our state monad `Redfin` and define it as follows[1]:

```
data Redfin a = Redfin { transform :: State -> (a, State) }
```

A computation of type `Redfin a` yields a value of type `a` and possibly alters the `State` of the REDFIN instruction-set architecture. The type `Redfin ()` describes computations that do not produce any value as part of the state transformation and thus only return `()` — the only value of type `()` (also known as the unit type); such computations directly correspond to state transformers.

For example, here is the state transformer T_{inc}:

```
incrementInstructionCounter :: Redfin ()
incrementInstructionCounter = Redfin $ \current -> ((), next)
  where
    next = current { instructionCounter = instructionCounter current + 1 }
```

In words, the state transformer looks up the value of the `instructionCounter` in the `current` state and replaces it in the `next` state with the incremented value. We can compose such primitive computations into more complex state transformers using Haskell's `do`-notation:

[1] A generic version of this monad is available in the standard module `Control.Monad.State`.

```
readInstructionRegister :: Redfin InstructionCode
readInstructionRegister = Redfin $ \s -> (instructionRegister s, s)

executeInstruction :: Redfin ()
executeInstruction = do
    fetchInstruction
    incrementInstructionCounter
    instructionCode <- readInstructionRegister
    decodeAndExecute instructionCode
```

Here `readInstructionRegister` reads the instruction code from the current state *without modifying it*, and is subsequently used in `executeInstruction`, which defines the semantics of the REDFIN execution cycle. We omit definitions of `fetchInstruction` and `decodeAndExecute` for brevity. The latter is a case analysis of 47 opcodes that returns the matching instruction.

The verification framework which we base on the state transformer semantics is designed as symbolic-first, i.e. we prioritise the ability to verify properties of REDFIN program by symbolic execution and treat testing and concrete simulation as secondary facilities. But in fact, the design of the framework and the SMT Based Verification (SBV) library allows to use the very same symbolic execution semantics for concrete execution too, by supplying concrete values instead of symbolic variables as inputs. Therefore, the symbolic simulation function (fig. 4.3) can be used for both testing and formal verification. The `simulate` function is essentially a modification of the $T_{\texttt{i:p}}$ state transformer semantics of a composite program, with an additional account for early cut-off after `steps` transition or after the `Halt` flag has been set. The latter condition is checked symbolically using the symbolic conditional operator `ite`: if the processor is halted we return the current state, otherwise we proceed with the next transition.

```
simulate :: Word -> State -> State
simulate steps state
  | steps == 0 = state
  | otherwise  = let halted    = readArray (flags state) (flagId Halt)
                     nextState = snd $ transform executeInstruction state
                 in ite halted
                        state
                        (simulate (steps - 1) nextState)
```

Figure 4.3: Implementation of the REDFIN state transformer

Now, when the general execution facilities are established, we discuss the semantics of several instructions below.

4.2.1 Halting the Processor

The instruction `halt` sets the flag `Halt`, which stops the execution of the current subroutine until a new one is started by a higher-level system controller that resets `Halt`.

```
halt :: Redfin ()
halt = writeFlag Halt true
```

The auxiliary function `writeFlag` modifies the flag:

```
writeFlag :: Flag -> SymbolicValue Bool -> Redfin ()
writeFlag flag value = Redfin $ \s -> ((), s')
  where
    s' = s { flags = writeArray (flags s) (flagId flag) value }
```

In the rest of the chapter we will use auxiliary functions `readRegister`, `writeRegister`, `readState`, etc.; they are simple state transformers defined similarly to `writeFlag` and `readInstructionRegister`.

4.2.2 Arithmetics

The instruction `abs` is more involved: it reads a register and writes back the absolute value of its contents. The semantics accounts for the potential integer overflow that leads to the *negative resulting value* when the input is -2^{63} (REDFIN uses the common two's complement signed number representation). The overflow is flagged by setting `Overflow`. We use SBV's symbolic *if-then-else* operation `ite` to *merge* two symbolic values — in this case two possible next states, one of which is a state with the `Overflow` flag set:

```
abs :: Register -> Redfin ()
abs reg = do
    state  <- readState
    result <- fmap Prelude.abs (readRegister reg)
    let (_, state') = transform (writeFlag Overflow true) state
    writeState $ ite (result .< 0) state' state
    writeRegister reg result
```

4.2.3 Conditional Branching

As an example of a control flow instruction consider `jmpi_ct`, which tests the `Condition` flag, and adds the provided signed `offset` to the instruction counter if the flag is set.

```
jmpi_ct :: SImm10 -> Redfin ()
jmpi_ct offset = do
    ic <- readInstructionCounter
    condition <- readFlag Condition
    let ic' = ite condition (ic + offset) ic
    writeInstructionCounter ic'
```

It is possible to write REDFIN programs directly in the semantics, however it is a difficult enterprise, akin to programming on the ISA level: all jumps have to be resolved manually; thus, in the next section we describe the design and implementation of an assembly language for REDFIN.

4.3 Simulation and formal verification

In the previous sections we have developed the facilities for modelling the REDFIN instruction-set architecture: how the state is represented, how the semantics of individual instructions is described in terms of state transformers and what syntax can be used to construct programs and specify their functional properties. However, we are still to discuss program simulation, testing and formal verification. In this section we will cover these topics and tie together the different parts of the verification framework.

The verification framework is designed with the following workflow in mind:

1. Develop programs in low-level REDFIN assembly, or in a high-level typed language embedded in Haskell.
2. Test REDFIN programs on concrete input values.
3. Define functional correctness and worst case execution time properties using the Expression EDSL and SBV.
4. Verify the properties or obtain counterexamples.

4.3.1 Energy estimation control task

For demonstrating the workflow, we continue to employ the energy estimation control task:

> Let t_1 and t_2 be two different time points (measured in ms), and p_1 and p_2 be two power values (measured in mW). Calculate the estimate of the total energy consumption during this period using linear approximation, rounding down to the nearest integer:
>
> $$energyEstimate(t_1, t_2, p_1, p_2) = \left\lfloor \frac{|t_1 - t_2| * (p_1 + p_2)}{2} \right\rfloor.$$

We can write programs in the low-level REDFIN assembly, or in a higher-level expression language. The former allows engineers to hand-craft highly optimised programs under tight resource constraints, while the latter brings type-safety and faster prototyping. We start with the high-level approach and define an expression that can be used both as a Haskell function and a high-level REDFIN expression:

```
energyEstimate :: Integral a => a -> a -> a -> a -> a
energyEstimate t1 t2 p1 p2 = abs (t1 - t2) * (p1 + p2) `div` 2
```

Thanks to polymorphism, we can treat `energyEstimate` both as a numeric function, and as an abstract syntax tree that can be *compiled* into a REDFIN assembly `Script`, as described in section 6.2, thus facilitating *specification* of functional properties of REDFIN programs. Naturally, the compiled program will require declaring the memory layout of the input variables:

```
energyEstimateHighLevel :: Script
energyEstimateHighLevel =
    let [t1, t2, p1, p2] = map varAtAddress [0, 1, 2, 3]
    in compile (energyEstimate t1 t2 p1 p2)
```

The `varAtAddress` function is an alias for the composition of data constructors `Var` and `MkVariable` which creates a variable in the abstract syntax and indicates that it is located at the given memory address. The `let` block declares four adjacent memory addresses: four input values $\{t_1, t_2, p_1, p_2\}$. We `compile` the high-level expression `energyEstimate` into the

assembly language by translating it to a sequence of REDFIN instructions. By convention, the calculated result will be placed in the register `r0`.

4.3.1.1 Program simulation

We can run simulation for at most 100 steps, initialising the program and data memory of the processor using the function `simulate` defined in section 4.2 (fig. 4.3):

```
main = do
    let dataMemory = [10, 5, 3, 5]
        finalState = simulate 100 $ boot energyEstimateHighLevel dataMemory
    putStrLn $ "R0: " ++ show (readArray (registers finalState) r0)
```

As the simulation result we get a `finalState`. We inspect it by printing relevant component: the result of the computation located in the register `r0`.

```
R0: 20
```

4.3.1.2 Formal verification

Simulating programs with specific inputs is useful for diagnostics and test, but SMT solvers allow us to verify the correctness for *all valid input combinations*. To demonstrate this, let us discover a problem in our energy estimation program. Consider the following total correctness[2] property:

> Assuming that values p_1 and p_2 are non-negative integers, the energy estimation subroutine must always terminate and return a non-negative integer value.

To check that the program meets this requirement, we symbolically execute the high-level energy estimation program to translate it into an SMT formula, and formulate the property:

```
ex4_5_1_1 = do
    [t1, t2, p1, p2] <- symbolics ["t1", "t2", "p1", "p2"]
    constrain $ t1 .>= 0
    constrain $ t2 .>= 0
```

[2] As opposed to partial correctness, which does not include termination, we also verify that the REDFIN core has halted, by checking the status of the `Halt` flag.

```
    constrain $ p1 .>= 0
    constrain $ p2 .>= 0
    let prog = assemble energyEstimateHighLevel
        steps = 100
        mem = mkMemory [(0, t1), (1, t2), (2, p1), (3, p2)]
    let initialState = boot prog defaultRegisters mem defaultFlags
        finalState = simulate steps initialState
        result = readArray (registers finalState) 0
        halted = readArray (flags finalState) (flagId Halt)
    pure $   halted
         .&& result .>= 0
```

Note how this time we declare the inputs $\{t_1, t_2, p_1, p_2\}$ as symbolic variables, constrain them to be non-negative and explicitely lay them out in memory. We form the initial state with the `boot` function supplying the program and the states of registers, memory and flags. Next, we extract the computed `result` and the value of the flag `Halt` from the `finalState`, and then assert that the processor has `halted`, and that the `result` is non-negative. The resulting SMT formula can be checked by Z3 in 3.0s[3]:

```
> proveWith z3 theorem
Falsifiable. Counter-example:
  t1 = 5190405167614263295 :: Int64
  t2 =                   0 :: Int64
  p1 =  149927859193384455 :: Int64
  p2 =  157447350457463356 :: Int64
```

Z3 has found a counterexample demonstrating that the program does not satisfy the above property. Indeed, the expression evaluates to a negative value on the provided inputs due to an *integer overflow*. We therefore refine the property:

> According to the spacecraft power system specification, p_1 and p_2 are non-negative integers not exceeding 1W. The time is measured from the mission start, hence t_1 and t_2 are non-negative and do not exceed the time span of the mission, which is 30 years. Under these assumptions, the energy estimation subroutine must return a non-negative integer value.

We need to modify time and power constraints accordingly:

```
constrain $ t1 .>= 0 .&& t1 .<= toMilliSeconds (30 % Year)
constrain $ t2 .>= 0 .&& t2 .<= toMilliSeconds (30 % Year)
```

[3]We use a laptop with 2.90GHz Intel Core i5-4300U processor, 8GB RAM (3MB cache), and the SMT solver Z3 v4.5.1 (64-bit).

```
constrain $ p1 .>= 0 .&& p1 .<= toMilliWatts (1 % Watt)
constrain $ p2 .>= 0 .&& p2 .<= toMilliWatts (1 % Watt)
```

Rerunning Z3 produces the desired QED outcome in 4.8s.

The refinement has rendered the integer overflow impossible; in particular, `abs` can never be called with -2^{63} within the mission parameters. Such a guarantee fundamentally requires solving an SMT problem, even if it is done at the type level, e.g. using *refinement types* [88].

The statically typed high-level expression language is very convenient for writing REDFIN programs, however, an experienced engineer can often find a way to improve the resulting code. In some resource-constrained situations, a fully hand-crafted assembly code may be required. As an example, consider the following low-level program:

```
energyEstimateLowLevel :: Script
energyEstimateLowLevel = do
    let { t1 = 0; t2 = 1; p1 = 2; p2 = 3 }
    ld r0 t1
    sub r0 t2
    abs r0
    ld r1 p1
    add r1 p2
    st r1 p2
    mul r0 p2
    sra_i r0 1
    halt
```

This program computes the energy estimate using only 9 instructions, whereas a direct unoptimised translation of the `energyEstimate` expression into assembly uses 79 instructions, most of them for stack manipulation.

4.3.1.3 Checking program equivalence

To support the development of hand-crafted code, we use Z3 to *check the equivalence of REDFIN programs* by verifying that they produce the same output on all valid inputs. This allows an engineer to optimise a high-level prototype and have a guarantee that no bugs were introduced in the process.

```
ex4_5_1_3 = do
    [t1, t2, p1, p2] <- symbolics ["t1", "t2", "p1", "p2"]
    constrain $ t1 .>= 0 .&& t1 .<= toMilliSeconds (1 % Year)
    constrain $ t2 .>= 0 .&& t2 .<= toMilliSeconds (1 % Year)
```

```
    constrain $ p1 .>= 0 .&& p1 .<= toMilliWatts (1 % Watt)
    constrain $ p2 .>= 0 .&& p2 .<= toMilliWatts (1 % Watt)
    let steps = 100
        mem = mkMemory [(0, t1), (1, t2), (2, p1), (3, p2)]
    let progLL = assemble energyEstimateLowLevel
        progHL = assemble energyEstimateHighLevel
    let initialStateLL = boot progLL defaultRegisters mem defaultFlags
        initialStateHL = boot progHL defaultRegisters mem defaultFlags
    let finalStateLL = simulate steps initialStateLL
        finalStateHL = simulate steps initialStateHL
    let resultLL = readArray (registers finalStateLL) 0
        resultHL = readArray (registers finalStateHL) 0
    pure $ resultLL .== resultLL
```

The `equivalence` check succeeds and takes 11.5s.

4.3.1.4 Worst-Case Execution Time analysis

Every call of the `executeInstruction` function advances the `clock` field of the `State` (see fig. 4.2) by the appropriate number of cycles, precisely matching the hardware implementation. This allows us to perform *best/worst-case execution timing analysis* using the optimisation facilities of SBV and Z3. As an example, let us determine the minimum and maximum number of clock cycles required for executing `energyEstimateLowLevel`. To make this example more interesting, we modified the semantics of the instruction `abs` and added 1 extra clock cycle in case of a negative argument.

```
timingAnalysis = optimize Independent $ do
  ... -- Initialise and run symbolic simulation
  minimize "Best case"  (clock finalState)
  maximize "Worst case" (clock finalState)
```

The total delay of the program depends only on the sign of $t_1 - t_2$, thus the best and worst cases differ only by one clock cycle. The worst case is achieved when the difference is negative ($t_1 - t_2 = -2$), as shown below. Z3 finishes in 0.5s.

```
Objective "Best case":
Optimal model:
  t1        = 549755813888
  t2        =  17179869184
  p1        =             0
  p2        =             0
  Best case =            12
```

```
Objective "Worst case":
Optimal model:
  t1         = 65535
  t2         = 65537
  p1         =     0
  p2         =     0
  Worst case =    13
```

4.3.2 Array sum

A number of control tasks require summarising data from sensors, which may be stored as arrays in memory. As a simple, yet interesting example, let us consider a program that calculates the sum of numbers in an array (fig. 4.4).

```
sumArrayLowLevel :: Script
sumArrayLowLevel = do
    let { pointer = 0; sum = 253; const_two = 255 } -- ; pointer_store
    ld_i r0 0
    st r0 sum
    ld r1 pointer
    add_si r1 1
    st r1 pointer

    -- compare the pointer variable to the constant 2 (stored in the cell 255)
    "loop"  cmplt r1 const_two
    -- if pointer == 2 then terminate
    goto_ct "end"

    ldmi r2 pointer
    add r2 sum
    st r2 sum
    ld r1 pointer
    sub_si r1 1
    st r1 pointer

    goto "loop"
    "end"  ld r0 sum
    halt
```

Figure 4.4: Array summation program

The array summation program implements the following algorithm: assuming the array length n is known and located in the cell 0, and the array is placed at address range $[2, n+2]$, the program loops through the range in reverse order and accumulates the sum in the memory cell 253, which, upon reaching the termination condition, is copied into the register `r0`. Since the array length is known in advance, it is possible to prove that the program terminates by loop unrolling. This property is essential for the applicability of formal verification by symbolic execution: if the number of iterations n were to depend on any symbolic variable, the symbolic execution algorithm would not terminate. We will discuss the problem of symbolic termination and the approaches to verification of non-terminating programs later in this thesis.

To specify the functional correctness of array summation we can express the algorithm in Haskell by computing the sum of a list of values. As in the energy estimation example, to compile the Haskell specification into assembly, we need to also specify how the inputs variables are laid out in the memory:

```
sum' :: (Foldable t, Num a) => t a -> a
sum' = foldl' (+) 0

sumArrayHighLevel :: Int -> Script
sumArrayHighLevel arraySize = do
    let xs = map varAtAddress [2..(fromIntegral arraySize) + 1]
    compile (sum' xs)
```

The `sum'` function closely matches the implementation of the corresponding function in Haskell's standard library `base`: we use this implementation in order to additionally point out the fact that the high-level program, when compiled, will essentially contain a sequence of instructions corresponding to the unrolled loop from the low-level program 4.4. This happens because the `foldl'` function, the strict left fold, is used here to express an arithmetic imperative loop.

To formulate theorems about array summation, it will be useful to construct a theorem schema — a higher-order template of a theorem which could be specialised to specific programs, constrains and statements:

> Let xs be the array elements, p be the program and s be a valid initial state. Then for all predicates P over values and for all predicates Q over REDFIN states, the following holds:
>
> $$(\forall x \in xs, P(x)) \implies Q(T_{\mathtt{p}}(s))$$
>
> I.e. if we constrain array elements with predicate P, then the state after executing the array summation program p will satisfy the predicate Q.

The schema can be formulated with SBV in as the following Haskell function, which adds the specific details:

```
theorem :: Int -> Script -> [Value] -> (Value -> Symbolic ())
        -> (State -> SBool) -> Symbolic SBool
theorem steps src summands constr statement = do
    sequence_ (zipWith ($) (repeat constr) summands)
    let mem = mkMemory
                (zip [2..] summands ++
                [(0, literal . fromIntegral $ length summands)] ++
                [(255, 2)])
    let prog = assemble src
    let initialState = boot prog defaultRegisters mem defaultFlags
        finalState = simulate steps initialState
    pure $ statement finalState
```

Figure 4.5: Array sum theorem schema

To form a valid initial state, we lay out the array in memory, put the array size in the location 0 and the constant 2 (used to check the loop termination condition in the low-level program) in the cell 255. The symbolic variables which form the array are constrained with the `constr` function, which corresponds to the predicate P. The statement of the theorem (predicate Q) is then applied to the final state.

Let us now verify a number of statements about low and high-level array summation programs and discuss how verification time is affected by the array size.

4.3.2.1 Integer overflow

Using the theorem schema 4.5, let us formulate and prove that integer overflow is guaranteed to not occur if the array is of length 15 and its elements are non-negative and do not exceed 1000[4]:

```
ex4_5_2_1 =
  let constr    x     = constrain (x .>= 0 .&& x .<= 1000)
      statement state =
          let halted = readArray (flags state) (flagId Halt)
              overflow = readArray (flags state) (flagId Overflow)
          in halted .&& sNot overflow
  in theorem 1000 sumArrayLowLevel 15 constr statement
```

The statement of the theorem specifies that the program terminates and that the `Overflow` flag is not set in the final state. Remember that due to the nature of the symbolic execution algorithm that SBV uses, the final state of the overflow flag will contain the symbolic merge

[4]These constrains look somewhat arbitrary and not motivated by performance bounds. We will discuss the performance limitations later in this section.

of all the states encountered during the execution of the program; thus the statement is sufficient to check for absence of overflow. The same property can be formulated for the high-level program. Note that it is vital to give a big enough execution bound (`steps`) for the program to terminate — here 1000 is enough, but an array of 53 elements would already require a larger number of steps.

4.3.2.2 Program equivalence

For the energy estimation task we were checking equivalence of the high-level compiled program and hand-crafted low-level assembly. Same could be done for the array sum programs, but let us demonstrate another option. We could directly use the `sum'` function as a specification and check its functional equivalence to the low-level program by symbolic execution:

```
ex4_5_2_3 :: Symbolic SBool
ex4_5_2_3 = do
  let names = map (("x" ++) . show) [1..15]
  summands <- symbolics names
  let constr    x     = constrain (x .>= 0 .&& x .<= 1000)
      statement state =
          let halted = readArray (flags state) (flagId Halt)
              result = readArray (registers state) r0
          in halted .&& result .== sum' summands
          theorem 1000 sumArrayLowLevel summands constr statement
```

Array size	Time Haskell	Time Compiled
6	0.07s	0.09
9	0.07s	0.2s
12	0.26s	1.12s
15	1.57s	8.07s
18	10.9s	1m 3.36s

Table 4.1: Verification time for array summation programs

In this case, the specification will not be compiled into assembly, but instead symbolically executed as Haskell code directly: this verification problem is easier for an SMT solver to handle, since there symbolically executed Haskell program produces a simpler SMT formula than the equivalent compiled assembly program. However, a microbenchmark (table 4.1) reveals that even though checking the equivalence with the Haskell expression indeed is faster, the verification time is still appears to be exponential in the length of the array.

4.3.3 Discussion

This chapter presents the coarse-grained monad semantics of the REDFIN ISA based and the verification framework for REDFIN programs based on this semantics. We have discussed the design and implementation of the framework, and the benefits of the formulating an ISA semantics as a monadic state transformer. This approach, while it is simple and natural for embedding a complicated stateful computation into a purely function language, presents a number of inconveniences for program verification with symbolic execution.

The verification case-studies we have presented in this chapter include an energy estimation program and an array summation program. Both these programs are interesting, since they belong to distinct classes if assessed by the termination condition. The energy estimation program is a typical subroutine for REDFIN: it is simple, serving a very specific computational task; and it is *always terminating*, i.e. it can be statically proved to only be executing for a known amount of steps. Formal verification of such programs has been the base task of this thesis. The array summation program's termination condition depends on the specification method we choose. We may consider the array to have a statically known length, as we did in this chapter, and then the program will trivially terminate in the number of steps needed for computing one addition multiplied by the array's length. However, it is desirable to have a more general specification that would universally quantify over the array's length, and prove instances of this specification for a range of length values. Unfortunately, the verification framework described in this chapter, while it is great for trivially terminating programs, is not suitable for this task. The state space explosion caused by the symbolic termination conditions is not easily resolved within the state-merging symbolic execution approach that we have chosen.

While formal verification of trivially terminating arithmetic programs is an important task in space software engineering, we would like to aim at verification of safety and functional correctness properties of the wider class of REDFIN programs, that do not trivially terminate. Looping programs with symbolic termination conditions also present an important class of REDFIN programs.

We conclude this chapter at this point, and transition to the next one, which describes the next milestone of this thesis. We will describe the new iteration of the verification framework for REDFIN programs, based on a novel semantics approach and leveraging a different

symbolic execution approach. We will specifically discuss verification of looping programs with symbolic termination conditions, finally arriving to the case-study of a stepper-motor control program deployed into an antenna pointing subsystem of a space satellite.

Chapter 5

REDFIN semantics and program verification with fine-grained state transformers

In this chapter, we present *fine-grained state transformers* — a novel approach to describing semantics of computations with effects. We have glanced over this approach in the section 3.3.2 to compare it with the coarse-grained state transformers we discussed in detail in chapter 4. Now, we give a more detailed account to fine-grained state transformers.

We first formulate the approach in Haskell and present simple intuition and examples. Further, we use the presented approach to formulate the semantics of REDFIN ISA in a polymorphic way that admits multiple interpretations and is highly amenable for static analysis. We then discuss these interpretations and the implementation of the verification framework for REDFIN ISA.

5.1 Fine-grained state transformers

Fine-grained state transformers provide a methodology for describing effectful computations in a polymorphic way, abstracting over the effects semantics and instead focusing on their

structure. Recall, that Haskell provides a hierarchy of type classes (discussed in section 2.2.4) to capture a polymorphic interface for effectful computations. Fine-grained transformers are orthogonal to this hierarchy, and can be used in conjunction with type classes to achieve even greater flexibility.

Ultimately, a fine-grained state transformer is a function, just like almost everything else in Haskell is. However, the type of this function is peculiar and employs a number of advanced type system extensions.

5.1.1 The `FS` type

The basic idea is to specify an effectful computation by the computational context it operates in, and by the actions it can perform over this context. A generic context can be thought of as a key-value store, that associates values of a certain type to a set of keys. An interaction with such a store can be boiled down to the operations of reading and writing the values associated with the keys. The flexibility of the approach comes to play when we remember that the context can be as complicated as we wish, and the reading and writing operations that are associated with it can have side-effects and thus, besides reading and writing, can perform some additional context-specific actions.

```
-- | Fine-grained state transformer
type FS
    (key :: Type)
    (control :: (Type -> Type) -> Constraint)
    (value :: [Type -> Constraint])
    (a :: Type) =
    forall (f :: Type -> Type).
    (control f, CS value a) =>
    (key -> f a) ->
    (key -> f a -> f a) ->
    f a

-- | Constrain the type 'a' by a list of constraints
type family CS (cs :: [Type -> Constraint]) (a :: Type) :: Constraint where
    CS cs a = CSGo cs (Any a) a

type family CSGo (cs :: [Type -> Constraint]) (acc :: Constraint) a :: Constraint where
    CSGo '[] acc _ = acc
    CSGo (x ': xs) acc a = CSGo xs (x a, acc) a
```

Figure 5.1: The type `FS` of a fine-grained state transformer

An `FS` computation is defined by the computational context `f`, the algebraic data type `key` of the keys and two parameterised type class constraints. The `value` parameter defines the constraints which determine the interface that will be used to define *pure* values the computation operates over. For example, it may include the equality constraint `Eq`, the numeric constraint `Num`, or something else. We use a type-level function `CS` to fold the list of value constraints into a single one automatically at compile time.

The most interesting ingredient is the `control` constraint, which accepts a type constructor (a type variable of kind `Type -> Type`) as a parameter. This variable can be, for example, instantiated with `Applicative` to only permit statically known shape of side effects. Or it could be instantiated with `Monad` to permit arbitrary dynamic effects. As we will see further, the `Selective` type class proves to be very useful here.

The `FS` type is existentially quantified in the type constructor variable `f`, which will be constrained by the `control` constraint and is the computational context the semantics will be interpreted in. The choice of `f` determines the interpretation of the polymorphic fine-grained state transformer. A value of `FS` type has two explicit arguments, which we will call the *read* and *write* callbacks. The first one, of type `key -> f a`, receives a key and returns a value in the computational context `f`. The read callback describes what does it mean to read a value associated with the key. The write callback, of type `key -> f a -> f a` receives a key and a *computation* of type `f a` which describes what needs to be done in order to compute the value to be written, and write it.

Component	Description	Haskell implementation
`key`	type of keys the computation operates with	sum type
`value`	value interface	list of parameterised constraints
`control`	control interface	parameterised constraint
`f`	computational context	existentially quantified type constructor
`read`	read callback	function of type `key -> f a`
`write`	write callback	function of type `key -> f a -> f a`

Table 5.1: Description of fine-grained stateful computation components

Through the use of ad-hoc polymorphism via type classes, rank-2 polymorphic functions and existential quantification, the `FS` type achieves high flexibility. We will demonstrate the benefits this flexibility brings by considering how the semantics of REDFIN ISA can be modelled with fine-grained state transformers.

5.2 REDFIN ISA semantics as a fine-grained state transformer

In the previous chapter, we have discussed a model of REDFIN ISA based in *coarse-grained* state transformers. That model, while suitable simple and suitable for verification of straight-line programs, suffers when many conditional jumps are involved. Also, since the coarse-grained semantics is inherently *monadic*, the model does not permit static analysis.

In this section, we model the REDFIN ISA semantics as a fine-grained state transformer. We will describe the semantics polymorphically. By varying the components from table 5.1, while keeping the `key` data type fixed, we will give several interpretations to the polymorphic semantics.

In the course of this section, we will describe more concretely the ingredients that the table 5.1 outlines. We get started by instantiating the `key` parameter with a concrete data type representing the data locations of REDFIN ISA.

5.2.1 Data types

The REDFIN ISA has six types of data locations, represented by the six data constructors of the `Key` data type (figure 5.2):

- `Reg`'s are the four general-purpose registers of REDFIN;
- `Addr` represents a potentially symbolic memory location;
- `F` refers to the field of the status register;
- `IC` is the instruction counter — during program execution it contains the address of the next instruction to be fetched from program memory;
- `IR` is the instruction register — during program execution it contains the binary instruction code of the fetched instruction;
- `Prog` is an address in program memory.

Note that the REDFIN ISA has separate memory regions for data and instructions. We, however, model them similarly and reuse the data type of addresses, even though the program addresses will never be accessed symbolically.

```haskell
data Key where
    -- | data register
    Reg :: Register -> Key
    -- | memory cell
    Addr :: Address -> Key
    -- | flag, a special boolean register
    F :: Flag -> Key
    -- | instruction counter
    IC :: Key
    -- | instruction register
    IR :: Key
    -- | program address
    Prog :: Address -> Key
```

```haskell
-- | Registers
data Register = R0 | R1 | R2 | R3

-- | Concrete memory address
newtype CAddress = CAddress Word8

-- | Immediate argument
newtype Imm a = Imm a

-- | Flags
data Flag
    = Halted
    | Condition
    | Overflow
    | DivisionByZero

-- | Binary instruction code
newtype InstructionCode =
    InstructionCode Word16
```

Figure 5.2: REDFIN ISA keys

The data type of keys in figure 5.2 account for potentially symbolic memory addresses, but we have not yet introduced how they are represented. We will do so soon, just before transferring to discussing the semantics of instructions.

5.2.2 Value type classes

To provide an interface to manipulate both concrete and symbolic values in a unified way, we reuse some of the standard Haskell type classes and define our own where there is no standard alternative. For arithmetic, we reuse Haskell's `Num` type class since it also gives us shallowly-embedded numeric literals. We, however, need our own equality and order type classes, since the standard ones, `Eq` and `Ord`, are fixed to the `Bool` result type. Therefore, we also define a type class `Boolean` for boolean values, which abstracts the usual connectives of boolean algebra.

```haskell
class Boolean a where
    -- | Conversion to and from 'Bool'
    toBool :: a -> Bool
    fromBool :: Bool -> a

    --| Constants
    true :: a
    false :: a

    -- | Negation
    not :: a -> a

    -- | Disjunction
    infixr 2 |||
    (|||) :: a -> a -> a

    -- | Conjunction
    infixr 3 &&&
    (&&&) :: a -> a -> a
```

```haskell
class BEq a where
    infix 4 ===
    (===) :: a -> a -> a

class BEq a => BOrd a where
    -- | Less then
    infix 4 `lt`
    lt :: a -> a -> a
    -- | Greater then
    infix 4 `gt`
    gt :: a -> a -> a
```

Figure 5.3: Type classes for Booleans, equality and order

The standard Haskell `Bool` data type is a simple instance of the `Boolean` type class. The REDFIN ISA semantics will be able to operate over concrete values (data type in figure 5.4), which are a *partial* instance of the `Boolean` class, forwarding the boolean connectives into the `CBool` data constructor and failing with an `error` in cases of numeric constructors.

```haskell
-- | Concrete values: either signed or unsigned integers, or booleans
data Concrete where
    CInt32 :: Int32 -> Concrete
    CWord :: Word16 -> Concrete
    CBool :: Bool -> Concrete
```

Figure 5.4: Concrete values

In the next section, we will define the data type of symbolic expressions, which we will make an instance of these type classes, alongside with the concrete values.

5.2.3 Symbolic values

A symbolic value is either a concrete value — a wrapper over signed and unsigned integers or a boolean, or a symbolic expression which may include variables, pointers or symbolic conditionals. The figure 5.5 defines the algebraic data types of symbolic values. The `Sym`

type has, among others, constructors for arithmetic, equality, order and boolean operations, and for symbolic variables (`SAny`), pointers (`SPointer`) and conditionals (`SIte`).

```
-- | Symbolic expressions
data Sym where
    SConst :: Concrete -> Sym
    SAny :: Text -> Sym
    SPointer :: Sym -> Sym
    SIte :: Sym -> Sym -> Sym -> Sym
    SAdd :: Sym -> Sym -> Sym
    SSub :: Sym -> Sym -> Sym
    SMul :: Sym -> Sym -> Sym
    SDiv :: Sym -> Sym -> Sym
    SMod :: Sym -> Sym -> Sym
    SAbs :: Sym -> Sym
    SEq :: Sym -> Sym -> Sym
    SGt :: Sym -> Sym -> Sym
    SLt :: Sym -> Sym -> Sym
    SAnd :: Sym -> Sym -> Sym
    SOr :: Sym -> Sym -> Sym
    SNot :: Sym -> Sym
```

Figure 5.5: Data type of symbolic values

We also implement a number of standard facilities, such as `subst` function that substitutes a variable for an expression, and others that perform constant-folding and simplification (table 5.2).

Name	Type	Description
`subst`	`Sym -> Text -> Sym -> Sym`	variable substitution
`tryFoldConstant`	`Sym -> Sym`	constant folding

Table 5.2: Programming interface of symbolic values

The `Sym` data type is an instance of the standard Haskell `Num` type class to support numeric literals and arithmetic.

```
instance Num Sym where
    x + y = SAdd x y
    x - y = SSub x y
    x * y = SMul x y
    abs x = SAbs x
    signum _ = error "Sym.Num: signum is not defined"
    fromInteger x = SConst (CInt32 $ fromInteger x)
    negate _ = error "Sym.Num: negate is not defined"
```

The `Boolean` instance is defined in a similar way. Having these instances in scope allows for constructing `Sym` expressions from shallowly-embedded Haskell expressions, which is

much easier than using the deeply-embedded syntax of `Sym` directly:

```
> let x = SAny "x" in x `gt` 0 &&& x `lt` 1000
SAnd (SGt (SAny "x" (SConst (CInt32 0)))) (SLt (SAny "x" (SConst (CInt32 1000))))
```

5.2.4 Memory representation

The data types modelling the REDFIN ISA which will be used for interpreting the fine-grained state semantics must account for the possible interpretations of the semantics. The primary, and also the most complicated, interpretation is symbolic execution, which requires a representation of symbolic memory. Therefore, the data types that model memory addresses must account for both concrete and symbolic representation.

We model the concrete REDFIN memory addresses with a `newtype` wrapping a value off type `Word8` — an unsigned integer of with 8.

```
newtype CAddress = CAddress Word8
```

The concrete memory addresses will be used by the simulation, control-flow analysis and symbolic execution backends. However, the symbolic execution backend will also require a symbolic representation of memory addresses to implement the semantics of the memory-indirect load instruction. Memory-indirect data access require symbolic address representation since it may introduce symbolic variables into the address expressions by converting the values stored in memory into addresses. Therefore, we define the general `Address` in the following way:

```
newtype Address = MkAddress (Either CAddress Sym)
```

An `Address` is either a concrete memory address — an unsigned integer of width 8, or a general symbolic expression which may contain symbolic variables. The expressions may also contain the `SAdd` and `SSub` constructors, with one of their arguments being a concrete offset constant.

In order to convert between integers and memory addresses, we define a type class of values that can be interpreted as addresses:

```
class Addressable a where
    toAddress :: a -> Maybe Address
    fromAddress :: Address -> a
```

and create the appropriate instances for the concrete addresses and values, and symbolic values.

We will refine the memory model by giving an interpretation to symbolic addresses in further sections, describing the specific backends.

5.2.5 Instruction and program semantics

In section 3.3.2, we have presented the semantics of several REDFIN instructions in terms of fine-grained state transformers. We classified the instructions according to their dataflow, classified by the `control` argument of the `FS` type. The instruction of the REDFIN ISA fall into the following categories of that classification:

- Linear dataflow and the `Functor` class:
- Static tree dataflow and the `Applicative` class:
- Selective tree dataflow and the `Selective` class:
- Dynamic dataflow and the `Monad` class:

Besides the semantics of the individual instructions, a semantics of an ISA must include the fetch-decode-execute loop, that enables executing complete programs, rather than just sequences of instructions. In the coarse-grained semantics (section 3.3.1), we defined the state transformers T_{fetch} and T_{inc}, which represent fetching the next instruction from program memory and incrementing the instruction counter. As it turns out, the T_{fetch} transformer is inherently monadic and is not much different from the semantics of the memory-indirect load instruction `LoadMI` (see section 3.3.2.4). Therefore, formulating this transformer as a fine-grained one brings no benefit: its semantics will depend on the whole program memory.

The fact that instruction fetching must remain coarse grained does not, however, hinder the amenability of programs for static analysis if we employ one trick. We would like to abstract away the instruction-fetching transformer T_{fetch} and treat the programs as if they were sequences of instructions. It turns out we can do that, if we pre-process the programs into a common intermediate representation used in compiles: a control-flow graph (CFG). A control-flow graph comprises basic blocks of code as nodes and control transfers, i.e. jumps,

as edges. Within a basic blocks there cannot be a jump, besides the last instruction in the block. The control flows into the first instruction of the block and can only flow out from the last.

5.3 Symbolic execution

We have carefully designed the formalism of the fine-grained state transformers in such a way so the ISA semantics implemented with them could be formulated independently of its interpretation. As we have seen earlier in this chapter, the type `FS` of fine-grained state transformers is parameterised by a computational context `f` and by the two callbacks `read` and `write` that govern the access to the context. These parameters are what determines the *interpretation* of the transformer. In this section, we present the interpretation which gives us a symbolic execution engine for REDFIN ISA.

The goal of this section is twofold: (i) we describe the design and implementation of the symbolic execution backend for fine-grained state transformer semantics of REDFIN ISA; (ii) we discuss the benefits this design provides over the coarse-grained symbolic execution framework described in chapter 4.

5.3.1 Memory representation

In previous sections we have discussed the type classes 5.2.2 that provide the shallowly-embedded values and the data types 5.2.3 that represent the deeply-embedded concrete and symbolic values and have instances of the interface type classes. We, however, mostly omitted the representation of the memory locations that will store these values during simulation and symbolic execution. We discuss the memory representation in detail here.

5.3.1.1 Memory configuration of REDFIN

The REDFIN ISA v2 follows the convention of Harvard Architecture and specifies two separate address spaces for data and program memory. The figure 5.6, an excerpt from the REDFIN v2 data sheet, displays the blocks of the core. In this work, our central concern is the instruction handler and the ALU. Note that we do not model the memory scrubber and

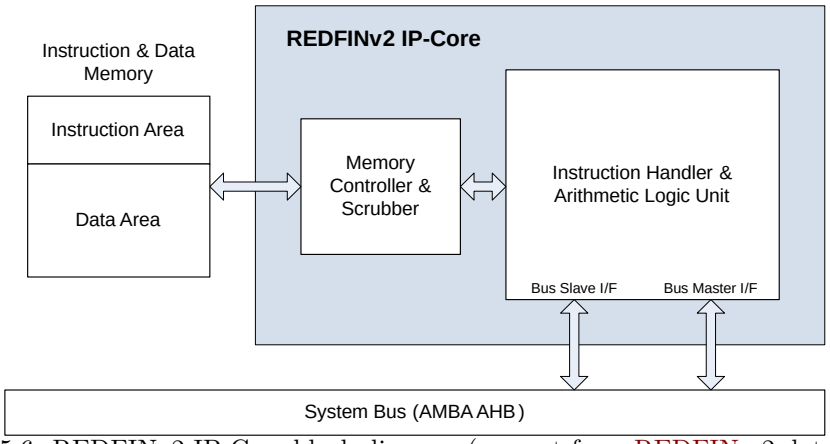


Figure 5.6: REDFINv2 IP-Core block diagram (excerpt from REDFIN v2 data sheet)

the system bus access. Incorporating the bus into the model is an interesting opportunity for future work.

The REDFIN core VHDL design is parameterised over the bit width of data words, denoted ABW, ranging form 8 to 64 bits. In our model, we fix the ABW to be 32 bits, which is the default value. The instruction code width is not configurable in the ISA and is set to 16 bit. For simplicity, we also fix the address space of the data memory to be 256 words and the program memory address space to 256 instructions. These limits are enough for our case-studies and can be extended if required.

On core initialisation, all location in the data memory will be set to 0. The program memory is initialised with zeroes too, and program subroutines are written to it by an external bus master. We do not model this behaviour, and assume the program memory to be containing the single subroutine to be executed, terminated by the `halt` instruction.

In REDFIN v2, a frame pointer register has been added to enlarge the addressable space of the data memory. We, however, keep the flat addressing mode used in REDFIN v1 in our model for simplicity. Extending the model to support the frame pointer addressing mode is considered for the future version of the model.

5.3.1.2 Concrete and symbolic memory addresses

We present the Haskell data types and related functions that implement the memory model of the verification framework. We omit some details, specifically the parts of the definitions that specify type class instances derivation for brevity.

We represent a concrete data memory address as a 8-bit unsigned integer, wrapped into a type constructor for additional type safety:

```
newtype CAddress = CAddress Word8
```

A symbolic memory address is either a concrete one, or an arbitrary symbolic expression:

```
newtype Address = MkAddress (Either CAddress Sym)
```

The **Address** type needs several type class instances. We only give one of them as an example here and discuss it in some detail further:

```
-- | Embed a concrete address
literal :: CAddress -> Address
literal a = MkAddress (Left a)

instance Num Address where
    fromInteger x = literal (fromInteger x)
    _ + _ = error "Address.Num: + is not defined"
    _ - _ = error "Address.Num: - is not defined"
    _ * _ = error "Address.Num: * is not defined"
    abs _ = error "Address.abs: abs is not defined"
    signum _ = error "Address.Num signum is not defined"
    negate _ = error "Address.Num: negate is not defined"
```

This instance enables us to use Haskell's native integer literals as concrete addresses, which is very useful in user-facing specifications. Not that we *forbid* arithmetic operations on addresses by triggering a runtime error. Instead, the addresses need to be *explicitly* cast to an arbitrary symbolic expression (of type **Sym**) for arithmetic manipulation:

```
class Addressable a where
    toAddress :: a -> Maybe Address
    fromAddress :: Address -> a

instance Addressable Sym where
    toAddress = Just . MkAddress . Right
    fromAddress (MkAddress a) =
        case a of
            Left (CAddress concrete) -> SConst (CWord $ fromIntegral concrete)
            Right sym -> sym
```

The **Addressable** type class hadles this casting. By this, we provide strong guarantees via Haskell's type system: we make sure that the **Num** instance of the symbolic expressions cannot be directly applied to symbolic addresses.

The `Address` type provides the representation for a single memory location. Remember, that we structure the semantics of the REDFIN ISA as a fine-grained state transformer. A data memory address becomes a specific kind of a location in the state, not very much different from other locations. We discuss the representation of states in the next section.

5.3.2 Symbolic execution context

We represent the state of the ISA as a context — a collection of typed variable bindings and related metadata. The bindings are exactly the type of keys we have been designing the semantics to work with in section 5.2. When evaluating a fine-grained state transformer, the context changes according to the transformer's specification. Concrete execution applies the effects of a transformer directly and produces a single resulting context on successful termination. Symbolic execution, on the other hand, produces a (possibly infinite) binary tree of contexts: a branch in the tree is created by a branching operation in the state transformer.

Conceptually, the representation of state is not very much different from the set-theoretic presentation we have seen in section 3.3.1. However, we consider the implementation details of the verification framework to be the focus of this thesis; therefore, we discuss the `Context` data type (figure 5.7) in detail here.

The context should contain the bindings and some additional data to support symbolic execution. We keep a map of program variables which we call `_store`: it maps variable names to symbolic expressions these variable are pointing to. This data structure will be essential for handling the semantics of indirect memory loads.

The last three components are only required for symbolic execution. The `_pathCondition` is a symbolic boolean expression that, for the current context, encodes its reachability from the initial context. A particular context is reachable if its path condition is a satisfiable formula. The `_constraints` are named symbolic boolean expressions that represent the user-specified or generated verification conditions, for example, restrictions on the ranges of program variables. Finally, the `_solution` will be filled in upon calling the external solver to check satisfiability of the path condition conjoined with the constraints. If the resulting conjunction is satisfiable, then this particular context is reachable; otherwise it represents an impossible (under the current constraints) state of execution and its children

states should not be explored.

```
data Context a = MkContext
    { -- | keys (like register names, memory cells) mapped to their (symbolic) values
      _bindings :: Map.Map Key a
    , -- | A store used for tracking symbolic points-to
      _store :: Map.Map Text a
    , -- | a boolean formula which must be satisfiable for this state to be reachable
      _pathCondition :: a
    , -- | a list of named boolean formulas, mostly used as preconditions
      --   and conjoined with _pathCondition
      --   when checking reachability
      _constraints :: [(Text, a)]
    , -- | a response from a solver, usually regarding
      --   satisfiability of _pathCondition s && conjoin (_constraints s)
      _solution :: Maybe SMTResult
    }
```

Figure 5.7: The data type representing the ISA states

Note that the type `Context` presented in the figure 5.7 is parameterised over the value type. We would usually instantiate it with the type `Sym` of symbolic expressions. We instantiate it with the type `Concrete` of concrete values in the efficient simulation backend, but we do not discuss this backend in the thesis. Potentially, the type variable `a` can be instantiated with a type representing some abstract domain for abstract interpretation. We leaved this avenue of research for future work.

As we noted earlier, the `_bindings` map associates values of type `Key` to expressions. We can extract the tracked memory locations form a context like with a selector function:

```
dumpMemory :: Context a -> [(Address, a)]
dumpMemory = catMaybes . map (uncurry getAddr) . Map.assocs . _bindings
  where
    getAddr (Addr a) v = Just (a, v)
    getAddr _ _ = Nothing
```

Similar selector functions are implemented to extract other parts of the architecture: registers, flags, instruction counter, etc.

5.3.2.1 Example: initial state of an array summation program

To perform simulation or symbolic execution of a program, we first need to specify the initial context. As an example, let us consider the initial state of a case-study program that

that calculates the sum of a static array of integers. We first present the state in a generic set-theoretic notation and then give the Haskell formulation of the corresponding context.

$$
\begin{aligned}
&\exists n : 0 \leq n < 253, \\
&\exists X_{min}, X_{max}, \\
&\forall i : 0 \leq i \leq n, \exists x_i : X_{min} \leq x_i \leq X_{max}
\end{aligned}
$$

Figure 5.8: A set-theoretic specification of an array of length n

The formula in figure 5.8 specifies an array of a fixed length ranging from 0 to 252 elements. The trailing memory locations will be used as for temporary variables by the program and thus need to kept unused. The array elements x_i are constrained to be in the range between X_{min} and X_{max}. The corresponding context specification is more verbose and includes the initialisation of other requires ISA locations, such as registers, flags and the program memory (with the program assembled from the source code shown on the right-had-side):

```
mkInitCtx :: Int32 -> Int32
          -> Int32 -> Context Sym
mkInitCtx n xMin xMax =
  let (vars, constrs) =
      array 1 n (\x -> (SGt x xMin) &&&
                       (SLt x xMax))
  in MkContext
    { _bindings =
        Map.fromList $
            [ (IC, 0)
            , (Reg R0, 0)
            , (Reg R1, 0)
            , (Reg R2, 0)
            , (Addr 0, SAny "n")
            , (Addr 253, 0)
            , (Addr 255, 1)
            ] ++ vars ++
            [ (F Halted, false)
            , (F Condition, false)
            , (F Overflow, false)
            ] ++
            mkProgram sumArrayLowLevel
    , _store = Map.empty
    , _pathCondition = true
    , _constraints = constrs ++
        [ ( "n"
          , (SGt (SAny "n") (-1)) &&&
            (SLt (SAny "n") (n + 1))
          ) ]
    , _solution = Nothing
    }
```

```
1  sumArrayLowLevel :: Script
2  sumArrayLowLevel = do
3      let pointer = 0
4          sum = 253
5          array_start = 255
6      let r0 = R0; r1 = R1; r2 = R2
7      ld r1 pointer
8
9      -- compare the pointer variable
10     -- to array_start
11     "loop"  cmplt r1 array_start
12     -- if pointer == array_start
13     -- then terminate
14     goto_ct "end"
15
16     ldmi r2 pointer
17     add r2 sum
18     st r2 sum
19     sub_i r1 1
20     st r1 pointer
21
22     goto "loop"
23     "end"  ld r0 sum
24     halt
```

Figure 5.9: The initial context for the array summation program and the program's source code

The most important parts to look at here are `_bindings`, `_pathCondition` and `_constraints`. The `_bindings` will contain the symbolic variables representing the array elements at locations 1 through n. The location 0 contains the variable n — the length of the array. We put the literal 1 in the location 255 to be used as array index. Other locations, such as the three registers and flags are explicitly initialised with 0 and $\bot$. Importantly, the path condition of the initial context is initialised with $\top$, meaning that the initial context is always reachable. Finally, the `_constraints` list will contain the constraints on the array elements and also the one on n.

So far we have discussed the syntactic machinery that allows specifying the individual states of execution: configuration of registers, flags and memory. Symbolic execution produces trees of contexts, which we call symbolic execution *traces*. We discuss them in the

next section.

5.3.3 Execution traces

The program on the right-hand-side of figure 5.9 contains a loop that traverses the array in reverses order and calculates the sum of the arrays elements. The instruction on line 14 is a conditional branch, which transfers the control to the label "end" on line 23 if the index in register `r1` reaches the start of the array. Symbolic execution of this program will produce a tree-shaped trace that encodes a class of concrete executions. Concrete execution scenarios can be obtained by instantiating symbolic variables that occur in the trace. Let us consider an example symbolic trace (figure 5.10) with the array length constrained to be $0 \leq n \leq 2$: at most two elements. The shape of traces will be similar for larger values of n, and we consider this small example purely for its didactic purpose. Note that the logic related to sum calculation is irrelevant for termination analysis. We, thus, exclude these instructions (lines 16, 17, 18, 20 and 24 after the label "end") from the trace for brevity.

The program contains a single loop that iterates through the array in the reverse order of indices and accumulates the sum. The array index is maintained in the register `r1` and the program is halted when the index reaches zero. Before starting the execution, the path condition is initialised with $0 \leq n \leq 2$, declaring the range of array lengths we are interested in. The comparison is performed by the `cmplt` opcode on line 11, which is followed by a branch in the execution tree: the left subtree conjoins the positive term $n < 1$ to the path condition, and the right subtree conjoins the term's negation, $\overline{n < 1}$. The left subtree represents by convention, the path of execution when the jump condition is true and thus the jump is performed; in this case, leading to the successful termination of the program with the `halt` instruction on line 24. The right subtree is feasible for $n = 1 \vee n = 2$ and represents the loop body which decrements the array index and again leads to the check of the termination condition on line 11. The process proceeds iteratively, and the path condition is conjoined with the terms encoding reachability of the program states. Finally, when the index reaches $n - 2$, it is no longer feasible to continue execution, since the path condition in the right-most subtree becomes unsatisfiable.

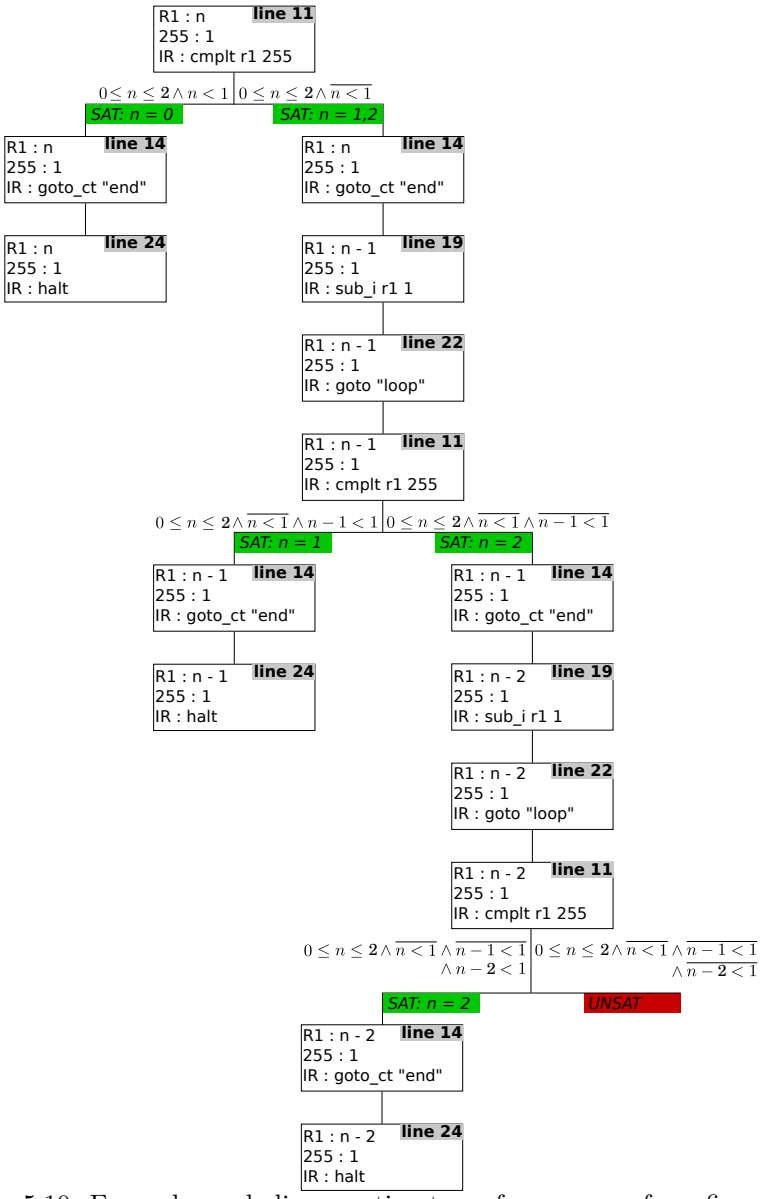


Figure 5.10: Example symbolic execution trace for program from figure 5.9

Symbolic execution traces are the basis for verification. They contain information about a class of executions of a program and are the source of answers to all our questions about the program's behaviour. However, the traces alone are not enough. By exploring a trace, we can *observe* the possible behaviours, but we, however, do not currently have a way of *asking questions* about these behaviours.

5.3.4 Specification syntax

In order to verify that a symbolic execution trace produced by a program only encodes *good* behaviours, we require a *language* to express what is good. In this thesis, we are interest in *safety* and *functional correctness* properties of REDFIN programs. For example, we would like to make sure that a certain program never causes an integer overflow, or that a successfully halted program always returns a result obeying a precise mathematical specification.

Ultimately, we require a language that allows us to formulate *invariants* that programs must obey. Informally, an invariant is a propositional formula that holds in every state the program could reach. We will give a model for the invariants in terms of symbolic execution traces in the following section 5.3.5, and justify in more detail its choice. For now, we restrict ourselves to the presentation of syntax.

5.3.4.1 Invariant syntax

The invariants need to specify properties over possible program executions. The syntax is two-level, with the top-level terms operating over whole program executions, and the bottom level terms, or atoms, referring to the ISA states. The top-level terms are conjunctions of expressions that universally-quantify over program executions. We borrow the notation of the global modality, $\mathcal{G}$ from Computation Tree Logic (CTL)[89] to emphasise that the enclosed atomic formula must hold *always*, i.e. for any feasible ISA state in every execution of the program.

$$
\begin{aligned}
\langle Invariant \rangle ::= {} & \forall \mathcal{G}\ \langle atom \rangle \\
| {} & \langle Invariant \rangle\ \wedge\ \langle Invariant \rangle
\end{aligned}
\qquad\qquad
\begin{aligned}
\langle atom \rangle ::= {} & \langle key \rangle \\
| {} & \langle sym \rangle \\
| {} & \neg\ \langle atom \rangle \\
| {} & \langle atom \rangle\ \vee\ \langle atom \rangle \\
| {} & \langle atom \rangle\ \wedge\ \langle atom \rangle \\
| {} & \langle atom \rangle\ =\ \langle atom \rangle \\
| {} & \langle atom \rangle\ >\ \langle atom \rangle \\
| {} & \langle atom \rangle\ <\ \langle atom \rangle
\end{aligned}
$$

Figure 5.11: Syntax of invariants and underlying atomic propositions

The atomic propositions, in addition to the usual boolean and relational connectives, include the first two productions for embedding a state key and a literal symbolic expression. These base productions allow referring to the keys of an ISA state, i.e. registers, memory locations, etc. and also referencing symbolic expressions which may contain symbolic variables.

The syntax of invariants could be easily reduced to just the atomic propositions. However, we argue that retaining the syntactic separation between invariants and the atomic proposition allows for better ergonomics. Additionally, this separation paves the path to extension of the logic to CTL, if ever deemed necessary.

5.3.5 Invariant semantics

The invariants (figure 5.11) presented in the previous section are intended to describe the safety and functional correctness properties of REDFIN programs. To perform verification of such properties, we need a *model* of the logic. We choose to give a model in terms of the symbolic execution traces described in section 5.3.3 and quantifier-free first-order logic formulas over the theory of bit-vectors, `QF_BV`, employed by the SMT-LIB [61] solver interface standard. The `QF_BV` theory contains the necessary primitives to reason about fixed-size bit-vectors, and is perfectly suited for ISA-level program verification. By giving a model in such a way, we effectively reduce the satisfiability problem of invariants to satisfiability modulo `QF_BV` theory, which is supported by many major SMT solvers.

The model is implemented as a translation procedure in our verification framework for REDFIN programs. The implementation details of this procedure lay out of scope of this thesis, and we, therefore, only give a high-level overview here.

5.3.5.1 Interpreting invariants over symbolic execution traces

As the syntax of invariants (figure 5.11) comprises two parts: (1) the top-level path-quantified productions, and (2) the underlying atomic formulas. The semantic interpretation is performed for these two constructions. We start with the atomic formulas, that are interpreted at one particular context that represents a single state. We then plug in that interpretation into the semantics of the top-level path quantifier.

The translation of the atomic proposition is syntax-directed recursive function that eliminates the productions of the $\langle atom \rangle$ grammar (figure 5.11). The first two productions are base cases: a key is extracted directly from the current state, and the symbolic expression is embedded directly. The logical and relational connectives are processed recursively. We give an excerpt of the implementation that translated an atomic proposition to a semantics symbolic expression following the aforementioned procedure:

```
-- | Evaluate an atomic formula at the given state
evalAtom :: Atom -> Context Sym -> Sym
evalAtom atom ctx = go true atom
  where
    go acc = \case
        AKey k -> getBinding k ctx
        ASym s -> s
        ANot p -> acc &&& not (evalAtom p ctx)
        AAnd x y -> acc &&& (evalAtom x ctx &&& evalAtom y ctx)
        AOr x y -> acc &&& (evalAtom x ctx ||| evalAtom y ctx)
        AEq x y -> acc &&& (evalAtom x ctx === evalAtom y ctx)
        AGt x y -> acc &&& (evalAtom x ctx `lt` evalAtom y ctx)
        ALt x y -> acc &&& (evalAtom x ctx `gt` evalAtom y ctx)
```

conjunctions, represented by `AAnd` constructors, translate to the conjunction operation (`&&&`), disjunctions to disjunctions, etc. The bindings of the context keys (`AKey` constructors) are interpreted by extracting the value associated with that key from the context; in a well-formed formula, the key-binders will always be direct children of relational constructors `AEq`, `AGt` and `ALt`. In case an embedded symbolic expression (`ASym` constructor) is encountered, it is interpreted without change.

The interpretation of the top-level syntax requires interpreting two cases:

- the $\forall\mathcal{G}$ (for all, globally) quantifier translates into a conjunction of its atomic child formula over the *whole trace*;
- the path-level conjunction is the only recursive production that is recursively translated to the semantic conjunction of its children.

We do not present the complete implementation of the translation to avoid cluttering the presentation with too many low-level details.

5.3.5.2 Interfacing with an off-the-shelf SMT solver

After invariants have been eliminated into first-order formulas over `QF_BV`, we prepare them for SMT solving. To check the validity of a formula we check satisfiability of its negation. Since the formulas we construct will always be large conjunctions of terms, the negation, by De Morgan's law, will be a disjunction. We take advantage of multicore processors by we splitting the disjunction into sub-terms and invoking several instances of the solver for them separately. If any solver returns `SAT`, then we have a counterexample to the initial formula's validity. Otherwise, all solvers return `UNSAT`, and thus the formula is valid. A solver may also return `UNKNOWN`, indicating that it got stuck. In this case, the initial specification may be reduced to construct a less difficult property.

5.3.6 Case-study: stepper-motor control program

Many programs targeting REDFIN share the distinctive feature of the energy estimation program considered in section 4.3.1, i.e. the existence of an *upper bound on execution time*, since their termination does not depend on input data. However, other programs may have a loop which is guarded by a termination condition that involves computation considering the input parameters of the program, thus making the loop *unbounded*.

Presence of unbounded loops makes program verification by symbolic execution considerably harder [54, p. 50:20], since the number of program execution paths becomes infinite. In this section we consider an example of a control program that drives a stepper motor and verify several of its essential safety properties.

5.3.6.1 Motor Control Algorithm

Stepper motors are often deployed as parts of antenna and solar panel pointing units in space satellites. We consider a program for controlling a motor with one degree of freedom. The algorithm takes three input parameters:

- $dist$ — the distance to move the motor,
- v_{max} — the maximal permitted velocity,
- a_{max} — the maximal permitted acceleration

and computes a series of displacement and velocity values that will be used to move the motor. Since the algorithm is designed for controlling a stepper motor, the calculations happen in *discrete time*, i.e. every iteration of Algorithm 1 corresponds to a time interval; thus the deceleration (i.e. braking) distance is computed as

$$s_{decel} = a_{max} \cdot \frac{decel\_steps \cdot (decel\_steps + 1)}{2},$$

where $decel\_steps = \frac{v}{a_{max}}$ is the number of decelerating iterations needed for a full stop.

The conditional statement on line 9 decides whether to accelerate, to keep the velocity, or to decelerate; see Fig. 5.13 for example plots of velocity and distance travelled against time. The spike at the bottom-right of the velocity plot illustrates the edge case covered by the conditional statement on line 18: if the velocity is zero, but the target distance has not yet been reached, the motor must be moved further.

Algorithm 1 Motor Control Algorithm

Input: $dist$, v_{max}, a_{max}

1: $s \leftarrow 0$
2: $v \leftarrow 0$
3: **while** *true* **do**
4: $\quad decel\_steps \leftarrow v/a_{max}$ ▷ Compute deceleration distance
5: $\quad s_{decel} \leftarrow a_{max} \cdot decel\_steps \cdot (decel\_steps + 1)/2$ ▷ based on the current velocity
6: $\quad$ **if** $decel\_steps \cdot a_{max} \neq v$ **then**
7: $\quad\quad s_{decel} \leftarrow s_{decel} + v$
8: $\quad v_{next} = min(v_{max}, dist, v + a_{max})$
9: $\quad$ **if** $s + s_{decel} + v_{next} \leq dist$ **then**
10: $\quad\quad v \leftarrow v_{next}$ ▷ Accelerate
11: $\quad$ **else if** $s + s_{decel} + v \leq dist$ **then**
12: $\quad\quad v \leftarrow v$ ▷ Keep velocity
13: $\quad$ **else** ▷ Decelerate
14: $\quad\quad$ **if** $v > decel\_steps \cdot a_{max}$ **then**
15: $\quad\quad\quad v \leftarrow decel\_steps \cdot a_{max}$
16: $\quad\quad$ **else**
17: $\quad\quad\quad v \leftarrow v - a_{max}$
18: $\quad$ **if** $v = 0$ **then**
19: $\quad\quad$ **if** $s \neq dist$ **then** ▷ Accelerate again to reach target
20: $\quad\quad\quad v \leftarrow min(dist - s, a_{max})$
21: $\quad\quad$ **else**
22: $\quad\quad\quad break$ ▷ Terminate execution
23: $\quad s \leftarrow s + v$

To deploy Algorithm 1 to REDFIN, it has been manually implemented in REDFIN assembly. The resulting assembly program comprises 85 lines of code and closely mirrors the high-level pseudocode. Figure 5.12 shows a fragment of the program's symbolic execution trace that corresponds to the decision whether to accelerate, keep the velocity, or decelerate the motor.

Figure 5.12: Symbolic execution trace of a code fragment with conditional branching.

The decision is performed by computing the resulting total distance travelled from start to stop, based on the action taken in the current time step. First, the total distance is

computed if the motor were to accelerate for one more time step and then decelerate in the subsequent time steps. If the computed total distance is less than or equal to *dist*, the decision to accelerate is committed. Otherwise, the algorithm checks whether the targeted distance can be met by maintaining the current velocity for one more time step. If even that would cause an overshoot, the decision for immediately commencing deceleration is taken. Fig. 5.13 illustrates this decision process by plotting the velocity and distance over time for a specific simulation run.

Figure 5.13: Velocity (v) and distance travelled (s) plotted against time (t)

5.3.6.2 Program termination and arithmetic safety

Mission-critical control programs are often specifically designed to satisfy a number of safety properties. Recall, the REDFIN core is intended to be deployed as part of subsystem of a space satellites. The satellite will usually have a conventional processing core which would serve as a master node, controlling the subsystems. In case of REDFIN the master node will trigger execution of subroutines, such as Algorithm 1. The termination of any REDFIN program will, by design, be controlled by the master node. We therefore verify *partial* correctness. i.e. prove that the program satisfies the invariant if it terminates.

We now describe the process of using the verification framework developed in this thesis to verify that the arithmetic operations involved in the algorithm Algorithm 1 do not trigger integer overflow and division by zero exceptions.

Arithmetic safety

Mission-critical control programs must be verified of absence of arithmetic errors under well-defined constraints. In section 4.3.1, we have verified the arithmetic safety of an energy estimation program. We now formulate a similar invariant:

$$\forall \mathcal{G}\ (Overflow = \bot\ \wedge\ DivisionByZero = \bot) \tag{5.1}$$

which states that "for all execution paths, the flags `Overflow` and `DivisionByZero` should be always unset".

To check the invariant (5.1), we need to employ the methodology described in section 5.3.5: symbolically execute the program from a well-formed initial state and then

evaluate the invariant property on that trace. However, in the specific case of Algorithm 1, we need to employ an additional technique.

One may notice that the control-flow structure of Algorithm 1 is very simple: it is an iterative process governed by one top-level unconditional **while** loop (line 3) and a single break point (line 22). We can be sure that the loop will be executed at least once, but, unfortunately, we do not have any syntactic termination guarantees. The termination condition of this loop depends on symbolic program variables, specifically v, s and $dist$. While the value of $dist$ is an input parameter of the algorithm and is fixed during symbolic execution, the other two variables will be modified at every iteration. In other words, the amount of iterations of the loop cannot be statically predicted and is potentially infinite [1]. Automatic verification of unbounded loops remains out of the scope of this thesis. We, however, can still perform verification of some safety properties of Algorithm 1 by considering a generic iteration of the top-level **while** loop.

5.3.6.3 Loop Invariant Verification

We mitigate the difficulties that the unbounded loop of Algorithm 1 poses to symbolic execution-based verification by considering an execution of an arbitrary iteration of this loop. By considering only one iteration, we unfortunately cannot verify functional correctness of the whole algorithm, but we can still achieve verification of safety properties like (5.1) and of more domain-specific ones.

In order to ensure that the motor will not introduce disturbances and will not lead the whole unit out of its normal mode of operation, the velocity and acceleration of the motor must be kept within safe limits. This verification condition is motivated by the correctness requirements of the whole space satellite unit.

More formally, the verification condition means that at any iteration t of the loop the values of the expressions v^t, velocity, and $|v^t_{next} - v^t|$, acceleration, must never exceed the parameters v_{max} and a_{max}, respectively.

This property is the loop invariant for the motor control program which ensures that velocity and acceleration always stay within their safe bounds. We formalise this invariant

[1] If we consider the whole integer domain. If we restrict ourselves to machine integers, the potential amount of iterations is bounded, but is still large.

as follows:

$$\forall \mathcal{G}\ v_{max}\ a_{max}\ v\ v_{next},\ v \leq v_{max} \wedge |v_{next} - v| \leq a_{max} \tag{5.2}$$

the property quantifies over all program states via the "all-globally" path quantifier $\forall \mathcal{G}$ and states that any state the stability condition must hold, i.e. the current velocity and acceleration must never exceed the maximum allowed ones.

Recall that we are allowing the top-level syntax of invariants to include conjunctions of path-quantified atomic formulas. We thus can combine the stability condition (5.2) with the one formalising arithmetic safety (5.1):

$$\begin{aligned} &\forall \mathcal{G}\ v_{max}\ a_{max}\ v\ v_{next},\ v \leq v_{max} \wedge |v_{next} - v| \leq a_{max} \\ \wedge\ &\forall \mathcal{G}\ (Overflow = \bot\ \wedge\ DivisionByZero = \bot) \end{aligned} \tag{5.3}$$

this combined property reads as follows:

- at any step, the velocity and acceleration must stay within bounds, **and**
- at any step, the `Overflow` and `DivisionByZero` flags must be unset

By verifying the property (5.3) for an arbitrary iteration of the loop, we will be able to conclude that any execution of the whole program satisfies it.

Chapter 6

Tool support

Alongside the research contribution of this thesis, we also provide a prototype software suite for developing, specifying and verifying REDFIN programs. The suite includes two libraries that describe the semantics of REDFIN ISA in terms of monadic state transformers and fine-grained store, and a Web-based IDE.
We propose two workflows to interact with the software suite:

- The libraries provide a module that could be used as an EDSL for developing REDFIN programs, specifying their behaviour and proving conformance to the specifications by symbolic execution.
- The IDE could be used for the same activities as the EDSLs, and additionally for interactive exploration of program symbolic execution trees and states.

We now describe the architecture of both the EDSL and the IDE, and provide sample interaction sessions.

6.1 Redfin Assembly

In this section we discuss the implementation of a simple assembly EDSL which facilitates programming the REDFIN sequencer by providing human-readable mnemonics for the instructions and an additional syntactic feature of labels for program locations with support

for forward and backward conditional jumps. In the implementation of the assembler we showcase an advanced feature of Haskell's type system, i.e. the support for type-level natural numbers, to achieve compile-time correctness verification of embedding opcodes and instruction arguments into the instruction code, thus eliminating possible bitvector width-related bugs.

The assembler translates a human-readable script of instruction mnemonics and goto-statements into a sequence of machine codes; to achieve this, the assembler performs the following two passes over the input script:

1. First, the assembler accumulates the labels and resolves them into pairs of the label's name and their corresponding instruction counter location. For example, if a label `"loop"` is located at line 42 of the script, it is going to be resolved into a pair `("loop", 42)`. The pairs are then arranged into a table for use in the second pass;

2. Second, the assembler translates every mnemonic in the input script into the corresponding instruction code and substitutes the goto-statements with the appropriate jump instructions with the offsets resolved according to the label table. The resulting sequence of machine codes is ready for being plugged into the REDFIN model.

The assembler is a stateful computation that needs to keep track of the current value of the instruction counter, the labels table and the machine code being constructed: the **`Script`** type (fig. 6.1) denotes exactly that by using the already familiar state monad abstraction.

```
data AssemblerState = MkAssemblerState
  { program            :: [(InstructionAddress, InstructionCode)]
  , labels             :: Labels
  , instructionCounter :: InstructionAddress
  }

type InstructionAddress = SymbolicValue (WordN 10)
type InstructionCode    = SymbolicValue (WordN 16)

type Labels = Map String InstructionAddress

type Script = State AssemblerState ()
```

Figure 6.1: Assembly EDSL types

The individual instruction mnemonics are now expressed as computations of type **`Script`**,

which construct instruction codes as bitvector concatenation of the opcode and the arguments, and then append the result to the machine code sequence. Figure 6.2 demonstrates the mnemonics for addition, loading of a signed immediate value into a register and a conditional jump. The (#) infix operator ensures the compatibility of bitvector sizes, i.e. that the opcode (of length 6) and the instruction arguments add up to a bitvector of length 16 (the instruction code length); moreover, since some instructions expect signed immediate arguments (in two's complement representation), we must convert the arguments into unsigned words with `fromSigned` to persuade GHC's type system that we know what we are doing. Note that `fromSigned` only changes the type of the argument and does not actually change the underlying value. Later in the pipeline, the instruction decoder will perform the reverse operation.

```
add :: Register -> MemoryAddress -> Script
add rX dmemaddr = instruction (0b000100 # rX # dmemaddr)

ld_i :: Register -> UImm8 -> Script
ld_i rX uimm = instruction (0b101111 # rX # uimm)

jmpi_cf :: SImm10 -> Script
jmpi_ct simm = instruction (0b110001 # fromSigned @10 simm)

instruction :: InstructionCode -> Script
instruction c = do
    s <- get
    let ic = instructionCounter s
    put $ s { program = (ic, c):program s
            , instructionCounter = ic + 1}
```

Figure 6.2: Example mnemonics

The types of `fromSigned` and (#) (Fig.6.3) are lightweight specifications for the behavour of these functions, containing information on signess and size of the bitvector arguments. The type of `fromSigned` constrains the type variable `n` to be a non-zero natural number[1] and ensures that if the input is a signed bitvector of size `n`, then the output will be an unsigned bitvector of the same size. The other function, (#), implements bitvector concatenation, and operates on both signed and unsigned betrverctors of non-zero length. The function's type ensures that the bitvectors have matching signess and that the size of the result is exactly the sum of the arguments' sizes. The `SymVal` typeclass from `Data.SBV` abstracts

[1] using the type classes from `GHC.TypeNats` module

the types `IntN`, `WordN` and other, non-bitvector, types of symbolic values.

```
fromSigned :: (KnownNat n, IsNonZero n) => SBV (IntN n) -> SBV (WordN n)

(#) :: (IsNonZero n, IsNonZero m, KnownNat n, KnownNat m
       , SymVal (bv n), SymVal (bv m)) =>
       SBV (bv n) -> SBV (bv m) -> SBV (bv (n + m))
```

Figure 6.3: Type-safe bitvector combinators

```
type Opcode = SymbolicValue (WordN 6)

type UImm8 = SymbolicValue (WordN 8)

type SImm10 = SymbolicValue (IntN 10)

type Register = SymbolicValue (WordN 2)

type MemoryAddress = SymbolicValue (WordN 8)
```

Figure 6.4: Recap of symbolic types used as mnemonics' arguments

As we saw in this section, embedding an assembly language into Haskell allows for a short and easy-to-maintain implementation which provides some correctness guarantees thanks to the Haskell's type system.

Assembly is great for implementing bespoke performance-tuned algorithms, especially for such resource-constrained environment like spacecraft control. However, we aim our model to be suitable not for programming new procedures, but rather for formal verification of the existing control programs targeting REDFIN. For verification to be possible, there must be means to *specify* the properties to be verified, and the next section describes another EDSL that we use for exactly this purpose: to specify functional properties of REDFIN programs.

6.2 Compiler for a language of expressions

As mentioned earlier in this thesis, REDFIN stands for "REDuced instruction set for Fixed-point and INteger arithmetic"; thus, a major verification task is making sure that we get our arithmetic right. In this section, we address this concern by developing the Expression EDSL — a simple language for arithmetic expressions that compiles to REDFIN assembly

and can be used for both implementing REDFIN programs and as a specification language for describing the functionality of hand-written assembly in a clear and concise way. The language will be used in conjunction with the facilities provided by SBV.

Consider the following simple spacecraft control task:

> Let t_1 and t_2 be two different time points (measured in ms), and p_1 and p_2 be two power values (measured in mW). Calculate the estimate of the total energy consumption during this period using linear approximation, rounding down to the nearest integer:
>
> $$energyEstimate(t_1, t_2, p_1, p_2) = \left\lfloor \frac{|t_1 - t_2| * (p_1 + p_2)}{2} \right\rfloor.$$

This task looks too simple, but in fact it has a few pitfalls that, if left unattended, may lead to the failure of the space mission. Examples of subtle bugs in seemingly simple programs leading to a catastrophe include 64-bit to 16-bit number conversion overflow causing the destruction of Ariane 5 rocket [3] and the loss of NASA's Mars orbiter due to incorrect unit conversion [4].

Let is describe the infrastructure required to write down formulas like this one in a subset of Haskell that we will be compiling to REDFIN assembly. The Expression EDSL comprises three roughly-defined components:

- Deeply-embedded abstract syntax of expressions
- Shallowly-embedded concrete syntax which allows to re-use Haskell's arithmetical notation
- A compiler from the deeply-embedded syntax into REDFIN assembly

6.2.1 Abstract Syntax of Expressions

The standard way of representing an abstract syntax in Haskell is by encoding the grammar as an algebraic data type. For our simple case, we need a type with four constructors (fig. 6.5): (i) integer literals, which will be compiled into signed immediate arguments; (ii) variables referring stored at a particular memory address; (iii) Binary operators, which

abstract instructions such as `add` and `mul`; and (iv) the last constructor to represent the only unary operation `abs`.

```
data Expression = Lit Literal
                | Var Variable
                | Bin BinaryOperator Expression Expression
                | Abs Expression

newtype Variable = MkVariable MemoryAddress

newtype Literal = MkLiteral SImm8

type BinaryOperator = Register -> MemoryAddress -> Script
```

Figure 6.5: Abstract syntax of the Expression language

It is possible to construct expressions directly as values of the `Expression` type. For example, an expression $x+1$ may be represented as `Bin add (Var 0) (Lit 1)`, supposing that the variable named x is stored at memory location `0`. However, it is clear that using the abstract syntax directly is cumbersome and counter-intuitive. Fortunately, there is an easy way to enrich the Expression EDSL to make it more convenient.

6.2.2 Reusing Haskell's syntax as concrete syntax

Haskell provides EDSL developers with an easy way to employ its own constructions of arithmetical operations, let-binding and other syntactic constructions via the type class mechanism. By defining the `Expression` type an instance of the `Num` class, which abstracts the arithmetical operations, we obtain a shallowly-embedded domain-specific language (DSL) for constructing expressions:

```
instance Num Expression where
    fromInteger = Lit . MkLiteral . fromIntegral
    (+)         = Bin add
    (-)         = Bin sub
    (*)         = Bin mul
    abs         = Abs
    signum x    = x `Prelude.div` Prelude.abs x
```

Figure 6.6: Shallow embedding of Expression into Haskell

With this embedding, we can reuse Haskell's let-binding to enrich ourselves with named variables for free, and construct the example expressions as `let x = Var 0 in x + 1`.

However fancy we make the syntax, it is still useless without semantics. In the next section, we will describe the translation of the expression terms into REDFIN assembly.

6.2.3 Compiling Expressions to Assembly

To be able to use the Expression EDSL as intended, we need to be able to translate it somehow into REDFIN assembly, or, more precisely, to implement a function of type `Expression -> Script`, which would enable us to perform simulation, testing and symbolic execution of the compiled Expressions just like if they were hand-written REDFIN assembly programs.

As we mentioned already, REDFIN is intentionally designed to be as simple as possible, to facilitate verification. This, however, puts us, as compiler developers, into a constrained position, since the ISA does not provide as with neither stack nor memory allocation. In order to compile a generic Expression into assembly, we need to emulate the stack and a rudiment temporary memory allocation system of only one cell. We will also need to purpose one of the REDFIN's registers for our needs.

```
newtype Temporary = MkTemporary   { fromTemporary :: MemoryAddress }

newtype Stack     = MkStack       { _pointer :: MemoryAddress }

data CompilerEnv  = MkCompilerEnv { _reg   :: Register
                                  , _tmp   :: Temporary
                                  , _stack :: Stack
                                  }
```

Figure 6.7: Embedded compiler infrastructure

6.2.3.1 Stack emulation

A stack is a last in, first out (LIFO) data structure that has numerous applications in computer science: from graph algorithms to programming languages, to low-level data buffers . In computer engineering, the stack usually refers to the *call stack* — a data structure that is maintained by the ISA implementation to keep track of the subprogram calls and their arguments, but can be used for other purposes too. Now, since the REDFIN ISA does not provide subprograms, the call stack is not needed; hence the architecture does not provide one.

To compile a generic Expression, we essential treat the possible arithmetic operations as inline subroutines, hence we need to be able to supply them with their arguments, and emulating a basic call stack is the conventional way to achieve that.

```
push :: Register -> Stack -> Script
push reg (MkStack pointer) = do
    stmi reg pointer
    ld reg pointer
    sub_si reg 1
    st reg pointer

pop :: Register -> Stack -> Script
pop reg (MkStack pointer) = do
    ld reg pointer
    add_si reg 1
    st reg pointer
    ldmi reg pointer

type BinaryOperator = Register -> MemoryAddress -> Script

applyBinary :: CompilerEnv -> BinaryOperator -> Script
applyBinary (MkCompilerEnv reg (MkTemporary tmp) stack) op = do
    pop reg stack
    st reg tmp
    pop reg stack
    op reg tmp
```

Figure 6.8: Embedded compiler infrastructure

The function `applyBinary` (fig. 6.8), expects its two arguments to be located on the stack: placing them there is the responsibility of the compilation function discussed further. We abstract binary operations as computations of type `Register->MemoryAddress->Script` since all binary operations in REDFIN, such as `add`, `mul` and others, require their first argument to be in a register, the second in the memory, and they store the result in the register. To generate the code applying the operation, we pop the arguments from the stack into the register and memory cell provided by the compiler environment, and then inline the code of the operation. The functions `push` and `pop` are the ones emulating the stack: they are implemented in terms of REDFIN assembly mnemonics, and the implementation is automatically inlined into the compiled script.

Before we further discuss the implementation of the compiler, consider an example of a program that adds to numbers (fig. 6.9) with an Expression shown on the left, and the corresponding compiled assembly on the right:

```
addHaskell :: Num a => a -> a -> a
addHaskell x y = x + y

addHighLevel :: Script
addHighLevel =
    let x      = varAtAddress 0
        y      = varAtAddress 1
    in compile (addHaskell x y)
```

```
ld_i 0 253          // pop R0 y
st 0 254            ld 0 254
// R0 ← x           add_si 0 1
ld 0 0              st 0 254
// push x           ldmi 0 254
stmi 0 254          // tmp ← y
ld 0 254            st 0 255
sub_si 0 1          // R0 ← pop x
st 0 254            ld 0 254
// R0 ← y           add_si 0 1
ld 0 1              st 0 254
// push y           ldmi 0 254
stmi 0 254          // R0 ← x + y
ld 0 254            add 0 255
sub_si 0 1          halt
st 0 254
```

Figure 6.9: An Expression and the corresponding assembly

The `addHaskell` function on the left is just an alias for the addition operation from the `Num` type class, and `addHighLevel` is the definition that emits the assembly program on the right by specifying the location of the input variables in memory and compiling the expression.

The assembly program on the right may seem as is too long for a one that just adding two numbers, so let us take a closer look to see what is going on. The first two lines initialise the stack by putting the stack pointer into the memory cell 254. Remember that `applyBinary` will pop both arguments from the stack, thus they need to be pushed there before (lines 5–16) by the compiler. After that, `applyBinary` specialised to the operation `add` is inlined: we pop y and store it at the temporary memory location `tmp` (located by default at address 255), pop x into the register `R0` and finally the `add` instruction performs the addition.

Let us now explain the implementation of the `compile` function which does most of the hard work for us. The core functionality is implemented by the `compileExpr` function (fig. 6.10), which recursively traverses the expression tree and compiles it into assembly. The leaf nodes are literals that translate to signed immediate arguments (line 4) and variables referring to memory addresses (line 5). Binary operators (lines 6–11) are compiled by compiling their arguments first, pushing the results (always ending up in the register) onto the stack and applying the operation with `applyBinary`. Same is done for the only unary operation `Abs`.

```
compileExpr :: CompilerEnv -> Expression -> Script
compileExpr env@(MkCompilerEnv reg tmp stack) expr =
  case expr of
    Lit (MkLiteral value) -> ld_si reg value
    Var (MkVariable  var) -> ld reg var
    Bin op x y -> do
        compileExpr env x
        push reg stack
        compileExpr env y
        push reg stack
        applyBinary env op
    Abs x -> do
        compileExpr env x
        abs reg
```

Figure 6.10: Compiling Expressions to assembly

To implement the rest of the `compile` function it is now enough to prepare the compiler environment by laying out the necessary tools (stack, temporary variable and register) in the memory and then calling `compileExpr`:

```
compile :: Expression -> Script
compile expr = do
  let init_pointer = _pointer (_stack defaultEnv) - 1
  ld_i r0 init_pointer
  st r0 (_pointer (_stack defaultEnv))
  compileExpr defaultEnv expr
  halt
  where defaultTmp :: Temporary
        defaultTmp = MkTemporary maxBound

        defaultStack :: Stack
        defaultStack = MkStack (maxBound - 1)

        defaultEnv :: CompilerEnv
        defaultEnv = initCompiler r0 defaultTmp defaultStack
```

Figure 6.11: Compiling Expressions to assembly

We place the temporary variable at the very end of the memory (`maxBound` of type `MemoryAddress`), and the stack pointer in the one before. Lines 3–4 initialise the stack pointer by putting the top of the stack into cell `maxBound - 2` and correspond to the lines 1–2 of the example program in fig.6.9. Our emulated stack "grows" from higher memory addresses to lower, but there is little opportunity for data corruption, since the stack will never grow bigger then two elements; however the user of the compiler needs to make sure

that they do not place the data into the five last cells of the memory.

6.3 Integrated Development Environment

In this section, we describe the Integrated Development Environment (IDE) for specifying and verifying REDFIN programs. We propose to use the IDE as the more user-friendly alternative to the low-level library interface of the verification framework. In fact, the IDE directly uses the framework itself as a Haskell library, and builds an interactive interface for program verification.

6.3.1 Motivation

Formal software verification requires the verification engineer to reason about all potential execution scenarios of the program under consideration. While the process of *specification*, i.e. developing a list of properties that the program needs to adhere too, is a creative pen-and-paper task, the consecutive tool-assisted verification of these properties can, in fact, be a cumbersome exercise. The process gets even more difficult if the program being verified poses a computationally hard task for the automated verification tool: every run of a tool can take hours. Therefore, it is desirable to make the verification process as interactive as possible, especially in the early stage of translating the informal human-produced statements into the formal language of the tool.

In case of symbolic execution-based tools, this interactivity manifests primarily in the ability to use the tool similarly to a traditional step-wise debugger. However, traditional debuggers step through only one specific program execution path, while a symbolic debugger must provide a way to interactively explore all feasible paths, and clearly display the path reachability conditions. A natural way to present this process to the user is by providing a tree diagram of the symbolic execution trace, together with the functionality to focus on a state and to continue execution from that state, generating all feasible child states. We discuss these features implemented in our verification IDE for REDFIN further in this section.

6.3.2 IDE features

The IDE provides the following features:

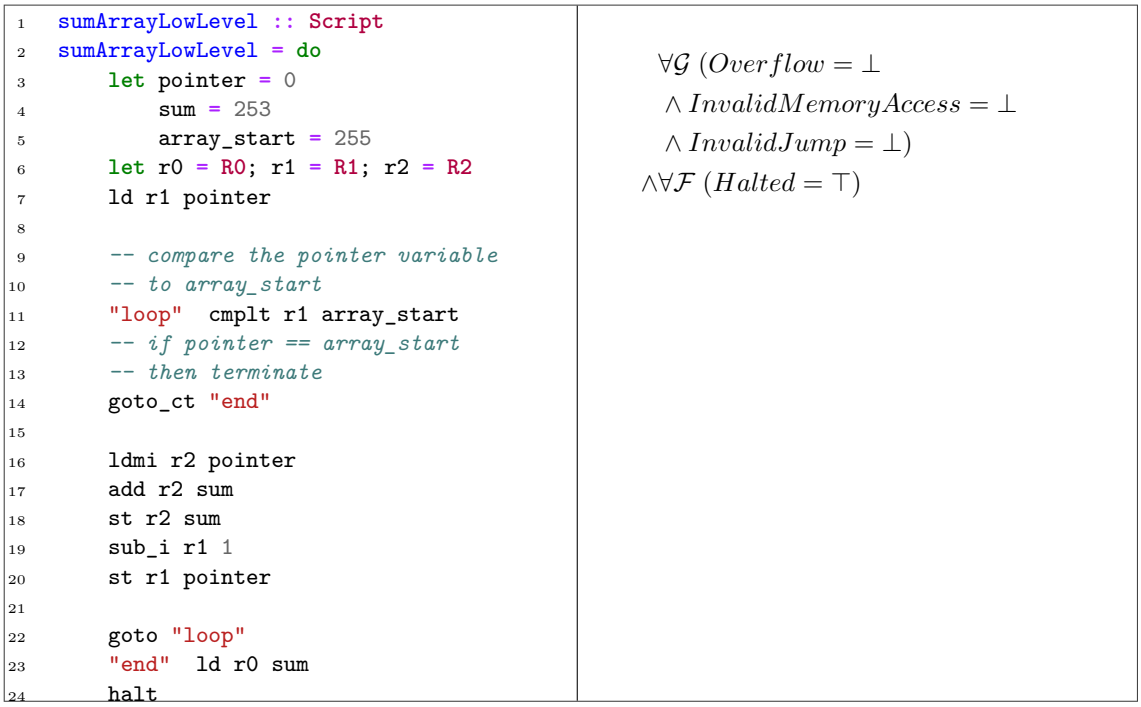
- loading of a REDFIN program from a user-supplied source code file;
- specification of the initial ISA state for the program;
- specification of symbolic constraints to perform the execution under;
- formulation of an invariant to verify;
- interactive exploration of the symbolic execution trace with the ability to pause and continue the execution from an arbitrary discovered state;
- support for saving work in form of projects for persistence.

We now proceed to a demonstration of the IDE user experience by considering the specification and verification process of two case-study REDFIN programs we have already discussed earlier in this thesis. We will present adapted excerpts from the IDE's interface and descriptions of contents of the individual widgets.

6.3.3 Demonstration: array sum program

We have already presented a verification case-study of a simple program to calculate the sum of an array in section 4.3.2. In that section, we have used the coarse-grained monadic semantics and the associated verification framework (see Chapter 4), via the framework's Haskell library interface. We now consider the same program, but use instead the verification framework based on the fine-grained semantics (discussed in Chapter 5), and the interactive IDE interface, which is the main subject of this section.

In this verification case-study, we are interested in *total safety*, i.e. that the program always terminates and the execution never causes the REDFIN core to panic. While the general safety condition for a REDFIN program would specify the absence of arithmetic overflow, division by zero, invalid data memory access, invalid program memory access and a number of other conditions, we can simplify it considering the operations that are actually used in the program.



```
sumArrayLowLevel :: Script
sumArrayLowLevel = do
    let pointer = 0
        sum = 253
        array_start = 255
    let r0 = R0; r1 = R1; r2 = R2
    ld r1 pointer

    -- compare the pointer variable
    -- to array_start
    "loop"  cmplt r1 array_start
    -- if pointer == array_start
    -- then terminate
    goto_ct "end"

    ldmi r2 pointer
    add r2 sum
    st r2 sum
    sub_i r1 1
    st r1 pointer

    goto "loop"
    "end"  ld r0 sum
    halt
```

$$\begin{aligned}&\forall \mathcal{G}\ (Overflow = \bot \\ &\quad \wedge InvalidMemoryAccess = \bot \\ &\quad \wedge InvalidJump = \bot) \\ &\wedge \forall \mathcal{F}\ (Halted = \top)\end{aligned}$$

Figure 6.12: The array summation program and its total safety condition

The figure 6.12 presents the program source code alongside with its total safety specification. The specification requires all execution states to not have overflow (can arise from `add` or `sub_i`), invalid memory access (`ld`, `ldmi` or `st` can cause that) or invalid jump (may be triggered by `goto` and `goto_ct`), and all execution paths have to end eventually terminate, i.e. have the *Halted* flag set.

The figure 6.13 presents the IDE interface displaying successful verification of the sum of three symbolic numbers. The program executed for at most 100 steps is checked to satisfy the property `G (![Overflow]) && F ([Halted])` (specified in the "Verifier" widget); in addition, the data and program memory safety are checked implicitly by the engine, adding up to the property from figure 6.12.

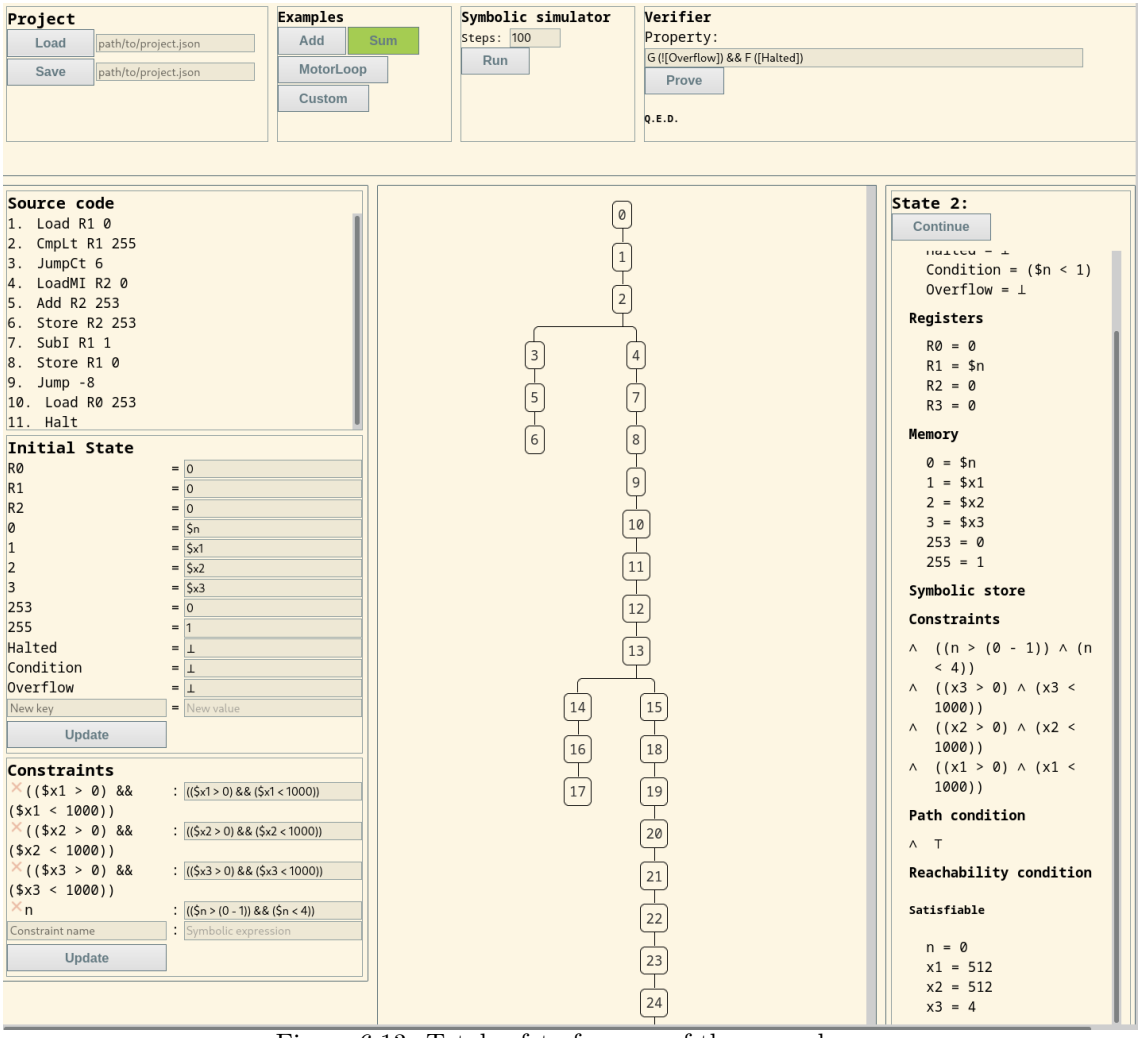


Figure 6.13: Total safety for sum of three numbers

The left-hand-side of the IDE window contains the "Initial State" and "Constraints" widgets, which are used to populate the ISA state with concrete and symbolic data and to specify symbolic constraints on program variables. For this example, we place the symbolic program variable `n` at address 0, and variables `x1`, `x2` and `x3` at address 1, 2 and 3. We constrain `n` to range from 0 to 3, and the array elements to range from 1 to 999:

$$
\begin{aligned}
&(x1 > 0 \wedge x1 < 1000) \\
\wedge\ &(x2 > 0 \wedge x2 < 1000) \\
\wedge\ &(x3 > 0 \wedge x3 < 1000) \\
\wedge\ &(n >= 0 \wedge n <= 3)
\end{aligned}
$$

These constraints are considered when checking reachability of states during symbolic execution. When verifying invariants, like the one in figure 6.12, the constraints become a conjunct in the atomic formulas inside path quantifiers. While widening the range of `n` would require us to specify more array elements, the constraints on the elements themselves can be widened if needed. The trace contains branches that are predicated on the value of `n`: with the left sub-tree always leading to a halt when `n` becomes zero.

The user is free to focus on any state in the trace an observe the ISA state and the program variables in the "State" widget on the right-hand-side. Additionally, the "State" widget shows an assignment of the program variables that can stir the execution into this state, enabling the user to use it in a unit-test.

Determining safe bounds for program variables is essential for successful verification. Consider the figure 6.14, where we remove the constraint on `x3` and re-run verification. The system reports that the invariant becomes falsifiable at states 19, 20, 30 and 31. We examine the state 19 (on the right-hand-side), and see that the execution of the `add` instruction on line 5 may lead to an overflow of the following constraint is satisfied:

$$
\begin{aligned}
Overflow =&(\&a1 > 0 \wedge \&a2 > (2147483647 - \&a1)) \\
&\vee (\&a1 < 0 \wedge \&a2 < (-2147483648 - \&a1))
\end{aligned}
$$

This constraint looks rather cryptic since it mentions the generated pointer variables `a1` and `a2` which are produced by the symbolic memory access. These variables can be resolved by looking at the "Symbolic store" section. To check this constraint for satisfiability, the system resolves the pointers and conjoins the path condition, generating the following

formula:

$$Overflow = ((x3 > 0 \wedge x2 > (2147483647 - x3)) \vee (x3 < 0 \wedge x2 < (-2147483648 - x3)))$$
$$\wedge (x1 > 0 \wedge x1 < 1000) \wedge (x2 > 0 \wedge x2 < 1000)$$
$$\wedge (x2 > 0 \wedge x2 < 1000) \wedge (n >= 0 \wedge n <= 3)$$

Note the absence of constraint on `x3`: this is the reason the overflow is possible in this case.

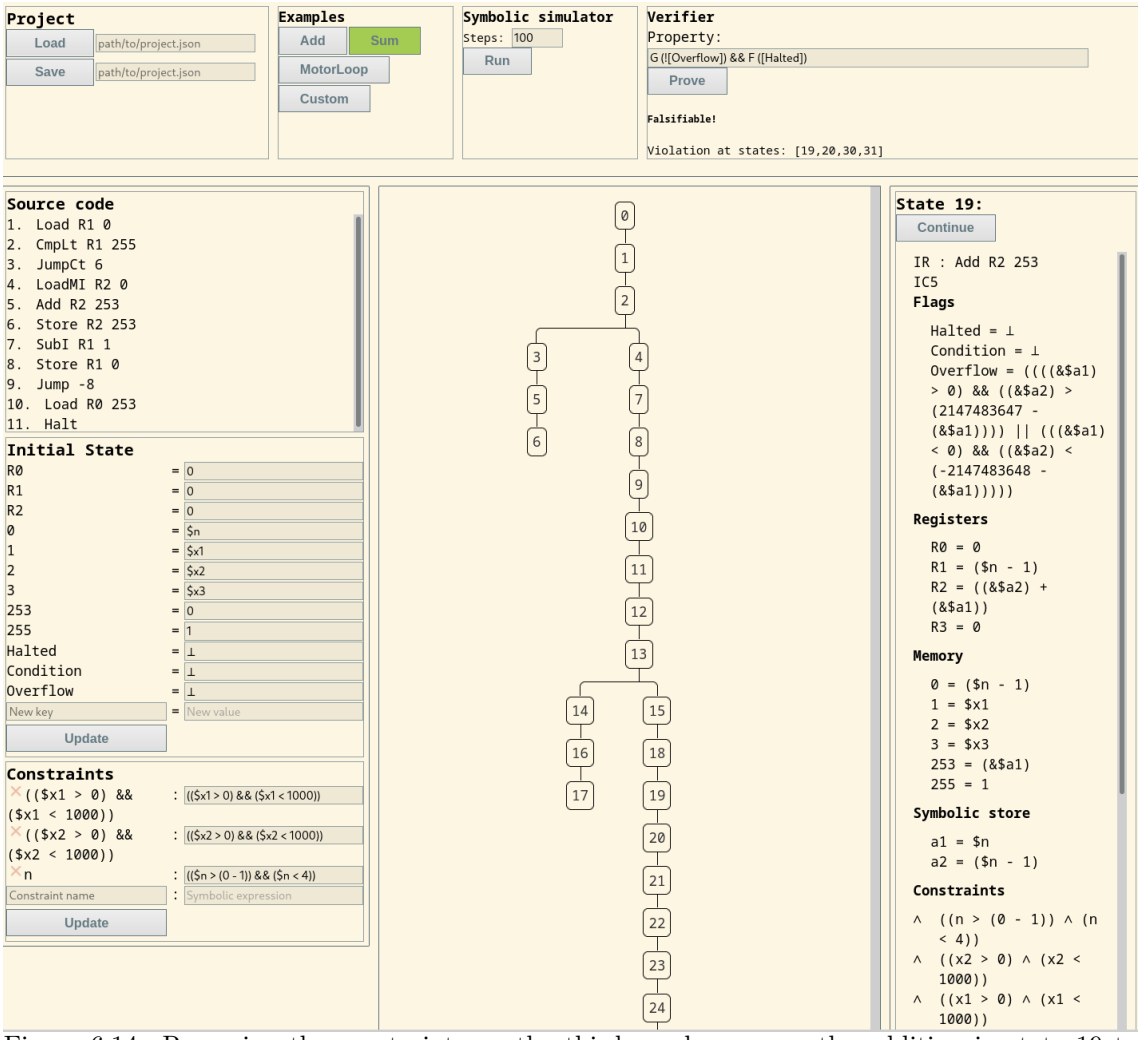


Figure 6.14: Removing the constraints on the third number causes the addition in state 19 to overflow

We have presented a demonstration of the IDE's functionality on a small instance of the array sum verification problem. The IDE is applicable to larger problems, including arbitrary-sized arrays and the stepper-motor control program discussed in the previous chapter.

We now briefly discuss the implementation of the IDE and how it communicates with the REDFIN verification framework.

6.3.4 IDE Implementation Overview

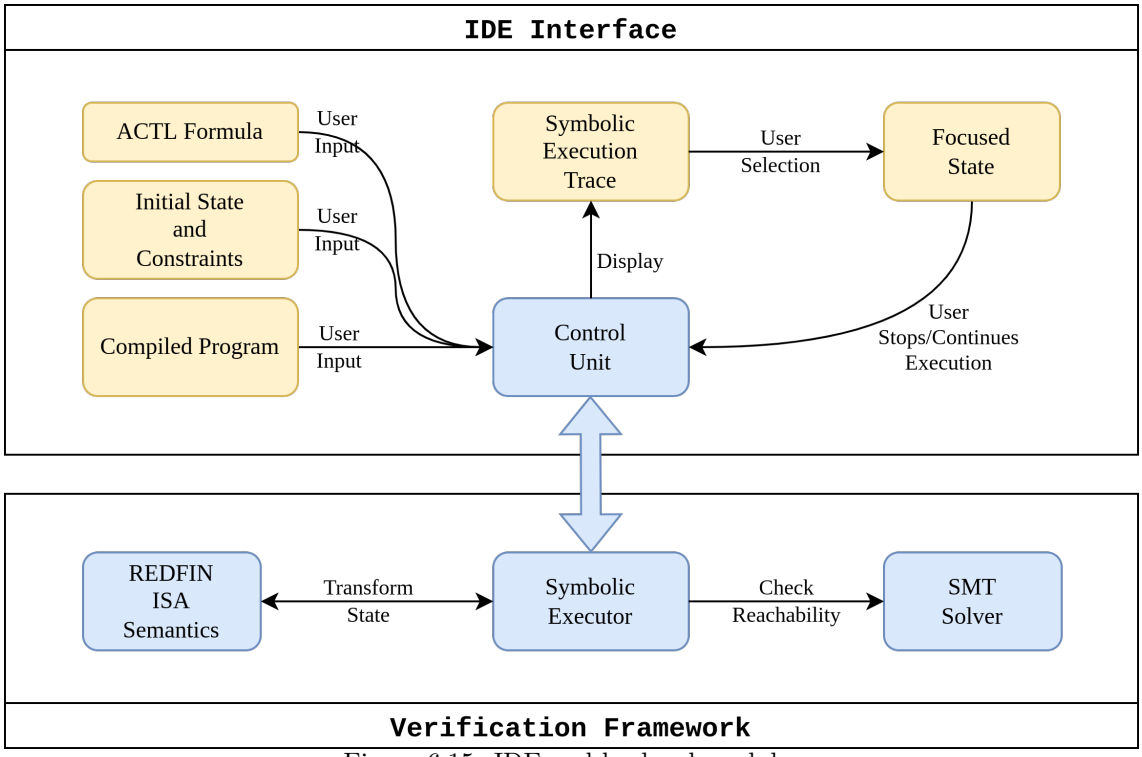


Figure 6.15: IDE and backend modules

The REDFIN IDE presents a user-friendly, visual and interactive interface to the underlying verification framework. The figure 6.15 displays a high-level diagram of the IDE and its connection to the verification framework. The yellow blocks are visible to the user in the IDE interface, and the blue ones are hidden. The control unit reads the user input in the form of a property to verify, the initial ISA state and constraints, and an assembly program. The control unit communicates with the verification framework to obtain a symbolic execution

trace of the program up to some depth and present it to the user. The user then has an opportunity to interactively explore the trace interactively and perform further execution from a particular leaf-state if required. Alternatively, the user can alter the input parameters and receive a new trace.

The verification framework comprises three large components: the ISA semantics, the symbolic execution engine and an external SMT solver connection. We refer the reader to chapters 4 and 5 for the design and implementation details.

Chapter 7

Conclusions and future work

7.1 Summary of contributions

This thesis describes a number of techniques, both existing and novel, and their application to formal verification of ISA-level programs. Our main case-study considers REDFIN — a specialised instruction-set architecture for subsystems of space satellites, and we verify a number of control programs targeting REDFIN. As the basis for formal verification, we build the semantics of REDFIN ISA — a formal and executable specification of the architecture's behaviour: what comprises the ISA state and how programs are executed. We use the same semantics for several purposes: a concrete execution backend for testing and simulation; a symbolic execution backend for formal verification; and also a number of static analyses.

We repeat the list contributions as already outlines in the Introduction:

7.1.1 Contributions

We instantiate the generic verification framework for the REDFIN ISA by providing the following:

- Semantics of REDFIN instruction set architecture implemented as a EDSL in Haskell;
- A tool chain for developing REDFIN programs comprising an assembler and a set of command-line tools for program simulation and testing;

- A specification language for functional properties of REDFIN programs that compiles to REDFIN assembly;
- A symbolic execution engine for REDFIN programs that supports verification of program equivalence, safety and liveness properties of programs and Worst-Case Execution Time analysis;
- An IDE that provides a single point of entry to the developed tools and an interactive explorer for symbolic execution traces and verification results.

Alongside that, the thesis contributes two novel functional programming techniques:

- Selective Applicative Functors [16] provide an abstraction for effectful computations with limited dynamic dependencies. The Haskell implementation of the verification framework for REDFIN uses Selective Applicative Functors in its symbolic execution engine;
- Fine-grained store abstraction is used as the metalanguage for defining the semantics of instructions in a way that allows multiple interpretations of the same semantics: efficient simulation, symbolic execution and static analysis.

The work presented in this thesis has a number of limitations. In the following sections we discuss the avenues of future work we consider to be most interesting and potentially fruitful.

7.2 Future work: application to other architectures

Our verification framework is not tied to one particular ISA, and can be instantiated to other instruction sets by implementing the relevant ISA-specific modules.

We see the emerging blockchain platforms as a promising domain for applying formal methods in general, and our framework in particular. Contemporary blockchain networks, such as, for example, Ethereum [17] and Algorand [18], provide infrastructure for developing *smart contacts* — decentralised applications that are stored on the blockchain. These applications are executed as part of the network's consensus algorithm and therefore benefit from security and censorship-resistance of the networks blockchain. The execution layer

of blockchain networks most often comprises a bytecode-style virtual machine akin to the Java Virtual Machine [90] with a number of domain-specific primitives. Importantly, the bytecode of the virtual machine will often be designed with an emphasis on program correctness, which makes it a fertile soil of formal methods-based verification methodologies to flourish [91]. That characteristic of blockchain networks, and also the fact that smart contracts security exploits can result if devastating capital loss, often makes bytecode-level program verification a business requirement for established blockchain projects [92].

7.3 Future work: Redfin modelling scope

The first body of future work we envision primary considers the modelling scope, i.e. *what* are we modelling and verifying.

7.3.1 System bus interaction

The REDFIN core is intended to be deployed as the local control unit into space satellite subsystems. The execution of REDFIN subroutines will be triggered by the system-wide processing unit, with the external signals delivered to REDFIN via the system bus (see Figure 7.1).As we have noted in section 5.3.1.1, we do not model the interaction with the system bus.

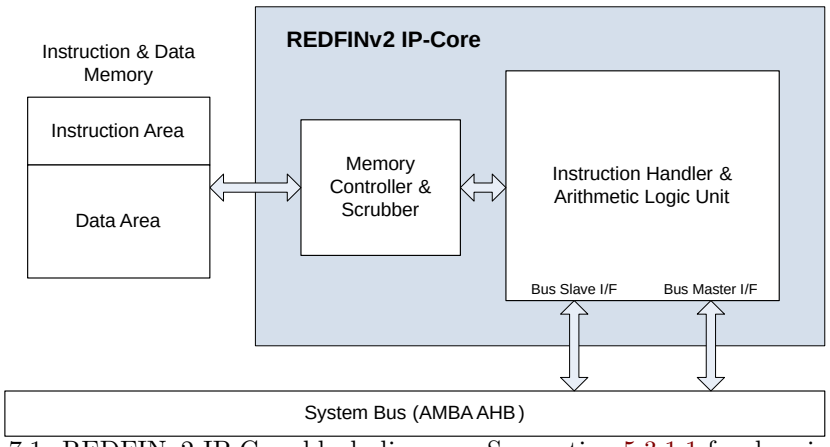


Figure 7.1: REDFINv2 IP-Core block diagram. See section 5.3.1.1 for description.)

Including the specification of the system bus interactions into the model presents an

interesting opportunity to wrap the REDFIN code into a *verified envelope*, with the system bus being the only unverified component. More specifically, such an extension of our work could study the problem of rigorously defining the interactions of verified and unverified systems.

7.3.2 Hardware synthesis

The models we build in this thesis consider REDFIN at the instruction-set architecture level. An interesting extension of our work is to implement a hardware synthesis procedure that would construct an implementation of the REDFIN core in a hardware description language. This synthesised implementation can be then crossvalidated with the existing bespoke implementation of REDFIN in VHDL for additional correctness guarantees.

7.4 Future work: formal verification techniques

The second avenue of future work considers the advances that can be made in *how* program verification is performed.

7.4.1 Automated proof of functional correctness for looping programs

The most significant case-study of this thesis is verification of the stepper-motor control program (section 5.3.6). The program calculates a series of displacement and velocities for a stepper-motor to move an antenna or a solar panel into a specific position. The termination condition of the program's algorithm (Algorithm 1) depends on the symbolic values associated with program variables that get altered at each iteration. This problem, known as *symbolic non-termination*, is the subject of active research across the formal software verification community. The traditional symbolic execution, which we use in our implementation, while great for automated proving, does not cope well with symbolic non-termination. There exists a number of techniques to mitigate this problem, which we have described in the Background chapter 2.1.4.2. In section 5.3.6.3, we discuss how we use a user-specified loop invariant to mitigate the non-termination problem up to some extent. Using other techniques, much more could be done here, and we believe it is possible to

achieve automated verification of functional correctness of looping REDFIN programs.

7.4.2 Reducing the trusted base: verified symbolic execution

Every verification tool has a *trusted base* — a part of the implementation that is not verified itself. Formal verification of a system involves checking that that system conforms to some form of a specification. The specification is usually stated as a theorem in some formal logic, and the verification tool checks that this theorem is valid. The implementation of this proof checker usually has to be *trusted* to be implemented correctly. If there is a bug in the proof checker, the verification tool may very well be *unsound*. i.e. it can construct a proof of $\bot$, the false proposition. If a proof of false can be constructed, anything can be proved.

The verification frameworks developed in this thesis have a large trusted base. We have to trust not only our own REDFIN semantics and implementations of the symbolic execution engine, but also a number of Haskell libraries [87] and the off-the-shelf SMT solver, Z3 [60]. Of course, this is also the case for many others verification tools, especially the push-button ones, like ours.

While completely eliminating the trusted base is impossible, there is a noticeable trend in modern literature on making the trusted base both (i) as small as possible and (ii) correct-by-construction.

One approach to reduce the trusted base is to use a theorem prover as the metalanguage for the implementation of a verification framework. Today, there exist a number of project leveraging the Coq [56] theorem prover, thus shifting the trust onto its core, which is itself designed to be small. Such project include the Verified Software Toolchain [93], which targets verification of C programs; the Bedrock project [34] that considers low-level assembly-like languages; and Katamaran [35], which aims to create a verified framework for ISA specification and program verification.

In this thesis, we use the Haskell programming language as the metalanguage for ISA semantics and as the implementation language of the symbolic execution engines. While we do employ advanced features of Haskell's type-system, such as type families and data kinds to enforce a number of constrains on the type level, there exists some other, more advanced machinery to enforce even stronger properties. The `hs-to-coq` [94] project sounds like a very exciting tool to employ for this avenue. It performs a source-level translation

of the total subset of the Haskell programming language into the Coq [56] proof assistant, so that the desired properties can be proved leveraging Coq's powerful dependently-typed metalanguage. A slightly different approach that avoids a translation to a different language would be to use the dependently-typed flavour of Haskell [95] to prove the desired properties inside Haskell itself.

Combining this direction, i.e. creating a *verified verifier*, with hardware synthesis 7.3.2 and system bus interaction 7.3.1 would yield a very interesting research project: a study of formally verified processing core implementation, derived from a formal specification of the core's ISA.

Bibliography

[1] J. Lechner, "Building robust GALS circuits : fault-tolerant and variation-aware design. Techniques for reliable circuit operation," Ph.D. dissertation, TU Wien, 2014.

[2] N. G. Leveson, "Role of Software in Spacecraft Accidents," *Journal of Spacecraft and Rockets*, vol. 41, no. 4, pp. 564–575, 2004.

[3] M. Ben-Ari, "The Bug That Destroyed a Rocket," *SIGCSE Bull.*, vol. 33, no. 2, pp. 58–59, Jun. 2001.

[4] NASA, "Mars Climate Orbiter Mishap Investigation Board Phase I Report," NASA, Tech. Rep., Nov. 1999.

[5] B. Kempa, P. Zhang, P. H. Jones, J. Zambreno, and K. Y. Rozier, "Embedding online runtime verification for fault disambiguation on robonaut2," in *Formal Modeling and Analysis of Timed Systems*, N. Bertrand and N. Jansen, Eds., Cham: Springer International Publishing, 2020, pp. 196–214, ISBN: 978-3-030-57628-8.

[6] N. Leveson, "A systems-theoretic approach to safety in software-intensive systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 66–86, 2004. DOI: 10.1109/TDSC.2004.1.

[7] IEEE P1076 Working Group, *Vhdl analysis and standardization group*, [Online; accessed 15-August-2021], 2021. [Online]. Available: https://ieee-p1076.gitlab.io/.

[8] IEEE, "Ieee standard for systemverilog–unified hardware design, specification, and verification language," *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pp. 1–1315, 2018. DOI: 10.1109/IEEESTD.2018.8299595.

[9] Bluespec Inc, *Bluespec hardware description language*, [Online; accessed 15-August-2021], 2021. [Online]. Available: https://github.com/B-Lang-org/bsc.

[10] A. Reid, R. Chen, A. Deligiannis, *et al.*, "End-to-end verification of processors with isa-formal," in *Computer Aided Verification*, S. Chaudhuri and A. Farzan, Eds., Cham: Springer International Publishing, 2016, pp. 42–58, ISBN: 978-3-319-41540-6.

[11] A. Armstrong, T. Bauereiss, B. Campbell, *et al.*, "Isa semantics for armv8-a, risc-v, and cheri-mips," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, 71:1–71:31, Jan. 2019, ISSN: 2475-1421. DOI: 10.1145/3290384. [Online]. Available: http://doi.acm.org/10.1145/3290384.

[12] M. J. S. Hunt Warren A. Kaufmann Matt and S. Anna, "Industrial hardware and software verification with acl2," *Phil. Trans. R. Soc. A.*, 2017. DOI: {https://doi.org/10.1098/rsta.2015.0399}.

[13] RISC-V International, *Risc-v website*, [Online; accessed 15-August-2021], 2021. [Online]. Available: https://riscv.org/.

[14] ESA, *ECSS-Q-ST-60-02C – ASIC and FPGA development*, https://ecss.nl/standard/ecss-q-st-60-02c-asic-and-fpga-development/, 2008.

[15] ESA, *ECSS-Q-ST-80C Rev.1 – Software product assurance*, https://ecss.nl/standard/ecss-q-st-80c-rev-1-software-product-assurance-15-february-2017/, 2017.

[16] A. Mokhov, G. Lukyanov, S. Marlow, and J. Dimino, "Selective applicative functors," *Proc. ACM Program. Lang.*, vol. 3, no. ICFP, Jul. 2019. DOI: 10.1145/3341694. [Online]. Available: https://doi.org/10.1145/3341694.

[17] Ethereum Foundation, *Ethereum virtual machine (evm)*, [Online; accessed 28-April-2021], 2021. [Online]. Available: https://ethereum.org/en/developers/docs/evm/.

[18] Algorand Foundation, *Teal*, [Online; accessed 28-April-2021], 2021. [Online]. Available: https://developer.algorand.org/docs/get-details/dapps/avm/teal/specification/.

[19] W3C; Mozilla; Microsoft; Google; Apple, *Webassembly*, [Online; accessed 28-April-2021], 2021. [Online]. Available: https://webassembly.org/.

[20] J. Contributors, *Junit 5 user guide*, [Online; accessed 13-September-2021], 2021. [Online]. Available: https://junit.org/junit5/docs/current/user-guide/.

[21] Q. Contributors, *Quickcheck: Automatic testing of haskell programs*, [Online; accessed 13-September-2021], 2021. [Online]. Available: https://hackage.haskell.org/package/QuickCheck.

[22] smallcheck Contributors, *Smallcheck: A property-based testing library*, [Online; accessed 13-September-2021], 2021. [Online]. Available: https://hackage.haskell.org/package/smallcheck.

[23] hedgehog Contributors, *Hedgehog: Release with confidence*, [Online; accessed 13-September-2021], 2021. [Online]. Available: https://hackage.haskell.org/package/hedgehog.

[24] P. Mishra and N. Dutt, *Processor Description Languages*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008, ISBN: 9780080558370.

[25] ARM Ltd., *A-profile architectures / exploration tools*, [Online; accessed 26-September-2021]. [Online]. Available: https://developer.arm.com/architectures/cpu-architecture/a-profile/exploration-tools.

[26] A. Reid, "Defining interfaces between hardware and software: Quality and performance," Ph.D. dissertation, School of Computing Science, University of Glasgow, Glasgow, Scotland, Mar. 2019. [Online]. Available: http://theses.gla.ac.uk/41068/.

[27] Anthony Fox, *L3 language manual*, [Online; accessed 2-October-2021], 2021. [Online]. Available: ttps://acjf3.github.io/l3/l3.pdf.

[28] A. Fox, "Directions in isa specification," in *Interactive Theorem Proving*, L. Beringer and A. Felty, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 338–344, ISBN: 978-3-642-32347-8.

[29] A. Fox, "Improved tool support for machine-code decompilation in hol4," in *Interactive Theorem Proving*, C. Urban and X. Zhang, Eds., Cham: Springer International Publishing, 2015, pp. 187–202, ISBN: 978-3-319-22102-1.

[30] D. Cock, "Lyrebird - assigning meanings to machines," in *5th International Workshop on Systems Software Verification, SSV'10, Vancouver, BC, Canada, October 6-7, 2010*, R. Huuck, G. Klein, and B. Schlich, Eds., USENIX Association, 2010. [Online]. Available: https://www.usenix.org/conference/ssv10/lyrebird%5C%E2%5C%80%5C%94assigning-meanings-machines.

[31] G. Klein, J. Andronick, K. Elphinstone, *et al.*, "Comprehensive formal verification of an os microkernel," *ACM Trans. Comput. Syst.*, vol. 32, no. 1, Feb. 2014, ISSN: 0734-2071. DOI: 10.1145/2560537. [Online]. Available: https://doi.org/10.1145/2560537.

[32] M. R. Barbacci, "Instruction set processor specifications (isps): The notation and its applications," *IEEE Transactions on Computers*, no. 1, pp. 24–40, 1981. DOI: 10.1109/TC.1981.6312154.

[33] J. Choi, M. Vijayaraghavan, B. Sherman, A. Chlipala, and Arvind, "Kami: A platform for high-level parametric hardware specification and its modular verification," *Proc. ACM Program. Lang.*, vol. 1, no. ICFP, Aug. 2017. DOI: 10.1145/3110268. [Online]. Available: https://doi.org/10.1145/3110268.

[34] A. Chlipala, "Mostly-automated verification of low-level programs in computational separation logic," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11, San Jose, California, USA: Association for Computing Machinery, 2011, pp. 234–245, ISBN: 9781450306638. DOI: 10.1145/1993498.1993526. [Online]. Available: https://doi.org/10.1145/1993498.1993526.

[35] S. Keuchel, G. Lukyanov, and D. Devriese, *Katamaran: Semi-automated verification of isa specifications*, https://pldi20.sigplan.org/details/rems-deepspec-2020/7/Katamaran-semi-automated-verification-of-ISA-specifications, 2020.

[36] J. Woodruff, R. N. Watson, D. Chisnall, *et al.*, "The cheri capability model: Revisiting risc in an age of risk," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14, Minneapolis, Minnesota, USA: IEEE Press, 2014, pp. 457–468, ISBN: 9781479943944.

[37] A. Fox and M. O. Myreen, "A trustworthy monadic formalization of the armv7 instruction set architecture," in *Interactive Theorem Proving*, M. Kaufmann and L. C. Paulson, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 243–258, ISBN: 978-3-642-14052-5.

[38] X. Feng, Z. Shao, A. Vaynberg, S. Xiang, and Z. Ni, "Modular verification of assembly code with stack-based control abstractions," in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '06, Ottawa, Ontario, Canada: Association for Computing Machinery, 2006, pp. 401–414, ISBN: 1595933204. DOI: 10.1145/1133981.1134028. [Online]. Available: https://doi.org/10.1145/1133981.1134028.

[39] Z. Ni and Z. Shao, "Certified assembly programming with embedded code pointers," in *POPL '06*, ser. POPL '06, Charleston, South Carolina, USA: Association for Computing Machinery, 2006, pp. 320–333, ISBN: 1595930272. DOI: 10.1145/1111037.1111066. [Online]. Available: https://doi.org/10.1145/1111037.1111066.

[40] G. C. Necula, "Proof-carrying code," in *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '97, Paris, France: Association for Computing Machinery, 1997, pp. 106–119, ISBN: 0897918533. DOI: 10.1145/263699.263712. [Online]. Available: https://doi.org/10.1145/263699.263712.

[41] T. Bourgeat, I. Clester, A. Erbsen, S. Gruetter, A. Wright, and A. Chlipala, *A multipurpose formal risc-v specification*, 2021. arXiv: 2104.00762 [cs.LO].

[42] B. Bond, C. Hawblitzel, M. Kapritsos, *et al.*, "Vale: Verifying high-performance cryptographic assembly code," in *26th USENIX Security Symposium (USENIX Security 17)*, Vancouver, BC: USENIX Association, Aug. 2017, pp. 917–934, ISBN: 978-1-931971-40-9. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/bond.

[43] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "Bap: A binary analysis platform," in *Computer Aided Verification*, G. Gopalakrishnan and S. Qadeer, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 463–469, ISBN: 978-3-642-22110-1.

[44] U. Degenbaev, *Formal specification of the x86 instruction set architecture, Formelle spezifizierung von dem x86-befehlssatz*, 2012. DOI: http://dx.doi.org/10.22028/D291-26338.

[45] S. Goel, W. A. Hunt, and M. Kaufmann, "Abstract stobjs and their application to isa modeling," *Electronic Proceedings in Theoretical Computer Science*, vol. 114, pp. 54–69, Apr. 2013, ISSN: 2075-2180. DOI: 10.4204/eptcs.114.5. [Online]. Available: http://dx.doi.org/10.4204/EPTCS.114.5.

[46] A. Contributors, *ACL2 Homepage*, https://www.cs.utexas.edu/users/moore/acl2/, Blog, 2021.

[47] S. Goel, A. Slobodova, R. Sumners, and S. Swords, "Balancing automation and control for formal verification of microprocessors," in *Computer Aided Verification*, A. Silva and K. R. M. Leino, Eds., Cham: Springer International Publishing, 2021, pp. 26–45, ISBN: 978-3-030-81685-8.

[48] S. Dasgupta, D. Park, T. Kasampalis, V. S. Adve, and G. Roşu, "A complete formal semantics of x86-64 user-level instruction set architecture," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019, Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 1133–1148, ISBN: 9781450367127. DOI: 10.1145/3314221.3314601. [Online]. Available: https://doi.org/10.1145/3314221.3314601.

[49] Two's complement, *Two's complement — Wikipedia, the free encyclopedia*, [Online; accessed 26-April-2021], 2021. [Online]. Available: https://en.wikipedia.org/wiki/Two%27s_complement.

[50] R. C. Seacord, *Secure Coding in C and C++*, 2nd. Addison-Wesley Professional, 2013, ISBN: 0321822137.

[51] CERT C Coding Standad, *Cert c coding standad*, [Online; accessed 28-April-2021], 2021. [Online]. Available: https://wiki.sei.cmu.edu/confluence/display/c/INT32-C.+Ensure+that+operations+on+signed+integers+do+not+result+in+overflow.

[52] T. Kapus, M. Nowack, and C. Cadar, "Constraints in dynamic symbolic execution: Bitvectors or integers?" In *Tests and Proofs*, D. Beyer and C. Keller, Eds., Cham: Springer International Publishing, 2019, pp. 41–54, ISBN: 978-3-030-31157-5.

[53] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," *Commun. ACM*, vol. 56, no. 2, pp. 82–90, Feb. 2013, ISSN: 0001-0782. DOI: 10.1145/2408776.2408795. [Online]. Available: https://doi.org/10.1145/2408776.2408795.

[54] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, 2018.

[55] G. D. Plotkin, "The origins of structural operational semantics," *The Journal of Logic and Algebraic Programming*, pp. 3–15, 2004, Structural Operational Semantics, ISSN: 1567-8326. DOI: https://doi.org/10.1016/j.jlap.2004.03.009. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1567832604000268.

[56] Coq Development Team, *The coq proof assistant*, [Online; accessed 28-April-2021], 2021. [Online]. Available: https://coq.inria.fr/.

[57] Agda Development Team, *The agda progarmming language*, [Online; accessed 28-April-2021], 2021. [Online]. Available: https://wiki.portal.chalmers.se/agda/pmwiki.php.

[58] E. Moggi, "Notions of computation and monads," *Inf. Comput.*, vol. 93, no. 1, pp. 55–92, Jul. 1991, ISSN: 0890-5401. DOI: 10.1016/0890-5401(91)90052-4. [Online]. Available: https://doi.org/10.1016/0890-5401(91)90052-4.

[59] P. Wadler, "Monads for functional programming," in *Program Design Calculi*, M. Broy, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 233–264, ISBN: 978-3-662-02880-3.

[60] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, 2008.

[61] C. Barrett, A. Stump, and C. Tinelli, "The SMT-LIB Standard: Version 2.0," Department of Computer Science, The University of Iowa, Tech. Rep., 2010, Available at www.SMT-LIB.org.

[62] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*, San Diego, CA: USENIX Association, Dec. 2008. [Online]. Available: https://www.usenix.org/conference/osdi-08/klee-unassisted-and-automatic-generation-high-coverage-tests-complex-systems.

[63] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "Exe: Automatically generating inputs of death," *ACM Trans. Inf. Syst. Secur.*, vol. 12, no. 2, Dec. 2008, ISSN: 1094-9224. DOI: 10.1145/1455518.1455522. [Online]. Available: https://doi.org/10.1145/1455518.1455522.

[64] C. S. Păsăreanu and N. Rungta, "Symbolic pathfinder: Symbolic execution of java bytecode," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '10, Antwerp, Belgium: Association for Computing Machinery, 2010, pp. 179–180, ISBN: 9781450301169. DOI: 10.1145/1858996.1859035. [Online]. Available: https://doi.org/10.1145/1858996.1859035.

[65] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," *SIGPLAN Not.*, vol. 47, no. 6, pp. 193–204, Jun. 2012, ISSN: 0362-1340. DOI: 10.1145/2345156.2254088. [Online]. Available: https://doi.org/10.1145/2345156.2254088.

[66] P. Godefroid and D. Luchaup, "Automatic partial loop summarization in dynamic test generation," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11, Toronto, Ontario, Canada: Association for Computing Machinery, 2011, pp. 23–33, ISBN: 9781450305624. DOI: 10.1145/2001420.2001424. [Online]. Available: https://doi.org/10.1145/2001420.2001424.

[67] T. Liu, M. Araujo, M. d'Amorim, and M. Taghdiri, "A comparative study of incremental constraint solving approaches in symbolic execution," in *10th Haifa Verification Conference (HVC), 2014*, Nov. 2014.

[68] A. Church, "A formulation of the simple theory of types," *Journal of Symbolic Logic*, vol. 5, no. 2, pp. 56–68, 1940. DOI: 10.2307/2266170.

[69] P. Hudak, S. Peyton Jones, P. Wadler, *et al.*, "Report on the programming language haskell: A non-strict, purely functional language version 1.2," *SIGPLAN Not.*, vol. 27, no. 5, pp. 1–164, 1992.

[70] R. M. Burstall, D. B. MacQueen, and D. T. Sannella, "Hope: An experimental applicative language," in *Proceedings of the 1980 ACM Conference on LISP and Functional Programming*, ser. LFP '80, Stanford University, California, USA: Association for Computing Machinery, 1980, pp. 136–143, ISBN: 9781450373968. DOI: 10.1145/800087.802799. [Online]. Available: https://doi.org/10.1145/800087.802799.

[71] G. Hutton, "A tutorial on the universality and expressiveness of fold," *Journal of Functional Programming*, vol. 9, Sep. 1999. DOI: 10.1017/S0956796899003500.

[72] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler, "A history of haskell: Being lazy with class," in *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, ser. HOPL III, San Diego, California: Association for Computing Machinery, 2007, pp. 12–1–12–55, ISBN: 9781595937667. DOI: 10.1145/1238844.1238856. [Online]. Available: https://doi.org/10.1145/1238844.1238856.

[73] G. Contributors, *Ghc users guide*, [Online; accessed 12-May-2021], 2021. [Online]. Available: https://ghc.readthedocs.io/.

[74] B. Blöndal, A. Löh, and R. Scott, "Deriving via: Or, how to turn hand-written instances into an anti-pattern," *SIGPLAN Not.*, vol. 53, no. 7, pp. 55–67, Sep. 2018, ISSN: 0362-1340. DOI: 10.1145/3299711.3242746. [Online]. Available: https://doi.org/10.1145/3299711.3242746.

[75] T. Sheard and S. P. Jones, "Template meta-programming for haskell," in *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, ser. Haskell '02, Pittsburgh, Pennsylvania: Association for Computing Machinery, 2002, pp. 1–16, ISBN: 1581136056. DOI: 10.1145/581690.581691. [Online]. Available: https://doi.org/10.1145/581690.581691.

[76] J. Svenningsson and E. Axelsson, "Combining deep and shallow embedding for edsl," in *Trends in Functional Programming*, H.-W. Loidl and R. Peña, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 21–36, ISBN: 978-3-642-40447-4.

[77] C. Mcbride and R. Paterson, "Applicative programming with effects," *J. Funct. Program.*, vol. 18, no. 1, pp. 1–13, Jan. 2008, ISSN: 0956-7968. DOI: 10.1017/S0956796807006326. [Online]. Available: https://doi.org/10.1017/S0956796807006326.

[78] P. Wadler, "Theorems for free!" In *FPCA*, vol. 89, 1989, pp. 347–359.

[79] D. Leijen and E. Meijer, "Domain specific embedded compilers," *ACM Sigplan Notices*, vol. 35, no. 1, pp. 109–122, 2000.

[80] R. Harper, "Boolean Blindness," https://web.archive.org/web/20110321191234/http://existentialtype.wordpress.com/2011/03/15/boolean-blindness/, 2011.

[81] B. Blöndal, A. Löh, and R. Scott, "Deriving via," in *Proceedings of the 11th ACM Haskell Symposium (Haskell'18)*, 2018.

[82] P. Wadler, "Monads for functional programming," in *Int'l School on Advanced Functional Programming*, Springer, 1995, pp. 24–52.

[83] P. Sewell, F. Z. Nardelli, S. Owens, *et al.*, "Ott: Effective tool support for the working semanticist," *SIGPLAN Not.*, vol. 42, no. 9, pp. 1–12, Oct. 2007, ISSN: 0362-1340. DOI: 10.1145/1291220.1291155. [Online]. Available: https://doi.org/10.1145/1291220.1291155.

[84] D. P. Mulligan, S. Owens, K. E. Gray, T. Ridge, and P. Sewell, "Lem: Reusable engineering of real-world semantics," in *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '14, Gothenburg, Sweden: Association for Computing Machinery, 2014, pp. 175–188, ISBN: 9781450328739. DOI: 10.1145/2628136.2628143. [Online]. Available: https://doi.org/10.1145/2628136.2628143.

[85] G. Roșu and T. F. Șerbănuță, "An overview of the k semantic framework," *The Journal of Logic and Algebraic Programming*, vol. 79, no. 6, pp. 397–434, 2010, Membrane computing and programming, ISSN: 1567-8326. DOI: https://doi.org/10.1016/j.jlap.2010.03.012. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1567832610000160.

[86] P. Wadler, "Comprehending monads," in *Proceedings of the 1990 ACM conference on LISP and functional programming*, ACM, 1990, pp. 61–78.

[87] L. Erkok, *SBV: SMT Based Verification in Haskell*, 2019. [Online]. Available: http://leventerkok.github.io/sbv/ (visited on 11/27/2017).

[88] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones, "Refinement types for Haskell," in *ACM SIGPLAN Notices*, ACM, 2014, pp. 269–282.

[89] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Logics of Programs*, D. Kozen, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, pp. 52–71, ISBN: 978-3-540-39047-3.

[90] Oracle, *The java® virtual machine specification*, [Online; accessed 28-April-2021], 2021. [Online]. Available: https://docs.oracle.com/javase/specs/jvms/se7/html/.

[91] M. Alturki, J. Chen, V. Luchangco, *et al.*, "Towards a verified model of the algorand consensus protocol in coq," English (US), in *Formal Methods- FM 2019 International Workshops - Revised Selected Papers*, ser. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Publisher Copyright: © Springer Nature Switzerland AG 2020.; 3rd World Congress on Formal Methods, FM 2019 ; Conference date: 07-10-2019 Through 11-10-2019, Germany: Springer, 2020, pp. 362–367, ISBN: 9783030549930. DOI: 10.1007/978-3-030-54994-7_27.

[92] Maker DAO, *Maker dao*, [Online; accessed 28-April-2021], 2021. [Online]. Available: https://security.makerdao.com/formal-verification.

[93] A. W. Appel, "Verified software toolchain," in *Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software*, ser. ESOP'11/ETAPS'11, Saarbrücken, Germany: Springer-Verlag, 2011, pp. 1–17, ISBN: 9783642197178.

[94] A. Spector-Zabusky, J. Breitner, C. Rizkallah, and S. Weirich, "Total haskell is reasonable coq," in *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, ser. CPP 2018, Los Angeles, CA, USA: Association for Computing Machinery, 2018, pp. 14–27, ISBN: 9781450355865. DOI: 10.1145/3167092. [Online]. Available: https://doi.org/10.1145/3167092.

[95] S. Weirich, A. Voizard, P. H. A. de Amorim, and R. A. Eisenberg, "A specification for dependent types in haskell," *Proc. ACM Program. Lang.*, vol. 1, no. ICFP, Aug. 2017. DOI: 10.1145/3110275. [Online]. Available: https://doi.org/10.1145/3110275.