A Multi-Backend Frontend for SMT Solvers in OCaml

João Maria Henriques Madeira Pereira

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisor(s): Prof. José Faustino Fragoso Femenin dos Santos
Prof. Pedro Miguel dos Santos Alves Madeira Adão

Examination Committee

Chairperson: Prof. Manuel Fernando Cabido Peres Lopes
Supervisor: Prof. José Faustino Fragoso Femenin dos Santos
Member of the Committee: Prof. Mikoláš Janota

November 2024

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

To my mom, dad, and sister,

Acknowledgments

First and foremost, I would like to express my deepest gratitude to my supervisors, Prof. José Fragoso Santos and Prof. Pedro Adão, for their invaluable guidance, expertise, and unwavering support, which have been fundamental throughout this journey. I am immensely grateful for the time and effort they dedicated to helping me grow academically and personally.

I would also like to extend my heartfelt appreciation to my parents and sister. Their endless encouragement and belief in me have supported me every step of the way. Without their unconditional support, this achievement would not have been possible.

A notable acknowledgement goes to Filipe Marques, whose work laid the foundation for this thesis. His time, assistance, and expertise were instrumental throughout this entire process.

A special word to my friends Leonor Barreiros, Filipa Cotrim, and José Afonso, whose invaluable friendship and companionship have been a source of motivation and joy throughout my academic journey. Your friendship has meant the world to me.

Finally, thank you to all my family members, friends, and everyone with whom I have had the privilege of working.

To each and every one of you – Thank you.

Resumo

Os solucionadores de Satisfiability Modulo Theories (SMT) têm-se revelado altamente eficazes e úteis em vários domínios, incluindo a verificação de software e hardware e a geração de casos de teste, entre outros. Consequentemente, inúmeras ferramentas precisam de interagir com solucionadores SMT durante a sua execução. Com o intuito de facilitar eficientemente esta interação na linguagem de programação OCaml, o SMT.ML foi desenvolvido como uma interface para a resolução de SMT que permite que um programa em OCaml interaja facilmente com um solucionador SMT. No entanto, a versão atual do SMT.ML suporta exclusivamente o solucionador SMT Z3. Isto pode impor restrições aos utilizadores, uma vez que o Z3 pode não ser o solucionador mais adequado para todas as teorias e aplicações. Para ultrapassar esta limitação, propomos uma versão melhorada do SMT.ML que inclui um backend adicional para o cvc5, um solucionador SMT de última geração que supera o Z3 em várias teorias. Para alcançar este objetivo, também desenvolvemos bindings em OCaml que permitam a interação direta entre o SMT.ML e o solucionador cvc5. Estes bindings não só beneficiarão o SMT.ML, mas também proporcionarão valor a programadores OCaml que pretendam incorporar o solucionador SMT cvc5 nos seus programas e outras ferramentas baseadas em OCaml.

Palavras-chave: Solucionadores SMT, Execução Simbólica, OCaml

Abstract

Satisfiability Modulo Theories (SMT) solvers have proven to be highly effective and helpful in various domains, including software and hardware verification and test-case generation, among others. Consequently, numerous tools need to interface with SMT solvers during their execution. With the intent of efficiently facilitating this interaction within the OCaml programming language, SMT.ML was developed as a frontend for SMT solving that allows an OCaml program to easily interact with an SMT solver. However, the current version of SMT.ML exclusively supports the Z3 SMT solver. This can pose constraints on users, as Z3 may not be the optimal solver for all theories and applications. To address this limitation, we propose an enhanced version of SMT.ML that features an additional backend for cvc5, a state-of-the-art SMT solver that outperforms Z3 in multiple theories. To achieve this, we also developed OCaml bindings that enable direct interaction between SMT.ML and the cvc5 solver. These bindings will not only benefit SMT.ML but also provide value to OCaml developers aiming to incorporate the cvc5 SMT solver into their programs and other OCaml-based tools.

Keywords: SMT Solvers, Symbolic Execution, OCaml

Contents

List of Tables

List of Figures

List of Listings

Acronyms

CDCL Conflict-Driven Clause Learning

CDF Cumulative Distribution Function

CVC Cooperating Validity Checker

DAG Directed Acyclic Graph

FFI Foreign Function Interface

GC Garbage Collection

IDL Interface Description Language

SMT Satisfiability Modulo Theories

Chapter 1

Introduction

1.1 Motivation

In the realm of formal verification and symbolic execution [1–3], *Satisfiability Modulo Theories* (SMT) solvers play a pivotal role in assisting developers to ensure the correctness of software systems [4–7]. As the complexity of modern software continues to escalate [8], the demand for robust and efficient tools for formal verification becomes increasingly pronounced. Within this context, the OCaml programming language [9] has gained prominence as a suitable platform for the development of high-performance and reliable software verification tools [10–12].

Frontends for SMT solvers [13–16] serve as practical interfaces that abstract the low-level intricacies of SMT solvers and provide developers with user-friendly mechanisms for formulating and expressing logical constraints in a high-level programming language. These frontends significantly simplify the interaction between developers and solvers, enabling a more intuitive and streamlined workflow for formal verification tasks. By handling complex SMT solver interactions, frontends allow developers to focus on higher-level logical reasoning and system design rather than solver-specific details, thereby promoting correctness and reliability.

SMT.ML is a frontend for SMT solving that features a single backend that uses the Z3 [17] solver. While SMT.ML successfully bridges the gap between OCaml and SMT solving, it currently supports a single SMT solver. Although Z3 is a widely used and powerful solver, this limitation constrains the versatility of the tool, as users may prefer or benefit from leveraging alternative SMT solvers.

Moreover, state-of-the-art SMT solvers have differences in their implementations and algorithms used to handle different theories [18–20], meaning that a solver's approach to efficiently solve a given problem is often unique when compared to other solvers. These dissimilarities can lead to solvers being tailored for specific target problems, thus having areas where they excel and outperform other SMT solvers [21].

1.2 Goals

Building on the motivation outlined above, this thesis aims to contribute to the development of SMT.ML by addressing its current limitation of supporting only a single SMT solver. To enhance the flexibility and expand the range of choices available to OCaml developers using SMT.ML, we propose extending its capabilities by integrating a new backend that supports the cvc5 SMT solver [22]. As the latest solver in the Cooperating Validity Checker (CVC) series, cvc5 has recently gained recognition as a versatile and efficient tool for solving complex satisfiability problems across a wide array of logical theories [23]. Furthermore, having multiple solvers with distinct characteristics and capabilities available will boost SMT.ML's overall flexibility and utility by allowing users to select the SMT solver that best suits a given satisfiability problem.

To achieve this objective, a secondary goal must be established. Currently, the cvc5 SMT solver lacks native bindings for the OCaml programming language. Such bindings are crucial for the development of a new SMT.ML backend for cvc5 as they are the bridge between the solver and the OCaml programming language. Consequently, a key component of this thesis involves the development of OCaml bindings for cvc5 to enable seamless interaction between OCaml programs and the solver. Although these bindings are necessary for the integration of a new SMT.ML backend for cvc5, they will also be valuable for any developers seeking to incorporate the cvc5 SMT solver into other OCaml-based tools.

1.3 Contributions

The specific contributions of this thesis are three-fold:

1. The integration of a new backend for the cvc5 SMT solver in SMT.ML. This thesis extends SMT.ML's current architecture by integrating a new backend that supports the cvc5 SMT solver. This integration significantly broadens the range of SMT solvers available to users, offering them the flexibility to select the most suitable solver for their specific needs. The new backend not only enriches SMT.ML 's functionality but also introduces a multi-backend design that allows other new SMT solvers to be easily added to SMT.ML, while maintaining a uniform and coherent structure that abstracts from users solver-specific implementation details.

2. The development of OCaml bindings for the cvc5 SMT solver. As a necessary step for the integration of the cvc5 backend, this thesis presents the development of native OCaml bindings for the cvc5 SMT solver. These bindings are crucial in the process of extending SMT.ML with a new backend as they enable seamless communication between the OCaml environment and the cvc5 solver. Moreover, they extend their utility beyond SMT.ML by offering OCaml developers the tools to integrate cvc5 into other OCaml-based projects. The development process is thoroughly documented, providing a valuable resource for future work involving the integration of other SMT solvers or similar tools into OCaml.

3. The evaluation of SMT.ML against an extensive dataset of SMT queries. To assess the effectiveness

and performance of the multi-backend SMT.ML version, this thesis includes a comprehensive evaluation using a large dataset of SMT queries. More concretely, we used a set of queries that were generated by a symbolic execution engine, Owi [5], while executing the Test-Comp 2023 [24] symbolic execution benchmark suite. This evaluation compares the performance of the newly integrated cvc5 backend with the existing Z3 backend, highlighting the strengths and weaknesses of each solver across various problem domains. The results offer insights into the practical implications of using different SMT solvers within SMT.ML, providing guidance for developers on selecting the appropriate solver for specific verification tasks.

1.4 Thesis Outline

We structure the remainder of this thesis as follows. Chapter 2 covers some relevant related work such as SMT solver applications, cvc5's architecture, and frontends for SMT solvers. Chapter 3 presents some background on language interoperability in the OCaml programming language, Z3's OCaml bindings, SMT.ML's overview and architecture, garbage collection in OCaml, and memory management in the cvc5 SMT solver. Chapter 4 presents the OCaml bindings for the cvc5 SMT solver. Chapter 5 provides a comprehensive overview of how SMT.ML incorporates multiple backends for various SMT solvers. In Chapter 6 we evaluate the correctness and efficiency of the developed OCaml bindings for the cvc5 SMT solver. Chapter 7 concludes the thesis, summarizing our contributions and pointing out potential avenues for future work.

Chapter 2

Related Work

There is a vast body of work in SMT solving, e.g., development of SMT solvers [17, 22, 25], applications for SMT solving [3, 26, 27], among others [13, 14, 28]. In this chapter, we focus mainly on some of the applications of SMT solvers (Section 2.1), on the cvc5 SMT solver alongside its predecessors (Section 2.2), and on the importance and role that frontends for SMT solvers play in SMT solving (Section 2.3).

2.1 SMT Solvers

The most well-known constraint satisfaction problem, and overall one of the most fundamental problems of computer science and mathematical logic, is the propositional satisfiability problem, also known as SAT. Introduced by Cook in 1971 as being an NP-complete problem [29], SAT consists of deciding whether a formula composed of Boolean variables and propositional logical connectives can be made *true* by assigning *true* or *false* to its variables [30].

Although SAT has become the target reduction problem for multiple combinatorial problems, some problems are more naturally described with more expressive logics, such as first-order logic. Solvers that make use of such formulations are commonly called *Satisfiability Modulo Theories*, or SMT solvers [31]. SMT solvers have gained significant attention in recent years due to their broad applicability and ability to address complex computational problems in a programmatic way. Their applicability ranges from software verification [4, 6, 11], test-case generation [32], to artificial intelligence [33, 34], and many others.

SMT solvers build upon the foundations laid by SAT solvers by incorporating decision procedures for various first-order theories, such as linear arithmetic, bitvectors, arrays, and uninterpreted functions. The combination of these theories allows SMT solvers to be more expressive, making them suitable for a wider range of applications. A notable approach in SMT solving is the *DPLL(T)* framework [35], which extends the classical DPLL procedure used in SAT solving to handle combinations of theories efficiently.

Historically, SMT solving has seen significant advancements driven by both academic research and

practical needs. Early SMT solvers like CVC [36] and its successors (CVC Lite [37], CVC3 [38], CVC4 [39], and the recent cvc5 [22]) have demonstrated the evolution from basic theory solvers to highly optimized, feature-rich tools. Similarly, Z3 [17], developed by Microsoft Research, has set a benchmark in the field due to its efficiency and integration with various software verification tools. Yices [25, 40], developed by SRI International, is another prominent solver known for its powerful support of different theories and optimization capabilities.

In addition to these solvers, the SMT community has fostered a competitive and collaborative environment through the annual SMT-COMP [23] competition, where solvers are evaluated on a diverse set of benchmarks [13]. These benchmarks cover a wide range of theories and application domains, helping to drive the innovation and improvement of SMT techniques. The competition has played a crucial role in identifying the strengths and weaknesses of current solvers, encouraging the development of more robust and efficient algorithms.

One key area of SMT solver application is software verification, where tools like Boogie [6], and Why3 [11] leverage SMT solvers to prove the correctness of programs. In automated test-case generation, tools like KLEE [32] use SMT solvers to systematically explore the execution paths of programs, ensuring thorough testing. Moreover, in the field of artificial intelligence, SMT solvers have been utilized for planning and scheduling tasks, such as in robotics [33] and automated reasoning [34].

Despite these advancements, SMT solvers still face challenges, particularly in handling the complexity of combining multiple theories and scaling to very large problem instances. Ongoing research is focused on improving the efficiency of solver algorithms, better handling of quantifiers, and extending the range of theories that can be efficiently solved. Furthermore, there is an increasing interest in applying SMT solvers to emerging areas such as cybersecurity, where they can be used for vulnerability analysis and protocol verification [41, 42].

Overall, the continued development and refinement of SMT solvers are critical for addressing the growing complexity of computational problems in various domains. The interplay between theory and practice in this field ensures that SMT solvers will remain a vital tool for both researchers and practitioners in the coming years.

2.2 Cooperating Validity Checker (CVC)

The Cooperating Validity Checker series of solvers has always been on the front line of state-of-the-art SMT solving and started with the Stanford Validity Checker (SVC) [36], followed by the first Cooperating Validity Checker (CVC) [43], CVC Lite [37], CVC3 [38], CVC4 [39], and the sixth (currently latest) generation of this validity checker series is cvc5 [22].

cvc5 represents a significant leap forward in the world of SMT solvers. Building upon the successful code base and architecture of its predecessor, CVC4 [39], it takes the legacy to new heights, and that is one of the main reasons for the deviation from the previous solvers naming convention (the use of capital

letters). cvc5 stands out among SMT solvers for its extensive support of theories, encompassing all standard SMT-LIB [13] theories as well as a wide array of non-standard ones and theory extensions (e.g., separation logic, theory of finite sets and relations, the extension of the theory of reals with transcendental functions, among others). Moreover, it extends beyond conventional SMT solving, boasting advanced capabilities like higher-order reasoning and syntax-guided synthesis (SyGuS)[44]. A high-level overview of its architecture and core components can be found in Figure 2.1.

At the heart of cvc5 lies the *SMT Solver* module, a pivotal component entrusted with the handling of all SMT queries. Beyond its primary role in checking satisfiability (SAT), this module is also tasked with constructing models for satisfiable input formulas and extracting critical components such as assumptions, cores, and proof objects for unsatisfiable (UNSAT) formulas. Its fundamental modules include:

- Preprocessor - is responsible for applying a sequence of satisfiability-preserving transformations to each formula within an input problem. These transformations encompass tasks such as simplifying the global formula, eliminating terms, and potentially converting the formula from one logic to another (e.g., from non-linear integer arithmetic to a bit-vector problem).

- Rewriter - the Rewriter is tasked with the conversion of terms into semantically equivalent normal forms using a set of predefined rules. All major components of cvc5 utilize this module to ensure they operate with normalized terms, which simplifies their implementation and enhances overall efficiency. The rewriting process involves applying rules that handle simplifications, such as $x + 0 \rightarrow x$, and operator eliminations, for example, $x \leqslant y \rightarrow y + 1 > x$ (assuming x and y have integer sort).

- Propositional Engine - serves as the coordinator for the CDCL(τ) SAT solver, responsible for taking the Boolean abstraction of the input formula and generating a satisfying assignment for that abstraction. Its key components include the Clausifier and a propositional SAT solver (more specifically, a customized variant of MiniSat [45]). The Clausifier's role is to transform the input Boolean abstraction formula into Conjunctive Normal Form (CNF), and this CNF representation is subsequently provided to the SAT solver. The extended version of MiniSat incorporates a Decision Engine, which is utilized to create tailored decision heuristics for the solver.

- Theory Engine - carries the responsibility of verifying the consistency and handling the theory literals asserted by the Propositional Engine. Within this module, there is a Combination Engine that coordinates multiple theories when necessary and a Quantifiers Module that manages input sub-formulas containing quantifiers. The Theory Engine also houses Theory Solvers, expanding cvc5's support to all theories standardized in SMT-LIB alongside specific non-standard theories and theory extensions.

Additionally to standard satisfiability checking, cvc5 provides functionalities such as abduction, interpolation, syntax-guided synthesis (SyGuS), and quantifier elimination. All of these features are implemented as solvers and are built on top of the SMT Solver module. The Abduction Solver and Interpolation Solver

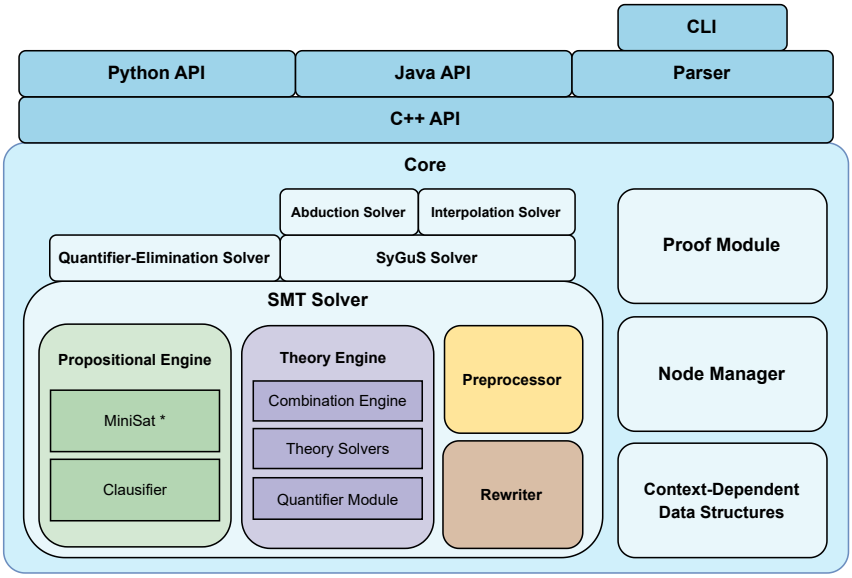

Figure 2.1: High-level overview of cvc5's architecture.

are both SyGuS-based and thus are built on top of the SyGuS Solver, as can be seen in the architecture diagram (Figure 2.1).

Within cvc5's core, there's also the *Proof Module*, which is a complete overhaul of its predecessor (CVC4) proof system. The former system suffered from incompleteness and various architectural limitations. This revamped module generates comprehensive proofs for nearly all of its theories, rewrite rules, internal SAT solver, and theory combination engines in a linear time frame at most. These proofs are sufficiently detailed to allow efficient verification, achievable within a polynomial time frame. Moreover, this module equips cvc5 with the capability to produce proofs in various formats, including those supported by LFSC [46] and Lean 4 [47] proof checkers, as well as Isabelle/HOL [48] and Coq [10] proof assistants.

Formulas and terms are represented as nodes in a directed acyclic graph (DAG) that is managed by the *Node Manager*, which is also a vital module in cvc5's core. In specific applications of SMT solvers, multiple satisfiability checks with similar assertions are necessary. To accommodate this, the SMT-LIB standard incorporates commands (push and pop) for saving and restoring sets of user-level assertions, enabling the solver to reuse prior work and reduce startup costs. In the case of cvc5, which relies heavily on the current assertions, it must save and restore the state when the user pushes or pops. Likewise, when the SAT solver makes decisions or backtracks, each theory solver must do the same. To facilitate these operations, cvc5 introduces the concept of context levels, which increase with each push and decrease with each pop. It also implements *Context-Dependent Data Structures* that function similarly to mutable data structures that are linked to context levels and automatically save and restore their state as the context level changes. This approach allows all state data associated with a level to be freed simultaneously by releasing the corresponding region, optimizing memory management.

cvc5 offers a streamlined, extensive, and fully-featured C++ API that not only serves as the primary interface for the Parser module but also forms the foundation for all other language bindings, including a

Python API and a Java API. Additionally, cvc5 supplies a command-line interface (CLI) built on top of the Parser module, supporting input languages such as SMT-LIB [13], SyGuS2, and TPTP [49].

In order to showcase its capabilities, cvc5's performance was compared against one of the most widely used solvers, Z3 (v4.8.12)[17], and also against CVC4 [39] to reason about performance improvements over its predecessor. The comparison involved using command-line options and time limits similar to those in SMT-COMP [23] competitions. cvc5 ended up outperforming the other solvers by solving the most significant number of benchmarks. However, there were some instances where it solved fewer benchmarks, notably in quantifier-free linear integer arithmetic, quantifier-free equality and bit-vector divisions, and quantifier-free string benchmarks.

2.3 Frontends for SMT Solvers

Frontends play an essential role in making SMT solvers more accessible to the wider computer science audience. These interfaces often come with user-friendly input languages that are both more expressive and closer to real-world problem domains than the logics of existing solvers, streamlining user interaction. Frontends also facilitate integration with high-level programming languages, allowing SMT-solving capabilities to be seamlessly embedded into applications and formal verification processes.

The SMT-LIB [13] language is a solver-agnostic textual format supported by multiple solvers. Its primary objectives include providing standardised descriptions of background theories for SMT solvers, promoting common input and output formats, and establishing a comprehensive benchmark library. One method for supporting multiple solvers is through the use of the SMT-LIB language, which involves serialising a given formula into an SMT-LIB format and running the most suitable solver on the generated file. However, this strategy is not effective for applications that require solving a large number of formulas as solver interaction is mediated through the file system, incurring heavy I/O overhead. When performance is critical, SMT solvers should be integrated into the client application's code through frontends ends that allow this, such as SMT.ML.

PySMT [15] and Smt-Switch [16] are examples of frontends that support multiple solvers in Python and C++, respectively. These tools provide users with a high-level API for interacting with various SMT solvers, abstracting away low-level solver-specific details. Both frontends support five SMT solvers:

- PySMT: Z3 [17], cvc5 [22], Yices2 [25], Bitwuzla [50], and MathSAT [51];
- Smt-Switch: Z3, cvc5, Yices2, MathSAT, and Boolector [52].

Additionally, PySMT implements a solver-agnostic optimisation layer that simplifies input formulas before querying them to the solver, an approach similar to the one in SMT.ML. By allowing users to choose between multiple solvers, these frontends provide greater flexibility and potential performance improvements over those that support a single solver. This further motivates the need to extend SMT.ML with support for additional solvers.

Solver-aided languages, such as Rosette [14] and Why3 [11], offer a flexible approach to writing programs that interact with SMT solvers to reason about logical formulas in multiple first-order theories. Rosette extends the Racket programming language [53] with a symbolic compiler that translates solver-aided programs into logical constraints, enabling seamless interaction with SMT solvers for tasks like program synthesis [44, 54] and test generation [55]. Similarly, Why3, an OCaml-based platform, supports formal program verification by translating high-level program specifications into verification conditions that various SMT solvers can process.

The development of effective frontends for SMT solvers presents several challenges. One of the primary difficulties lies in balancing user-friendliness with the need to expose the full capabilities of the underlying solvers. Frontends must simplify problem formulation without unduly restricting expressiveness or computational efficiency. Another challenge is integrating these tools into existing software development environments. As SMT solvers are increasingly adopted in fields such as software engineering and systems design, frontends must be designed to integrate seamlessly with existing workflows.

In summary, frontends for SMT solvers are vital components of the SMT ecosystem, acting as intermediaries that connect advanced formal reasoning tools with end-users. As SMT solvers evolve and expand into new application domains, the development of robust, flexible, and user-friendly frontends will be essential to unlocking their full potential.

Summary

In this chapter, we presented the related work. We began by highlighting some of the key applications of SMT solvers [4, 6, 11, 32–34] and outlined the current state-of-the-art SMT solvers [17, 22, 25]. Following this, we introduced the CVC series of SMT solvers and detailed the architecture of the most recent solver in this series, cvc5 [22]. Finally, we analysed several popular frontends for SMT solvers [11, 13–16], emphasising their role in enhancing the accessibility of SMT solvers. The next chapter will provide background on language interoperability in OCaml, discuss Z3's OCaml bindings, offer an overview of SMT.ML, introduce the basics of garbage collection and its management in OCaml, and describe how the cvc5 SMT solver handles memory management.

Chapter 3

Background

This chapter provides some background on language interoperability in OCaml (Section 3.1), giving examples of how one can achieve seamless interoperability between OCaml and the C++ programming language. Then, it introduces Z3's OCaml bindings, how they work, and some problems associated with them (Section 3.2). Finally, in Section 3.3, we introduce SMT.ML's architecture and explain how it addresses some of the problems in Z3's OCaml bindings introduced in the previous section.

3.1 Language Interoperability in OCaml

Nowadays, it has become common to use multiple programming languages when developing software. Since every programming language has its pros and cons, combining multiple programming languages allows developers to leverage the pros of some languages while simultaneously avoiding their cons. For example, high-level languages offer a high level of abstraction, making it easier for developers to write code without having to worry about the underlying implementation details, but usually sacrifice some performance for the ease of development. In contrast, low-level languages provide features such as pointer arithmetic and the ability to access memory directly, allowing developers maximum control over hardware resources and to write highly performant and efficient code, but also introduce a steeper learning curve and are much more prone to errors (e.g., memory leaks, buffer overflows). This is why high-level programming languages (e.g., Python, JavaScript) are used for tasks such as web development and data analysis, whereas compilers and network protocols are written using low-level languages (e.g., C, C++, Rust).

However, different programming languages can be hard to combine due to differences in data representation, memory models, runtime environments, and type systems, which, most of the time, result in performance overhead, rendering the problem of cross-language interoperability as a challenging one [56]. Several tools and techniques address the challenge of cross-language interoperability, including:

- Foreign Function Interfaces (FFIs): a mechanism that allows programs to invoke functions or use

libraries written in other programming languages.

- Multi-language runtimes: an environment that allows developers to write code in multiple languages and execute it in a single runtime system [57, 58].
- Interface Description Language (IDL): a specification language used to describe a software component's interface in a language-independent manner, allowing different pieces of software written in different programming languages to communicate with one another [59].

We will set our focus on OCaml's *Foreign Function Interface* (FFI) [9] to C [60]. To do this, we are going to use two examples: a simple one to introduce OCaml's FFI syntax and a more complex one that illustrates the seamless integration of another programming language's (such as C++) code with OCaml code.

OCaml has an FFI that allows user-defined primitives, written in C, to be linked and called from OCaml functions, and that also allows these C primitives to call back to OCaml code. To understand how this mechanism works, let us consider a simple example. Suppose we want to call a C function that prints an integer value to the terminal from OCaml code. In order to do that, one has to follow these steps:

1. Declare on the C side the function that will be exported and that contains the desired functionality. From now on, we will also refer to these functions as *stubs*. Considering our simple example, the C function could be defined like in Listing 3.1. All declarations for this type of C functions follow the format of

   $$\texttt{value}\ c\_fname\,(\texttt{value}\ arg1,\ ...,\ \texttt{value}\ argx)$$

   where `value` corresponds to C's representation of all OCaml values. All parameters given to C functions from OCaml are of type `value`. It encodes objects of several base types (integers, floating-point numbers, strings, etc.), and it can also encode OCaml data structures (lists, records, etc.).

2. On the OCaml side, we have to declare an `external` declaration to be able to invoke the C function. The `external` declarations follow the following style

   $$\texttt{external}\ caml\_f : type\ =\ \text{“}c\_fname\text{”}$$

   where *type* is an OCaml function type. For our example, the external would look like the one in Listing 3.2. Note that the name given to the OCaml side function (*caml_f*) and the name of the stub (*c_fname*) do not need to be the same.

Having completed these steps, we can now call the function print_int on OCaml's side, and that will result in the invocation of the function print_int_stub on the C side.

The process for implementing C functions that interact with OCaml can be divided into three stages:

```c
#include <stdio.h>
#include <caml/mlvalues.h>

CAMLprim value print_int_stub(value v)
{
    int i = Int_val(v);
    printf("%d\n", i);
    return Val_unit;
}
```

Listing 3.1: Simple C primitive example.

```
external print_int : int -> unit = "print_int_stub"
```

Listing 3.2: External declaration for simple C primitive example.

1. Decoding the primitive arguments to extract C values from the received OCaml values
2. Computing the result for that given function (what is normally the function body)
3. Encoding the return value as an OCaml value

The first and third stages are not necessary when developing with a single programming language. However, as mentioned before, a multi-language paradigm introduces new challenges, and differences in data representation are one of them. In order to be able to use C native types in our defined primitives, we must first convert the OCaml values that are passed to the C function into C values so that these can be used in the remainder of the function body. The same applies to the third stage; to be able to use the C values resulting from the function call in our OCaml code, we must convert these into OCaml values. Some conversion macros are provided to convert the type `value` into C native types in the include header `<caml/mlvalues.h>`; for instance, if we wanted to convert a type `value` to a C integer we would use the macro `Int_val()` and, if we later wanted to convert that C integer back into a type `value` (so that we could use it in OCaml) we would use the macro Val_int(). The syntax for the conversion macros follows this style: for a C native type *foo*, `Foo_val()` is the macro that converts `value` into *foo* and the macro that converts from type *foo* into `value` is `Val_foo()`. A simple C function that takes an OCaml integer value as an argument and prints it could be implemented using the code presented in Listing 3.1.

Main Example

To see how we can easily integrate code written in another programming language into our OCaml program, let's consider a more complex example that creates and manipulates C++ singly-linked lists with OCaml. In Listing 3.3, we define several functions for creating and interacting with linked lists from the C++ Standard Template Library (`std::list`) that we want to call from our OCaml code. The first step to achieve this is to write the stubs that make use of OCaml's FFI. While OCaml's FFI is primarily intended for interoperability with the C programming language, it is worth noting that C++ provides a convenient feature that enables functions to have C linkage through the use of the `extern "C"` clause. This allows us to encapsulate our stubs within this clause, enabling seamless interaction between OCaml and C++.

```cpp
/* Instantiates a new list */
list<int> * init_list(int val){
    list<int> * l = new list<int>();
    l->push_front(val);
    return l;
}

/* Prepends a value to an existing list */
list<int> * prepend(list<int> * l, int val){
    l->push_front(val);
    return l;
}

/* Returns the length of a list */
size_t length(list<int> * l){
    return l->size();
}

/* Pops first element of a list */
int pop(list<int> * l){
    int val = l->front();
    l->pop_front();
    return val;
}
```

Listing 3.3: Partial C++ Singly Linked Lists example.

```cpp
CAMLprim value caml_stub_init_list(value v) {
  list<int> * p = init_list(Int_val(v));
  return Val_list(p);
}

CAMLprim value caml_stub_prepend(value v, value l) {
  list<int> *p = List_val(l);
  list<int> *q = prepend(p, Int_val(v));
  return Val_list(q);
}

CAMLprim value caml_stub_length(value l) {
  list<int> *p = List_val(l);
  int len = length(p);
  return Val_int(len);
}

CAMLprim value caml_stub_pop(value l){
  list<int> *p = List_val(l);
  int val = pop(p);
  return Val_int(val);
}
```

Listing 3.4: Stubs for C++ functions in Listing 3.3.

```cpp
static value Val_list(list<int> * p) {
  return caml_copy_nativeint((intnat) p);
}

static list<int> * List_val(value v) {
    return (list<int> *)Nativeint_val(v);
}
```

Listing 3.5: C functions for conversion of pointers to type `value`.

The complete example, together with all the required code to use it from OCaml, is given in the appendix (Listings 8.1 and 8.2).

In Listing 3.4, we can see the stubs for the functions represented in Listing 3.3 that will allow us to use the linked lists we create using C++ in OCaml. One should pay particular notice to the mechanism used to represent C pointers on the OCaml side. In earlier versions of OCaml, it was possible to deal with such pointers by simply casting them to type `value`; however, this stopped being supported in version 5.0 of OCaml. We deal with pointers that point outside of the heap by wrapping them within native integers (boxed 32-bit integers on a 32-bit platform and boxed 64-bit integers on a 64-bit platform). In the example of Listing 3.5, we define the functions `Val_list` and `List_val` that convert pointers from type `list<int>` to type `value` and vice versa. There are other ways to deal with these cases, such as storing the pointers in OCaml blocks (using special macros from the `<caml/memory.h>` header file), identifying the blocks with a special tag, and then use the blocks as OCaml values (see section 22.2.3 of [9] for further details).

The second step is to create an OCaml file with all the external declarations needed to interact with the stubs we created. Listing 3.6 illustrates these declarations for the functions given in Listing 3.3. Once we have the OCaml declarations for the stubs, we can create a simple OCaml program that creates a linked list, prepends elements, pops an element, and prints its size and content, shown in Listing 3.7.

Finally, to run our OCaml program, we need to compile it, and for that, we can use a tool like Dune

```
type cpp_list

external init_list :
    int -> cpp_list = "caml_stub_init_list"
external prepend :
    int -> cpp_list -> cpp_list = "caml_stub_prepend"
external length :
    cpp_list -> int = "caml_stub_length"
external pop :
    cpp_list -> int = "caml_stub_pop"
external print_list :
    cpp_list -> unit = "caml_stub_print_list"
external delete_list :
    cpp_list -> unit = "caml_stub_delete_list"
```

Listing 3.6: OCaml external declarations for stub code.

```
open Llists

let () =
  let head =
    init_list 3 |> prepend 2 |> prepend 1 in
  ignore (pop head);
  let size = length head in
  Printf.printf "size: %d\n" size;
  print_list head;
  delete_list head;
```

Listing 3.7: OCaml program that uses C++ linked lists.

```
(executable
  (name toy)
  (libraries llists)
  (modules toy))

(library
  (name llists)
  (flags (:standard -ccopt "$LDFLAGS" -cclib -lcpp_lists))
  (foreign_stubs
    (language cxx)
    (names stubs_lists))
  (modules llists))
```

Listing 3.8: Dune configuration file for toy example.

(OCaml's build system) that automatically finds and links in libraries required for our code to run. Since we are using an external file containing our C++ for the linked lists (cpp_lists.cpp), we first need to compile it into a shared object file (with a .so extension) so that the linker can find the symbols (functions) we created (since we are using a dynamic library this linking will be done at runtime), we also have to make a configuration file (dune), telling Dune that we are using foreign stubs written in C++ and instructing it on where to find the shared object file (using the `-cclib` and `-ccopt` flags). The dune file used for this example is in Listing 3.8.

In Figure 3.1, we can see a diagram that illustrates the interactions between the OCaml side and the C++ side of our example. On the OCaml side, we only have access to a pointer for the C++ list structure and pass it to our stub code whenever we want to perform an operation. This is why we need to have stubs that serve as an interface between the OCaml code and the C++ code that actually performs the computations on the C++ side. Without them, we would not be able to manipulate anything on the OCaml side because we only have access to a boxed integer representing an address of the C++ side (i.e., the memory address where the structure is stored). For instance, if we wanted to add a new node to the linked list from the OCaml side, we could use the `prepend` external function declared in Listing 3.6. This function receives an integer and a `cpp_list`, which is simply a pointer to the list we want to add a new value to. The `caml_stub_prepend` stub converts the arguments received from type `value` to the appropriate C++ types and then calls the corresponding C++ function that actually adds a new element to the list. Without a stub that performs this operation, we would not be able to achieve this from the OCaml side.

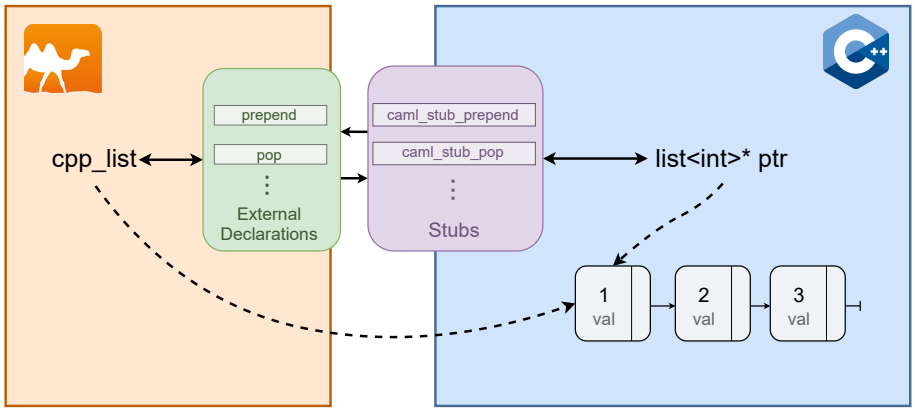

Figure 3.1: Interaction diagram between an OCaml program, the respective stubs, and a C++ program.

3.2 OCaml Bindings for Z3

The Z3 Theorem Prover [17], developed by Microsoft Research, stands as a highly efficient SMT solver that holds a prominent position in the industry. Its wide range of capabilities includes support for several theories, such as linear and real arithmetic, bit-vectors, arrays, datatypes, and uninterpreted functions. While its default input method for interacting with the Z3 solver is the SMT-LIB [13] language, it also officially supports bindings for multiple programming languages. These bindings enable users to interact with the solver through API calls from languages like C++, Java, Python, .NET, and OCaml, which act as proxies for communication with the solver via a C-based API. In this section, we will focus on Z3's bindings for OCaml.

While Z3's OCaml bindings are comprehensive, covering a wide range of features, specific modules play a pivotal role in their utilization, such as the `Solver` and the `Expr` modules. The `Solver` module provides a set of functions that enable direct interaction with the solver, like creating a solver, adding one or multiple constraints, checking the satisfiability of the constraints added, and potentially getting a model or extracting unsatisfiable cores from the asserted formulas. In order to add constraints to the solver, one must first encode them into an expression that the solver can process, and that is where the module `Expr` is used. The module `Expr` defines a type `Expr.expr` that encodes general terms, which are the building blocks for every formula. This type can encode multiple things, ranging from a simple *true* boolean value to more complex expressions such as $g(a, f(b)) = b \wedge f(b) = c$.

In Listing 3.9, we demonstrate a simple program that makes use of Z3's OCaml bindings to check the trivially satisfiable formula $x > 0$. We start by creating a context and a solver with no additional initial configurations. The context variable created (`ctx`) is an essential object in the Z3 API; it is used to manage Z3 objects, ensuring that all other objects created and operations performed are bound to the same instance of the environment. Note that the context variable is used in all operations that create other Z3 objects; this allows Z3 to correctly manage the resources, scope, and life-cycle of the objects involved in the logical assertions, solving, and other operations.

```
let ctx = mk_context [] in
let solver = Solver.mk_simple_solver ctx in

let int_sort = Arithmetic.Integer.mk_sort ctx in
let x : Expr.expr = Expr.mk_const_s ctx "x" int_sort in
let zero : Expr.expr = Arithmetic.Integer.mk_numeral_i ctx 0 in

let formula : Expr.expr = Arithmetic.mk_gt ctx x zero in
Solver.add solver [formula];
match Solver.check solver [] with
| Solver.SATISFIABLE -> Printf.printf "SAT.\n"
| _ -> Printf.printf "UNSAT/UNKNOWN.\n"
```

Listing 3.9: Simple Z3 OCaml bindings example.

After creating a context and a solver, we create a sort type (in this instance, `int_sort`) that we will use to specify the type for the possible values of the constant `x`, and we also create an expression that encodes the integer 0. Having created all the terms that we will need, we can construct the formula $x > 0$. Finally, we add the formula to the solver and check its satisfiability. This example is not only useful to help us understand how Z3's OCaml bindings work but also because it illustrates how the typing of the different Z3 expressions is done. It is clear to see that on the OCaml side, different expressions are all encoded using the type Expr.expr, and type checking is later performed by Z3 during runtime using information provided by sort variables like the one we defined. For instance, by applying `int_sort` to constant `x`, we restricted possible values of `x` to integers, and Z3 will consider this internally. Let us now consider some of the problems of this encoding.

No Type Information is Kept

Using type Expr.expr for every expression on the OCaml side can quickly become a problem, as we will now see. Let us consider an example that makes use of uninterpreted functions (EUF), another theory supported by Z3. In Listing 3.10, we have an OCaml program that uses Z3 to check the satisfiability of a formula that contains an uninterpreted function (`foo`).

Uninterpreted functions are symbolic function representations that have no predefined semantics. Their behaviour is not explicitly defined but is instead characterized by the constraints and relationships imposed on them within a logical formula. They act as placeholders for any possible function that satisfies these constraints. In our example, we just define the sort of the argument that the function receives and the sort of the return value, so foo is a function that receives an integer and returns an integer.

In the example of Listing 3.10, our goal is to check the satisfiability of the following formula:

$$\texttt{foo(value)} + 2 \geqslant 2$$

To do so, we must first construct the term `foo(value)` by creating `value` and applying the function `foo` to it. This is done through the function `mk_app` that receives a context, a function declaration, and a list of Z3 expressions (`Expr.expr`) and also returns a Z3 expression. Since the application function takes in a list of Z3 expressions, we can apply function `foo` to any value as long as it is encoded into a Z3 expression.

```
let ctx = mk_context [] in
let solver = Solver.mk_simple_solver ctx in

let int_sort = Arithmetic.Integer.mk_sort ctx in
let two = Arithmetic.Integer.mk_numeral_i ctx 2 in
(* Function Declaration: foo : int -> int *)
let foo = FuncDecl.mk_func_decl_s ctx "foo" [ int_sort ] int_sort in

let str_sort = Seq.mk_string_sort ctx in
let value = Symbol.mk_string ctx "value" in
let str_const = Expr.mk_const ctx value str_sort in

let foo_app = Expr.mk_app ctx foo [ str_const ] in
(* foo(value) + 2 >= 2 *)
let formula = Arithmetic.mk_ge ctx
    (Arithmetic.mk_add ctx [ foo_app ; two ]) two in

Solver.add solver [formula];
match Solver.check solver [] with
| Solver.SATISFIABLE -> Printf.printf "SAT.\n"
| _ -> Printf.printf "UNSAT/UNKNOWN.\n"
```

Listing 3.10: Type violation of function declaration using Z3 OCaml bindings.

Regardless of the Z3 sorts of the provided terms, OCaml will not signal any errors because it is not able to check whether the Z3 expressions have the same sort as the sort defined in the uninterpreted function declaration. In our example, we create an ill-typed expression by passing an expression of string sort as an argument to function `foo`, and, as already mentioned, OCaml's type-checking mechanism will not be able to detect this type violation. If we compile and run our example, the program will crash during runtime, and an exception will be thrown stating that there is a sort mismatch between the function declaration and the sort of the supplied argument.

No Support for Concurrency

Another problem in the OCaml bindings for Z3 is the lack of support for concurrency. To illustrate this limitation, let's examine another scenario. In Listing 3.11, we have an OCaml program that uses two threads to query a solver concurrently. One of the threads assesses the satisfiability of a trivially satisfiable expression, while the other thread does the same but for an unsatisfiable expression. Note that because we have a single solver and a single context variable, there is the need to reset the solver after checking for the satisfiability of one of the formulas; otherwise, the subsequent results would be influenced by prior checks.

Z3's behavior when dealing with concurrency is undefined, and since the bindings do not offer a concurrency control mechanism if we run the code from our example, we are not guaranteed to have a correct outcome; more often than not, it may result in a runtime error, such as a segmentation fault.

3.3 SMT.ML

We are now going to introduce SMT.ML, a frontend for SMT-solving that serves as an abstracted constraint-solving wrapper and uses Z3 [17] as its backend solver. SMT.ML allows users to build logical formulas and

```
1  (* Check 0 = 0 *)
2  let sat_formula_thread () =
3    let sat_formula = Boolean.mk_eq ctx zero zero in
4    check_formula solver sat_formula
5  (* Check 0 != 0 *)
6  let unsat_formula_thread () =
7    let unsat_formula = Boolean.mk_not ctx (Boolean.mk_eq ctx zero zero) in
8    check_formula solver unsat_formula
9
10 let () =
11   let sat_thread = Thread.create sat_formula_thread () in
12   let unsat_thread = Thread.create unsat_formula_thread () in
13   Thread.join sat_thread;
14   Thread.join unsat_thread
```

Listing 3.11: Concurrent program using Z3 OCaml bindings.

check their satisfiability, all while abstracting the intricate syntax for interacting directly with the Z3 solver.

In Figure 3.2, we present a high-level overview of the architecture of SMT.ML. SMT.ML can function as a module in other OCaml programs or operate independently as a standalone tool. When used as a command line tool, SMT.ML expects an input file in the SMT-LIB [13] language, which is then parsed by the Parser module (step 0 in Figure 3.2), and the results are displayed on the command line. Conversely, when used as a module, the given logical expressions are directly transmitted to the Simplifier module without the need for prior parsing.

The Simplifier module is responsible for applying simplifications to the formula in an attempt to reduce complexity and make it easier to handle the terms of the formula, all this while preserving its original semantics. This is done in step 1 of Figure 3.2. For example, when dealing with bit-vectors, if we consider the operators `concat` and `extract`, a possible simplification applied to the formula is:

$$\texttt{concat}(\texttt{extract}(x, h, m),\ \texttt{extract}(x, m, l)) = \texttt{extract}(x, h, l)$$

Once the formula is simplified, it is then passed to the Cache module. This module is responsible for storing previously queried formulas to the solver so that their results can be later reused instead of

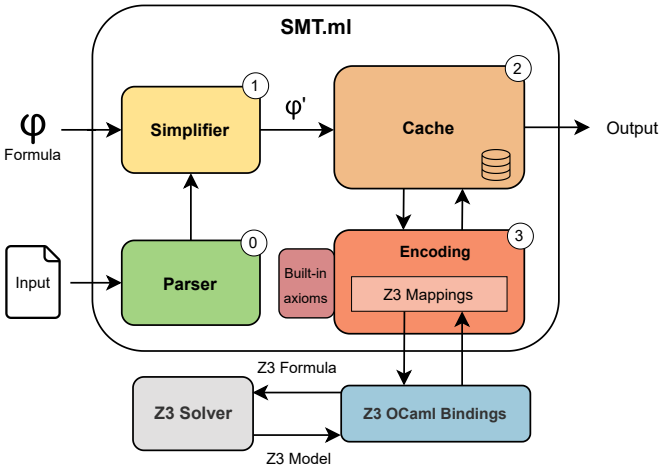

Figure 3.2: High-level overview of SMT.ML's architecture.

NUMERAL VALUES
$num ::= i8 \mid i32 \mid i64 \mid f32 \mid f64$

TYPES
$t ::=$ `Ty_int` | `Ty_str` | `Ty_bitv` int | `Ty_bool` | `Ty_fp` int | `Ty_real`
| `Ty_unit` | `Ty_list` | `Ty_app`

VALUES
$v \in \mathcal{V}_{smt} ::=$ `true` | `false` | `unit` | `int` | `real` | `str` | `num` | `list` v

EXPRESSIONS
$e \in \mathcal{E}_{smt} ::=$ $v \mid x_t \mid$ `unop`$(\texttt{op}, t, e) \mid$ `binop`$(\texttt{op}, t, e, e)$
$\mid$ `triop`$(\texttt{op}, t, e, e, e) \mid$ `relop`$(\texttt{op}, t, e, e) \mid$ `cvtop`$(\texttt{op}, t, e)$
$\mid$ `naryop`$(\texttt{op}, t, \texttt{list}\ e) \mid$ `list` e

COMMANDS
$c \in \mathcal{C}_{smt} ::=$ `assert` $e \mid$ `check_sat` (`list` e) $\mid$ `declare`$(x_t) \mid$ `exit`
$\mid$ `get_model` $\mid$ `get_value` $e \mid$ `pop` $int \mid$ `push` $int \mid$ `reset`

Figure 3.3: SMT.ML's syntax.

performing redundant computations. Before checking if a given formula has been previously computed, all formulas go through a normalization process [61]. The normalization process involves the simplification of complex expressions, the standardization of variable names, and the removal of irrelevant details from the input. This allows cache hits to be maximized.

Lastly, in step 3 of the diagram, the formula is transformed into a Z3 formula, using the Z3 OCaml bindings, so that the solver can reason about it. As seen in Section 3.2, this includes encoding all terms of the formula into Z3 types (e.g., `Expr.expr`). After the conversion, the resulting formula can be queried to the Z3 solver. The model that Z3 returns is then stored in the cache so that it can later be reused.

Figure 3.3 presents the syntax of SMT.ML. There are two main syntactic categories: expressions, that denote values, and commands, that represent instructions given to the solver.

Expressions Expressions in SMT.ML are built using values and operators. Values, v, include boolean constants (`true` and `false`), integers (`int`), reals (`real`), strings (`str`), numeral constants (`num`), and lists of values. Numerals are machine integers (8, 32 and 64-bit) or IEEE 754 floating-point numbers [62] (32 and 64-bit). Expressions, e, are constructed by combining values with *symbolic variables* of the form x_t, where x denotes the variable and t its type.

Expressions are constructed through the application of a variety of operators. SMT.ML includes unary (`unop`$(\texttt{op}, t, e)$), binary (`binop`$(\texttt{op}, t, e, e)$), ternary (`triop`$(\texttt{op}, t, e, e, e)$), and n-ary (`naryop`$(\texttt{op}, t, \texttt{list}\ e)$) operators. Additionally, it includes relational operators (`relop`$(\texttt{op}, t, e, e)$) for comparisons and conversion operators (`cvtop`$(\texttt{op}, t, e)$) for type casting or conversion between value types.

Commands SMT.ML provides a set of commands to manipulate and interact with the SMT solvers. Commands, c, include typical solver instructions such as `assert` e, that asserts a given formula to the solver, and `check_sat` (`list` e), that checks the satisfiability of a list of formulas. The command `declare`(x_t) is used to declare a new symbolic variable x of type t. Additional commands include `exit`,

```
let solver = Solver.create () in
let x = mk_symbol Symbol.("x" @: Ty_int) in
let formula = x > make (Val (Int 0)) in

let r = Solver.check solver [ formula ] in
match r with
| `Sat -> Printf.printf "sat\n"
| `Unsat | `Unkown -> Printf.printf "unsat/unknown\n"
```

Listing 3.12: Simple SMT.ML example.

used to terminate the interaction with the solver, `get_model`, to retrieve a model when given a list of satisfiable formulas, and `get_value` e, that returns a satisfiable value for the given expression. For state manipulation, SMT.ML provides the `push` int and `pop` int, commands to introduce or remove a number of assertion levels, respectively. Finally, the `reset` command is used to reset the solver's state allowing for a fresh set of assertions and commands to be processed.

Let us now consider some of the advantages of using SMT.ML instead of Z3's OCaml bindings directly.

User-Friendly Syntax

SMT.ML abstracts interactions with the Z3 solver, which results in a simpler and more concise syntax. This enables developers to set their focus on developing programs that use high-level logical constraints without having to delve into the intricate details of Z3's extensive OCaml bindings.

For instance, Listing 3.12 illustrates how we could create a program that builds the formula $\mathrm{x} > 0$ and queries it to the solver but this time using SMT.ML's syntax. When compared to the program of Listing 3.9 that used Z3's bindings, we can see that we are able to achieve the same goal but by using a much more concise and readable syntax.

Static Type-Checking

Another advantage of using SMT.ML instead of Z3's OCaml bindings directly is that SMT.ML's expressions are typed OCaml expressions with different types for specific primitives (e.g., integers, strings, booleans), meaning that OCaml's compiler can reason about them and correctly type-check them. For instance, we can see in Listing 3.12 that every element used in the formula queried to the solver is annotated with its respective type (this is done through the `@:` operator).

This allows expressions to be correctly associated with their types and ensures that any ill-typed expressions are detected at compile time, thus preventing runtime crashes due to type violations. We have already seen this is possible when using Z3's OCaml bindings directly.

Support for Additional Operators

Besides encapsulating interactions with the Z3 SMT solver, SMT.ML supports additional operators that Z3's API does not provide and that developers may need when constructing and interacting with logical formulas. One example of an operator that is not supported natively by Z3's API but is implemented in

```
let encode_ceil_unop =
  let x_int = Arithmetic.Real.mk_real2int ctx x in
  Boolean.mk_ite ctx
    (Boolean.mk_eq ctx (Arithmetic.Integer.mk_int2real ctx x_int) x)
    x_int
    Arithmetic.(mk_add ctx [ x_int; Integer.mk_numeral_i ctx 1 ])
```

Listing 3.13: `Ceil` operator encoding in SMT.ML.

SMT.ML is the `Ceil` unary operator that maps a real number x to the smallest integer greater than or equal to x (commonly represented as $\lceil x \rceil$). In Listing 3.13, we present a snippet of an SMT.ML function that encodes the `Ceil` unary operator for real-typed values into Z3 expressions.

3.4 Garbage Collection in OCaml

Garbage collection (GC) is the process of reclaiming unused storage space in order to promote program efficiency and optimization [63]. At their core, garbage collectors identify and reclaim memory that is no longer in use by programs, which otherwise would lead to memory leaks and eventual degradation of system performance. Garbage collection is such a critical process that many programming languages already feature garbage collectors in their design in order to automate this entire process [64], especially in programming languages where direct memory manipulation is abstracted away from programmers (i.e., OCaml, Python, Java), allowing for safer and more reliable code. John McCarthy first introduced garbage collection in 1959 to simplify memory management in the Lisp programming language [65], and it has since then become a widely researched topic with various garbage collection algorithms being introduced [66]. In this section, we will focus on the OCaml programming language and how it handles garbage collection.

Like most modern programming languages, OCaml performs garbage collection automatically, meaning that users do not have to worry about manually managing memory allocation/deallocation. This significantly reduces the risk of memory leaks and improper memory usage. OCaml programs dynamically allocate blocks of memory (i.e., contiguous sequences of words in RAM) to store values like tuples and records, among others. As the number of requests for block allocation grows, it is the garbage collector's job to identify blocks that are no longer being used and can, therefore, be freed, allowing the program to continue executing in its normal course of action. We can think of the heap as a pool of memory regions that the operating system has reserved for the execution of a given program. Whenever a value needs to be allocated, this pool is checked to see if there are any free blocks available to satisfy this request. If there are no available blocks, then the garbage collector is invoked to determine which blocks are still being used and which ones are not. Several garbage collection algorithms can achieve this goal, and all of them have advantages and disadvantages that can significantly impact performance.

OCaml has a generational garbage collector [67, 68] that employs a mark-and-sweep algorithm to detect which blocks are still in use and which ones are not. It is called *generational* because OCaml's

garbage collector divides the available heap space into two different memory regions:

- Minor heap - a smaller, fixed-size region where most of the blocks are allocated initially;
- Major heap - a larger, variable-sized region to accommodate blocks that have been reachable for longer.

In functional programming, it is common for “young” blocks to have short lifespans, becoming unreachable and eligible for garbage collection shortly after their creation. Meanwhile, blocks that survive initial collections (“old” blocks) are likely to remain in use for a longer period. This is often referred to as the *generational hypothesis*. By segregating blocks based on their expected lifespans, garbage collectors are able to optimize memory management and handle each group of blocks differently.

Minor Heap

The minor heap consists of one contiguous chunk of virtual memory containing a sequence of OCaml blocks. It is where new objects (up to a certain size) are allocated, and if memory is available, the allocation is quick and constant-time. However, if the minor heap is full, all live blocks (which, according to the generational hypothesis, should be few) are moved to the major heap since they are likely to have a longer lifespan. Typically, the garbage collector *stops the world* (i.e., halts the application while it runs). Hence, it is crucial for the garbage collection process to finish quickly so as to minimize interruptions and to allow the application to resume promptly.

Major Heap

The major heap is where blocks with longer lifespans reside. It consists of multiple non-contiguous chunks of virtual memory, each containing live blocks mixed with regions of free memory. The major heap is generally much larger than the minor heap, which means that garbage collection takes longer but also occurs less frequently. This is based on the assumption that objects in the major heap tend to be reachable for longer periods. A mark-and-sweep algorithm is used to trace the liveness of blocks; this algorithm consists of three phases:

- *Mark* phase - in this phase, all blocks are scanned and marked by setting a bit in the tag of the block header (known as the colour tag);
- *Sweep* phase - the phase where all blocks that were not marked as being live are collected;
- *Compact* phase - in this phase, live blocks are relocated to a fresh heap in order to avoid fragmentation and improve memory locality. This phase happens much less frequently than the *mark* and *sweep* phases. Still, it is also crucial as without it, it would be possible to have several smaller blocks available that are not able to accommodate a bigger object.

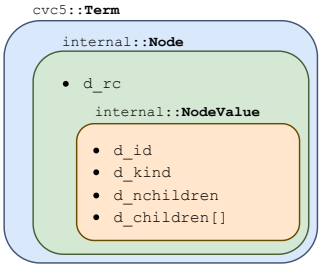

Figure 3.4: cvc5 internal `Term` structure.

The *mark* and *sweep* phases are executed incrementally, allowing them to run over smaller parts of the major heap without halting the entire program. Only the *compact* phase requires stopping the world, ensuring blocks can be moved without that being noticed by the application. Still, again, this operation is rare, so it does not have a significant impact on program performance.

3.5 Memory Management in cvc5

SMT solvers are tasked with determining the satisfiability of logical formulas with respect to background theories, a task that often requires substantial computational resources. Additionally, solvers must perform memory management to handle large, complex formulas and avoid resource exhaustion efficiently. In this section, we will focus on the mechanisms employed by the cvc5 [22] SMT solver to achieve efficient memory management.

In the cvc5 SMT solver, all formulas are internally represented as nodes and are stored in a pool of nodes that is managed by an instance of the `NodeManager` class. Nodes are a subclass of the more general `Term` class that is used, at the API level, to represent all types of solver expressions and whose internal structure is depicted in Figure 3.4. For instance, the formula $x > 1$ is built using two `Term` objects (x and 1) and is also itself a `Term` object. Analysing the internal structure of the `Term` class, we can see that it essentially serves as a wrapper for the `internal::Node` class. Additionally, the `internal::Node` class acts as a reference-counted encapsulation for the `internal::NodeValue` class, storing the node's reference count as an attribute (`d_rc`). Finally, the `internal::NodeValue` class is the one that actually contains the implementation of the node, holding several attributes such as its unique identifier (`d_id`), kind (`d_kind`), number of children (`d_nchildren`), and child nodes (`d_children[]`).

Reference counting [69] is a memory management technique used to track the number of references or pointers to a resource such as an object or block of memory. Each object holds a reference count, and each time a new reference to the object is created (e.g., when a pointer is copied via assignment), the pointed-to object's count is incremented. Conversely, when an existing reference to an object is eliminated, the count is decremented (illustrated in Figure 3.5). When an object's reference count reaches zero, it indicates that the memory it occupies can be reclaimed, as no other objects reference it. References from an object with a zero reference count do not affect an object's liveness. Thus, when an object's

memory is reclaimed, any objects it references will also have their reference counts decremented. This can lead to the recursive decrementing of reference counts and, consequently, object reclaiming (e.g., in Figure 3.5, setting the count of the object with value $x > 4$ to zero would also imply decrementing the count of the object representing the literal 4, and, therefore, both object's memory would be reclaimed).

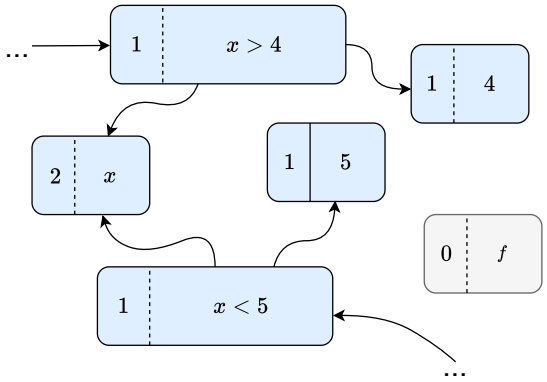

Figure 3.5: Reference counting illustration.

This mechanism allows memory to be automatically reclaimed when objects are no longer needed, which is the case in the cvc5 solver. Whenever a node's reference count gets decremented and becomes zero, the node is considered to be a "zombie" node, and it is placed in a separate pool alongside other nodes whose count is also zero. The nodes in this pool are periodically reclaimed; more concretely, they are reclaimed whenever the pool's size reaches a pre-determined, fixed number. Such efficient memory management mechanisms are crucial for maintaining the solver's efficiency and effectiveness, particularly as the complexity and size of the formulas it tackles continue to grow.

Summary

This chapter provided essential background for the remainder of this thesis. We began by discussing the mechanisms and challenges associated with language interoperability in OCaml. Following this, we introduced the OCaml bindings for Z3, one of the most widely used SMT solvers, and explained their usage along with potential issues. We then presented SMT.ML, detailing its architecture, its supported syntax and how it addresses some of the challenges associated with Z3's OCaml bindings. This was followed by an overview of garbage collection fundamentals and their implementation in the OCaml programming language. Finally, we described memory management practices in the cvc5 SMT solver. The next chapter will introduce the cvc5 OCaml bindings, a set of C++ and OCaml functions that connect the OCaml programming language with the cvc5 SMT solver.

Chapter 4

OCaml Bindings for cvc5

In this chapter, we describe our implementation of the cvc5 OCaml bindings. We begin in Section 4.1 by presenting the C++ API containing all the necessary stubs for the bindings. Subsequently, in Section 4.2, we present the cvc5 OCaml library and provide an overview of its structure. In Section 4.3, we introduce the topic of cross-language garbage collection, describing how it is relevant and can affect the behaviour of the proposed bindings.

4.1 C++ cvc5 Stub-API

The communication between programs written in different programming languages is more complex than between programs written in the same language, as it requires additional steps to coordinate the representation of data in both languages. In our case, we want our new cvc5 OCaml frontend to communicate with the C++ API of cvc5. To achieve this, we use the OCaml's Foreign Function Interface (FFI) to manage the interaction between the OCaml frontend and each function of the cvc5 API, ensuring that every time a cvc5 function is called from the OCaml frontend, the following steps are taken:

1. The OCaml arguments given to the function are decoded into C++ values;
2. The C++ function is executed with the given C++ values;
3. The returned value is encoded back as an OCaml value.

We refer to this set of functions, which bridge the gap between OCaml and cvc5, collectively as the *Stub-API*. Figure 4.1 illustrates the architecture of the Stub-API, which can be divided into two main components:

- a set of *stub functions* that wrap each API function into a new function to be used from the OCaml side;

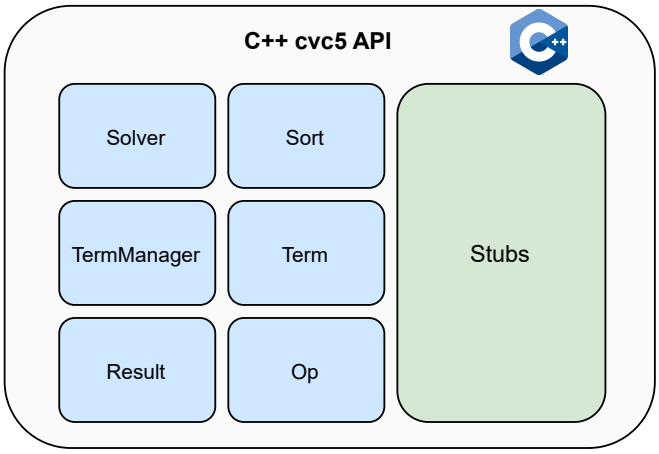

Figure 4.1: C++ cvc5 stub-API architecture.

- a set of *stub classes* that are meant to represent cvc5 values on the OCaml side (e.g., the solver object, formulas, expressions etc).

The cvc5's C++ API is extensive, including not only highly used SMT features, such as the theories of bitvectors and arithmetic, but also new experimental features currently under development, such as support for syntax-guided synthesis (SyGuS) problems [44] and the theory of separation logic [70, 71]. In this thesis, we have decided to focus only on the main theories supported by cvc5 because our goal is for SMT.ML to serve as the largest common denominator among various SMT solvers. Hence, we need only support theories that are implemented by multiple solvers. If a particular solver includes specialized theories that others do not, it is not necessary to add them to SMT.ML. Accordingly, we currently support the following cvc5 classes:

1. `TermManager` - This class serves as the primary entry point for the cvc5 API, providing methods to create the majority of cvc5 objects (e.g., `Term`, `Sort`, `Op`, etc.).

2. `Solver` - This class offers methods for direct interaction with the solver object, including asserting formulas (e.g., `assertFormula`) and checking their satisfiability (e.g., `checkSat`, `checkSatAssuming`).

3. `Term` - This class encapsulates formulas and expressions generated by the term manager, offering methods for effective interaction and manipulation (e.g., `getSort`, `isBooleanValue`).

4. `Sort` - This class represents the sort of a given term in a formula, offering methods for effective interaction and manipulation (e.g.,`substitute`, `isString`).

5. `Result` - This class encapsulates a three-valued solver result (sat/unsat/unknown) and provides methods for extracting these values from a `Result` object (e.g., `isSat`, `isUnsat`).

6. `Op` - This class represents cvc5 indexed operators used in bit-vector and floating-point arithmetic.

```
class Term : public cvc5::Term {
  public:
    TermManager* _tm;
    Term(cvc5::Term t, TermManager* tm) : cvc5::Term(t) {
      if (tm != NULL) { _tm = tm; tm->addRef(); }
      else { _tm = NULL; }
    }
    ~Term() {}
    void * operator new(size_t size,
          struct custom_operations *ops,
          value *custom){
      *custom = caml_alloc_custom(ops, size, 0, 1);
      return Data_custom_val(*custom);
    }
};
```

Listing 4.1: Class for `Term` objects in the stub-API.

```
#define Term_val(v) ((Term*)Data_custom_val(v))

static void term_delete(value v){
  Term* term = Term_val(v);
  if (term->_tm != NULL) {
    // decrement the reference count of the term manager
    term->_tm->rc--;
    try_delete_tm(term->_tm);
  }
  delete term;
}

static struct custom_operations term_operations =
{
   "https://cvc5.github.io/",
   &term_delete,
   custom_compare_default,
   custom_hash_default,
   custom_serialize_default,
   custom_deserialize_default,
   custom_compare_ext_default,
   custom_fixed_length_default
};
```

Listing 4.2: Custom operations struct for `Term` object blocks.

The classes that encode cvc5 API objects all follow the same structure. In Listing 4.1, we have the Stub-API class that encapsulates `cvc5::Term` objects. This class publicly inherits from the corresponding cvc5 API class (in this particular case, `cvc5::Term`). It has an overloaded operator `new` that makes use of OCaml's FFI to allocate a fresh new block of memory, with room for `size` bytes of data, using the `caml_alloc_custom` function. This operator also receives a structure that specifies the operations to handle the newly allocated block. In Listing 4.2, we exemplify how these operations structures are constructed; more concretely, we have the custom operations structure for `Term` objects. The structure contains multiple fields for operations on OCaml blocks. These operations can include serialisation and deserialisation of blocks, comparison between blocks, and finalisation of blocks. OCaml's FFI offers a set of default operations that one can use when no special behaviour is needed. In our case, we only specify a custom finaliser for blocks and set the other functions as the default ones. The finaliser holds a function that is called when a block becomes unreachable and is about to be reclaimed. These functions allow additional instructions to be performed right before a block is about to be freed. These finalisers allow additional instructions to be executed just before a block is freed. In our implementation, we replace the default finalisers with custom ones that implement a reference-counting system used to ensure a correct block collection order. This is detailed in Section 4.3.

The stubs in our API follow the same structure as those outlined in Section 3.1. Listing 4.3 contains the code for a stub that creates a new term representing the boolean value $true$. The process begins by decoding the argument received from OCaml, which is an instance of `TermManager`, using `TermManager_val`. Following this, we allocate a new custom block containing a `Term` object and return the pointer to this newly allocated block. Although it may seem that we have skipped the final step of encoding the return value back into an OCaml value, this step is handled by the operator `new` using the `Data_custom_val()` function. This function encodes the reference to the newly allocated block as a value that can be used on

```
1  /* Create a term with value true */
2  CAMLprim value ocaml_cvc5_stub_mk_true(value v){
3    CAMLparam1(v);
4    CAMLlocal1(custom);
5    TermManager* tm = TermManager_val(v);
6    CVC5_TRY_CATCH_BEGIN;
7    new(&term_operations, &custom) Term(tm->mkTrue(), tm);
8    CAMLreturn(custom);
9    CVC5_TRY_CATCH_END;
10 }
```

Listing 4.3: cvc5 stub that creates a term with value $true$.

the OCaml side.

4.2 OCaml cvc5 API

We will now take a deeper look at the OCaml API for the cvc5 SMT solver. This section is divided into two main parts. In Subsection 4.2.1, we look at the set of external declarations that directly refer to the stubs (which were analysed in Section 4.1). The second part refers to the cvc5 OCaml library, which aims to provide a more organised and user-friendly experience to developers who intend to use these bindings (Subsection 4.2.2).

4.2.1 External Declarations

Let us first look at the set of external declarations for the OCaml cvc5 API. As mentioned in Section 3.1, one of the steps that must be completed to achieve interoperability between OCaml and C/C++ is to define external declarations.

These sets of declarations are crucial when interfacing OCaml with C/C++ as they are the bridge between both environments and allow developers to invoke C/C++ functions from the OCaml runtime. The declarations follow a consistent structure, specifying the OCaml function type and the name of the corresponding C/C++ stub. Below we present the external declaration for the stub in Listing 4.3:

```
1  external mk_true : term_manager -> term = "ocaml_cvc5_stub_mk_true"
```

In this particular example, the function type is `term_manager` $\rightarrow$ `term`, where `term_manager` and `term` are pointers to custom data allocated blocks, meaning that this stub receives as an argument a pointer to a term manager object and returns a pointer to a term object. The stub's name is `ocaml_cvc5_stub_mk_true` (as defined in Listing 4.3). Having completed these steps, we are now able to create a `cvc5::Term` object with the boolean value $true$ using an OCaml program. However, as we will discuss in Subsection 4.2.2, using these declarations directly may not always be ideal, as there are situations where some processing of the arguments is necessary.

```
external mk_real : term_manager -> (int64[@unboxed]) -> (int64[@unboxed]) -> term
    = "ocaml_cvc5_stub_mk_real" "native_cvc5_stub_mk_real"
```

Listing 4.4: Example external declaration with unboxed arguments.

```
CAMLprim value native_cvc5_stub_mk_real(value v, int64_t num, int64_t den){
  CAMLparam1(v);
  CAMLlocal1(custom);
  TermManager* tm = TermManager_val(v);
  CVC5_TRY_CATCH_BEGIN;
  new(&term_operations, &custom) Term(tm->mkReal(num, den), tm);
  CAMLreturn(custom);
  CVC5_TRY_CATCH_END;
}

CAMLprim value ocaml_cvc5_stub_mk_real(value v, value num, value den){
  return native_cvc5_stub_mk_real(v, Int64_val(num), Int64_val(den));
}
```

Listing 4.5: Stubs for external declaration in Listing 4.4

Optimisation with Unboxed Values

As we have mentioned in Section 3.1, all OCaml objects are represented by the `value` type in C, hence it is necessary a first stage where every stub decodes the received OCaml object. This is done because, typically, primitives in OCaml are wrapped inside another data structure that holds extra metadata about the value, a technique known as boxing [72].

Boxing facilitates interaction with the garbage collector, but it also introduces an extra level of indirection to access the wrapped value. In order to avoid this expense, there is a way to instruct the OCaml native-code compiler to do this automatically, allowing us to pass unboxed arguments to the stub functions. Conversely, it is also possible to tell OCaml to expect an unboxed return value and automatically encode it. Although this might seem to be just a way of performing these operations automatically, delegating the responsibility of boxing values to the compiler means that the compiler can often decide to suppress it entirely.

In Listing 4.4, we have an example of an external declaration that uses this mechanism to instruct the OCaml compiler to expect two unboxed arguments of type `int64` when using the function `mk_real`. This does not mean that the compiler will always use the unboxed values; it simply gives it the responsibility to decide whether or not those arguments should be boxed. That is why we need to supply the name of two C++ stubs (in Listing's 4.4 concrete example, `ocaml_cvc5_stub_mk_real` and `native_cvc5_stub_mk_real`), more concretely, one that assumes the arguments have been boxed and another that does not. Listing 4.5 shows the code for these stubs. Note that `native_cvc5_stub_mk_real` expects unboxed values of type `int64_t`, while in `native_cvc5_stub_mk_real` we still have to extract the boxed value (using `Int64_val`).

4.2.2 cvc5.ml

Having a set of external declarations referencing C/C++ stubs is sufficient for invoking C/C++ functions from the OCaml runtime. However, there are scenarios where this approach is impractical. In this

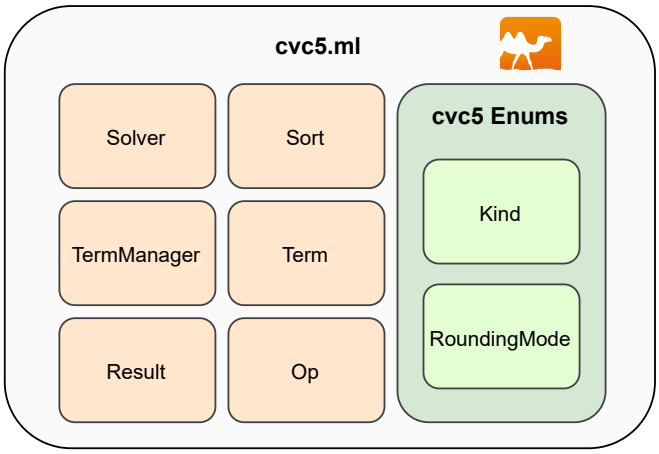

Figure 4.2: OCaml cvc5 library architecture.

section, we will discuss the OCaml cvc5 library, which serves as the main interface for OCaml developers to interact with the cvc5 SMT solver. We will now introduce the primary features this library offers to developers.

Modular Structure

The OCaml cvc5 library is organised using modules, as illustrated in Figure 4.2. Its architecture closely resembles the architecture of the Stub-API (shown in Figure 4.1), with the key difference being the inclusion of modules for the cvc5 C++ API enumerations: `Kind` and `RoundingMode` modules. In cvc5's C++ API, kinds (i.e., expression operators) and rounding modes (for expressions involving floating-point numbers) are defined as enumerations, meaning they are internally represented by integers. For example, creating the expression $1.0 + 2.0$ with a rounding mode of "round to nearest even" using the corresponding integer identifiers would look like this:

```
1 let n1 = Term.mk_fp tm 1. in
2 let n2 = Term.mk_fp tm 2. in
3 let expr = Term.mk_term tm 123 [| 0; n1; n2 |]
```

Note that in this example, when using the function `mk_term` to create the expression, we must use two integer constants (`123` and `0`), representing the operator for floating-point addition and the rounding mode for "round to nearest even," respectively. This approach is not ideal, as it requires developers to manually map these integers to the appropriate operators. To address this, we developed two additional modules that map the internal integer identifiers from the cvc5 C++ API to OCaml types. With these new modules, creating the same expression becomes a simpler and more intuitive process:

```
1 let n1 = Term.mk_fp tm 1. in
2 let n2 = Term.mk_fp tm 2. in
3 let expr = Term.mk_term tm Kind.Floatingpoint_add
4     [| RoundingMode.Rne; n1; n1 |]
```

```
(** Create a String constant from a string which may contain SMT-LIB
    compatible escape sequences like [\u1234] to encode unicode characters.

    Parameters: - The string this constant represents
                - (optional) A boolean that determines whether the escape sequences in the
                string should be converted to the corresponding unicode character *)
val mk_string : TermManager.tm -> ?useEscSequences:bool -> string -> term
```

Listing 4.6: Module interface entry for `mk_string` function in cvc5.ml.

This allows developers to easily identify the appropriate operators, making the code more readable by using descriptive operation names instead of opaque integer values.

Additionally, these two modules are automatically generated each time the cvc5.ml library is built. This is achieved by iterating through the corresponding enumerations and writing to a separate OCaml file the definitions of the types and functions that convert such types to and from the integers used internally by cvc5's C++ API. Since these enumerations are extensive, containing numerous fields, automating this generation ensures that any changes made to the enumerations in the C++ API are immediately reflected in cvc5.ml, promoting robustness and flexibility.

Documentation

As previously mentioned, cvc5.ml is organized using modules, enabling functions and types to be clearly separated and assigned specific roles within the library's overall structure. OCaml supports the definition of module interfaces through the creation of a file with the same name as the module implementation but with a .mli extension. This allows developers to specify which parts of the module implementation should be accessible to other modules. If no module interface is provided, the entire module implementation becomes public by default.

We follow this approach in cvc5.ml, where we have a well-defined public module interface. Since this interface contains all the public declarations, it is considered good practice to include documentation that outlines the specifications for the functions. For example, Listing 4.6 shows the module interface entry for the `mk_string` function from the `Term` module in cvc5.ml. This entry is annotated with a comment that briefly describes the function's behaviour and details the parameters it accepts. The comments for functions in cvc5.ml are based on the official documentation of the cvc5 C++ API, with appropriate modifications to account for differences between the two programming languages.

This documentation is particularly valuable for developers using cvc5.ml, as it allows them to quickly verify the specifications of a function without needing to refer to the C++ API documentation and map the corresponding functions. Additionally, it helps to clarify any ambiguities regarding the usage of specific parameters. For instance, in Listing 4.6, the function includes an optional boolean argument that might otherwise be confusing. The accompanying documentation resolves this issue by explaining the purpose of each parameter and, in some cases, specifying the possible values that a parameter can take.

OPAM Package

One of the key features of cvc5.ml is its availability as an OPAM package.[1] OPAM [73], the package manager for OCaml, streamlines the installation, management, and maintenance of OCaml libraries and applications. Its role in the OCaml ecosystem is crucial, as it manages project dependencies, ensuring the correct versions of libraries are installed and enables different projects to use distinct sets of dependencies without conflict.

A primary objective of this thesis was to make cvc5.ml accessible as an OPAM package, thereby broadening its availability to a wider range of OCaml developers. This significantly simplifies dependency management and removes the need for developers to build cvc5.ml from source.

A package must be self-contained to be accepted as an OPAM package. This requires that all dependencies are either managed by the user or explicitly listed and built along with the package. Meeting this requirement was essential for cvc5.ml's inclusion in the OPAM repository. The cvc5 SMT solver depends on several external libraries, including the CaDiCaL SAT solver [74] and LibPoly [75], a C library for polynomial computations. Typically, a user building cvc5 can use a script provided in the solver's source code that automates the fetching and building of these dependencies. However, because such external fetching is not permitted within an OPAM package, our build configuration for cvc5.ml cannot rely on this script. Instead, we developed a build script that uses Dune, OCaml's build system, to compile all required dependencies locally, ensuring that cvc5.ml remains fully self-contained.

4.3 Cross-Language Garbage Collection

When dealing with a multi-language paradigm, certain challenges that are not present in a single-language environment arise and must be considered. One of the primary concerns is the interoperability between all programming languages. This introduces challenges in things such as data representation, function calling conventions, and error handling across language boundaries. Different programming languages feature distinct characteristics, such as syntax, type systems, and memory management strategies. These differences require a deep understanding of each language's strengths and limitations when designing cross-language systems. Memory management is a particularly crucial aspect, as different programming languages handle it in various ways.

As discussed in Section 3.4, OCaml performs automatic garbage collection. This means that during execution, the OCaml garbage collector automatically reclaims memory blocks that it identifies as no longer being reachable. This feature simplifies memory management for the programmer, as they do not need to manually free memory, reducing the likelihood of memory leaks and other related issues, but it also raises some challenges when combining OCaml with another programming language in an interoperability scenario.

[1] https://opam.ocaml.org/packages/cvc5/

```
1  /* Create a constant with sort 'sort' and name 'n' */
2  CAMLprim value ocaml_cvc5_stub_mk_const_s(value v, value sort, value n){
3    CAMLparam3(v, sort, n);
4    CAMLlocal1(custom);
5    TermManager* tm = TermManager_val(v);
6    cvc5::Sort* s = Sort_val(sort);
7    CVC5_TRY_CATCH_BEGIN;
8    new(&term_operations, &custom)
9      Term(tm->mkConst(*s, String_val(n)), tm);
10   CAMLreturn(custom);
11   CVC5_TRY_CATCH_END;
12 }
```

Listing 4.7: cvc5 stub that creates constants with a given sort and name.

On the other hand, the cvc5 SMT solver was developed using the C++ programming language. In Section 3.5, we outlined how cvc5 manages memory internally. Since C++ does not feature automatic garbage collection, that responsibility falls on the developers. This manual approach offers fine-grained control over memory usage but also increases the risk of memory-related errors, such as leaks, dangling pointers, and double deletions. When integrating OCaml with cvc5, it is essential to bridge these two different memory management paradigms effectively. This involves ensuring that memory allocated in one language is correctly managed and released when no longer needed, regardless of which language's runtime is currently in control. Special attention must be paid to the lifetime of objects and ensuring that no premature deallocation or memory leaks occur.

The OCaml FFI establishes rules and provides directives for "living in harmony with the garbage collector" [9]. Our stubs for the cvc5 OCaml bindings adhere to these guidelines. For instance, let us consider Listing 4.7, which illustrates the stub code for creating a constant with a specified sort and name. One of the rules is that functions with parameters or local variables of type `value` must start with a call to one of the `CAMLparam` macros and return with one of the `CAMLreturn` macros. This is precisely the case in Listing 4.7; it starts with the `CAMLparam3` macro because this stub receives three arguments with type `value` and it returns with the `CAMLreturn` to return the custom block that was allocated in this function. Another rule states that local variables of type `value` declared inside a function must be declared with one of the `CAMLlocal` macros. Consequently, we employ the `CAMLlocal1` macro to declare a `value` variable, which points to the new custom block to be allocated and returned. There are more rules that indicate

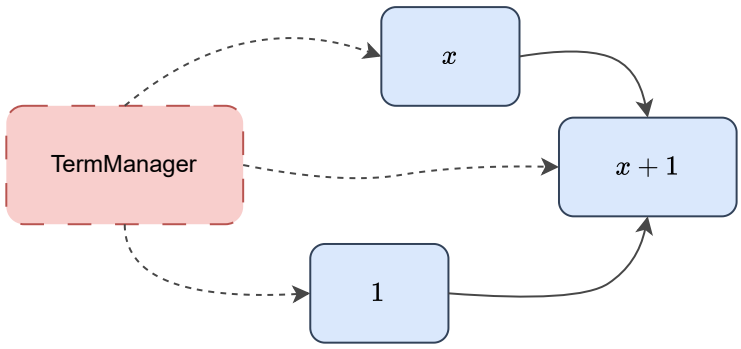

Figure 4.3: Incorrect garbage collection order example.

```
class TermManager : public cvc5::TermManager {
public:
  /* Used for API-level reference counting */
  std::atomic<unsigned long> rc;
  TermManager() : cvc5::TermManager() { rc = 1; }
  ~TermManager() {}
  void * operator new(size_t size,
        struct custom_operations *ops,
        value *custom){
    *custom = caml_alloc_custom(ops, size, 0, 1);
    return Data_custom_val(*custom);
  }
  void addRef() { rc.fetch_add(1, std::memory_order_release); }
};

static void try_delete_tm(TermManager* tm){
  if (tm->rc == 0) { delete tm; }
}

static void delete_tm(value v){
  TermManager* tm = TermManager_val(v);
  tm->rc--;
  try_delete_tm(tm);
}
```

Listing 4.8: Stub-level reference counting of `TermManager` instances.

how to register global variables, for example. All these rules and directives are used to inform the garbage collector of the memory blocks that are being used to ensure that they are not wrongfully reclaimed.

As we have outlined in Section 3.5, the cvc5 SMT solver uses an internal reference counting mechanism to identify which objects are still being used and which ones can be marked for deletion. This means that whenever an object is deleted, the count of all other objects it referenced has to be decremented to ensure a correct count across all objects that are still reachable. Let us consider the example of Figure 4.3, where we have a `TermManager` object that manages three terms: x, 1, and $x + 1$. In this scenario, we allocate four custom OCaml blocks, one of which holds the term manager and three for the remaining terms. Since OCaml blocks can be deallocated in any order once they are no longer reachable, if the term manager block is collected before the other blocks, this can lead to undefined behaviour. Specifically, when the remaining terms are collected, the counts of all objects referencing them should be decremented. However, if the term manager has already been reclaimed, this can cause runtime errors, such as segmentation faults, because we would be accessing memory that has already been freed.

To address this issue, we implement a reference counting mechanism at the stub level. Specifically, we track the number of blocks each term manager references. Listing 4.8 shows the class representing cvc5 `TermManager` instances at the stub level. Each object of this class includes an attribute that maintains the count of all blocks referenced by a given term manager. To prevent premature collection of term managers, we check if there are still reachable blocks referenced by a term manager. This check is performed using OCaml block finalisers. Block finalisers are functions that are invoked when a block becomes unreachable and is about to be collected. They are useful for running additional code just before a block is reclaimed, especially when the timing of reclamation is uncertain (e.g., closing a file descriptor). In our implementation, the block finaliser verifies whether it is safe to collect a particular term manager. Collection is allowed only if the term manager's count is zero; otherwise, it indicates that there are still reachable blocks referenced by this term manager.

Summary

In this chapter, we presented our implementation of the OCaml bindings for the cvc5 SMT solver. First, we explained how we developed the C++ stubs that use OCaml's FFI and are essential in bridging the OCaml programming language and the cvc5 solver. Subsequently, we outlined the structure of the OCaml API that references the stubs and the cvc5.ml library that users can install if they intend to incorporate cvc5 in their OCaml programs. Lastly, we explained how we address the challenges that arise from garbage collection in a multi-language paradigm. The next chapter introduces a new version of SMT.ML, featuring multiple backends, including a new backend for the cvc5 SMT solver that uses the bindings we described in this chapter.

Chapter 5

Multi-Backend SMT.ML

In this chapter, we explain how we extended SMT.ML to support multiple backends. Initially, SMT.ML only supported a single backend for the Z3 SMT solver [17]. Currently, in addition to Z3, it also supports the cvc5 [22], Bitwuzla [50], and Colibri2 [76] SMT solvers. At the core of this extension is our new encoding module that is parametric on a generic solver interface. To add support for a new backend, one must write a wrapper around that backend that implements the API required by the parametric encoding module. We describe this parametric encoding in Section 5.1 and our wrapper for cvc5.ml that supports the parametric API in Section 5.2.

5.1 Parametric Encoding

Perfect SMT solvers do not exist. Each SMT solver has strengths and weaknesses, excelling in certain areas while underperforming in others. Consequently, the best SMT solver for a given problem depends on the specific characteristics of the problem being analysed. To address this, SMT.ML provides a unified and straightforward interface for interacting with multiple SMT solvers through a single frontend, allowing developers to leverage each solver's strengths and select the most suitable one for their problem.

At the core of SMT.ML lies the Encoding module, a module responsible for translating SMT.ML's expressions and commands into the expressions and commands of each solver backend. Instead of developing an encoding module for each solver backend, SMT.ML's encoding module is parametric on a *Core Solver API*, $\mathcal{S}$, that solvers are expected to implement, streamlining the addition of new solver backends and avoiding code duplication. Having established this API, obtaining the encoding of SMT.ML's logic into the logic of a specific solver becomes straightforward: it suffices to plug the target solver's implementation of the API $\mathcal{S}$ into SMT.ML.

Naturally, we do not expect solver developers to implement our Core Solver API in their native tools. Instead, we develop wrappers around those solvers that do implement the expected API and plug the obtained wrapped solvers into SMT.ML. Figure 5.1 illustrates this process, showing how different SMT

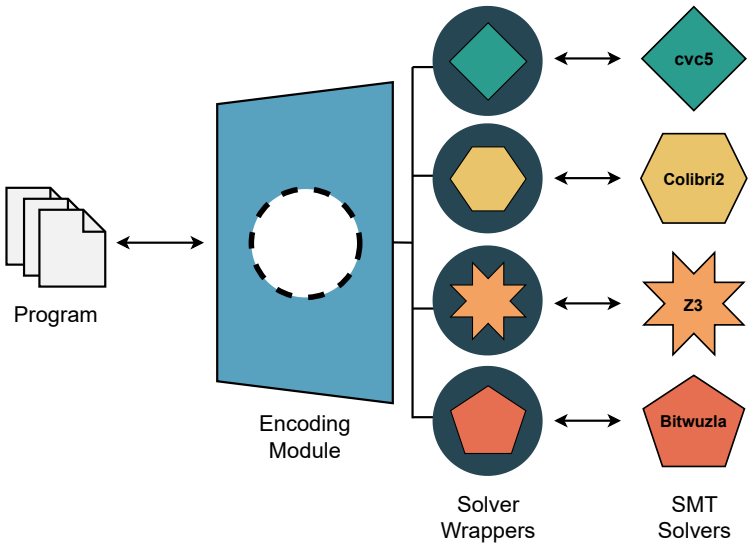

Figure 5.1: Overview of SMT.ML's parametric Encoding module.

solvers, each with its own specific API, can be integrated into SMT.ML by first being wrapped within modules that implement our Core Solver API.

5.1.1 Solver-Agnostic API

The Core Solver API $\mathcal{S}$ lists all functions on solver values, expressions, and commands on which the translation of SMT.ML expressions and commands depends. By establishing this common API, SMT.ML can maintain consistency and interoperability across different solver backends. The Core Solver API lists functions that cover tasks such as initialising a solver instance, asserting formulas, encoding operations in the target solver's logic (e.g., binary operations), checking the satisfiability of formulas, and retrieving results, forming a comprehensive set of functions that solver wrappers are expected to implement.

We can divide the required functions into the following four main categories:

- *Values:* The functions in this category are responsible for mapping SMT.ML primitive values, $v \in \mathcal{V}_{smt}$, into values from the corresponding target solver.
- *Operators:* These functions map SMT.ML operators to the corresponding solver operators. For example, solver wrappers are expected to implement the addition operator, which, when given two solver expressions, returns a target solver expression representing their sum.
- *Commands:* The functions in this category map SMT.ML commands, $c \in \mathcal{C}_{smt}$, into commands that manipulate and interact with the target solver.
- *Lifting:* This category contains functions that map solver values back to SMT.ML values, $v \in \mathcal{V}_{smt}$. Such functions are essential when performing model extraction tasks, for instance, as they allow models to be constructed using SMT.ML's native values.

```
1  module Int : sig
2      val v : int -> term
3
4      val neg : term -> term
5
6      val add : term -> term -> term
7
8      val sub : term -> term -> term
9  end
```

Listing 5.1: Core Solver API functions to handle integer operations

Listing 5.1 illustrates a fragment of the OCaml implementation of the Core Solver API $\mathcal{S}$ signature. Specifically, it includes the signatures of the function used to create solver-specific integer values (function `v` on line 2), as well as the functions for constructing the negation of an integer (`neg` on line 4) and the addition and subtraction of two integer expressions (`add` on line 6 and `sub` on line 8, respectively). It is important to note that all functions use an uninterpreted type `term`, which corresponds to the type of the wrapped solver expression. By establishing a clear API that separates the implementation of SMT.ml from that of the backend solvers, we make our implementation easier to extend and more resilient to changes in the backend solvers.

5.1.2 Parametric Translation

Using the Core Solver API $\mathcal{S}$, we implement a generic translation for converting SMT.ML constructs into the corresponding solver-specific constructs. We formalise the main translation for expressions as a function:

$$T_{\mathcal{S}} : \mathcal{E}_{smt} \rightarrow \mathcal{S}.Expr$$

that receives an SMT.ML expression $e \in \mathcal{E}_{smt}$ and generates a target solver expression $te \in \mathcal{S}.Expr$. Figure 5.2 presents the parametric translation rules for SMT.ML expressions.

Values v, and symbols s, are translated into their counterparts in the target solver $\mathcal{S}$, using the functions $\mathcal{S}.val(v)$ and $\mathcal{S}.symbol(s)$, respectively. Unary operators *uop e* are translated into the application of the solver's unary operator to the translation of the operation argument $T_{\mathcal{S}}(e) = te$, which is computed using the Core Solver API function $\mathcal{S}.unop$, which receives as inputs a unary operator uop and a solver expression te and generates the solver expression that denotes the application of uop to te.

The translation follows a similar process for the remaining operators: the operation's arguments are first translated into native solver expressions, after which the operator's translation is applied to them, resulting in an expression that the solver being used can reason about.

5.1.3 Backend-independent Optimisations

SMT.ML's usefulness extends beyond its capability to interact with multiple SMT solvers through a unified syntax. A significant aspect of its design is the inclusion of backend-independent optimisations that enhance performance when checking the satisfiability of logical formulas. These optimisations

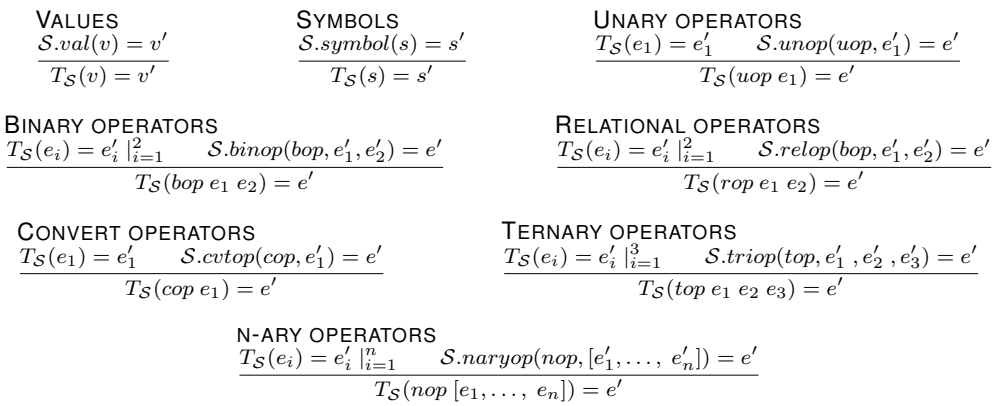

Figure 5.2: Parametric translation rules for SMT.ML.

are universally applied, regardless of the chosen solver backend, ensuring consistent performance improvements across different solvers. SMT.ML features two key optimisations: expression simplifications and caching.

Expression Simplifications

In applications that interact with SMT solvers, the solver's performance often becomes the primary bottleneck, with the problem's size and complexity significantly affecting the solver's efficiency [77]. To counter this, SMT.ML features a set of simplifications that are applied to expressions in an attempt to reduce their overall complexity, while maintaining their original meaning.

At the core of our expression-simplification algorithm is a set of simplification rules $r \in \mathcal{R}$ of the form: $f\ ?\ e_1 \rightarrow e_2$, meaning that if the formula f holds, then the expression e_1 can be rewritten as e_2. For instance, the rule:

$$r_1 \equiv h - l = |x|\ ?\ \texttt{extract}(x, h, l) \rightarrow x$$

states that the expression $\texttt{extract}(x, h, l)$ can be rewritten as x if $h - l$ coincides with the size of x. In order to apply a simplification rule $f\ ?\ e_1 \rightarrow e_2$ to a given expression e under an execution context where the formula f' holds, we proceeds as follows:

1. find a substitution θ such that $\theta(e_1) = e$;
2. check if $f' \implies \theta(f)$;
3. replace e with $\theta(e_2)$.

For instance, applying r_1 to $\texttt{extract}(\texttt{concat}(y, z), |y| + 2, 0)$ yields the expression $\texttt{concat}(y, z)$ with substitution $\theta = [x \mapsto \texttt{concat}(y, z), l \mapsto 0, h \mapsto |y| + 2]$ under the formula context $f' \equiv |z| = 2$.

Currently, SMT.ML performs constant folding for every theory it supports (215 rules), meaning that concrete values are simplified as much as possible, and comes with 62 additional simplification rules,

spanning the theories of bit-vectors (28 rules), floating point arithmetic (3 rules), boolean (2 rules), strings (2 rules), and 27 generic rules which can be applied to multiple theories. The rules must be carefully designed to ensure that the simplifications are sound (i.e., the simplified expression denotes the same values as the original one) and to make sure that the simplification process is not infinite, meaning that rule application does not generate simplification loops, such as $e \xrightarrow{r_1} e' \xrightarrow{r_2} e$, where e is the original expression, e' is the simplified expression, and r_1 and r_2 are two simplification rules.

With a set of sound and terminating simplification rules established, designing our expression-simplification procedure becomes straightforward. Figure 5.3 outlines this procedure: the main loop iterates through the set of simplification rules, applying those whose constraints are satisfied. The loop terminates once no further rules can be applied to the expression at hand, indicating that a fixed point has been reached.

```
1: procedure SIMPLIFY(f)
2:     f' ← f
3:     for all r ∈ Rules do
4:         f' ← apply r to f'
5:     if f' ≠ f then
6:         return SIMPLIFY(f')
7:     else
8:         return f'
```

Figure 5.3: Expression simplification algorithm.

5.1.4 Caching

Caching and Normalisation

Caching of intermediate satisfiability results is a standard technique used in SMT solvers and solver clients to improve performance [78, 79]. However, it is not common for identical formulas to be queried multiple times, even in applications that make an intensive use of SMT solvers. To address this, formula caching systems [61] typically implement normalisation strategies with the goal of maximising cache hits. SMT.ML comes with its own formula caching system equipped with a normalisation procedure that performs:

- *Standardisation of associative operators:* a standard order is imposed on expressions that include such operators. For instance, considering the logical or operator (commonly denoted by $\vee$), we have that $(x \vee y) \vee z = x \vee (y \vee z)$. In SMT.ML, expressions that include chained associative operators are always rewritten to ensure that the leftmost operations are performed first.

- *Variable renaming:* variables are renamed to ensure structurally identical formulas with different variable names are considered equal.

```
1 let mk_or hte1 hte2 =
2   let x = Binary (Or, hte1, hte2) in
3   try Hashtbl.find table x
4   with Not_found -> Hashtbl.add table x x; x
```

Listing 5.2: Hash-consing constructor for boolean disjunction.

```
1 let hash (hte : t) = hte.tag
2
3 let equal (hte1 : t) (hte2 : t) = hte1.tag == hte2.tag
4
5 let view (hte : t) : expr = hte.node [@@inline]
```

Listing 5.3: Functions for hash-consed expressions

Caching Expressions via Hash-consing

In addition to minimizing the number of queries, another way to enhance the performance of solver clients is to reduce the number of formulas and expressions created at runtime. In fact, solver clients often generate a large number of formulas, frequently with repeated elements. As the number of queries grows, memory consumption increases, significantly impacting solver client's performance; a prime example of this is symbolic executors [80]. The standard technique to reduce the memory impact of solver systems is the use of *hash-consing* [81], a technique that ensures that no two physical copies of the same expression are ever created by storing expressions in a hash table. SMT.ML includes a hash-consing module that prevents the duplication of identical expressions. To this end, whenever an SMT.ML expression constructor is called, it checks whether the expression already exists, and, if it does, returns the previously stored expression.

Listing 5.2 illustrates this process for the `Or` constructor. We define the `mk_or` hash-consing constructor, which builds a boolean disjunction between two hash-consed expressions. In line 2, we construct the binary expression, and in line 3, we attempt to retrieve a previously constructed expression from the hash-consing table. If the expression is not found (line 4), we add it to the table and return the value constructed in line 2. SMT.ML only allows creating expressions through these smart constructors, ensuring that every expression is correctly hash-consed.

Additionally, because hash-consed expressions are unique, they can be identified by a single integer. This significantly enhances the performance of expression comparison. Listing 5.3 presents several functions for operating on hash-consed expressions, including the `equal` function, which is used for expression comparison. As demonstrated, this comparison is reduced to a simple integer comparison between the identifiers of the expressions, regardless of the expression complexity.

5.2 Solver Wrapper for cvc5.ml

In order to extend SMT.ML's architecture with a new SMT solver, a wrapper for that solver must be developed. This wrapper contains the implementation of the functions listed by SMT.ML's Core Solver API $\mathcal{S}$ using the target solver's native OCaml bindings. The cvc5 solver is no exception. To integrate the cvc5 within SMT.ML, we developed a wrapper that uses cvc5.ml, our OCaml bindings for the cvc5 solver (Chapter 4).

Implementing the solver wrapper for cvc5 involves developing an OCaml module that defines the

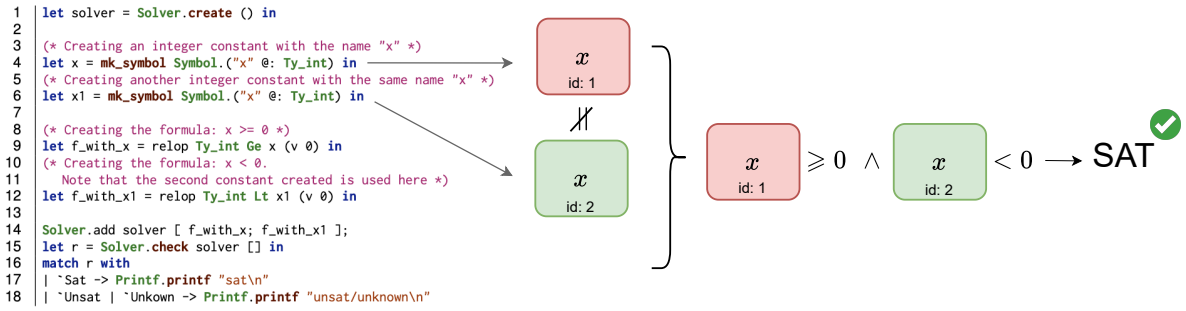

Figure 5.4: Example of an SMT.ML program with no variable bookkeeping.

functions of the Core Solver API $\mathcal{S}$ using the functions from the OCaml bindings for cvc5. Most of the time, it suffices to map the API function to the corresponding function in the bindings. However, even in these cases, the process is not always straightforward because the binding function might require additional arguments, arguments with different types, or arguments in a different order. Below, we show the implementation of the API function `add` for bitvector expressions:

```
let add t1 t2 = Term.mk_term tm Kind.Bitvector_add [| t1; t2 |]
```

We note that this function is implemented using the `mk_term` function from the cvc5 OCaml bindings, which is a more general function and hence requires the additional argument `Kind.Bitvector_add` to specify the addition operation. Furthermore, the two sub-expressions of the addition expression, `e1` and `e2`, are passed in a list, whereas the API function receives them as two separate arguments.

5.2.1 No Support for Bookkeeping of Variables

When integrating new solver backends into SMT.ML, it is crucial to ensure they adhere to the specifications outlined by SMT.ML's Core Solver API. This task is often challenging due to the nuanced design choices that different SMT solvers exhibit.

The cvc5 SMT solver API, for example, does not internally index declared variables by their names. This allows for the definition of two variables with the same name and sort that remain independent and represent different values within the same formula. In the cvc5 solver, bookkeeping of variables by their name is only done when the SMT-LIB frontend is used, ensuring compliance with SMT-LIB standards. For instance, Figure 5.4 illustrates an SMT.ML program using the cvc5 backend to create two integer variables named "x" and subsequently check the satisfiability of the formula: $x \geqslant 0 \wedge x < 0$. Since the cvc5 solver delegates the responsibility of tracking variables to the user, running this program without any modifications to the SMT.ML wrapper for cvc5 would incorrectly indicate that the formula is satisfiable. In contrast, in Z3, the bookkeeping of variables is performed internally, meaning that executing the program of Figure 5.4 would always correctly yield an unsatisfiable result. Note that such differences can affect the performance of SMT.ML when employing different backends.

```
1 let make_symbol (symbol_table : sym_tbl) (s : Symbol.t) : M.term =
2     match Hashtbl.find_opt symbol_table s with
3     | Some sym -> sym
4     | None ->
5       let sym = S.const s.name (get_type s.ty) in
6       Hashtbl.replace symbol_table s sym;
7       sym
```

Listing 5.4: SMT.ML mappings function to create constants.

To ensure appropriate bookkeeping of variable names in cvc5, we have to maintain an *internal map* associating variable names with the corresponding variables created by cvc5. This can be implemented either in OCaml, at the Solver Wrapper level, or in C++, at the Stub-API level. In this thesis, we have implemented both approaches. Below, we describe the two approaches in detail and discuss their relative merits.

Bookkeeping Variables at the Solver Wrapper Level In order to bookkeep variables at the Solver Wrapper Level, we have included in cvc5's solver wrapper a hash table that maps variable names to the corresponding cvc5 variables. Listing 5.4 presents the code for the function responsible for creating variables. This function begins by checking whether an entry for the desired variable already exists in the hash table (line 2). If an entry is found, the corresponding variable is returned (line 3). Otherwise, the solver's wrapper is used to create the variable (line 5), and the table is subsequently updated (line 6) to prevent the creation of duplicate variables in the future. This mechanism ensures that variables with the same name maintain consistent meanings across formulas, thereby avoiding issues such as the one depicted in Figure 5.4, which could otherwise lead to elusive bugs.

Bookkeeping Variables at the Stub-API Level In order to bookkeep variables at the stub-API level, we have included in cvc5's stub API a hash table that maps variable names to the corresponding cvc5 variables. Since this hash table will be queried a significant number of times in general, instead of implementing our own hash table in C++, we used Google's `dense_hash_map` from the Google SparseHash library [82], which offers several efficient hash table implementations. Essentially, this code modification mirrors the changes made at the OCaml level. Listing 5.5 presents a fragment of the implementation of the function `ocaml_cvc5_mk_const_s`, which is the stub API function responsible for creating new variables. As in the OCaml level, before creating a variable, we check if it already exists in the hash table (line 3 and 4). If it exists, we return the corresponding variable (line 5); if not, we create a new variable (line 7), insert it into the hash table (line 8), and return it.

Approach Comparison While the two approaches may seem equivalent at first sight, they are not because C++ is substantially more performant than OCaml. Even if the performance difference is marginal per hash table interaction, because the hash table is going to be accessed extremely often, these marginal gains compound and become significant. In the evaluation chapter, we compare the two

```
1  CAMLprim value ocaml_cvc5_stub_mk_const_s(value v, value sort, value n) {
2    ...
3    auto existingTerm = tm->getTerm(termName);
4    if (existingTerm != nullptr) {
5      CAMLreturn(existingTerm);
6    } else {
7      cvc5::Term* newTerm = new cvc5::Term(tm->mkConst(*s, termName));
8      tm->addTerm(termName, newTerm);
9      new(&term_operations, &custom) Term(*newTerm, tm);
10   }
11 }
```

Listing 5.5: Stub-API variable bookkeeping implementation.

approaches quantitatively.

Summary

In this chapter, we presented a multi-backend version of SMT.ML. First, we explained how we modified SMT.ML's architecture to support multiple solver backends through the use of a central parametric module. Subsequently, we outlined SMT.ML's Core Solver API $\mathcal{S}$ that every solver wrapper is expected to implement in order to be integrated within SMT.ML, and the set of translation rules used by the parametric module to translate SMT.ML expressions into target solver expressions. Afterwards, we described the two main optimisations SMT.ML implements. Lastly, we explained how the wrapper for the cvc5 SMT solver was developed in order to add its backend to SMT.ML, outlining two possible strategies to deal with the bookkeeping of cvc5 variables. The next chapter presents our evaluation of the correctness and performance of the OCaml bindings for cvc5, as well as the impact of the new cvc5 backend on SMT.ML.

Chapter 6

Evaluation

In this chapter, we assess the correctness and efficiency of the cvc5 OCaml bindings and the impact that a new backend for the cvc5 SMT solver has on SMT.ML. We begin this chapter by describing our experimental setup. Then, we answer the following evaluation questions:

- **EQ1:** Are the cvc5 OCaml bindings correctly implemented?
- **EQ2:** Are the cvc5 OCaml bindings efficiently implemented?

6.1 Experimental Setup

This section presents our experimental setup. We begin by describing the environment we used during the evaluation process (Section 6.1.1). Subsequently, we describe the datasets used to assess the correctness of the cvc5 OCaml bindings and to compare the performance of our new SMT.ML cvc5 backend with the existing Z3 backend (Section 6.1.2).

6.1.1 Experimental Environment

Our experimental environment consisted of a Macbook Air with the 14.5 Sonoma macOS, 16GB of RAM, and an Apple M2 ARM64 CPU. For compiling SMT.ML, we used the OCaml 5.2.0 compiler. For the SMT solvers, we employed cvc5 version 1.2.0 and Z3 version 4.13.0.

6.1.2 Datasets

To assess the correctness of the developed cvc5 OCaml bindings and to measure the performance of the new SMT.ML backend for the cvc5 SMT solver, we utilized a subset of queries generated by Owi [5], a symbolic execution engine, during the execution of the Test-Comp 2023 [24] benchmark suite. This dataset comprises approximately 2.3 million queries produced throughout the symbolic execution process.

Table 6.1: Summary of the queries from the Test-Comp 2023 dataset.

Theory	*Expected result*			**Total**
	# SAT	**# UNSAT**	**# Unknown**	
QF_BV	1 940 874	311 532	11	2 252 417
QF_FP	122	23	6	153
QF_BVFP	35	13	2	50
Other	17	17	-	34

Given that SMT.ML 's primary application lies in symbolic execution, this dataset offers a robust basis for evaluating the performance of the new cvc5 backend. Additionally, the large volume of queries contributes to a high level of confidence in assessing the correctness of the cvc5 OCaml bindings.

Table 6.1 provides an overview of the distribution of queries generated during the symbolic execution of the Test-Comp 2023 benchmark suite. Most of these queries involve the theory of bitvectors (QF_BV), a prevalent theory in symbolic execution tasks. A subset of the queries also involves the theory of floating-point arithmetic (QF_FP), while a smaller portion tests a combination of both bitvectors and floating-point arithmetic (QF_BVFP). Additionally, Table 6.1 categorizes the queries by their expected results (SAT/UNSAT/Unknown). While this provides a high-level analysis of the query content, a deeper analysis of the queries' structure offers further characterisation of this benchmark. Specifically, we can classify each query based on additional parameters that influence the complexity of the formulas and, consequently, the solver's performance.

To further characterize each query, we employed two additional metrics beyond the theory tested and the expected result:

- *Number of variables declared:* This metric refers to the number of variables declared within each query.
- *Average number of operations per assertion:* This metric refers to the average number of operations within each assertion, which we will also refer to as "nodes" hereafter.

These metrics have a direct impact on the complexity of the formulas queried to the solver, thus offering valuable insights about the characteristics of the dataset, which can explain the solvers' performance. To this end, we plotted the two metrics mentioned earlier: the number of declared variables and the average number of nodes per assertion. Figure 6.1 provides a box plot representation of these metrics, supplemented by individual data points and the mean value for each metric.

As illustrated in Figure 6.1, the number of declared variables usually ranges between ten and a hundred variables, although some queries include significantly more. Notably, the average number of nodes per assertion displays a much broader distribution, reflecting the diversity in the complexity of the formulas. This metric ranges from single digits to several thousands, indicating that certain queries impose a significantly higher computational burden on the solver. These distributions emphasize the varying complexity of the queries in the dataset. Furthermore, the presence of outliers in all three metrics

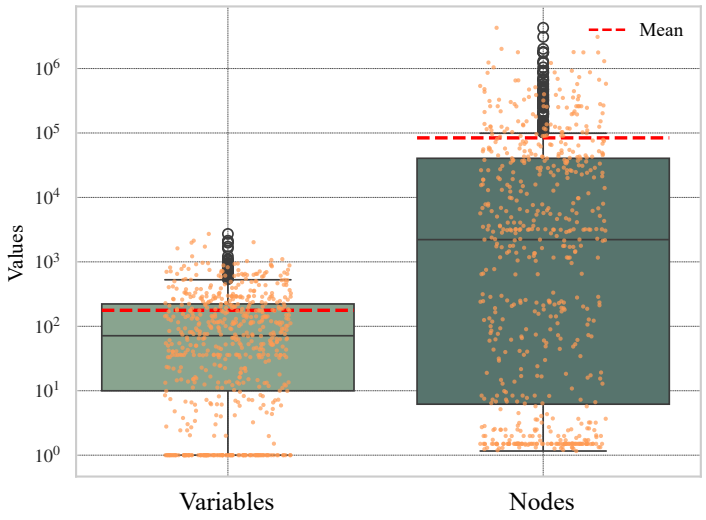

Figure 6.1: Box plot representations for the additional metrics regarding the subset of queries from Test-Comp 2023.

also suggests that while most queries are relatively simple, there are specific instances that are far more complex and could significantly impact the performance of the SMT solver.

6.2 EQ1: Are the cvc5 OCaml Bindings Correctly Implemented?

In this section, we assess the correctness of the cvc5 OCaml bindings. First, we introduce our evaluation methodology (Section 6.2.1). Then, we detail the obtained results (Section 6.2.2).

6.2.1 Evaluation Methodology

To assess the correctness of the developed bindings for the cvc5 SMT solver, we compared the results obtained when executing the query dataset from Test-Comp 2023 [24] in the new SMT.ML cvc5 backend with the expected result for each query. Each query is annotated with its expected result (SAT/UNSAT/Unknown), allowing us to compare the obtained result with the expected one. Furthermore, we also ran the same set of queries directly on the cvc5 SMT solver and the Z3 SMT solver. This further increases the confidence level in the correctness of the results.

6.2.2 Results

After running the Test-Comp 2023 benchmark using the SMT.ML backend for the cvc5 solver, we verified that all obtained results matched the expected outcomes annotated for each query, yielding a correctness rate of 100%. Additionally, we compared the results from the SMT.ML cvc5 backend with those obtained by executing the same benchmark directly on the cvc5 and Z3 SMT solvers. The results were consistent across all tests, further confirming the correctness. The Test-Comp 2023 dataset, consisting of approxi-

mately 2.3 million queries, provides a sufficiently large sample size to ensure a high level of confidence in these findings.

6.3 EQ2: Are the cvc5 OCaml Bindings Efficiently Implemented?

In this section, we assess the efficiency of cvc5's OCaml bindings by comparing SMT.ML's new backend for cvc5 against the backend for the Z3 solver. We first introduce our evaluation methodology (Section 6.3.1). Then, we compare the results obtained from both backends (Section 6.3.2) on our benchmark of SMT formulas. This comparison includes both bookkeeping variable strategies introduced in Section 5.2.

6.3.1 Evaluation Methodology

To evaluate the efficiency of cvc5's OCaml bindings, we began by benchmarking both the cvc5 and the Z3 SMT solver directly with the subset of queries generated by symbolically executing the Test-Comp 2023 dataset. Subsequently, we executed the SMT.ML on the same benchmark using both the cvc5 and Z3 backends. The comparative analysis of the number of problems solved and the total execution time, in conjunction with the independent results from each solver, provides insights into the relative performance of the two backends.

To further analyze the results, we established nine categories of SMT tests. The process involved considering two metrics—the number of declared variables and the average number of nodes per assertion. We assigned three qualitative levels to each metric: *Low*, *Medium*, and *High*. Given that we have two metrics and each metric can have three different levels, this results in nine distinct classes of formulas. For instance, at one extreme, we have the class corresponding to the easiest formulas, characterized by a low number of declared variables and a low average of nodes per assertion. Conversely, at the other extreme, we have the class representing the most challenging formulas, characterized by a high number of declared variables and a high average of nodes per assertion.

The thresholds for the qualitative levels were determined by analysing the value distribution of each metric on our dataset of formulas. More concretely, we determined for each metric its the lower and upper terciles (corresponding to the 33^{rd} and 66^{th} percentiles, respectively). Values falling at or below the lower tercile were classified as *Low*, those between the lower and upper terciles as *Medium*, and values above the upper tercile as *High*.

6.3.2 Results

Figure 6.2 contains a CDF (Cumulative Distribution Function) plot that shows the number of problems solved over time by both the cvc5 and Z3 SMT solvers when used directly as independent tools. Analysing the overall results, we conclude that the cvc5 SMT solver achieves a performance improvement of around 30% over the Z3 solver.

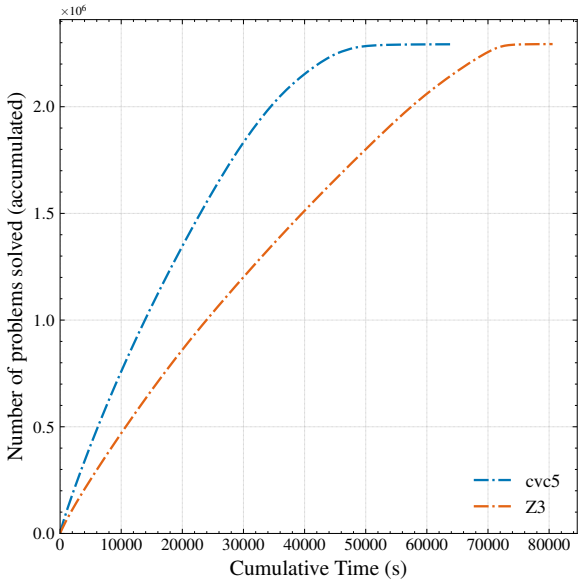

Figure 6.2: Results for the cvc5 and Z3 SMT solvers.

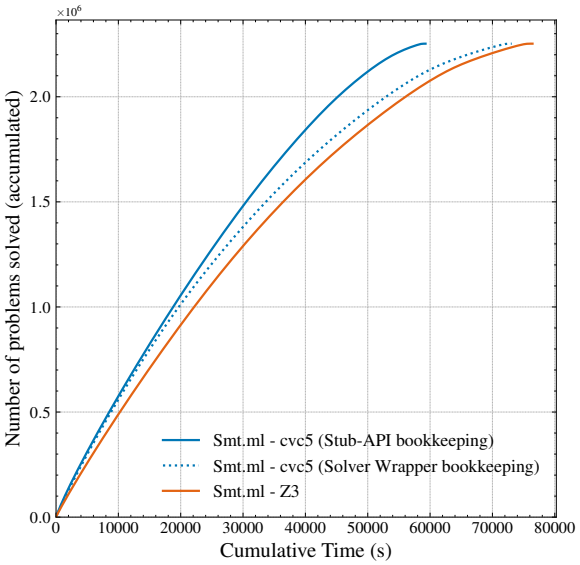

Figure 6.3: Results for Z3 backend and both versions of the cvc5 backend

Figure 6.3 contains a CDF plot that illustrates the number of problems solved over time by SMT.ML using the Z3 backend, the cvc5 backend with Solver Wrapper variable bookkeeping, and the cvc5 backend with the Stub-API variable bookkeeping. Analysing the overall results, we conclude that the cvc5 backend with Solver Wrapper bookkeeping achieves a performance improvement of approximately 5% over the Z3 backend, while the cvc5 backend with Stub-API bookkeeping achieves a performance improvement of around 29%.

To better understand the scenarios in which the cvc5 backend outperforms the Z3 backend, we generated heatmap plots depicting the average speedups between both backends across the nine established qualitative categories. Figure 6.4 contains two heatmap plots: the left heatmap illustrates the average speedups achieved by the cvc5 backend with Solver Wrapper variable bookkeeping, while the right depicts the average speedups with Stub-API variable bookkeeping. For example, in the left heatmap, the bottom-left cell indicates that for formulas with a *low number of declared variables* and a *low average number of nodes per assertion*, the cvc5 backend using the Solver Wrapper strategy achieves a 17% performance improvement over the Z3 backend.

Analysis From these results, we conclude the following:

- When using variable bookkeeping at the Solver Wrapper level, performance differences between the Z3 and cvc5 backends are generally marginal across most categories. Notably, in the category with the easiest formulas (i.e., both a low number of variables and average number of nodes per assertion), the cvc5 backend achieves a speedup of 17%. However, for the most complex formulas (i.e., those with a high number of variables and average number of nodes per assertion), the cvc5 backend significantly underperforms compared to the Z3 backend, achieving a negative speedup

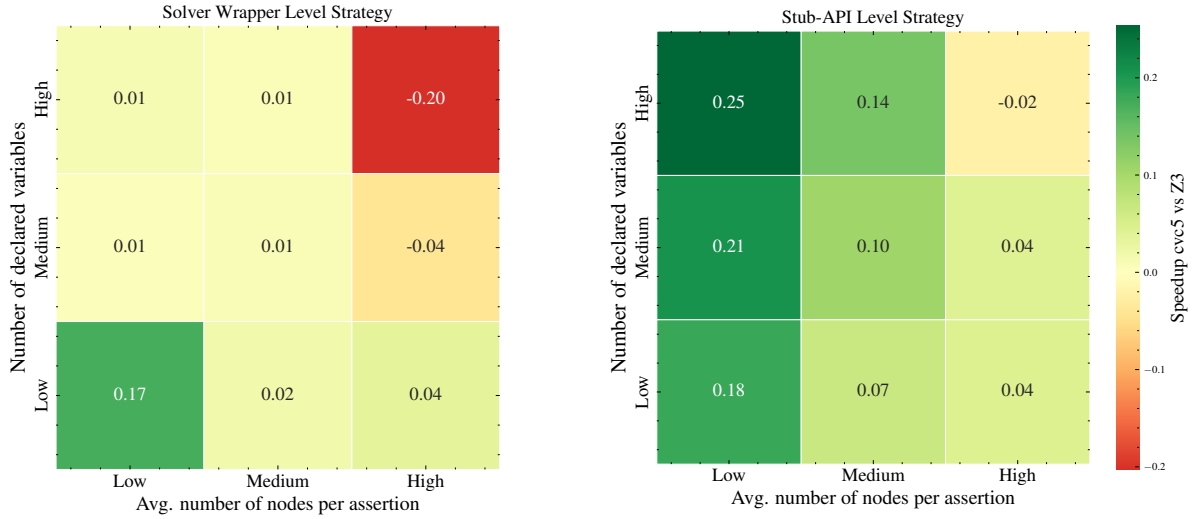

Figure 6.4: Heatmap plots with the speedup results for each variable bookkeeping strategy.

of 20%.

- When using variable bookkeeping at the Stub-API level, the cvc5 backend delivers significant performance improvements across all formula categories, except for the category of the most challenging formulas.

These results are expected as the impact that the hash table has on the cvc5 backend is more pronounced in problems where a high number of variables are declared and in assertions containing a high number of operations, as this directly translates into more hash table accesses that negatively impact performance. Conversely, for formulas with fewer declared variables and operations, the performance of the cvc5 backend is expected to more closely resemble that of the cvc5 SMT solver when used directly.

In conclusion, we observe that the performance difference between SMT.ML with the cvc5 backend using Stub-API bookkeeping and SMT.ML with the Z3 backend is consistent with the performance difference between cvc5 and Z3 when applied directly to the benchmark. In the former case, cvc5 achieves a 29% speedup, while in the latter, it achieves a 30% speedup, essentially the same. These results underscore the efficiency of our OCaml bindings for cvc5. If they had not been implemented as efficiently as those of Z3, the performance gap between the two tools would have narrowed. This did not happened, confirming the efficiency of our binding.

Discussion on Impact of Variable Bookkeeping The significant impact of the chosen variable bookkeeping strategy on backend performance is surprising. The large discrepancy between the performance results raises questions about their accuracy. To verify the negative impact of using bookkeeping at the Solver Wrapper level, we applied the same strategy to the Z3 backend and measured its performance. Figure 6.5 shows a CDF plot comparing the performance of both solver backends using the two different variable bookkeeping strategies (Solver Wrapper level and Stub-API level).

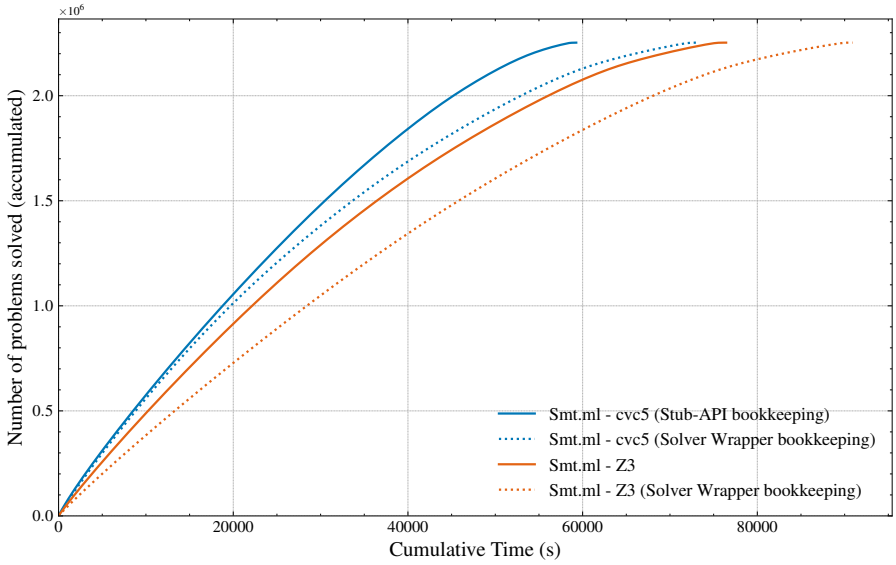

Figure 6.5: Results including Z3 SMT.ML backend when employing variable bookkeeping at the Solver Wrapper level.

The results confirm that the Solver Wrapper variable bookkeeping strategy negatively impacts the performance of both backends. Specifically, for the Z3 backend, using this strategy leads to a substantial performance decline compared to its original backend implementation. This outcome is expected, as the Z3 solver already manages and tracks variables internally, making the additional hash table operations unnecessary and detrimental to performance.

Finally, an important takeaway from these results is the significant impact that the chosen strategy for variable bookkeeping has on the backend's performance, making it a critical factor to consider when implementing solver backends.

Summary

In this chapter, we evaluated the OCaml bindings of cvc5 and their impact on SMT.ML. To assess the correctness and efficiency of cvc5's OCaml bindings, we ran a comprehensive set of queries generated during the symbolic execution of the Test-Comp 2023 benchmark suite. Furthermore, we compared the performance of SMT.ML's cvc5 and Z3 backends, offering valuable insights into how each backend performed depending on the characteristics of the input problem. Additionally, we assessed the impact that the selected variable bookkeeping strategy has on the performance of the cvc5's SMT.ML backend. The following chapter concludes this document by summarising the contributions of this thesis and identifying promising directions for research and future work.

Chapter 7

Conclusions and Future Work

In this chapter, we summarize the main findings and conclusions of this thesis (Section 7.1). Additionally, we identify promising directions for future work and research (Section 7.2).

7.1 Conclusions

Satisfiability Modulo Theories (SMT) solvers have become widely used tools in various domains, including software verification, formal methods, and artificial intelligence. Their ability to solve complex, often undecidable problems in a programmatic manner makes them valuable for users and developers. However, despite their undeniable utility, integrating SMT solvers into systems can present significant challenges. This difficulty primarily stems from the inherent complexity of the solvers themselves, which often involve sophisticated and intricate design choices. Additionally, the complexity of the APIs that SMT solvers expose require users to have a deep understanding of both the solvers' internals and the specific domain of application. To alleviate these challenges, frontends for SMT solvers have been developed, offering a more accessible entry point for users. These frontends abstract much of the complexity associated with SMT solvers, providing a simplified interface that lowers the learning curve and facilitates easier integration into various systems. Furthermore, having a frontend that supports multiple solvers enables users to select the solver that better suits their particular problem, providing flexibility and adaptability in problem-solving approaches.

Considering the aforementioned challenges, we make significant contributions to the development of SMT.ML, an OCaml frontend for SMT solvers. SMT.ML is a frontend that supports four SMT solvers: Bitwuzla [50], Colibri2 [76], cvc5 [22], and Z3 [17]. It enables users to seamlessly switch between which solver they want to use and it does all this through a single and unified syntax, meaning that users do not have to rewrite their entire programs if they want to change the solver they are using. The specific contributions of this thesis are as follows:

1. **The new parametric encoding of SMT.ML.** As part of this thesis, we re-implemented the core

encoding of SMT.ML to make it independent of the target backend solver. This new parametric architecture allowed for extending SMT.ML with support not only for cvc5, which was accomplished in the context of this thesis, but also for Bitwuzla and Colibri2.

2. **The integration of a new backend for the cvc5 SMT solver in SMT.ML.** Leveraging ou new parametric architecture, we extended SMT.ML with support for the cvc5 SMT solver.

3. **The development of OCaml bindings for the cvc5 SMT solver.** As a necessary step for the integration of the new cvc5 backend, we developed OCaml bindings for the cvc5 SMT solver. These bindings are essential for extending SMT.ML with a new backend, as they enable smooth interoperability between the OCaml environment and the cvc5 solver. Additionally, these bindings offer OCaml developers the capability to incorporate cvc5 into other OCaml-based projects without having to use SMT.ML. The development process is comprehensively documented, serving as a valuable resource for future efforts to integrate other SMT solvers or similar tools into OCaml.

4. **The evaluation of SMT.ML.** To assess the effectiveness and performance of SMT.ML, we have conducted a comprehensive evaluation using a dataset of approximately 2.3 million SMT queries. These queries were generated by the symbolic execution engine Owi [5] during the execution of the Test-Comp 2023 [24] symbolic execution benchmark suite. The evaluation compares the performance of the newly integrated cvc5 backend with the existing Z3 backend, focusing on their strengths and weaknesses within the domain of symbolic execution—a key use case for SMT.ML. The results offer valuable insights into the practical implications of using different SMT solvers within SMT.ML, providing developers with guidance on selecting the most appropriate solver for specific verification tasks.

7.2 Future Work

The conducted research paves the way for future work in improving and expanding SMT.ML. The following sections outline potential directions for further enhancing SMT.ML 's capabilities.

7.2.1 Extending SMT.ML with Support for more SMT Solvers

In the future, we plan to extend SMT.ML with support for additional SMT solvers. Offering a wider range of solvers to SMT.ML users allows them to select the one that better suits their given input problem. The annual SMT-COMP competition [23] provides valuable insights into actively maintained solvers' performance across various theories, offering a reliable reference for identifying potential candidates to integrate into SMT.ML. Currently, there is already an ongoing effort to integrate the Alt-Ergo [12] solver within SMT.ML.

7.2.2 Combining SMT.ML Backends During Runtime

Dissimilarities in internal solver implementations and algorithms used to handle different theories can lead to solvers being tailored for specific target problems, thus having areas where they excel and outperform other solvers. This means that being able to combine multiple backends during runtime would allow users to capitalize on the strengths of each solver, leading to an overall improvement in SMT.ML's performance.

We consider two possibilities that would allow interoperability between the backends used. One option is to statically analyse the formula that will be queried and classify it according to the expressions that compose it. This would allow us to determine which theories are being tested and choose the strongest solver accordingly. There already exists work in this area of determining the empirical hardness of SMT problems according to their input characteristics; for instance, MachSMT [83] is an algorithm selection tool for SMT solvers that employs machine learning models to compute a ranking of which solver is most likely to solve a particular problem the fastest.

Another option is to assess one solver's performance during execution and make an online decision based on performance metrics to decide whether to switch to another solver or keep using the current one. To achieve this, we could employ a multi-armed bandit problem model [84], a model for algorithms that make decisions over time under uncertainty. In our case, the uncertainty factor lies in the solver's performance, and our decision would be based on it.

Bibliography

[1] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[2] C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[3] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.

[4] F. Marques, J. Fragoso Santos, N. Santos, and P. Adão. Concolic execution for webassembly. In *36th European Conference on Object-Oriented Programming (ECOOP 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[5] L. Andrès, F. Marques, A. Carcano, P. Chambart, J. Fragoso Femenin dos Santos, and J.-C. Filliâtre. Owi: Performant Parallel Symbolic Execution Made Easy, an Application to WebAssembly. *The Art, Science, and Engineering of Programming*, 9(2), Oct. 2024. URL `https://hal.science/hal-04627413`.

[6] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures 4*, pages 364–387. Springer, 2006.

[7] L. Erkök. Sbv: Smt based verification in haskell. *Software library*, 2019.

[8] T. Mens. On the complexity of software systems. *Computer*, 45(08):79–81, 2012.

[9] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, K. Sivaramakrishnan, and J. Vouillon. *The OCaml system release 5.0: Documentation and user's manual*. PhD thesis, Inria, 2022.

[10] Y. Bertot and P. Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[11] J.-C. Filliâtre and A. Paskevich. Why3—where programs meet provers. In *Programming Languages and Systems: 22nd European Symposium on Programming, ESOP 2013, Held as Part of the*

European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings 22, pages 125–128. Springer, 2013.

[12] S. Conchon, A. Coquereau, M. Iguernlala, and A. Mebsout. Alt-ergo 2.2. In *SMT Workshop: International Workshop on Satisfiability Modulo Theories*, 2018.

[13] C. Barrett, A. Stump, C. Tinelli, et al. The smt-lib standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*, volume 13, page 14, 2010.

[14] E. Torlak and R. Bodik. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 135–152, 2013.

[15] M. Gario and A. Micheli. Pysmt: a solver-agnostic library for fast prototyping of smt-based algorithms. In *SMT workshop*, volume 2015, 2015.

[16] M. Mann, A. Wilson, Y. Zohar, L. Stuntz, A. Irfan, K. Brown, C. Donovick, A. Guman, C. Tinelli, and C. Barrett. Smt-switch: a solver-agnostic c++ api for smt solving. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 377–386. Springer, 2021.

[17] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[18] Y. Ge and L. De Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In *Computer Aided Verification: 21st International Conference, CAV 2009, Grenoble, France, June 26-July 2, 2009. Proceedings 21*, pages 306–320. Springer, 2009.

[19] L. Kovács, S. Robillard, and A. Voronkov. Coming to terms with quantified reasoning. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 260–270, 2017.

[20] R. E. Bryant, D. Kroening, J. Ouaknine, S. A. Seshia, O. Strichman, and B. Brady. Deciding bit-vector arithmetic with abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems: 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24-April 1, 2007. Proceedings 13*, pages 358–372. Springer, 2007.

[21] A. Høfler. Smt solver comparison, 2014.

[22] H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, et al. cvc5: A versatile and industrial-strength smt solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 415–442. Springer, 2022.

[23] M. Bromberger, F. Bobot, , et al. The International Satisfiability Modulo Theories Competition (SMT-COMP)(2024). https://smt-comp.github.io/.

[24] D. Beyer. Software testing: 5th comparative evaluation: Test-comp 2023. *Fundamental Approaches to Software Engineering LNCS 13991*, page 309, 2023.

[25] B. Dutertre. Yices 2.2. In *International Conference on Computer Aided Verification*, pages 737–744. Springer, 2014.

[26] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley. Automatic exploit generation. *Communications of the ACM*, 57(2):74–84, 2014.

[27] R. Nieuwenhuis and A. Oliveras. On sat modulo theories and optimization problems. In *Theory and Applications of Satisfiability Testing-SAT 2006: 9th International Conference, Seattle, WA, USA, August 12-15, 2006. Proceedings 9*, pages 156–169. Springer, 2006.

[28] A. Cimatti, A. Franzén, A. Griggio, R. Sebastiani, and C. Stenico. Satisfiability modulo the theory of costs: Foundations and applications. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 99–113. Springer, 2010.

[29] S. A. Cook. The complexity of theorem-proving procedures. In *Logic, Automata, and Computational Complexity: The Works of Stephen A. Cook*, pages 143–152. 2023.

[30] A. Biere, M. Heule, and H. van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.

[31] L. De Moura and N. Bjørner. Satisfiability modulo theories: An appetizer. In *Brazilian Symposium on Formal Methods*, pages 23–36. Springer, 2009.

[32] C. Cadar, D. Dunbar, D. R. Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[33] W. He, Z. He, H. Wu, and H. Wang. Improved neural machine translation with smt features. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.

[34] D. Bryce, S. Gao, D. Musliner, and R. Goldman. Smt-based nonlinear pddl+ planning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 29, 2015.

[35] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll (t). *Journal of the ACM (JACM)*, 53(6):937–977, 2006.

[36] C. Barrett, D. Dill, and J. Levitt. Validity checking for combinations of theories with equality. In *Formal Methods in Computer-Aided Design: First International Conference, FMCAD'96 Palo Alto, CA, USA, November 6–8, 1996 Proceedings 1*, pages 187–201. Springer, 1996.

[37] C. Barrett and S. Berezin. Cvc lite: A new implementation of the cooperating validity checker: Category b. In *Computer Aided Verification: 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004. Proceedings 16*, pages 515–518. Springer, 2004.

[38] W. Damm and H. Hermanns. *Computer Aided Verification: 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590. Springer, 2007.

[39] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. Cvc4. In *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*, pages 171–177. Springer, 2011.

[40] B. Dutertre and L. De Moura. The yices smt solver. *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, 2(2):1–2, 2006.

[41] K. Weitz, D. Woos, E. Torlak, M. D. Ernst, A. Krishnamurthy, and Z. Tatlock. Scalable verification of border gateway protocol configurations with an smt solver. In *Proceedings of the 2016 acm sigplan international conference on object-oriented programming, systems, languages, and applications*, pages 765–780, 2016.

[42] S. Szymoniak, O. Siedlecka-Lamch, A. M. Zbrzezny, A. Zbrzezny, and M. Kurkowski. Sat and smt-based verification of security protocols including time aspects. *Sensors*, 21(9):3055, 2021.

[43] A. Stump, C. W. Barrett, and D. L. Dill. Cvc: A cooperating validity checker. In *Computer Aided Verification: 14th International Conference, CAV 2002 Copenhagen, Denmark, July 27–31, 2002 Proceedings 14*, pages 500–504. Springer, 2002.

[44] R. Alur, R. Bodik, G. Juniwal, M. M. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. *Syntax-guided synthesis*. IEEE, 2013.

[45] N. Eén and N. Sörensson. An extensible sat-solver. In *International conference on theory and applications of satisfiability testing*, pages 502–518. Springer, 2003.

[46] A. Stump, D. Oe, A. Reynolds, L. Hadarean, and C. Tinelli. Smt proof checking using a logical framework. *Formal Methods in System Design*, 42:91–118, 2013.

[47] L. d. Moura and S. Ullrich. The lean 4 theorem prover and programming language. In *Automated Deduction–CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings 28*, pages 625–635. Springer, 2021.

[48] T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002.

[49] G. Sutcliffe. The tptp problem library and associated infrastructure: from cnf to th0, tptp v6. 4.0. *Journal of Automated Reasoning*, 59(4):483–502, 2017.

[50] A. Niemetz and M. Preiner. Bitwuzla. In *International Conference on Computer Aided Verification*, pages 3–17. Springer, 2023.

[51] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, and R. Sebastiani. The MATHSAT 4 SMT solver: Tool paper. In *Computer Aided Verification: 20th International Conference, CAV 2008 Princeton, NJ, USA, July 7-14, 2008 Proceedings 20*, pages 299–303. Springer, 2008.

[52] A. Niemetz, M. Preiner, C. Wolf, and A. Biere. Btor2, btormc and boolector 3.0. In *International Conference on Computer Aided Verification*, pages 587–595. Springer, 2018.

[53] M. Felleisen, R. B. Findler, M. Flatt, S. Krishnamurthi, E. Barzilay, J. McCarthy, and S. Tobin-Hochstadt. The racket manifesto. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[54] R. Bodík and B. Jobstmann. Algorithmic program synthesis: introduction. *International journal on software tools for technology transfer*, 15:397–411, 2013.

[55] J. F. Santos, P. Maksimović, T. Grohens, J. Dolby, and P. Gardner. Symbolic execution for JavaScript. In *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming*, pages 1–14, 2018.

[56] D. Chisnall. The challenge of cross-language interoperability. *Communications of the ACM*, 56(12): 50–56, 2013.

[57] M. Grimmer, R. Schatz, C. Seaton, T. Würthinger, M. Luján, and H. Mössenböck. Cross-language interoperability in a multi-language runtime. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 40(2):1–43, 2018.

[58] E. Meijer and J. Gough. Technical overview of the common language runtime. 2001.

[59] M. Slee, A. Agarwal, and M. Kwiatkowski. Thrift: Scalable cross-language services implementation. *Facebook white paper*, 5(8):127, 2007.

[60] B. W. Kernighan and D. M. Ritchie. The c programming language. 2002.

[61] W. Visser, J. Geldenhuys, and M. B. Dwyer. Green: reducing, reusing and recycling constraints in program analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 1–11, 2012.

[62] IEEE. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, 2019.

[63] J. Cohen. Garbage collection of linked data structures. *ACM Computing Surveys (CSUR)*, 13(3): 341–367, 1981.

[64] L. P. Deutsch and D. G. Bobrow. An efficient, incremental, automatic garbage collector. *Communications of the ACM*, 19(9):522–526, 1976.

[65] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.

[66] R. Jones, A. Hosking, and E. Moss. *The garbage collection handbook: the art of automatic memory management*. CRC Press, 2023.

[67] K. Sivaramakrishnan, S. Dolan, L. White, S. Jaffer, T. Kelly, A. Sahoo, S. Parimala, A. Dhiman, and A. Madhavapeddy. Retrofitting parallelism onto ocaml. *Proceedings of the ACM on Programming Languages*, 4(ICFP):1–30, 2020.

[68] A. W. Appel. Simple generational garbage collection and fast allocation. *Software: Practice and experience*, 19(2):171–183, 1989.

[69] P. R. Wilson. Uniprocessor garbage collection techniques. In *International Workshop on Memory Management*, pages 1–42. Springer, 1992.

[70] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Computer Science Logic: 15th International Workshop, CSL 2001 10th Annual Conference of the EACSL Paris, France, September 10–13, 2001, Proceedings 15*, pages 1–19. Springer, 2001.

[71] R. Piskac, T. Wies, and D. Zufferey. Automating separation logic using smt. In *Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings 25*, pages 773–789. Springer, 2013.

[72] Y. Minsky, A. Madhavapeddy, and J. Hickey. *Real World OCaml: Functional programming for the masses*. " O'Reilly Media, Inc.", 2013.

[73] F. Tuong, F. Le Fessant, and T. Gazagnaire. OPAM: an OCaml packa manager. In *ACM SIGPLAN OCaml Users and Developers Workshop*, 2012.

[74] A. Biere, T. Faller, K. Fazekas, M. Fleury, N. Froleyks, and F. Pollitt. Cadical 2.0. In *International Conference on Computer Aided Verification*, pages 133–152. Springer, 2024.

[75] D. Jovanovic and B. Dutertre. Libpoly: A library for reasoning about polynomials. In *SMT*, pages 28–39, 2017.

[76] F. Bobot, B. Marre, G. Bury, S. Graham-Lengrand, and H. R. Ait El Hara. Colibri2. webpage: `https://colibri.frama-c.com`, source code: `https://git.frama-c.com/pub/colibrics`.

[77] A. Wilson, A. Noetzli, A. Reynolds, B. Cook, C. Tinelli, and C. W. Barrett. Partitioning strategies for distributed smt solving. In *FMCAD*, pages 199–208, 2023.

[78] H. Palikareva and C. Cadar. Multi-solver support in symbolic execution. In *Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings 25*, pages 53–68. Springer, 2013.

[79] E. Rakadjiev, T. Shimosawa, H. Mine, and S. Oshima. Parallel SMT solving and concurrent symbolic execution. In *2015 IEEE Trustcom/BigDataSE/ISPA*, volume 3, pages 17–26. IEEE, 2015.

[80] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing Mayhem on Binary Code. In *2012 IEEE Symposium on Security and Privacy*, pages 380–394, 2012. doi: 10.1109/SP.2012.31.

[81] J.-C. Filliâtre and S. Conchon. Type-safe modular hash-consing. In *Proceedings of the 2006 Workshop on ML*, ML '06, page 12–19, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595934839. doi: 10.1145/1159876.1159880. URL https://doi.org/10.1145/1159876.1159880.

[82] Google SparseHash. https://github.com/sparsehash/sparsehash.

[83] J. Scott, A. Niemetz, M. Preiner, S. Nejati, and V. Ganesh. Machsmt: A machine learning-based algorithm selector for smt solvers. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 303–325. Springer, 2021.

[84] A. Slivkins et al. Introduction to multi-armed bandits. *Foundations and Trends® in Machine Learning*, 12(1-2):1–286, 2019.

Chapter 8

Appendix

8.1 OCaml and C++ interoperability

```cpp
#include <iostream>
#include "cpp_lists.h"

using namespace std;

/* Instantiates a new list */
list<int> * init_list(int val){
    list<int> * l = new list<int>();
    l->push_front(val);
    return l;
}

/* Prepends a value to an existing list */
list<int> * prepend(list<int> * l, int val){
    l->push_front(val);
    return l;
}

/* Prints the contents of a list */
void print_list(list<int> * l){
    list<int>::iterator it;
    for(it = l->begin(); it != l->end(); it++){
        cout << *it << " ";
    }
    cout << endl;
}

/* Returns the length of a list */
size_t length(list<int> * l){
    return l->size();
}

/* Pops first element of a list */
int pop(list<int> * l){
    int val = l->front();
    l->pop_front();
    return val;
}

/* Deletes a list */
void delete_list(list<int> * l){
    delete l;
}
```

Listing 8.1: C++ Singly-linked lists example.

```cpp
#include "cpp_lists.h"

extern "C" {
    #include <caml/mlvalues.h>
    #include <caml/memory.h>
    #include <caml/alloc.h>

    static value Val_list(list<int> * p) {
      return caml_copy_nativeint((intnat) p);
    }

    static list<int> * List_val(value v) {
        return (list<int> *)Nativeint_val(v);
    }

    CAMLprim value caml_stub_init_list(value v) {
      list<int> * p = init_list(Int_val(v));
      return Val_list(p);
    }

    CAMLprim value caml_stub_prepend(value v, value l) {
      list<int> *p = List_val(l);
      list<int> *q = prepend(p, Int_val(v));
      return Val_list(q);
    }

    CAMLprim value caml_stub_length(value l) {
      list<int> *p = List_val(l);
      int len = length(p);
      return Val_int(len);
    }

    CAMLprim value caml_stub_pop(value l){
      list<int> *p = List_val(l);
      int val = pop(p);
      return Val_int(val);
    }

    CAMLprim value caml_stub_print_list(value l) {
      list<int> *p = List_val(l);
      print_list(p);
      return Val_unit;
    }

    CAMLprim value caml_stub_delete_list(value l) {
      list<int> *p = List_val(l);
      delete_list(p);
      return Val_unit;
    }
}
```

Listing 8.2: C stubs for C++ code in Listing 8.1.