Verified Compilation and Optimization of Floating-Point Kernels

Dissertation zur Erlangung des Grades des Doktors der Ingenieurwissenschaften
der Fakultät für Mathematik und Informatik der Universität des Saarlandes

vorgelegt von
Heiko Becker

Saarbrücken, September 2022

Tag des Kolloquiums

TBD

Dekan der Fakultät für Mathematik und Informatik:

Prof. Dr. Jürgen Steimle

Prüfungsausschuss

Vorsitzender:	Prof. Dr. Jan Reineke
Gutachter:	Assoc. Prof. Dr. Eva Darulova
	Assoc. Prof. Dr. Magnus. O. Myreen
	Prof. Dr. Sebastian Hack
	Assoc. Prof. Dr. Michael Norrish
Akademischer Mitarbeiter:	Dr. Emanuele D'Osualdo

Abstract

When verifying safety-critical code on the level of source code, we trust the compiler to produce machine code that preserves the behavior of the source code. Trusting a verified compiler is easy. A rigorous machine-checked proof shows that the compiler correctly translates source code into machine code.

Modern verified compilers (e.g. CompCert and CakeML) have rich input languages, but only rudimentary support for floating-point arithmetic. In fact, state-of-the-art verified compilers only implement and verify an inflexible one-to-one translation from floating-point source code to machine code. This translation completely ignores that floating-point arithmetic is actually a discrete representation of the continuous real numbers.

This thesis presents two extensions improving floating-point arithmetic in CakeML. First, the thesis demonstrates *verified compilation* of elementary functions to floating-point code in: *Dandelion*, an automatic verifier for polynomial approximations of elementary functions; and *libmGen*, a proof-producing compiler relating floating-point machine code to the implemented real-numbered elementary function. Second, the thesis demonstrates *verified optimization* of floating-point code in: *Icing*, a floating-point language extending standard floating-point arithmetic with optimizations similar to those used by unverified compilers, like GCC and LLVM; and *RealCake*, an extension of CakeML with Icing into the first fully verified optimizing compiler for floating-point arithmetic.

Zusammenfassung

Bei der Verifizierung von sicherheitsrelevantem Quellcode vertrauen wir dem Compiler, dass er Maschinencode ausgibt, der sich wie der Quellcode verhält. Man kann ohne weiteres einem verifizierten Compiler vertrauen. Ein rigoroser maschinen-überprüfter Beweis zeigt, dass der Compiler Quellcode in korrekten Maschinencode übersetzt.

Moderne verifizierte Compiler (z.B. CompCert und CakeML) haben komplizierte Eingabesprachen, aber unterstützen Fließkommaarithmetik nur rudimentär. De facto implementieren und verifizieren hochmoderne verifizierte Compiler für Fließkommaarithmetik nur eine starre eins-zu-eins Übersetzung von Quell- zu Maschinencode. Diese Übersetzung ignoriert vollständig, dass Fließkommaarithmetik eigentlich eine diskrete Repräsentation der kontinuierlichen reellen Zahlen ist.

Diese Dissertation präsentiert zwei Erweiterungen die Fließkommaarithmetik in CakeML verbessern. Zuerst demonstriert die Dissertation *verifizierte Übersetzung* von elementaren Funktionen in Fließkommacode mit: *Dandelion*, einem automatischen Verifizierer für Polynomapproximierungen von elementaren Funktionen; und *libmGen*, einen Beweis-erzeugenden Compiler der Fließkommacode in Relation mit der implementierten elementaren Funktion setzt. Dann demonstriert die Dissertation *verifizierte Optimierung* von Fließkommacode mit: *Icing*, einer Fließkommasprache die Fließkommaarithmetik mit Optimierungen erweitert die ähnlich zu denen in unverifizierten Compilern, wie GCC und LLVM, sind; und *RealCake*, eine Erweiterung von CakeML mit Icing als der erste vollverifizierte Compiler für Fließkommaarithmetik.

Acknowledgements

This thesis would not exist without all of the support I received along the way and I am deeply grateful to everyone that helped me get to this point.

First and foremost, I could have not undertaken this journey without my advisor, Eva Darulova. Your hands-on advise was always spot-on and I would not have been able to conduct the research presented in this thesis without your continuous and amazing advise. I am also grateful to my co-advisor, Magnus Myreen. Thank you for all the technical and non-technical advise, for showing me a new area of interactive theorem proving, and for introducing me to the CakeML community.

Next, I thank my thesis examiners, Sebastian Hack and Michael Norrish, for reviewing my thesis. I also thank Jan Reineke for acting as the chair of the examination board, and Emnuele D'Osualdo for being the academic member of the committee. My thanks also go to Debasmita Lohar, Fabian Ritter, Ahana Gosh, Anastasiia Izycheva, and Philipp Fuchs for helping me proof-read the thesis.

None of the work presented in this thesis would exist without the amazing collaborators I got to work with during my PhD: Zachary Tatlock, Anastasiia Volkova, Jean-Baptiste Jeannin, Ivan Gavran, Rupak Majumdar, and Pavel Panchekha; as well as all the undergraduate and graduate students that I mentored and worked with in research projects: Raphael Monat, Nikita Zyuzin, Joachim Bard, Robert Rabe, Nathaniel Bos, and Mohit Tekriwal. All of you have influenced my academic career and personal development, and your contributions to the research we have conducted together were invaluable.

My time at MPI-SWS started with me doing a research internship in Eva's group, and I still fondly remember my first visit day to Kaiserslautern. I am grateful to Manohar Vanga, Arpan Gujarati, and Aastha Mehta for giving me the best possible welcome at the institute on that day. Also, thank you Aastha for being an amazing office maturing during the time we overlapped.

I am grateful for all the friends and colleagues I made in the AVA group and generally at MPI-SWS during my PhD. You all pushed me through many paper deadlines, supported me with listening to my proof-related babblings and we had the most amazing coffee chats together. I especially thank Debasmita Lohar, Rosa Abbasi, Anastasiia Izycheva, and Clothilde Jeangoudoux from the AVA group, and my colleagues Hai Dang, Ivan Gavran, and Ahana Gosh.

I also thank my friends from the compiler lab, Fabian Ritter, and Tina Jung with whom I am proud to have done most of my undergraduate studies and PhD studies together. Thank you for every lunch break and every coffee chat we had.

As a PhD student at MPI-SWS, I benefited greatly from the amazing support by our IT and office staff, and our scientific support. They all have helped me in so many ways such that I could always focus on my research work. Thank you Claudia Richter, Annika Meiser, Gretchen Gravelle, Christian Klein, Carina Schmitt, and Rose Hobermann.

Academically, I would not be where I am at today, if I had not gotten in touch with research already during my first years at university. Therefore I thank everyone that advised and mentored me during my undergraduate studies, especially Sigurd Schneider, Ivan Beschastnikh, Jasmin Blanchette, Sebastian Hack, and everyone from the compiler design lab.

During my PhD I also got introduced to the most amazing research communities, the CakeML group, and the FPBench group. Thank you to the FPBench community for the interesting monthly chats and talks that showed me the different facets of finite-precision research; and thank you to the CakeML community for the amazing developer meetings and for answering even the craziest HOL4 questions of mine.

Finally, I also thank all my family for their support during my PhD. My parents, Hermann Josef und Karin Becker, my sister Sandra with her husband Patrick, my brother Jens, my parents-in-law Roman and Christel Breunig, my sister-in-law Anna and her husband Philipp. The person I am most grateful for having in my life is my wife Nora. Without her I would not be the person that I am today and I am proud of every challenge we have overcome and every step we have taken together so far. Hopefully many more exciting adventures will follow.

Contents

CHAPTER 1

Introduction

To run source code on a machine, one can either interpret it directly, or translate source code to machine code using a compiler. If we opt for using a compiler and run it on source code, we expect that the compiler generates machine code with the behavior specified in our source code. However, Yang et al. [121] have shown that compilers can produce machine code that does not have the same behavior as the source code they started off with due to compiler bugs. While a compiler bug is not particularly harmful in everyday coding, a compiler bug can have significant impact on safety-critical code, since such code is part of, e.g., airplanes, surgical robots, and self-driving cars.

One way to ensure that safety-critical code behaves as expected is through formal verification. This formal verification is most tractable on the level of source code. To not invalidate the guarantees established by verification of the source code, it is mandatory that the machine code produced by the compiler behaves like the initially given source code. As an example, a system designer may prove that a proximity sensor for an autonomous car always raises an alarm before a car crash. Clearly, the machine code generated by the compiler for this proximity sensor must still always raise an alarm before a car crash and not randomly stop working, making it mandatory that the compiler preserves guarantees of source code and is free of compiler bugs.

The absence of compiler bugs cannot be proven by even the most rigorous testing. Testing can only prove the presence of bugs, but never their absence. But, a way of showing that a compiler is bug-free is to formulate a rigorous mathematical proof that the compiler preserves the behavior of its input source code. From such a proof, we can conclude that the compiler a) contains no bug, and b) preserves guarantees proven about the source code. We call a compiler that provably preserves the behavior of source code a *verified compiler*, and we call the accompanying proof a *compiler correctness proof*[1].

[1] In fact, the only code that Yang et al.[121] could not find any bugs in was the code of a verified compiler.

Proving compiler correctness with pen-and-paper is a major challenge, because pen-and-paper proofs are in general error prone and compiler correctness proofs specifically require substantial bookkeeping when done properly. Instead, compiler correctness is commonly proven using a so-called interactive theorem prover (ITP).

An ITP implements a logic as a set of rules and these rules are used to perform proofs in the ITP. This set of logical rules is usually justified by an underlying mathematical theory and therefore trusted by users of the ITP. Within a subset of the ITP's trusted logic one can usually write functional programs and prove properties of these programs with the ITP. The first benefit of an ITP for proving compiler correctness is that because the ITP is trustworthy, any proof performed with the ITP is generally more reliable than a pen-and-paper argument. The second benefit of an ITP is that the ITP makes it easier to update and maintain proofs if the implementation of the compiler changes.

Verfied compilers are not a new topic, and many different verified compilers have been developed [89, 90, 66, 74, 124, 113]. The two most mature verified compilers are CompCert [74], which implements a version of the C-language standard [59], and CakeML [113], which implements a dialect of Standard ML [88]. Both CompCert and CakeML handle interesting programs and support features like integer and word arithmetic, Boolean checks, memory, I/O, and rich control structures.

Example. Now that we have covered some ground for verified compilers, let us look back at our initial motivation: compiling safety-critical source code to machine code while preserving correctness guarantees. As an example, we look at a small kernel that computes the x coordinate when translating polar to cartesian coordinates[2]:

```
fun polToCart_x(radius: double, theta: double): double = let
  val pi = 3.14159265359
  val radiant = (theta * (pi / 180.0))
  in (radius * cos (radiant)) end
```

Function `polToCart_x` could for example be used in an autonomous car. A key challenge of verified compilation for our example kernel is that the kernel is implemented in (64-bit double) floating-point arithmetic.

To understand why verified compilation of floating-point arithmetic is challenging, we first look at what makes floating-point arithmetic tricky in the first place. Floating-point arithmetic is commonly used to represent real-numbers in computers. However, because some real numbers like, e.g., π are infinite floating-point arithmetic cannot represent

[2]The kernel is based on the `polarToCarthesian,x` benchmark of FPBench [31]

all real-numbers in finite hardware. Thus, discrete floating-point arithmetic necessarily approximates the continuous real-numbers. This approximation comes at the price of an unavoidable difference between the idealized real-number, and its discrete floating-point representation, the so-called *roundoff error*. It is this very roundoff error that makes implementing floating-point arithmetic in a verified compiler hard.

Before the work done in this thesis, both CompCert and CakeML were not able to reason about the roundoff error of floating-point arithmetic at all. One particularly striking observation is that both CompCert and CakeML compile floating-point arithmetic one-to-one into machine code, i.e., each floating-point operation in the source code corresponds to one floating-point operation in the produced machine code. This is in stark contrast to how any other arithmetic, and source code program in general, is handled as a verified compiler generally attempts to optimize the machine code for the target architecture. Further, we have explained that floating-point arithmetic is used to implement real-numbers, and thus one would expect the compiler correctness theorem to relate floating-point machine code to real-number arithmetic. However, neither CakeML nor CompCert establish such a relation between floating-point machine code and real-number arithmetic.

To remedy these deficiencies of verified compilers, this thesis presents two complementary extensions of floating-point arithmetic in a verified compiler: First, we demonstrate how a verified compiler can fully automatically compile a mathematical function like sin and cos into floating-point machine code, if no hardware equivalent is available. Second, we extend floating-point arithmetic in verified compilers such that performance of floating-point machine code can be improved using optimizations. Both of these extensions ultimately relate the produced floating-point machine code to the real-number semantics of the initial program. We illustrate both extensions on the example code from above.

Extension I: Verified Compilation of Mathematical Functions. If we use function `polToCart_x` in an autonomous car and compile it with a state-of-the-art verified compiler like CakeML or CompCert, the compiler can only try to replace the call to `cos` by a general purpose library function. However, in an embedded setting, as is the case for autonomous cars, such a library function might not be available. If a library function is available, it may be inefficient as a general purpose function must account for all corner cases of floating-point arithmetic.

If we take a step back, and carefully revise the setting, in which function `polToCart_x` is meant to be used, one notices that in an autonomous car the designer can rely on the inputs `radius` and `theta` of function `polToCart_x` being bounded as the car is likely only

ever to be used on earth, limiting the possible values the inputs may reach. This allows the designer to state an input constraint restricting the input values, for example[3]:

$$1.0 \leq \texttt{radius} \leq 10.0 \wedge 0.0 \leq \texttt{theta} \leq 360.0$$

Using this input constraint it is possible to compute accurate bounds on the inputs of the call to the `cos` function, and replace the general-purpose `cos` function with a custom *polynomial approximation* that is use-case specific. This polynomial approximation is more efficient as it does not need to handle all of the corner cases of floating-point arithmetic.

Prior to this work, both CakeML and CompCert were unable to make use of input constraints and thus cannot compute custom polynomial approximations. To make matters worse, coming up with a custom polynomial approximation manually is hard, and therefore there exist automatic approximation algorithms. One example for such algorithms are so-called Remez-like algorithms [94] and these are implemented in tools like metalibm [69] and Sollya [25]. A key issue, both for verified compilers, and of polynomial approximations in general, is that custom polynomial approximations introduce a second source of error, the so-called *approximation error*. The approximation error bounds the difference between an elementary function, and its polynomial approximation in real-number arithmetic. Therefore, for a given elementary function, a Remez-like algorithm computes both a polynomial approximation and a bound on the approximation error. As an example, using Sollya we can approximate the cos function on $[0, 1]$ with the polynomial $p(x) = 0.784 \times 10^{-2} \times x^3 - 0.547 \times x^2 + 9.238 \times 10^{-3} \times x + 0.999$ with an approximation error of 3.809×10^{-4}.

Remez-like algorithms are known to compute the best-possible approximations for elementary functions, but so-far, they have not been verified. Specifically, suppose that for a given elementary function, a Remez-like algorithm computes a polynomial approximation and an approximation error. A key question is whether the approximation error is a true upper bound to the difference between the elementary function and its polynomial approximation.

Contributions (Extension I). The first extension described in this thesis tackles the problem of verifying polynomial approximations computed by Remez-like algorithms and compiling these approximations to floating-point machine code using the CakeML verified compiler. To this end, we first present *Dandelion*, a verified checker for polynomial approximations (Chapter 3). Dandelion proves approximation error bounds fully automatically and computes results fast with a verified binary.

[3]The input constraint is taken from `polarToCarthesian,x` benchmark of FPBench[31]

The key challenge in developing Dandelion is to come up with an automatic and verifiable method to prove real-numbered inequalities of the form $\max_x |f(x) - p(x)| \leq \varepsilon$ where $f(x)$ is a real-numbered elementary function, $p(x)$ is a polynomial, and ε is the constant approximation error. This challenge is further complicated by the fact that elementary functions are often defined as infinite sums, which rules out straight-forward techniques based on evaluation.

Building upon Dandelion, we extend CakeML with a proof-producing compiler for polynomial approximations, called *libmGen* (Chapter 4). For a given elementary function and its polynomial approximation, libmGen automatically implements the polynomial in CakeML. Our tool libmGen further proves a specification that relates the CakeML source code to the real-numbered elementary function it implements with an accuracy bound that includes both the approximation error and the roundoff error.

The main technical challenge in developing libmGen is to properly automate the translation from elementary functions to CakeML source-code, with an accompanying accuracy proof. This requires a combination of analyses to ensure that a valid accuracy bound is computed which accounts for both the approximation error and the roundoff error.

Example Continued. Before explaining the second extension of the thesis, we briefly return to our example. With the tools from the first extension described in this thesis, CakeML implements function `polToCart_x` as:

```
fun polToCart_x(radius: real, theta: real): real = let
 val pi = 3.14159265359
 val radiant = (theta * (pi / 180.0))
 val cos_rad = ((radiant^3 * 3.3222216089257017e-09) +
           ((radiant^2 * 0.2026423215866089) +
           ((radiant * -1.2732393741607666) + 1.3056366443634033)))
 in (radius * cos_rad) end
```

If we compile this program to machine code with CompCert or CakeML, we get machine code that exactly implements the original source code. While verified compilers can relax the required one-to-one correspondance between source and machine code by performing conservative, bit-accurate optimizations (e.g., replacing $2 \times x$ with $x + x$) they preserve the *IEEE-754 semantics* [58] of their input programs in all other cases. This approach to compilation is in stark contrast to the rich set of so-called *fast-math optimizations* used by unverified compilers such as GCC [40] and LLVM [70]. When applying

fast-math optimizations, an unverified compiler can, for example, replace the last computation of variable `cos_rad` (`(radiant * -1.2732393741607666) + 1.3056366443634033`) by an `fma` instruction (`fma(radiant, -1.2732393741607666, 1.3056366443634033)`). The `fma` instruction `fma(a,b,c)` computes a locally more accurate version of `(a * b) + c`, as the result is rounded to floating-point only once. An unverified compiler may use the `fma` instruction as the instruction is said to be locally more accurate and faster. However, this very increase in accuracy changes the global roundoff error of the computation, which in turn disallows introduction of `fma` instructions by a verified compiler.

Extension II: Verified Optimizations of Floating-Point Arithmetic. The fast-math optimizations performed by GCC and LLVM cannot straight-forwardly be applied by CompCert and CakeML as they necessarily reorder arithmetic expression, which in turn prevents proving a one-to-one correspondance between source code and machine code. Intuitively, this is because a floating-point multiplication `a * b` internally has to be treated by the compiler as `rnd (a × b)`, where × is a real-number multiplication, and `rnd` translates real-numbers into floating-point constants. Consequently, expression `rnd (a × (rnd (b × c)))` is not equivalent to `rnd (rnd (a × b) × c)`.

However, if we take a step back and look at the context in which our example code (and safety-critical floating-point code in general) is used, we argue that the code for an autonomous car must anyway be able to tolerate (bounded) noise on the input values, as no sensor reading will ever be perfectly accurate. Thus, both the code itself that computes values from sensor readings, as well as all of its dependants must be able to tolerate a certain, *bounded noise*. Rephrasing this observation a bit, the programmer is indifferent to the exact version produced by the compiler as long as the machine code adheres to a tolerable noise bound, i.e., there is no one-to-one correspondance needed. Ideally, a verified compiler uses this very noise bound as its correctness criterion for verified optimizations of floating-point programs.

Contributions (Extension II). In the second part of the thesis, we extend the CakeML compiler with a proof-producing floating-point optimizer. To do so, we first design a novel floating-point semantics, dubbed *Icing*, which is the first semantics that can verify fast-math-style optimizations as they occur in GCC and LLVM while being backwards compatible to verified compilers (Chapter 5).

The key challenge that we faced when designing Icing is coming up with a relaxed version of the IEEE-754 floating-point semantics, which integrates with verified compilers and optimizes floating-point programs with full control for the end-user. As Icing is

destined to be used in a verified compiler, all of the above must be done while keeping the proof burden for the end-user as small as possible.

Because Icing itself does not reason about accuracy bounds, we integrate it with the CakeML compiler and a verified accuracy analysis to provide end-to-end guarantees about optimized floating-point code in our extension to CakeML, called *RealCake* (Chapter 6).

The key technical challenge in designing RealCake is to properly integrate all the necessary tooling required for proving accuracy guarantees for floating-point machine code. While this may seem simple at a first glance, the key challenge here is that all tools involved must be unified inside CakeML using delicate simulation proofs. To further complicate matters, all of the work has to happen without breaking the already existing tools built around the CakeML compiler.

Structure of This Thesis

The content of this thesis is based on previously peer-reviewed publications. Below we[4] list each chapter and the publications on which it is based. Each of these chapters starts with an explanation of which parts of the original paper have been kept, and which have been removed or extended. Below, we give an overview of the thesis and which publications are included in each chapter.

- Chapter 1 contains primarily new content but reuses formulations from the introductions of our papers which are the basis of Chapter 3, Chapter 5, and Chapter 6.
- Chapter 2 gives background information on key technologies used throughout the thesis. The chapter presents new content.
- Chapter 3 presents *Dandelion*, a fully automated verifier for polynomial approximations computed by Remez-like algorithms. The content is based on the publication

  Heiko Becker, Mohit Tekriwal, Eva Darulova, Anastasia Volkova, and Jean-Baptiste Jeannin. "Dandelion: Certified Approximations of Elementary Functions". In: *Conference on Interactive Theorem Proving (ITP)*. 2022. DOI: 10.4230/LIPIcs.ITP.2022.6

  Artifact: https://github.com/HeikoBecker/Dandelion

[4]While I have been the main author of the publications included in the thesis, all scientific contributions are referred to using "we" because none of the work would have been possible without all of my collaborators.

- Chapter 4 presents *libmGen*, the proof-producing compiler from real-numbered elementary functions to floating-point machine code. The chapter presents new content and is not based on a scientific publication.

- Chapter 5 presents the *Icing* language that supports fast-math-style optimizations in a verified compiler. The chapter is based on the publication

  Becker, Heiko and Darulova, Eva and Myreen, Magnus O. and Tatlock, Zachary. "Icing: Supporting Fast-Math Style Optimizations in a Verified Compiler". In: *Conference on Computer Aided Verification (CAV)*. 2019. DOI: 10.1007/978-3-030-25543-5_10

  Artifact: https://gitlab.mpi-sws.org/AVA/icing

- Chapter 6 presents *RealCake* the first verified optimizing compiler relating floating-point machine code with its unoptimized real-numbered equivalent. The chapter is based on the publication

  Heiko Becker, Robert Rabe, Eva Darulova, Magnus O Myreen, Zachary Tatlock, Ramana Kumar, Yong Kiam Tan, and Anthony Fox. "Verified Compilation and Optimization of Floating-Point Programs in CakeML". in: *European Conference on Object-Oriented Programming (ECOOP)*. 2022. DOI: 10.4230/LIPIcs.ECOOP.2022.1

  Artifact: Heiko Becker, Robert Rabe, Eva Darulova, Magnus O. Myreen, Zachary Tatlock, Ramana Kumar, Yong Kiam Tan, and Anthony Fox. "Verified Compilation and Optimization of Floating-Point Programs in CakeML (Artifact)". In: *Dagstuhl Artifacts Series* 8.2 (2022). DOI: 10.4230/DARTS.8.2.10

- Chapter 7 concludes the thesis and highlights potential future work. The chapter presents new content.

CHAPTER 2

Background

Before going into the contributions of the thesis, we give an overview of the key technologies and concepts used throughout. This chapter starts with a primer on interactive theorem proving, which is the key technology used for all of the contributions, then the chapter explains the main ideas of floating-point arithmetic, and finally, the chapter gives an introduction to the analysis of roundoff errors.

2.1. Interactive Theorem Proving

The key technology used throughout this thesis is interactive theorem proving. On a high-level, interactive theorem proving performs rigorous mathematical proofs with the help of a computer; proofs are performed using a so-called *Interactive Theorem Prover* (ITP).

Notable results have been formally proven with ITPs like, e.g., the Kepler conjecture [48], the four-color-theorem [43], and the seL4 operating system microkernel [65]. The main purpose of an ITP is to assist the user by keeping track of the current state of the proof and ensuring that no mistakes are made. We explain this key functionality with an example.

We prove the closed form for the sum of the first n natural numbers, discovered by Gauss:

$$\forall n. \sum_{i=0}^{n} n = \frac{n * (n+1)}{2}$$

The summation can be easily defined as a function:

```
fun sum n = if n = 0 then 0 else n + sum (n -1)
```

and in an ITP we state the closed form from above as the proof goal

```
∀ n. sum n = (n * (n + 1)) / 2
```

The proof of the goal is commonly done by induction, and the ITP keeps track of the two subgoals:

$$\frac{}{\texttt{sum 0 = (0 * (0 + 1)) / 2}}\ \text{Base}$$

$$\frac{\texttt{sum n = (n * (n + 1)) / 2}}{\texttt{sum (n + 1) = ((n+1) * ((n + 1) + 1)) / 2}}\ \text{Step}$$

The goal labeled "Base" corresponds to the induction base, and the goal labeled "Step" corresponds to the induction step. A key benefit is that after telling the ITP that we perform an induction on n, the ITP automatically generates the proof goals "Base" and "Step". However, if we look closely at the induction step, it is apparent that the ITP only started the induction, but did not automatically perform additional simplification steps as one would have done in a pen-&-paper proof.

In general, the exact steps required to finish our example proof vary depending on which ITP is used. This difference in proof steps stems from the different underlying logics used to justify theorems in ITPs and how these logics are implemented.

One way to implement such an underlying logic is the calculus of inductive constructions [102], which is for example used in the ITPs Coq [116], and Agda [115]. Another way to implement such a logic follows the architecture presented in the Logic for Computable Functions (LCF) which has been first described by Milner [87]. In LCF, a small, trusted kernel of logical rules is used to justify each and every proof step in the ITP. ITP's following this key idea of a trusted small kernel are called "LCF-style" provers, and example ITPs are Isabelle/HOL [119], HOL-Light [117], and HOL4 [118]. HOL4 was used to develop the tools presented in this thesis and thus we give an intuition of how proofs are performed inside HOL4.

As an LCF-style prover, HOL4 implements a small, trusted, logic kernel and an overview of the rules used by the HOL4 kernel is given by Gordon and Melham [44][1]. Intuitively, these rules represent elementary logical inference rules, from which all other operations are derived. Exactly as in our example, functions are first-class citizens in HOL4 and are reasoned upon during proofs. Figure 2.1 shows the HOL4 proof for our example theorem.

The text `closed_sum` between the `Theorem` keyword and the `:` defines the name for the theorem to reference it in later proofs, and the text between the `:` and the `Proof` keyword is the goal to be proven using HOL4. In the goal, `!` is universal quantification, `DIV` is division of natural numbers, and `SUC` refers to the "successor" function from Peano

[1] An up-to-date explanation of the kernel for the current release of HOL4 is also provided in the logic manual on the official HOL4 webpage [118].

```
Theorem closed_sum:
  ! n. sum n = (n * (n + 1)) DIV 2
Proof
  Induct_on `n` >> gs[sum_def]
  >> `SUC n * (SUC n + 1) = (SUC n + 1) + n * (SUC n + 1)`
    by gs[MULT_CLAUSES]
  >> qspec_then `2` mp_tac ADD_DIV_ADD_DIV
  >> impl_tac >- gs[]
  >> disch_then (once_rewrite_tac o single o GSYM)
  >> `SUC n * 2 + n * (n + 1) = SUC n * (SUC n + 1)` suffices_by gs[]
  >> `SUC n * 2 = SUC n + SUC n` by gs[]
  >> `n * (n + 1) = SUC n * n` by gs[]
  >> `n * (SUC n + 1) = SUC n * n + n` by gs[]
  >> simp[]
QED
```

Figure 2.1: Example Theorem and Proof in HOL4

arithmetic, i.e., `SUC n` is the next natural number after `n`. The text between the `Proof` and the `QED` keyword contains the separate steps necessary to prove the theorem. In HOL4, proofs are performed using so-called tactics that change the current goal until it is equivalent to logical true ($\top$). Thus we call each of the steps of the proof a tactic, and they are combined using the "then" operator (`>>`) to form the overall proof.

In summary, the proof in Figure 2.1 proceeds by induction (`Induct_on `n``), and then performs simplification steps about which we will not go into detail here. In fact, throughout the thesis we usually only show theorem statements and leave the exact proof-steps to the formal developments.

2.2. Compiler Verification

The key expectation we have when running a compiler is that the compiler faithfully translates the source program into machine language and will not introduce new behaviors. Overall, for a particular compiler `cc`, compiler verification rigorously proves this property in an ITP as a theorem relating the behavior of the source language of `cc` to the behavior of the produced machine language. Compiler verification proves the theorem

Theorem 2.1 (`cc` correct)**.**

$$\texttt{p} \Downarrow^s r \Leftrightarrow \texttt{cc (p)} \Downarrow^m r$$

where `p` is a source language program, $\Downarrow^s$ is the semantics of the source language, and r the result of evaluating `p` under the semantics $\Downarrow^s$. Further, `cc (p)` is the machine program obtain from running the compiler `cc` on program `p`, $\Downarrow^m$ is the semantics of the machine language, and r the result. It is necessary to formally specificy the semantics of both source and target programs inside the ITP to have a formal way of reasoning about the behaviors of both program versions.

Both CompCert and CakeML have deterministic source languages [73, 113]. To reduce the complexity of proving Theorem 2.1, it is common to have $\Downarrow^s$, and $\Downarrow^m$ be deterministic languages, and prove explicitly the implication

Theorem 2.2 (`cc` forward simulation)**.**

$$\texttt{p} \Downarrow^s r \Rightarrow \texttt{cc (p)} \Downarrow^m r$$

This implication is called a *forward simulation* as it proves a simulation following the direction of the compiler, i.e., going from a property of the source program to a property of the machine code produced by the compiler. The inverse direction, a so-called *backwards simulation*, is in general more complicated to prove [73], but if the semantics are deterministic, it is established as a simple corollary from determinism and Theorem 2.2.

In this thesis, we base our work on the CakeML compiler, and therefore we give a more thorough introduction to its key ideas. CakeML is implemented and verified inside the HOL4 ITP, and compiles to machine code for x86, RISC-V, ARMv7, ARMv8, MIPS, and the verified Silver processor [78] from a dialect of Standard ML [88], which we refer to as CakeML source.

Just like any other compiler, CakeML compiles programs by reading them from a file, and parsing them into their source-code representation. In addition to this classic method of compiling programs, the CakeML compiler provides an alternative way of compiling to machine code through its proof-producing synthesis [2]. The proof-producing synthesis translates HOL4 functions into CakeML source programs, with an equivalence proof. These programs can then be compiled into machine code with the compiler.

Looking back at our general compiler correctness theorem (Theorem 2.1), machine code compiled with CakeML has the same properties as the source code. This allows for a very neat setting, which is not possible when using an unverified compiler, where we prove properties about the behavior of the source program, e.g., the program never

crashes when run with a certain kind of inputs, and the compiler proves for us that the machine code has the exact same behavior. This property of verified compilers is key to making verification of machine code implementations easier. To this end, CakeML provides so-called characteristic formulae [104] (CF) to prove Hoare-like specifications of CakeML source programs encoded in HOL4. Similar to other verification frameworks like, e.g., VST [6] and Iris [63], CF comes with a library of HOL4 tactics to prove triples of the form $< P,\mathtt{s},Q >$ where P is a pre-, and Q a post-condition for program `s`.

Throughout this thesis, we will refer to the trio consisting of the CakeML compiler, the proof-producing synthesis, and the characteristic formulae as the *CakeML ecosystem*.

2.3. IEEE-754 Floating-Point Arithmetic

Floating-point arithmetic is standardized in IEEE-754 [58]. The standard defines that a floating-point number is described by a triple (s, m, e), representing the fraction $-1^s * m * 2^{e-p}$. In hardware, s is a 1-bit word, representing the sign, word m is the so-called mantissa, and word e the biased exponent. Exponent e is said to be biased as it is offset by the constant p. The number of bits of m and e, and the value of p differ based on the precision of the target format. IEEE-754 defines single precision (m: 24 bits, e: 8 bits, p: 127) and double precision (m: 53 bits, e: 11 bits, p: 1023) representations. In hardware, one bit of the mantissa is not stored explicitly, but determined by the value of e. If the exponent is 0, this so-called implicit bit is 0, otherwise it is 1. Floating-point numbers where the implicit bit and the exponent are 0 are called sub-normal numbers, all other numbers are called normal numbers.

According to IEEE-754, each floating-point operation must be computed in hardware as if it was computed using real numbers and then rounded to the target representation. IEEE-754 defines different rounding modes that determine how a floating-point representation for a real number is found. The most commonly used is round to nearest with ties to even. We use a $\tilde{x}_p$ to denote real-number x rounded to precision p, and $a\ \tilde{\circ}_p\ b$ to denote the real-number binary operation $\circ$ applied to the arguments and the result rounded to precision p. According to IEEE-754, a binary floating-point operation performed in round to nearest, ties-to-even can be abstracted as

$$a\ \tilde{\circ}_p\ b = (a \circ b) * (1 + \delta) + \gamma \quad \text{where } |\delta| \leq \varepsilon_p \text{ and } |\gamma| \leq \zeta_p \tag{2.1}$$

The constants ε_p and ζ_p are defined for each and every floating-point type in IEEE-754. If the resulting number is a normal floating-point number, ζ_p is 0, and for subnormal numbers, ε_p is 0. The perturbations by δ and γ model the rounding to a floating-point

number. While there are some cases where the result of rounding can be computed exactly, their number is negligible in comparison to those that have to be rounded, leading to *roundoff errors*.

The IEEE-754 standard also defines five different exceptions that may occur during computations:

- `Invalid`: a mathematically undefined operation is performed (e.g., square-root of a negative number)
- `Div-0`: a division by zero occured
- `Overflow`: the rounded result value cannot be represented in the target format
- `Underflow`: the rounded result is a sub-normal number
- `Inexact`: the rounded result is different from the exact real-number result

A computation that raises an `Invalid` returns a so-called *Not-a-number special-value* (NaN), whereas `Div-0` and `Overflow` return one of $\{\infty, -\infty\}$. Generally, evaluation is resumed if one of these exceptional values occurs, but they can lead to unexpected computation results[2]. The latter two exceptions (`Underflow` and `Inexact`) do not return exceptional values and act as mere signals to the programmer.

2.4. Analysis of Finite-Precision Computations

We have seen that roundoff errors are unavoidable for floating-point arithmetic, and that exceptional values should be avoided. In general, this is true for any datatype used to represent real numbers, and these are commonly called finite-precision computations. In general, analysis of finite-precision computations tries to statically determine how large the roundoff error of a computation may get. To this end, a roundoff error analysis computes an upper bound to

$$\max_x |f(x) - \tilde{f}(\tilde{x})| \tag{2.2}$$

where $f(x)$ is a real-numbered function, and $\tilde{f}(\tilde{x})$ its finite-precision counterpart. Clearly, if x were unbounded, the roundoff error would become infinity. Thus roundoff error analysis requires knowing range bounds for input variables to compute non-trivial bounds. Further, if an exceptional value occurs, the value of $\tilde{f}(\tilde{x})$ becomes undefined and the roundoff error is infinite. Therefore roundoff error analysis simultaneously has to check for exceptional values while computing roundoff errors.

[2]The `NaN` value for example is not reflexive, i.e., the Boolean check `NaN = NaN` always returns false.

Error analysis is still an ongoing research topic, with various tools and different tradeoffs available. So far, the tools are limited to either straight-line kernels [37, 111, 34], or they provide limited support for conditionals and loops [120]. Here, we focus on the certified floating-point dataflow analysis tool FloVer [12] as its implementation is available in the HOL4 theorem prover.

In its initial version, FloVer was designed as an external validation tool for unverified roundoff error analysis. We have since extended it with a roundoff error analysis whose result is automatically validated. Our explanation focuses on how FloVer computes upper bounds on roundoff errors but applies to all other tools that use a forward dataflow analysis as well. FloVer's input programs are straight-line numerical kernels consisting of constants, variables, unary negation, binary $+, -, *, /$, let-bindings and so-called fused-multiply-add instructions. A fused-multiply-add (`fma`) is a ternary operation, and `fma(a,b,c)` computes a locally more accurate version of `(a * b) + c` by only rounding the final result of the computation. Orthogonal to the supported operations, FloVer analyzes so-called mixed-precision programs, where different parts of the program are executed with different finite-precisions. As an additional finite-precision, FloVer supports analysis of fixed-point arithmetic through its generalized error computations.

For a single finite-precision operation $\tilde{\circ}_p$, FloVer computes an upper bound to Equation 2.2:

$$\max_{x \in P} |(a \circ b) - (\tilde{a} \;\tilde{\circ}_p\; \tilde{b})|$$

which FloVer translates into

$$\max_{x \in P} |((a \circ b) - (\tilde{a} \circ \tilde{b})) + \text{error}(a \;\tilde{\circ}\; b, p)|$$

The left-hand-side $((a \circ b) - (\tilde{a} \circ \tilde{b}))$ is called the propagation error, as it is the error accumulated in the arguments to $\circ$, and function error on the right-hand-side computes the newly contributed error of $\circ$. Here, p is the finite-precision type of operation $\tilde{a} \;\tilde{\circ}_p\; \tilde{b}$. For a floating-point precision p, function error computes an upper bound using Equation 2.1. To support fixed-point arithmetic, FloVer generalizes Equation 2.1 in its generalized error model. For the purpose of this thesis, we limit our explanations to the analysis of floating-point arithmetic.

In Equation 2.1, the roundoff error depends on both the real-numbered range of the arguments, and the error contributed by the operation. Thus FloVer splits roundoff error analysis into a real-numbered range analysis, and a roundoff error analysis based on these ranges. What is more, fixed-point arithmetic requires real-numbered ranges to check for overflows. Both analyses are implemented and verified for interval arithmetic [91].

Verifying Error Analysis Results. FloVer takes as input a certificate containing the analyzed real-number function f, a mixed-precision type assignment Γ, a real-number range analysis result $\mathcal{R}$, and an implementation error analysis result $\mathcal{E}$. Given a certificate, FloVer checks for the following points of failure in the analysis result:

- *wrong mixed-precision assignment* (ME): As the kernel uses different finite-precision arithmetics, the assigned types can be incomptable, i.e., a computation with 32-bit floating-point values may use a 64-bit floating-point value.
- *real-numbered division-by-zero* (R0): The real-number program may contain a potential division-by-zero. For soundness of the analysis we rule out these exceptional cases.
- *finite-precision division-by-zero* (F0): Similarly, due to implementatiton errors, the overall error may become large enough that while the real-number execution is fine, its finite-precision counterpart runs into a division-by-zero exception.
- *finite-precision overflow* (F∞): The real-number program can be run fine, but the values become too large to be representable in a chosen finite-precision, which leads to runtime overflow errors.
- *real-numbered ranges are unsound* (RA): The real-numbered range analysis may fail to compute sound upper bounds on the actual ranges for a subexpression of the real-numbered program.
- *implementation error estimate is unsound* (EA): Similar to above, the implementation error analysis may have failed to compute sound upper bounds on the roundoff error for a subexpression of the program.

FloVer rules out each of these failures by checking a correctness certificate produced by its unverified analysis.

In FloVer, errors *RA* and *EA* are checked by recomputing the analysis results $\mathcal{R}$ and $\mathcal{E}$ inside the logic of HOL4 using interval arithmetic. FloVer checks that *ME* does not occur via a simple type-inference algorithm. The type inference checks that whenever an arithmetic operation is performed on a mixed-precision expression, that the target precision is at least as accurate as the argument precisions. Errors *R0*, *F0*, and *F*∞ are checked in FloVer by the real-numbered range analysis, and the roundoff error analysis respectively.

FloVer's rigorous soundness proof shows once and for all that if a certificate is checked correctly, no failure was made by the unverified roundoff error analysis, i.e., no error

can occur in the real-number program, no exceptional value can occur at runtime for the finite-precision counterpart, and both the real-number ranges $\mathcal{R}$ and the roundoff error analysis $\mathcal{E}$ are sound upper bounds to the ranges and errors inferred by FloVer. As reference for soundness, we give a semantics based on Equation 2.1 and its dual for fixed-point arithmetic. In a separate proof, FloVer connects its floating-point semantics to a HOL4 formalization of IEEE-754 floating-point semantics [49] and we show that roundoff error bounds validated by FloVer are valid for IEEE-754 floating-point semantics as well.

CHAPTER 3

Dandelion
Certified Approximations of Elementary Functions

This chapter is based on our paper titled *Dandelion: Certified Approximations of Elementary Functions*, which has been published at ITP'22. The work was done in collaboration with Mohit Tekriwal, Eva Darulova, Anastasia Volkova, and Jean-Baptiste Jeannin. Mohit Tekriwal has contributed proofs to the formal development under my supervision, and Eva Darulova, Anastasia Volkova and Jean-Baptise Jeannin have provided feedback on revisions of the writing.

- Section 3.1 is based on the ITP submission with minor rewordings.
- Section 3.2 contains the discussion of Harrison's preliminary work from the ITP submission; the background on polynomial approximations is new.
- Section 3.3, Section 3.4, and Section 3.5 are based on the ITP submission with minor rewordings and reformulations for clarity.
- Section 3.6 is an extended version of the ITP submission and contains the original description of the CakeML binary and an additional part about speeding up HOL4 computations with Karatsuba multiplications.
- Section 3.7, and Section 3.8 are based on the ITP submission with minor rewordings and reformulations for clarity.
- Section 3.9 is new.

3.1. Introduction

Elementary functions, like sin and exp, are defined in real-numbers using infinite sums and irrational constants (e.g., π). Consequently they cannot be directly translated

into finite-precision arithmetic. Some compilers provide general-purpose floating-point implementations of elementary functions. These general-purpose implementation usually are software-defined approximations, and because of the required generality they can become inefficient. In resource-constrained settings, as they occur in embedded systems, real-numbers must be implemented using fixed-point arithmetic, where often no general-purpose math-library is available. If no library function is available, or the implementation is not useful in the context at hand, developers opt for custom polynomial approximations.

Coming up with a custom polynomial approximation manually is tricky. For once, any polynomial approximation necessarily cannot compute the exact result of an elementary function, and this difference in results is called the *approximation error*:

$$\forall x \in P, |f(x) - p(x)| \leq \delta \tag{3.1}$$

Here, f is the elementary function approximated with polynomial p, and δ is the approximation error. In general, polynomial approximations are computed for a particular range of inputs, and therefore, Equation 3.1 constrains the input x to be from a precondition P.

As an alternative to manual approximations, approximation tools like, e.g., metalibm [69] and Sollya [25], automatically compute a custom polynomial approximation and an upper bound to the approximation error. Different approaches to compute polynomial approximations and bounds on the approximation error exist, and the known most-accurate are so-called Remez-like algorithms [100]. In a high-assurance setting, as is common when dealing with verified compilers, a key question is whether Equation 3.1 is true, i.e., whether the approximation error δ computed by a Remez-like algorithm is a true upper bound to the difference between the elementary function f, and its polynomial approximation p[1].

This chapter presents *Dandelion*, an automatic certificate checker for results of Remez-like algorithms. Dandelion is implemented and fully-verified in the HOL4 theorem prover [67]. A certificate for Dandelion contains an elementary function, its polynomial approximation, a precondition, and an approximation error, computed using a Remez-like algorithm. We prove once and forall the correctness theorem that if Dandelion succeeds to validate a certificate, Equation 3.1 is true.

Dandelion's certificates are minimal: they only contain the input and output of a Remez-like algorithm. Further, Dandelion certifies the known most-accurate approximations computed using a Remez-like algorithms. Previous approaches focused on manual

[1] Muller, Jean-Michel [94](page 52) warns:"[...] even if the outlines of the [Remez] algorithm are reasonably simple, making sure that an implementation will always return a valid result is sometimes quite difficult[...]".

proofs [53]; verify only Chebyshev approximations [20], which are not as accurate as those computed by Remez-like algorithms; or base their verification mainly on interval arithmetic [82]. Dandelion is the first tool that automates the approach of Harrison [53], and thus the key challenge that Dandelion solves is *automation*; Dandelion is based on polynomial zero finding and checks certificates fully-automatically — there is no user interaction required.

It may seem that one should rather verify an implementation of a Remez-like algorithm instead of verifying its results. However, a correctness proof for a particular implementation is always specific to implementation choices, and thus, such a proof usually does not scale to other implementations. Actually, the proof would need to be redone (or at least repaired) with every implementation change. In contrast, as Dandelion verifies results of Remez-like algorithms, implementation choices do not matter, which makes Dandelion widely applicable.

In previous work, Harrison manually verified an implementation of the exponential function in the HOL-Light theorem prover [53]. So far, the technique used by Harrison, while general, has not been automated. Dandelion reuses the key ideas of the approach, providing full automation.

One may think that automating the manual approach of Harrison is simple. However, Dandelion solves two key technical challenges: *Speed* — It is generally known that computations in ITPs are slower than unverified computations. This is especially true when dealing with real-numbered computations. *Computability* — Harrison's manual proof uses non-computable functions to formalize key concepts.

We solve the problem of speeding up the computations by extracting a verified binary with the CakeML compiler [113]. The verified binary provides the same correctness guarantees as our in-logic implementation of Dandelion, but computes results significantly faster: Dandelion's binary checks certificates on average within 6 minutes. We solve the problem of non-computable definitions by defining computable versions, and proving them equivalent to their non-computable counterparts.

Dandelion certifies results of any implementation of a Remez-like algorithm. We evaluate Dandelion on polynomial approximations generated from benchmarks from FPBench [31] and the work by Izycheva *et al.* [60]. The evaluation shows that Dandelion is fast, and that approximation errors of an elementary function f and its polynomial approximation p are usually in the same order of magnitude as the infinity norm ($\max_{x \in P} |f(x) - p(x)|$). We encode the original proof-goal of Harrison as a certificate for Dandelion and the proof is reduced to a single line of code in Dandelion.

Contributions. In summary, this chapter contributes:

- a HOL4 implementation of Dandelion, a verified certificate checker for polynomial approximations;
- a verified binary extracted using CakeML to make certificate checking fast; and
- an evaluation of Dandelion's performance on a set of benchmarks, comparing it with the state-of-the-art.

3.2. Manual Verification of Polynomial Approximations

Before giving a high-level overview of Dandelion, and explaining the detais behind our approach, we quickly review how polynomial approximations are computed, and paraphrase the key points of Harrison's manual proof.

3.2.1. Polynomial Approximations

In mathematics, a polynomial approximation replaces an elementary function, like sin and exp, with a polynomial, i.e., a finite series that computes similar results up-to a certain degree of accuracy. The most well-known, and simplest method for coming up with polynomial approximations is Taylor's theorem:

> **Theorem 3.1** (Taylor's theorem)**.** If n is a natural number, f a n-times differentiable, univariate function, and c a real-number, then there exists a univariate function r, such that
>
> $$f(x) = f(c) + \sum_{i=1}^{n} \frac{f^{(i)}(c) \times (x-c)^i}{i!} + r(x)$$

In Theorem 3.1, $f^{(i)}$ is the i-th derivative of f, and $i!$ is the factorial of i. The term $f(c) + \sum_{i=1}^{n} \frac{f^{(i)}(c) \times (x-c)^i}{i!}$ from Theorem 3.1 is commonly referred to as the degree n *Taylor polynomial* of f at c. Function r is called the remainder of the approximation and gives a bound on the approximation error[2]. In Dandelion, we use a special case of Taylor's theorem where $c = 0$. This special case is commonly referred to as a *McLaurin series*.

The book by Muller [94] provides a general overview of different approximation techniques. Here we focus on Remez-like approximation algorithms as they are known to compute most accurate results and thus are the target of Dandelions certificate checking.

[2] Some presentations of Taylor's theorem give more explicit characterizations of the remainder term, but we opt for a simpler presentation here.

3.2.2. Remez Algorithm

In contrast to the simple statement of Theorem 3.1 from which Taylor polynomials are computed, Remez algorithm is an iterative approximation algorithm. Our explanation of Remez algorithm is based on the one given by Muller ([94], p. 53). The inputs to the algorithm are a real-valued function f that is to be approximated, and a set of points $x_0, \ldots, x_n$, where $x_i \leq x_{i+1}$. The algorithm then tries to find an approximating polynomial p where $p(x_i) = f(x_i)$. To this end, the algorithm solves the equation system

$$\begin{pmatrix} 1 & x_0^1 & \ldots & x_0^n \\ \ldots & & & \\ 1 & x_n^1 & \ldots & x_n^n \end{pmatrix} * \begin{pmatrix} p_0 \\ \ldots \\ p_n \end{pmatrix} - \begin{pmatrix} f(x_0) \\ \ldots \\ f(x_n) \end{pmatrix} = \begin{pmatrix} +\varepsilon \\ \ldots \\ (-1)^{n+1}\varepsilon \end{pmatrix}$$

for the unknown values of $p_0, \ldots, p_n$ and ε. The solution is a polynomial $p(x) = p_0 + p_1 \times x + p_2 \times x^2 \ldots$, and an error ε. The error ε is part of the solution of the equation system but may not yet be an accurate bound on the approximation error. Therefore, Remez algorithm computes a bound on the approximation error δ by finding the extremas of $|f(x) - p(x)|$. To decide whether the polynomial p is a good enough approximation or whether to continue the iteration, the algorithm checks whether the error polynomial $|f(x) - p(x)|$ equi-oscillates between δ, i.e., whether the extremal values of $|f(x) - p(x)|$ are δ. If this is the case, the polynomial p and the error δ are returned, otherwise iteration continues with the a new set of points, where $x_0, \ldots, x_n$ are the extremal points of $|f(x) - p(x)|$.

The overall algorithm may seem simple, but we highlight two key challenges in verifying an implementation of a Remez-like algorithm: First, the algorithm is iterative and only returns a valid result in the final step, which makes verifying the iterative process hard. Second, while we have described the algorithm in terms of real-numbers, its implementation is usually done in some finite-precision, so a correctness argument must also relate finite-precision results to their real-number counterparts.

3.2.3. Manual Proof by Harrison

Coming back to verification of polynomial approximations, Harrison [53] has manually verified an approximation p of the exponential function, which was presented by Tang [114], showing that:

$$\forall x.x \in [-0.010831, 0.010831] \Rightarrow |((e^x - 1) - p(x)| \leq 2^{-33.2}$$

The manual verification by Harrison is split into two steps. First, Harrison simplifies the overall proof goal to a proof about polynomials, by replacing $e^x - 1$ with a high-accuracy truncated Taylor series q. By truncating the series after the 7th term, the approximation error of the series becomes 2^{-58} and the overall proof-goal is reduced from $|((e^x - 1) - p(x)| \leq 2^{-33.2}$ with the triangle inequality to

$$|q(x) - p(x)| \leq 2^{-33.2} - 2^{-58} \tag{3.2}$$

The difference between $q(x)$ and $p(x)$ itself is a polynomial $h(x)$, and thus this first step reduces the overall proof goal to proving an upper bound on the polynomial h. A key observation by Harrison is that h represents the difference between two polynomial approximations, and consequently, the points where h attains its maximum value (i.e., its extremal points) are those where the approximation error is the largest. To prove Equation 3.2, it thus suffices to reason about the extremal points of h.

To reason about the extremal points of h, Harrison proved two well-known mathematical theorems in HOL-Light, which we restate here for completeness. The first theorem proves that a real-valued polynomial p defined on the closed interval $[a, b]$ attains its extremal values at the outer points a and b and the points where the first derivative is zero:

Theorem 3.2.
Let p be a differentiable, univariate polynomial defined on $[a, b]$, and M a real number, then

$$\begin{aligned}
&|\, p(a) \,| \leq M \ \wedge \ |\, p(b) \,| \leq M \ \wedge \\
&(\forall x.\ a \leq x \leq b \ \wedge \ p'(x) = 0 \ \Rightarrow \ |\, p(x) \,| \leq M) \ \Rightarrow \\
&(\forall x.\ a \leq x \leq b \ \Rightarrow |\, p(x) \,| \leq M)
\end{aligned}$$

The second theorem proves that the exact number of zeros of a polynomial is computed from its so-called *Sturm sequence*. The Sturm sequence ss of polynomial p is defined recursively as

$$ss_0 = p \qquad ss_1 = p' \qquad ss_{i+1} = -\,\mathtt{rem}\,(ss_{i-1}, ss_i)$$

where `rem` computes the remainder of the polynomial division $\frac{ss_{i-1}}{ss_i}$. Computation stops once the remainder becomes the constant 0 polynomial. The second theorem proven by Harrison is called *Sturm's theorem*:

```
(* (1.0 <= radius <= 10.0) ∧ (0.0 <= theta <= 360.0) *)
fun polToCart_x(radius: real, theta: real): real = let
  val pi = 3.14159265359
  val radiant = (theta * (pi / 180.0))
  val cos_rad = cos (radiant)
  in (radius * cos_rad) end
```

Figure 3.1: Example program computing the x-coordinate when translating polar to cartesian coordinates

Theorem 3.3 (Sturm's theorem)**.**
Let p be a differentiable, univariate polynomial defined on $[a, b]$, and ss the sturm sequence for p. If p has non-zero values on both a and b, and its derivative is not the constant zero function, then the set of zeros of p has size $V(a, ss) - V(b, ss)$.

Function $V(a, ss)$ in Theorem 3.3 computes the number of sign changes when evaluating the polynomials in the list ss on value a.

To prove the final inequality, Harrison computes unverified guesses for both the Sturm sequence of $h'(x)$ and the zeros of $h'(x)$ using Maple, and manually validates them in HOL4 using Theorem 3.3. By knowing the number of zeros, and their values, Harrison then provably derives an upper bound on the extremal values of polynomial h using Theorem 3.2, which in turn validates the approximation error bound.

3.3. Overview

Before explaining the details of Dandelion, we give a high-level overview of its key contributions and how Dandelion automates the proof by Harrison. We base our explanations on the motivating example from Chapter 1, which we repeat in Figure 3.1.

The example code converts polar to cartesian coordinates, and returns the resulting x component. This code could for example be part of an autonomous car or a drone, and inaccuracies in the conversion of coordinates may have catastrophic effects [95]. In an embedded setting the code from Figure 3.1 could be implemented in fixed-point arithmetic, which, however, does not come with standard and efficient library implementations of elementary functions [60]. Hence, an engineer may approximate the function cos on line 8 in Figure 3.1. In Figure 3.2, function cos has been replaced by a custom polynomial

```
fun polToCart_x(radius: Fixed, theta: Fixed): Fixed = let
 val pi = 3.14159265359
 val radiant = (theta * (pi / 180.0))
 val cos_rad = (1.3056366443634033 +
  (radiant * (-1.2732393741607666 +
  (radiant * (0.2026423215866089 +
  (radiant * 3.3222216089257017e-09))))))
 in (radius * cos_rad) end
```

Figure 3.2: Example with polynomial approximation for cos

approximation, which can for instance be computed using the state-of-the-art synthesis tool Daisy [60][3].

Daisy internally calls a Remez-like algorithm to generate the polynomial approximation of cos, but the approximation algorithm and Daisy itself are not (formally) verified. To prove that the approximation is correct, we use Dandelion. As Dandelion is a certificate checker, it is straight-forward to extend Daisy with a function to generate the certificate shown in Figure 3.3. The certificate encodes the elementary function to be approximated (f), the polynomial approximation (p), the approximation error (ε), the input range for the approximation (I), and an additional parameter n which we explain later. Note that the input interval I recorded in the certificate captures the direct inputs to the elementary function cos and is thus different from the input interval in the comment of Figure 3.1, which captures inputs to the overall function `polToCart_x`.

Using its verified binary, Dandelion validates the example certificate in 31 seconds and proves the HOL4 theorem

Theorem 3.4.

$$\forall x.\ x \in I(x)\ \Rightarrow\ |\cos(x) - p(x)|\ \leq\ \varepsilon$$

If the approximation error had not been correct, the binary would emit an error message, explaining which part of the validation failed.

Going back to the example certificate in Figure 3.3, the certificate uses only a single elementary function (cos). In general, Dandelion supports more complicated elementary

[3]Daisy's input language is Scala. For consistency, we present all our examples as CakeML source code, but note that a straight-forward translation from CakeML source to Scala is easily implemented for the subset of the language that we use in our examples.

```
cos_cert = <|
 f := Fun Cos (Var "radiant"); n := 32;
 (* p (x) ~ 1.305 - 1.273 * x + 0.202 * x^2 + 3.322 * 10^-9 * x^3 *)
 p := [5476237/4194304; -5340353/4194304; 1699887/8388608;
     3740489/1125899906842624];
 (* ε ~ 0.306 *)
 ε := 766133524584849981160987377038947873961143126798 7/(25 * 10^48);
 (* ~ x in [0, 6.284] *)
 I := [("radiant", (0, 314159265359/50000000000))]; |>
```

Figure 3.3: Certificate for the approximation of cos in the example

function expressions, like $\exp(x \times \frac{1}{2})$, and $\sin(x-1)+\cos(x+1)$. Exactly as for Remez-like algorithms, Dandelion only requires the functions to be univariate, i.e., the certificate can only have a single free variable. Any approximation tool that can generate these certificates can be used to generate inputs for Dandelion, and Dandelion can be used independently of a particular approximation algorithm implementation.

The approach used by Dandelion follows the structure of the manual proof by Harrison [53] (Section 3.2.3 overviews the main theorems and ideas). The presented high-level approach is general, but a key challenge solved by Dandelion is to automate each step and extend them to more complex expressions. We explain the high-level ideas of how Dandelion automates Harrison's proof next.

As in Harrison's approach, Dandelion splits the proof into two parts. In the first phase, Dandelion replaces elementary functions in the certificate by high-accuracy approximations computed inside HOL4 to reduce the overall goal to reasoning about a polynomial. The second phase then proves that the approximation error is a correct upper bound on the extremal points of the resulting polynomial by finding zeros of the derivative and bounding the number of zeros with Sturm sequences. The key differences between Dandelion and the proof by Harrison is that Dandelion supports more elementary functions, i.e., $\exp$, $\sin$, $\cos$, $\ln$, and $\tan^{-1}$, and that its certificate checking is fully automated and does not require any user interaction or additional proofs. Figure 3.4 gives an overview of the automatic computations done by Dandelion and we explain them at a high-level for our running example from Figure 3.1.

To prove the overall correctness theorem (Theorem 3.4), Dandelion first computes a computable high-accuracy polynomial approximation for cos, denoted by `q`, using a

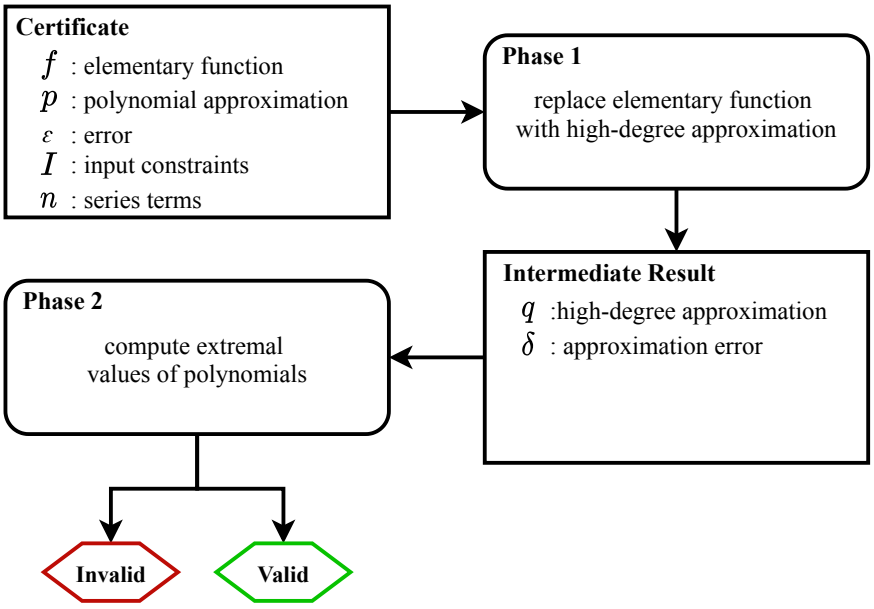

Figure 3.4: Overview of Dandelion toolchain

truncated Taylor series of degree n, with an approximation error of `3.77e-3`. Generally, the certificate parameter n defines the number of series terms computed for the truncated Taylor series in Dandelion. Exactly as for Harrison's manual proof, Dandelion proves an upper bound on the difference between q and the target polynomial approximation p (Equation 3.2). Coming up with a general approach for computing accurate truncated Taylor series of arbitrary elementary functions was a major challenge for Dandelion—we implemented a library of general purpose Taylor series for the supported elementary functions. Given a target degree, Dandelion automatically computes a polynomial implementation and proves an approximation error for the truncated series. We explain the first phase in more detail in Section 3.4.

In the second phase, Dandelion computes an upper bound on the polynomial $h(x) = q(x) - p(x)$ exactly like Harrison by reasoning about the zeros of its derivative h', using Sturm sequences to bound the number of zeros of h'. Based on the number of zeros, Dandelion uses an (unverified) oracle to automatically come up with a list of zeros of h'.

To prove the final bound on h, Dandelion checks for each zero of h' that the value of h at this point is smaller than or equal to the residual error $\varepsilon -$ `3.77e-3`. A key challenge of the checks in the second phase is that Harrison's definition of Sturm sequences is defined as a non-computable predicate, involving an existentially quantified definition of polynomial division. Dandelion implements a computable version of both polynomial

division and Sturm sequences, and relates them to Harrison's predicates with equivalence proofs. We explain how Dandelion computes the Sturm sequences automatically and how Dandelion estimates zeros of polynomials in more detail in Section 3.5.

Dandelion's computations heavily rely on real-number operations. To increase their performance we have implemented a function that reduces real-number multiplications to natural number multiplications and computes the result efficiently using Karatsuba multiplications. Further, computing the Sturm sequence is the most computationally expensive part of Dandelion, which we found to be impractical to do in logic. Thus we extract a verified binary using the CakeML compiler for the second phase of Dandelion only. For the extraction to work, we translated the HOL4 definitions of the second phase into CakeML source code via CakeML's proof-producing synthesis tool [2]. We explain the simplification of real-number multiplications and the extraction with CakeML in Section 3.6.

3.4. Automatic Computation of Truncated Taylor Series

As in Harrison's manual proof, the first phase of Dandelion replaces the elementary function in the certificate by a high-accuracy polynomial approximation. The crucial difference is that Dandelion automates all of the manual steps, which we explain in this section.

When Dandelion checks a certificate for function f with input range I, every occurence of an elementary function f is automatically replaced with a truncated Taylor series $t_{f,n}$. The parameter n of truncated Taylor series $t_{f,n}$ is part of the certificate and specifies the number of terms computed for the truncated series, for example, if n is 32 Dandelion truncates the Taylor series of f after the 32nd term. The final result of the phase is a high-accuracy polynomial approximation of function f, $q_{f,n}$, and an overall approximation error $\delta_{f,n}$. We implement the first phase in a HOL4 function `approxAsPoly` and prove soundness of `approxAsPoly` once and for all in HOL4:

Theorem 3.5 (First Phase Soundness)**.**

$$\texttt{approxAsPoly}\ (f,\ I,\ n) = \texttt{Some}(q_{f,n}, \delta_{f,n}) \Rightarrow$$
$$(\forall x.\ x \in I \Rightarrow |\, f(x) - q_{f,n}(x)\,| \leq \delta_{f,n})$$

The theorem shows that if `approxAsPoly` succeeds and returns $q_{f,n}$ and $\delta_{f,n}$, then the approximation error on input range I between $f(x)$ and $q_{f,n}(x)$ is upper bounded by $\delta_{f,n}$.

To explain how `approxAsPoly` is implemented, we take f to be a single elementary function, like sin and exp. In this first, simple case, the final polynomial approximation $q_{f,n}$ and the truncated Taylor series $t_{f,n}$ are the same. Afterwards, we extend function `approxAsPoly` to compute a single polynomial approximation of more complicated elementary function expressions like $\exp(x * \frac{1}{2})$. The more complicated step combines interval analysis and propagation of polynomial errors with the truncated Taylor series from the simple case. Throughout this section, we use f to refer to the elementary function from the certificate, $t_{f,n}$ as truncated Taylor series, $\delta_{t,n}$ as the approximation error of the series, $q_{f,n}$ as polynomial approximation, and $\delta_{f,n}$ as the overall approximation error.

3.4.1. Truncated Taylor Series for Single Elementary Functions

Overall, Dandelion automatically computes a truncated Taylor series for the elementary functions sin, cos, exp, $\tan^{-1}$, and ln [4]; the series expansions for exp and ln existed already in HOL4 prior to Dandelion and we port the series for $\tan^{-1}$ from HOL-Light. For sin and cos we prove series based on textbook descriptions. Both the truncated series $t_{f,n}$ and the approximation error $\delta_{t,n}$, depend on the approximated elementary function f, as well as the number of series terms n from the certificate. Formally, Dandelion once and for all proves a truncated Taylor series for each elementary function as a McLaurin series:

Theorem 3.6.

$$\forall x\ n.\ Pre(x) \Rightarrow f(x) = \sum_{i=0}^{n} (\frac{f^i\ 0}{i!} \times x^i) + \delta_{t,n}(x)$$

Here, f^i is the i-th derivative of f, and the approximation error $\delta_{f,n}(x)$ is soundly bounded from the remainder term of Taylor's theorem (Theorem 3.1) for input value x. Predicate Pre is a precondition constraining the values for which function f can be approximated by the truncated series. We show all of the theorems proven for Dandelion in Figure 3.5. The series for exp and ln are marked with a * because they are based on the HOL4 standard library proofs about exp and ln, but we extend them with a precondition to make the approximation error computable for Dandelion.

When approximating an elementary function f by its truncated series, function `approxAsPoly` always ensures that the precondition Pre is true based on the theorems in Figure 3.5: The series for sin and cos have no preconditions; the series for exp requires

[4] Dandelion currently does not support tan, as a straight-forward reduction to sin(x)/cos(x) did not work out.

Theorem 3.7 (sin McLaurin series)**.**

$$\forall\, x\ n.\ \top \Rightarrow \sin(x) = \sum_{i=0}^{n}\left(\frac{x^i}{!i} \times \texttt{if even } i \texttt{ then } 0 \texttt{ else } -1^{\frac{i-1}{2}}\right) + \frac{|x|^n}{!n}$$

Theorem 3.8 (cos McLaurin series)**.**

$$\forall\, x\ n.\ \top \Rightarrow \cos(x) = \sum_{i=0}^{n}\left(\frac{x^i}{!i} \times \texttt{if even } i \texttt{ then } -1^{\frac{i-1}{2}} \texttt{ else } 0\right) + \frac{|x|^n}{!n}$$

Theorem 3.9 (exp McLaurin Series*)**.**

$$\forall\, x\ n.\ 0 \le x \Rightarrow \exp(x) = \sum_{i=0}^{n}\left(\frac{x^i}{!i}\right) + \frac{\exp(x)}{!n} \times x^n$$

Theorem 3.10 (ln McLaurin Series*)**.**

$$\forall\, x\ n.\ 0 < x\ \wedge\ 0 < n \Rightarrow \ln(1+x) = \sum_{i=0}^{n}\left(-1^{i+1} \times \frac{x^i}{i}\right) + -1^{n+1} \times \frac{x^n}{n}$$

Theorem 3.11 ($\tan^{-1}$ McLaurin Series)**.**

$$\forall\, x\ n.\ |x| \le 1 \Rightarrow \left|\tan^{-1}(x) - \sum_{i=0}^{n}\left(\texttt{if even } i \texttt{ then } 0 \texttt{ else } -1^{\frac{i-1}{2\times i}} \times x^i\right)\right| \le \frac{|x|^n}{1-|x|}$$

Figure 3.5: McLaurin series proven for Dandelion

inputs to be non-negative; the series for ln requires arguments greater than 1; and the series for $\tan^{-1}$ requires arguments in $(-1, 1)$.

When checking a certificate, function `approxAsPoly` automatically computes an upper bound to the approximation error $\delta_{f,n}$. Function `approxAsPoly` also must return the McLaurin series as a polynomial following Harrison's formalization. This technicality is required because Dandelion's second phase reuses Harrison's formalization of polynomial

arithmetic to simplify proofs. To return an appropriate result in `approxAsPoly`, we prove once and for all that the truncated series from Theorem 3.6 can be implemented in Harrison's polynomial datatype:

Theorem 3.12.

$$\forall\, n.\ \sum_{i=0}^{n}(\frac{f^i\,0}{i!} * x^i) = t_{f,n}(x)$$

Theorem 3.12 proves that polynomial $t_{f,n}$ implements the truncated Taylor series of f on the left-hand side for an arbitrary number of approximation steps n. We prove versions of Theorem 3.12 for each elementary function supported by Dandelion. Putting it all together, in the simple case, where f is a single elementary function, the proof of First Phase Soundness (Theorem 3.5) is a simple combination of the Theorems in Figure 3.5 and Theorem 3.12.

3.4.2. Approximations of More Complicated Expressions

If we zoom out to the big picture, Dandelion so far only approximates a single elementary function with `approxAsPoly`. In contrast, Remez-like algorithms generally can compute an approximation for a compound function or an expression, as long as it remains univariate. Compared to approximating individual functions, e.g., exp, an overall expression approximation can be more accurate, sometimes avoiding undesirable effects such as cancellation. Hence, Dandelion should also be able to certify those. We explain how Dandelion approximates more complicated elementary expressions with truncated Taylor series using $\exp(y \times \frac{1}{2}) - 1$ on the interval $[1, 2]$ as an example[5].

From Theorems 3.9 and 3.12 Dandelion knows how to automatically compute a polynomial approximation $t_{\exp,n}$ and an approximation error $\delta_{\exp,n}(x)$ for the exponential function for a given input range on the argument. In our example, the input argument is $y \times \frac{1}{2}$, and thus the value of $\delta_{\exp,n}(x)$ depends on the range of this expression, which Dandelion computes automatically using interval arithmetic [91].

As interval analysis, we reuse the formalization of interval arithmetic from FloVer [12] (Section 2.4) and extend it with range bounds for elementary functions. For our example Dandelion also needs to compute a range bound for $\exp(y \times \frac{1}{2})$. In general, because elementary functions are defined non-computably in HOL4, we have to rely on a trick to compute interval bounds. To compute interval bounds for elementary functions, Dandelion

[5]Currently, Dandelion does not generally support divisions, hence we explicitly represent $\frac{y}{2}$ as $y \times \frac{1}{2}$.

reuses our formalized truncated Taylor series. From Theorem 3.6, we know for a single elementary function f that

$$f(x) = t_{f,n}(x) + \delta_{f,n} \tag{3.3}$$

From this inequality, we derive a bound on $f(x)$ in the interval $[a, b]$

$$t_{f,n}(a) \leq f(x) \leq t_{f,n}(b) + \delta_{f,n} \tag{3.4}$$

Note that Equation 3.4 does not need to subtract $\delta_{f,n}$ for the lower bound as Equation 3.3 is an equality and $\delta_{f,n}$ is usually positive. Equation 3.4 holds for monotone f only, and thus we cannot apply it to sin and cos as they are periodic. For both functions, interval analysis returns the closed interval $[-1, 1]$. For $\tan^{-1}$, Dandelion also relies on the coarse upper bound of $[-2, 2]$, because we can only inaccurately bound π in HOL4, as we will explain in an instant. Dandelion's interval analysis is proven sound once and for all in HOL4.

With the interval analysis, we can soundly compute a polynomial approximation for exp, $t_{\exp,n}$ on the range of $y \times \frac{1}{2}$. Dandelion automatically composes the polynomial $y \times \frac{1}{2}$ with $t_{\exp,n}$ to obtain a polynomial $q_{\exp(y\times\frac{1}{2}),n}$ with approximation error $\delta_{\exp(y\times\frac{1}{2}),n}$. However, we still need to come up with a polynomial approximation p and an approximation error for the full function $\exp(y \times \frac{1}{2}) - 1$. In our example, Dandelion treats the constant 1 as a polynomial returning 1, and automatically computes the polynomial difference of $q_{\exp(\ldots),n}$ and $q_{1,n}$. The global approximation error δ for the difference of $q_{\exp(\ldots),n}$ and $q_{1,n}$ depends on the approximation errors accumulated in both polynomials. In a final step, Dandelion automatically computes an upper bound on the global approximation error by propagating accumulated errors through the subtraction operation.

Generally, Dandelion implements an automatic approximation error analysis inside function `approxAsPoly` that propagates accumulated approximation errors. The propagation is implemented for basic arithmetic and elementary functions to support, e.g., expressions like $\exp(x) + \sin(x - 1)$. For basic arithmetic, the propagation, while conceptually different from roundoff errors follows the same rules and intuitions as the analysis implemented in FloVer. In fact, the correctness proof of error propagation for basic arithmetic reuses theorems proven for FloVer.

Computing Propagation Errors for Elementary Functions. When propagating accumulated approximation errors through calls to exp and ln we simply exploit their monotonicity properties and some basic trigonometric identities to compute a bound, e.g., $\exp(x+y) = \exp(x) \times \exp(y)$. For $\tan^{-1}$ we currently do not propagate errors in Dandelion

and raise an exception, simply because of a lack of supporting trigonometric identities verified in HOL4. Both sin and cos require extra care as they are not monotonically increasing, but periodic. To accurately propagate approximation errors through sin and cos, our soundness proof relies on properties of $\sin(x)$ and $\cos(x)$ where x is in the interval $[0, \frac{\pi}{2}]$ and x is an accumulated approximation error. This does not pose a true limitation of Dandelion as errors larger than $\pi/2$ would anyway be undesirable and impractical. For the correctness proof of `approxAsPoly` (Theorem 3.5) to succeed, however, Dandelion must automatically prove that accumulated errors are less than or equal to $\pi/2$. This poses a challenge as in HOL4 π is defined non-computably using Hilbert's choice operator: if $0 \leq x \leq 2$ and $\cos(x) = 0$, then π is $2 \times x$. To solve this problem, we reuse the truncated Taylor series of $\tan^{-1}$ and the fact that $\tan^{-1}(1) = \pi/4$ to compute a lower bound r in HOL4, where $r \leq \pi$. At certificate checking time, when propagating the error $\delta_{f,n}$ through sin and cos, Dandelion checks $\delta_{f,n} \leq \frac{r}{2}$, which by transitivity proves that $\delta_{f,n} \leq \frac{\pi}{2}$.

3.4.3. Extending Dandelion's First Phase

All of the truncated Taylor series proven in Dandelion are for single applications of an elementary function. For a particular application it may be beneficial to add special cases to compute a single, more accurate, truncated Taylor series of an elementary function like $\exp(\sin(x))$ instead of computing a truncated series for each function separately.

In Harrison's original proof this would require manually redoing a large chunk of the proof work whereas for Dandelion such an extension amounts to 4 steps: Proving the truncated Taylor series as in Figure 3.5, implementing and proving correct the polynomial $t_{f,n}$ as in Theorem 3.12, extending `approxAsPoly` with the special case for the new elementary function, and finally using the theorems proven for the first two steps to extend First Phase Soundness (Theorem 3.5) with a correctness proof for the new case. Complexity of the proofs only depends on the complexity of the series approximation. Dandelion then automatically uses the new series approximation whenever the approximated function is encountered in a certificate, and the global soundness result of Dandelion still holds without any required changes. The second phase directly benefits from adding additional approximations as more accurate Taylor series decrease the approximation error of the first phase.

3.5. Validating Polynomial Errors

For a certificate consisting of an elementary function f, polynomial approximation p, approximation error ε, input constraints I, and truncation steps n, the first phase of Dandelion computes a truncated Taylor series $q_{f,n}$ and an approximation error $\delta_{f,n}$, which is sound by Theorem 3.5. Both $q_{f,n}$ and p are polynomials, and following Harrison's terminology, we refer to their difference $q_{f,n}(x) - p(x)$ as the error polynomial $h(x)$. In the second phase, Dandelion's function `validateZerosLeqErr` automatically finds an upper bound to the extremal values of $h(x)$ and compares this upper bound to the residual approximation error $\varepsilon - \delta_{f,n}$, which we refer to as γ. We prove soundness of the second phase once and for all as a HOL4 theorem:

Theorem 3.13 (Second Phase Soundness)**.**

$$\texttt{validateZerosLeqErr}(q_{f,n}, \texttt{p}, \texttt{I}, \gamma) = \top \Rightarrow$$
$$\forall x.\ x \in \texttt{I}(x) \Rightarrow |\, q_{f,n}(x) - \texttt{p}(x) \,| \leq \gamma$$

Before going into the details of how function `validateZerosLeqErr` automatically validates the residual error γ, we quickly recall the key real analysis result which we rely on for its implementation: on a closed interval $[a, b]$, a differentiable polynomial p can reach its extremal values at the outer points $p(a)$, $p(b)$, and the zeros of p's first derivative p' (Theorem 3.2). To find the extremal values of $h(x)$, function `validateZerosLeqErr` thus needs to automatically find *all* zeros of $h'(x)$.

Function `validateZerosLeqErr` splits finding the extremal values and validating γ into three automated steps:

1. Compute the number of zeros using Sturm's theorem (Theorem 3.3 in Section 3.2)
2. Validate a guess of the zeros computed by an unverified, external oracle
3. Compute an upper bound on extremal values (using the validated zeros) and compare with γ

Conceptually, the second phase automates the main part of Harrison's manual proof, and the key step is computing Sturm sequences automatically in the first step. Next, we explain the ideas behind automating each of the steps.

3.5.1. Bounding the Number of Zeros of a Polynomial

Dandelion bounds the number of zeros of a polynomial using Sturm's theorem (Theorem 3.3). A key challenge in developing this part of Dandelion was ensuring that the

```
sturm_seq (p, q, n) =
  if n = 0 then
    if (rm (p, (1/q[deg q]) * q) = 0 ∧ q <> 0) then SOME []
    else None
  else let g = - (rm (p, (1/q[deg q]) * q)) in
    if g = 0 ∧ ~ q = 0 then Some []
    else if (q = 0 ∨ (deg q < 3)) then None
    else case sturm_seq (q, g, n-1) of
      None => None
      |Some ss => Some (g::ss)
```

Figure 3.6: HOL4 definition of Sturm sequence computation

Sturm sequence is computable inside HOL4. In HOL-Light, Harrison defines Sturm sequences as a non-computable predicate `STURM` that existentially quantifies results, and thus the predicate can only be used to validate results in a manual proof.

In Figure 3.6, we show how Dandelion computes Sturm sequences. Function `rm (p,q)` computes the remainder of the polynomial division of `p` by `q`, `deg p` is the degree of polynomial `p`, and `q[n]` is the extraction of the `n`-th coefficient of `q`. As each polynomial division operation decreases the degree of the result by at least 1, the Sturm sequence for a polynomial p has a maximum length of $\texttt{deg}(p) - 1$, as computation starts with p and its first derivative p'. Function `sturm_seq` is therefore initially run on polynomial $h'(x)$, $h''(x)$, and $\texttt{deg}(h') - 1$[6]. If `sturm_seq(h', h'', deg h'-1)` returns list `sseq`, the complete Sturm sequence is `h'::h''::sseq`, and Dandelion computes the number of zeros of e' as its variation on the input range, based on Theorem 3.3.

We have proven once and for all that the results obtained from `sturm_seq(p', p'', n)` satisfy Harrison's non-computable predicate `STURM`. Thus we can reuse Harrison's proof of Sturm's theorem. Harrison's Sturm sequences also use a non-computable predicate for defining the result of polynomial division, and we prove it equivalent to a computable version in Dandelion, inspired by the one provided by Isabelle/HOL [119]. Ultimately, Dandelion uses these two equivalence proofs to reuse Harrison's proof of Sturm's theorem, which we ported from HOL-Light.

[6] We add the additional parameter `n` to the function as "fuel" such that termination of the function is automatically proven by HOL4.

3.5.2. Finding Zeros of Polynomials

Given the numbers of zeros nz for $h'(x)$, Dandelion next finds their values. As zero-finding is highly complicated even in non-verified settings, Dandelion uses an external oracle to come up with an initial guess of the zeros. These initial guesses are presented as a list of confidence intervals $[a, b]$, where $h'(x)$ is speculated to have a zero between a and b. The algorithm computing the confidence intervals need not be verified, as the intervals themselves are easily validated by Dandelion. To validate a list of guesses Z, Dandelion again relies on a result from real-number analysis ported from Harrison's HOL-Light development: Function f has a zero in the interval $[a, b]$, if its first derivative f' changes sign in the interval $[a, b]$. Thus, to check whether a confidence interval $[a, b]$ in Z contains a zero of h', we need to check whether h'' changes sign in the interval.

Dandelion validates the confidence intervals Z for h' using a computable function that checks automatically for each element $[a, b]$ in Z that $h''(a) * h''(b) \leq 0$, which is equivalent to a sign-change of h'' in the interval. If the number of zeros found using Sturm's theorem is nz, Dandelion checks that this sign change occurs at least nz times in Z.

While we do prove our approach for finding zeros of polynomials sound, Dandelion is necessarily incomplete. One known source for incompleteness are so-called multiple roots as they occur in, e.g., $p(x) = (x - 1)^2$. Harrison's formalization implicitly relies on the polynomial being squarefree and Dandelion inherits this limitation. This issue could potentially be addressed using the approach of Li, Passmore, and Paulson [75], though it would require reproving Sturm's theorem for non-squarefree polynomials.

3.5.3. Computing Extremal Values

In the final step, Dandelion uses the validated confidence intervals Z which contain all zeros of $h'(x)$ to compute an upper bound to the extremal values of $h(x)$. For an interval $[a, b]$, Dandelion would ideally bound the error of $h(x)$ in $[a, b]$ as the maximum of $h(a)$, $h(b)$, and $h(y)$, where y is a zero of $h'(x)$. However, we have only confidence intervals for the zeros available, and not their exact values. Therefore, Dandelion's computation of an upper bound to $h(x)$ is more involved, and we base it on a theorem of Harrison. Harrison's theorem is a generalization of Theorem 3.2 for polynomial p, with derivative p', on interval $[a, b]$, where zeros are bounded by confidence intervals in Z:

```
validateZerosLeqErr (h, I, numZeros, zeros, eps) =
 2  let mAbs = max (abs (fst I)) (abs (snd I));
      realZeros = findN numZeros
 4     (λ (u,v). poly (diff h) u * poly (diff h) v ≤ 0) zeros;
      B = poly (MAP abs (diff h)) mAbs;
 6    K = getMaxAbsLb h realZeros;
      e = getMaxWidth realZeros;
 8    globalErr = max (abs (poly h (fst I)))
                      (max (abs (poly h (snd I)))
10                     (K + B * e))
    in if ¬ (validBounds I realZeros ∧ recordered (fst I) realZeros (snd I))
12   then (Invalid "Zeros not correctly spaced", 0)
     else if LENGTH realZeros < numZeros then
14      (Invalid "Did not find sufficient zeros", 0)
     else if globalErr ≤ eps
16   then (Valid, globalErr)
     else (Invalid "Bounding error too large", 0)
```

Figure 3.7: Function for computing a bound on the error polynomial $h(x)$ in Dandelion

Theorem 3.14.

$$
\begin{aligned}
&(1) \quad (\forall x.\ a \le x \le b \wedge f'(x) = 0 \Rightarrow \exists (u,v).\ (u,v) \in Z \wedge u \le x \le v) \wedge \\
&(2) \quad (\forall x.\ a \le x \le b \Rightarrow |\, p'(x) \,| \le B) \wedge \\
&(3) \quad (\forall [u,v].\ [u,v] \in Z \Rightarrow a \le u \wedge v \le b \wedge |\, u - v \,| \le e \wedge |\, f(u) \,| \le K) \ \Rightarrow \\
&\qquad \forall x.\ a \le x \le b \Rightarrow |\, p(x) \,| \le \max(|\, f(a) \,|, |\, f(b) \,|, K + B \times e)
\end{aligned}
$$

The theorem can be used to prove an upper bound on the error polynomial $h(x)$ which then can be compared to the residual error γ. For Dandelion, we automatically compute the values described by the assumptions to compute an overall bound on $h(x)$. We implement this computations in a function `validateZerosLeqErr` in Figure 3.7, and explain each of its computation steps on a high-level, based on the assumptions of Theorem 3.14.

The first assumption (1) from Theorem 3.14 states that the confidence intervals in Z contain only valid zeros. Function `validateZerosLeqErr` computes a list of valid confidence intervals from the unvalidated intervals in line 3 of Figure 3.7. Based on assumption

(2), Dandelion computes B by evaluating $|h'(x)|$ on $\max(|a|, |b|)$ (line 5). Following assumption (3), Dandelion computes K as the maximum value of evaluating the error polynomial h on the lower bounds of the confidence intervals in Z (line 6), and a value e as the maximum value of $|u - v|$ for each $[u, v]$ in Z (line 7). Dandelion then computes the overall bound on the error polynomial h as $\max(h(a), h(b), K + B \times e)$ (line 8). The code in lines 11 to 14 of Figure 3.7 checks that the confidence intervals are in ascending order (line 11), and that exactly the required number of zeros were found (line 13). To validate the residual error γ, it then suffices to check whether $\max(h(a), h(b), K + B \times e) \leq \gamma$ (line 15).

Overall, we prove once and for all soundness of Dandelion as

Theorem 3.15 (Dandelion Soundness)**.**

$$\texttt{Dandelion(f,p,I,}\varepsilon\texttt{,n)} = \texttt{true} \;\Rightarrow\; (\forall x.\; x \in \texttt{I} \Rightarrow |\texttt{f}(x) - \texttt{p}(x)| \leq \varepsilon)$$

The proof of Theorem 3.15 uses the triangle inequality to combine the theorem First Phase Soundness (Theorem 3.5) with the theorem Second Phase Soundness (Theorem 3.13).

This concludes our discussion of how Dandelion automates the key steps of Harrison's proof. Before comparing Dandelion with state-of-the-art tools, we first take a closer look at the performance of Dandelion's HOL4 implementation.

3.6. Extracting a Verified Binary with CakeML

In general, computations performed in interactive theorem provers are said to be slower than those in unverified languages. When performing first tests with Dandelion, using HOL4's native evaluation, we noticed that especially the Sturm sequence computation required a significant amount of computation time.

Via manual inspection, we found that the computations are bottlenecked (in-part) by the efficiency of real-number computations in HOL4. As every intermediate result is computed exactly by HOL4, the size of the numbers that HOL4 uses for computations grows quickly. Some numbers would span around six thousand digits, which lead to a single real number operation taking up-to 6 hours to compute a result.

Under the hood, HOL4 treats real numbers as fractions during computation, stopping at uncomputable terms, e.g., elementary functions. Every operation on these fractions relies on an efficient implementation of natural number multiplications and we optimize it by implementing Karatsuba multiplications.

The key insight for this approach is that any fractional multiplication $\frac{a}{b} \times \frac{c}{d}$ can be turned into multiplications of natural numbers with an additional sign:

$$\frac{a}{b} \times_{\mathbb{R}} \frac{c}{d} = \text{sgn}(\frac{a}{b}) \times_{\mathbb{R}} \text{sgn}(\frac{c}{d}) \times_{\mathbb{R}} \frac{|a| \times_{\mathbb{N}} |c|}{|b| \times_{\mathbb{N}} |d|}$$

where sgn is the sign function, returning -1 for negative and 1 for positive inputs. Our implementation of Karatsuba multiplication significantly improves performance of Dandelions polynomial computations, e.g., for some tests we essentially halved the running time of multiplying a polynomial by a constant. On a global scale, however, the effect on Dandelion's computations was smaller, but still noticable. For a degree three polynomial, we reduced the overall evaluation time to validate the approximation error with Dandelion from 47 to 43 minutes, and for a degree five polynomial, we went from 92 minutes to 69 minutes. We still found the performance of Dandelion to not yet be optimal.

To further improve performance of Dandelion, we chose to use the proof-producing synthesis [2] implemented in the CakeML verified compiler [113]. The synthesis procedure translates HOL4 functions into their CakeML counterpart, with an equivalence proof. These translated CakeML functions are compiled into machine code with the CakeML compiler and, as CakeML is fully verified, the machine code enjoys the same correctness guarantees as its HOL4 equivalent.

During our earlier tests with the Karatsuba multiplication, we noticed that the Sturm sequence computations in the second phase are the most computationally expensive task of Dandelion. Therefore, we use CakeML's proof-producing synthesis to extract a verified binary for the computations described in Section 3.5. To communicate results of the first phase with the binary, we implemented an (albeit unverified) lexer and parser that reads-in results from the first phase.

3.7. Evaluation

We have described how we speed up Dandelions automatic validation of polynomial approximations from Remez-like algorithms. In this section, we demonstrate Dandelion's usefulness with three separate experiments, showing that Dandelion fully automatically validates

1. certificates generated by an off-the-shelf Remez-like algorithm(Section 3.7.1)
2. certificates for more complicated elementary function expressions (Section 3.7.2)
3. certificates for less-accurate techniques (Section 3.7.3)

All the results we report in this section where gathered on a machine running Ubuntu 20.04, with an 2.7GHz i7 core and 16 GB of RAM. All runnning times are measured using the UNIX `time` command as elapsed wall-clock time in seconds.

3.7.1. Validating Certificates of a Remez-like Algorithm

In our first evaluation, we show that Dandelion certifies accurate approximation errors from an off-the-shelf Remez-like algorithm. We generate certificates by combining the Daisy static analyzer with the Sollya approximation tool [25], and extend Daisy with a simple pass that replaces calls to elementary functions with an approximation computed by Sollya. As a Remez-like algorithm, we use the `fpminimax` [21] function in Sollya. We evaluate Dandelion on numerical kernels taken from the FPBench benchmark suite [31], and the benchmarks used by an unverified extension of Daisy with approximations for elementary functions [60]. These benchmarks represent kernels as they occur in, e.g., embedded systems, and thus they benefit from custom polynomial approximations. The original work by Izycheva *et al.* [60] synthesizes polynomial approximations whose target error bounds are usually larger than those inferred by Sollya. Hence, Dandelion could validate Daisy's bounds as well, but for the sake of the evaluation we choose more challenging, tighter bounds. For each benchmark, Daisy creates a certificate for each approximated elementary function, amounting to a total of 96 generated certificates.

Sollya's implementation of `fpminimax` can be configured to use different degrees for the generated approximation and different formats for the coefficients of the approximation. In our evaluation we approximate elementary functions with a degree 5 polynomial, storing the coefficients with a precision of 53 bits. All input ranges used in the certificates are computed by Daisy without modifying them, except for exp, where we disallow negative intervals, i.e., if Daisy wanted to approximate on $[-x, y]$, we change it to $[0, y]$ as Dandelion currently does not support negative exponentials. This simplification can be fixed by straight-forward range reductions that are independent to the approximations computed by Remez-like algorithms. For each such approximation, Daisy creates a certificate to be checked by Dandelion.

The MetiTarski automated theorem prover [4] is a tool that provides the same level of automation as Dandelion, but relies on a different technology for proving inequalities. MetiTarski is based on the Metis theorem prover [57] and can output proofs in TSTP format [112]. It relies on an external decision procedure to discharge some goals, and for our experiments we used Mathematica. In general, MetiTarski checks real-number

Function	#	Dandelion			MetiTarski		CoqInterval	
		Verified	HOL4(s)	Binary(s)	Verified	Time(s)	Verified	Time(s)
$\tan^{-1}$	2	1	11.62	20.36	2	6.80	2	1.74
cos	28	25	202.35	251.33	25	3.15	26	1.91
exp	21	18	39.58	212.54	10	5.35	20	1.58
log	8	0	0	0	5	4.99	8	1.68
sin	31	27	17.83	295.66	25	6.32	31	1.78
Total	90	71			67		87	

Table 3.1: Overview of certificates validated with Dandelion, MetiTarski, and CoqInterval

inequalities that may contain elementary functions, thus we compare the number of certificates validated by Dandelion to those that can be checked by MetiTarski.

Further, the CoqInterval package [82] proves inequalities about elementary functions in the Coq theorem prover [116]. In contrast to Dandelion's technique based on high-accuracy Taylor polynomials and Sturm sequences, CoqInterval is based on interval arithmetic with optional interval bisections and high-accuracy Taylor polynomials. Further, Dandelion is verified in HOL4, while CoqInterval is implemented in Coq. To compare the two approaches, our evaluation also includes the CoqInterval package. Our evaluation excludes a similar approach formalized in Isabelle/HOL [56] because we could not come up with a straight-forward translation of our certificates as inputs to the tool. We have manually tested some of our benchmarks and expect it to produce results similar to CoqInterval.

Our results are given in Table 3.1. The left-most column of Table 3.1, contains the name of the elementary function approximated by Daisy, and the second column, labeled with a # contains the number of certificates with unique input ranges generated for the elementary function. The next three columns, headed "Dandelion", contain the number of certificates validated by Dandelion, the average HOL4 running time for the first phase in seconds, and the average running time of the binary for the second phase in seconds. The next two columns, headed "MetiTarski", contain the number of certificates validated by MetiTarski, and the average running time in seconds. The final two columns, headed "CoqInterval", contain the number of certificates validated by CoqInterval, and the average running time in seconds.

Our evaluation truncates Taylor series after 32 terms in `approxAsPoly`. In general, we found six times the degree of the computed approximation to be a good estimate for when to truncate Taylor series in Dandelion. Our use of 32 instead of 30 is a technical detail, as some Taylor series require both the number of series terms n, as well as $\frac{n}{2}$ to be even.

In general, the number of series terms has to be significantly higher than the degree of the approximated polynomial to make the approximation error of the first phase almost negligible.

Overall, we notice that each of the tools in our evaluation certifies a slightly different set of approximations. In total, CoqInterval certifies most of the benchmarks, but Dandelion successfully checks one cosine certificate that CoqInterval fails to check. While Dandelion validates more certificates than MetiTarski, both MetiTarski and CoqInterval validate certificates that are currently out of reach for Dandelion. This is mostly due to the first phase of Dandelion. Even though we used general, widely known truncated Taylor series for all supported elementary functions, Dandelion fails to validate certificates for the ln function. We have inspected the generated certificates, and Dandelion cannot compute a high-accuracy polynomial approximation for 5 of them because they do not satisfy the precondition of Dandelion's Taylor series. For the remaining 3 certificates, we ran into issues with Sollya's computation of the confidence intervals for the zeros. On a high-level, the problem originates from the derivative of the error polynomial $h(x)$ being very close to 0, leading to a huge number of zeros found, i.e., computation not terminating within a reasonable amount of time. To demonstrate that Dandelion still certifies errors for the ln function, we add an example in Section 3.7.2.

While Dandelion cannot certify errors for the ln function in this part of the evalution, this is not a conceptual limitation, as polynomial approximations are commonly paired with an argument reduction strategy. While verification of these strategies is orthogonal to validating results of an Remez-like algorithm, they could be used to reduce the input range of the approximated elementary function into a range that Dandelion can certify. More generally, we have done the heavy lifting of automating the computations and implementing the general framework, such that adding more accurate Taylor series to Dandelion amounts to mere proof engineering, modulo coming up with Taylor series in the first place. For the certificates for $\tan^{-1}$, cos, sin, and exp, we notice that the average running time is in the order of minutes, making certificate checking with Dandelion's verified binary feasible.

We compare the approximation errors recorded in the certificates with the infinity norm computed by Sollya, which is the most-accurate estimate of the approximation error [24]. Overall, Dandelion certifies an approximation error in the same order of magnitude as the infinity norm for 61 certificates. For the remaining 10, the error is a sound upper bound. In general, infinity norm-based estimates are known to be the most accurate and their verification requires more elaborate techniques than Sturm sequences [24]. Consequently we would not expect Dandelion to be able to always certify infinity norms.

Function	Range	Deg.	Prec.	∞-norm	Error	HOL4	Binary
$\cos(x+1)$	$[0, 2.14]$	5	53	3.06E-5	3.06E-5	169	63
$\sin(x-2)$	$[-1, 3.00]$	5	53	2.05E-3	2.91E-3	93	68
$\ln(x+\frac{1}{10})$	$[1.001, 1.1]$	3	32	1.08E-7	1.08E-7	2775	1773
$\exp(x\times\frac{1}{2})+\cos(x\times\frac{1}{2})$	$[0.1, 1.00]$	5	53	2.03E-9	4.45E-9	711	5
$\tan^{-1}(x)-\cos(\frac{3}{4}\times x)$	$[-0.5, 0.5]$	5	53	1.18E-5	1.18E-5	24	2308

Table 3.2: Functions approximated with Sollya using `fpminimax` and certified with Dandelion

To measure the influence of the approximation degree and precision used by Sollya, we also ran the evaluation for an approximation degree of 3 and 5, with precisions of 53 and 23 each. Overall, the running time significantly decreases when decreasing the degree from 5 to 3, going from average running times of minutes to average running times of seconds. Decreasing the precision of the coefficients further speeds up evaluation, though not as significant as decreasing the degree did. This suggests that higher coefficient accuracies can easily be used for generating polynomials with Remez-like algorithms, and lower degree polynomials should be preferred for fast validation.

3.7.2. Validating Certificates for Elementary Function Expressions

In our second evaluation, we demonstrate that Dandelion can also certify approximation errors for complicated elementary function expressions. We validate with Dandelion approximation errors for random examples involving elementary functions and arithmetic. Polynomial approximations are again generated by Sollya.

An overview of our results is given in Table 3.2. The table shows the approximatied function, then Sollya's parameters (the input range, the target degree (Deg.), the target precision (Prec.)), and then the infinity-norm (∞-norm) of the approximation. The final columns summarize the Dandelion results, giving the certified approximation error, and the running time in seconds of the first phase (HOL4) and the second phase (Binary).

Overall, we notice that the certified approximation error is on the same order of magnitude as the infinity norm for all examples. We also notice that performance for both phases varies across the different examples. For the first phase this is often due to how the input ranges are encoded in the certificate. We noticed that HOL4 is very sensitive to how the fractions representing real numbers are encoded when performing computations. Similarly, performance of the second phase greatly varies depending on

the complexity of the error polynomial computed by the first phase. We observe that perfomance improves with both smaller degree polynomials, and smaller representations of the polynomial coefficients.

The results in Table 3.2 exclude examples where two elementary functions are composed with each other asn in, e.g., $\exp(\sin(x-1))$. This is because Dandelion computes the global high-accuracy approximation in the first phase via polynomial composition of an approximation for exp and sin. While this is theoretically supported by Dandelion, we found that the polynomial composition leads to an exponential blow-up in the degree of the error polynomial. Even for the innocuously looking example $\exp(\sin(x-1))$, the second phase could not validate a polynomial approximation within 24 hours. This clearly motivates the use of more elaborate Taylor series if compound elementary functions need to be certified by Dandelion. In general, settings where elementary functions like those in Table 3.2 are used could potentially be made more accurate and be validated faster with custom Taylor series.

3.7.3. Validating Certificates for Simpler Approximation Algorithms

Remez-like algorithms are known to be the most accurate approximation algorithms. However, less accurate approaches are still in use today, and as such interesting targets for verification. Bréhard *et al.* [20] certify Chebyshev approximations in the Coq theorem prover, where their approach requires some manual proofs. We demonstrate that Dandelion also certifies Chebyshev approximations on some random examples by computing Chebyshev approximations with Sollya's function `chebyshevform`. The results are shown in Table 3.3.

Again, we first give Sollya's parameter and the infinity norm, then we give the error certified by Dandelion, and the execution times for the first and second phase.

Table 3.3 also includes the approximation certified by Harrison [53], labeled with a *. The polynomial has degree 3, but we leave the precision empty as it is not generated by Sollya, and we do not provide an infinity norm. The only difference to the proof from Harrison is that we prove the bound only for positive x, as Dandelion currently does not handle exponentials on negative values. The lower bound of the range is 0.003, instead of 0 to rule out a 0 on the lower bound (as $\exp(x) - 1 = 0$ for $x = 0$), which we must exclude by Theorem 3.14. Harrison's manual proof of the polynomial approximation then reduces to a single line running Dandelion on the encoding.

Function	Range	Deg.	Prec.	∞-norm	Error	HOL4	Binary
$\cos(x)$	$[0, 2.14]$	5	53	3.17E-7	3.22E-7	169	63
$\sin(x+2)$	$[-1.5, 1.5]$	5	53	4.47E-4	7.60E-4	142	138
$\sin(3 \times x) + \exp(x \times \frac{1}{2})$	$[0, 1]$	3	53	2.45E-2	2.48E-2	54	1897
$\exp(x) - 1^*$	$[0.003, 0.01]$	3			$2^{-33.2}$	133	<1

Table 3.3: Chebyshev approximations certified with Dandelion, Harrison's example labeled with *

3.8. Related Work

Throughout this chapter, we have already hinted at immediate related work for Dandelion. In this section, we explain the key conceptual differences between Dandelion and its immediate related work and put Dandelion into the greater context. In general, Dandelion touches upon two key research areas in interactive and automated theorem proving: techniques for approximating elementary functions and techniques for proving theorems involving real-numbered functions.

Approximating Elementary Functions. The work on approximating elementary functions can be distinguished among two axes: whether or not the work provides rigorous machine-checked proofs, and whether the work is fully automated or requires user intervention.

Fully automated, rigorous machine-checked proofs, similar to Dandelion are provided by the work by Bréhard *et al.* [20]. They develop a framework for proving correct Chebychev approximations of real number functions in the Coq theorem prover [116]. A key difference to Dandelion is that the technique cannot certify approximations of Remez-like algorithms, which can in general provide more accurate approximations. Also developed in the Coq theorem prover, Martin-Dorel and Melquiond [83] verify polynomial approximations using the CoqInterval [82] package. They develop a fully automated tactic for proving approximations inside floating-point mathematical libraries correct and we have compared Dandelion's results with those of CoqInterval in Section 3.7. The key technical difference is that the work by Martin-Dorel and Melquiond uses a combination of interval arithmetic, Taylor approximations, and floating-point computations to prove approximations, while Dandelion relies on Sturm sequences. An approach similar to CoqInterval was also formalized in Isabelle/HOL by Hölzl [56].

For manual proofs, versions of the exponential function have been verified by Harrison [51], and Akbarpour *et al.* [3]. The manual proof by Harrison [53] layed out the foundations for Dandelion. Harrison's work has also been extended by Chevillard *et al.* [24]. Instead of verifying approximation errors for polynomials, they use so-called sum-of-squares decompositions [52] to certify infinity norm computations. A major limiting factor for their work was finding accurate enough Taylor polynomials which we found to not be a major issue for our approach.

Coward *et al.* [28] use the MetiTarski automated theorem prover [4] to verify accuracy of finite-precision hardware implementations of elementary functions. MetiTarski provides proofs in machine-readable form using the TSTP format [112] instead of being developed inside an interactive theorem prover like HOL4. A major conceptual difference is that the verification done by Coward *et al.* reasons about bit-level accuracy of the hardware implementation, while Dandelion reasons about real-number functions and polynomials. Together with a verified roundoff error analysis like FloVer [12], Dandelion could be extended to verify finite-precision implementations of elementary functions, and together with Daisy [60] verification could possibly be lifted to entire arithmetic kernels.

A different style of approximations without rigorous machine-checked proofs is provided by Lim and Nagarakatte [77]. Instead of computing specialized polynomial approximations, they focus on correctly-rounded, general purpose approximations. Such approximations are not build for specific use-cases, but rather meant to replace the functions provided in mathematical libraries. At the time of writing, their approach is not formally verified, but they do provide a pen-and-paper correctness argument for their code generation. The CR-libm [32] library also provides unverified alternatives of correctly rounded mathematical libraries, and Muller [94] gives a general overview of the techniques for implementing elementary functions.

(Automated) Real-Number Theorem Proving. Dandelion heavily relies on HOL4's support for real-number theorem proving. Below we list some alternatives for proving properties of real-numbers in both interactive and automated theorem proving systems. In the HOL-family of ITP systems, Harrison [52] has formalized sum-of-squares certificates for the HOL-Light [117] theorem prover. His approach relies on semidefinite programming to find a decomposition of a polynomial into a sum-of-squares polynomial. Both Isabelle/HOL and PVS have been independently extended with implementations of Sturm sequences [39, 97, 96]. Their main focus is not on verification of polynomial approximations, they rather use Sturm sequences to prove properties about roots of polynomials, non-negativity, and monotonicity.

Previously we have already mentioned the MetiTarski automated theorem prover [4], as an example of an automated theorem prover for real-numbered functions. However, MetiTarski is not the only automated prover for real-numbered functions. Real-numbered functions are also supported by, e.g., dReal [41], and z3's SMT theory for real-numbers [93].

3.9. Discussion

We have presented the first contribution of this thesis, Dandelion. Dandelion certifies approximation errors for polynomial approximations of elementary functions and supports inputs from the known most-accurate Remez-like algorithms. Through its verified binary, Dandelion certifies bounds within a reasonable amount of time.

The next chapter combines Dandelion's results with a verified roundoff error analysis. We extend CakeML with a proof-producing that translates a real-numbered elementary function into floating-point machine code with accuracy guarantees.

CHAPTER 4

Verified Generation of libm Kernels

This chapter contains new content. The proofs and implementation have been performed for the thesis and are not published elsewhere. Both the writing and formal development have been performed by me.

4.1. Introduction

An integral part of any floating-point implementation in a compiler is a library of mathematical functions (short *libm*). A libm commonly implements elementary functions like sin, cos, and exp. In general, a function is implemented in a libm if it cannot be translated directly into floating-point hardware instructions, and therefore it must be implemented in software instead.

Before our work, the CakeML compiler did not provide a library of mathematical functions. CakeML only supports floating-point primitives that have a one-to-one correspondence with hardware instructions, i.e., addition, subtraction, multiplication, division, square root, and absolute values.

Implementating a libm for an unverified compiler is already a tricky task on its own. A key challenge in implementing an elementary function like, e.g., sin, is making sure that the libm implementation returns a result that is as-close-as-possible to the idealized, real-numbered result. The work by Lim et al. [77] is the most recent work on "correct-by-construction" libm implementations. Their tool, Rlibm, claims to generate code that computes the correctly-rounded result of mathematical functions for different target formats; i.e., the result of running one of their libm implementations is the most accurate result that can be computed for a particular format. However, verification of the results of Rlibm is still a major challenge.

In this work, we relax the stringent correctness requirement of Rlibm, and say that for CakeML any polynomial implementation of an elementary function is fine as long as we

can certify its accuracy with a rigorous machine-checked proof. Further, while Rlibm generates general-purpose implementations, we focus on polynomial approximations that are designed for a specific use-case; they must only be correct for a limited set of inputs.

The key challenge to verify a particular use-case-specific polynomial approximation in CakeML is two-fold: First, the correctness proof must relate a real-valued polynomial approximation to its elementary function with a proof bounding the approximation error; second, the proof must relate a floating-point polynomial implementation to the aforementioned real-valued polynomial approximation.

A first solution addressing this key challenge is presented by Appel and Bertot [5]. Their approach verifies numerical implementations using the VST framework [7] and Flocq [18] in the Coq theorem prover. To this end, they verify a C-implementation of a sqrt algorithm by first manually proving a correspondance between the C-code implementation and a functional model using VST. Then, the functional model is verified with respect to IEEE-754 floating-point semantics using Flocq and Gappa [37]. With the CompCert [74] compiler, the verified C-implementation can be compiled to machine code. In general, the approach by APpel and Bertot could be used to verify libm implementations, but, the approach still involves a large chunk of manual proof work.

A key takeaway point of the work by Appel and Bertot is that they split the overall correctness into proofs about the C-code implementation of a floating-point polynomial, and proofs relating a polynomial implementation to its real-valued counterpart. Following this separations of concerns, we want to automate both steps inside the CakeML compiler to automatically generate use-case specific implementations of elementary functions.

With Dandelion (Chapter 3), we can automatically prove error bounds for real-valued poynomials and elementary functions, and with the FloVer tool (Section 2.4) we can validate finite-precision roundoff error bounds, automating the most complicated parts. The only piece missing from the puzzle is a relation between FloVer's idealized finite-precision semantics, and CakeML's floating-point semantics.

In previous work, we have shown that floating-point tools generally benefit from being used together [8] and therefore, our technical key idea is to combine the outputs of Dandelion and FloVer and pairing them with a simulation proof relating FloVer's idealized IEEE-754 semantics to CakeML floating-point arithmetic. To this end, we implement *libmGen*, the first fully automated proof-producing compiler for libm function implementations as a CakeML extension.

Dandelion and FloVer were both designed to certify results of unverified tools, and do not compute the results themselves. For Dandelion, this dependency cannot be resolved, and we will necessarily rely on a generator for polynomial approximations. For FloVer,

```
cos_example = <|
 transc := cos (x);
 poly := [
   4289449735 / (2 pow 32); (* ∼0.9987 *)
   139975391 / (2 pow 33); (* ∼0.0162 *)
   -2408138823 / (2 pow 32); (* ∼-0.5606 *)
   2948059219 / (2 pow 35) ]; (* ∼0.0857 *)
 eps := 582015 / (2 pow 31)^-1; (* ∼2.71 * 2^-4 *)
 iv := [ ("x", ( 0.1 , 1))]; |>
```

Figure 4.1: Example certificate for cos approximation; for readability we translate constants to reals in comments

however, we notice that the (unverified) analysis of finite-precision errors and FloVer's validation steps practically coincide. We remove the dependency on an external unverified analysis to generate certificates with an extension of FloVer that generates (unverified) analysis certificates. These certificates are then validated with FloVer's verified certificate checking pipeline.

Contributions. In summary, this chapter contributes:

- an extension of FloVer to generate (unverified) analysis certificates;
- an end-to-end simulation relating Dandelion's real-valued elementary functions to CakeML-produced floating-point machine code;
- a fully automated proof-producing method that implements libm functions from a Dandelion certificate.

4.2. Overview

Before diving into the details of how libmGen is implemented and verified, we give a high-level overview of how our proof-producing compiler libmGen works. As an example, we implement the elementary function $\cos(x)$ on the interval $[0.1, 1]$ with a degree 3 polynomial in CakeML. Figure 4.1 shows the Dandelion certificate for our example. In the certificate in Figure 4.1, `transc` is the approximated elementary function, `poly` is the polynomial approximation computed with Sollya [25], `eps` is an unverified bound on the

```
fun cos x = 0x3FEFF579E0E00000 (* ∼0.9987 *) +
   (x * (0x3F90AFB5BE000000 (* ∼0.0162 *) +
    (x * (0xBFE1F12908E00000 (* ∼-0.5606 *) +
     (x * 0x3FB5F6FA0A600000 (* ∼0.0857 *))))))
```

Figure 4.2: CakeML program generated automatically by libmGen for the example; for readability we translate constants to reals in comments

approximation error, and `iv` describes the input constraint. From this certificate, libmGen automatically generates the CakeML source code implementation in Figure 4.2, where constants are the coefficients of the polynomial from Figure 4.1 rounded to 64-bit double words.

In addition to generating CakeML source code, libmGen also proves a specification theorem relating executions of the generated CakeML code to the real-valued cos function:

Theorem 4.1 (CakeML specification of cos)**.**

$$0.1 \leq w \leq 1 \ \wedge \ \texttt{finite}\ w \Rightarrow$$
$$\texttt{evaluate cosCode}\ [w] = v \ \wedge \ |v - \cos(w)| \leq \texttt{eps} + \delta$$

As the approximation and roundoff errors inferred by FloVer and Dandelion are only valid for arguments within the range specified in Figure 4.1, Theorem 4.1 assumes that the program is only run on inputs within these constraints. Further, the theorem assumes that input values are finite floating-point double constants, i.e., no exceptional values like `NaN` or $\pm\infty$ as this would invalidate the result of the roundoff error analysis. The theorem shows that the generated CakeML-code (Figure 4.2) always returns a floating-point value v and that the difference between floating-point value v and the real-valued cos function is upper bounded by the approximation error from the certificate (`eps`) plus the roundoff error δ.

To prove Theorem 4.1, libmGen performs three separate steps, which we illustrate on a high-level in Figure 4.3. Given a Dandelion certificate consisting of an elementary function f, polynomial approximation p, input constraints I, and an approximation error ε, libmGen first validates the certificate with Dandelion. Second, the polynomial from the certificate is translated into FloVer's input expressions using function `poly2FloVer`, and libmGen automatically infers and validates a roundoff error bound using FloVer. Third, libmGen translates FloVer's floating-point expression into CakeML code using function `flover2CML`, and automatically proves a program specification exactly as in Theorem 4.1.

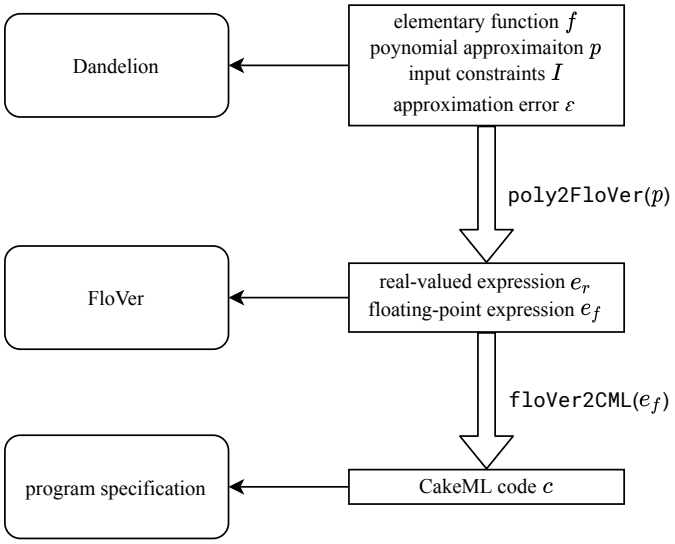

Figure 4.3: Overview of components of libmGen

Our tool libmGen should compute a verified roundoff error analysis result without relying on an additional external tool to generate an analysis certificate. Therefore, we extend FloVer with an unverified implementation of a roundoff error analysis that encodes its result as a certificate for FloVer's checking pipeline. In libmGen, we validate this certificate using FloVer's verified certificate checking pipeline. While the translation functions `poly2FloVer` and `flover2CML` have straight-forward implementations, the crux of libmGen lies in proving the correct simulations for the translation functions to prove the program specification fully automatically. We have already presented the soundness proofs for Dandelion (at the end of Section 3.5) and FloVer (Section 2.4) and thus we will not go into further details here.

In the rest of this chapter, we will walk, step-by-step, through the simulations, explaining how libmGen automatically translates elementary functions approximations into CakeML source code with accuracy proofs (Section 4.3). Afterwards, we show in our evaluation how libmGen generates CakeML code for some small elementary function examples (Section 4.4).

4.3. Simulations in libmGen

Computationally, libmGen simply runs the tools we have presented so far one after another. We only extend FloVer with unverified certificate generation to avoid relying on an additional external tool. To implement the unverified certificate generation, we

have simply copied the validation functions of FloVer and replaced their checking part with code that returns the computed values. The key technical challenge of libmGen is to automatically prove a program specification relating double floating-point machine code to real-numbered elementary functions.

On a high-level, when implementing an elementary function, libmGen automatically instantiates the following general theorem:

Theorem 4.2 (libmGen specification)**.**
Let f be an elementary function defined on range I, p a polynomial approximation, and y an element of I, then

$$
\begin{aligned}
&(1) \quad \max_{x \in I} |f(x) - p(x)| \leq \varepsilon \wedge \\
&(2) \quad (\texttt{poly2FloVer}(p), [x \mapsto [y]_{\texttt{real}}]) \Downarrow^{\text{real}} v_r \wedge \\
&\qquad (\texttt{poly2FloVer}(p), [x \mapsto y]) \Downarrow^{\text{double}} v_f \wedge \\
&(3) \quad |v_r - v_f| \leq \delta \Rightarrow \\
&(4) \quad \texttt{evaluate}\,(\texttt{floVer2CML}\,(\texttt{poly2FloVer}\ p\ ),\ y) = v \wedge \\
&\qquad |v - f([y]_{\texttt{real}})| \leq \varepsilon + \delta
\end{aligned}
$$

Theorem 4.2 assumes that ε is the verified approximation error for $|f(x) - p(x)|$ (1), the polynomial p translated to FloVer can be run under real-numbered and floating-point semantics (2), and δ is the verified roundoff error of the translated polynomial (3). From these assumptions, the theorem shows (4) that the CakeML code generated from the FloVer polynomial can be successfully run on input value y to obtain the result value v. Further, the difference between v and the result of evaluating the elementary function f on the real-number equivalent of y is upper bounded by $\varepsilon + \delta$, i.e., the sum of the approximation error and the roundoff error.

When generating a CakeML implementation for a Dandelion certificate, libmGen automatically generates proofs for assumptions (1), (2) and (3) of Theorem 4.2. Assumption (1) is proven by running Dandelion on the input certificate; and assumptions (2) and (3) are proven by translating the polynomial to FloVer, generating an unverified analysis certificate for the translated polynomial, and checking the certificate using FloVer.

We prove Theorem 4.2 once and for all and the proof relies on two key simulations. The first simulation relates Dandelion's idealized real-numbered polynomials to their translation to FloVer via `poly2FloVer`:

Theorem 4.3 (`poly2FloVer` Bisimulation)**.**
Let p be a polynomial, then

$$p(v) = r \Leftrightarrow (\texttt{poly2FloVer}\ p, [x \mapsto v]) \Downarrow^{\text{real}} r$$

Theorem 4.3 shows that the real-valued polynomial p and its translation to FloVer must always return the same value if the translated expression is evaluated under real-number semantics ($\Downarrow^{\text{real}}$) with the free variable x bound to the input v of the polynomial. We have proven Theorem 4.3 once and for all via an induction on p, i.e., we show Theorem 4.3 first for the constant 0 polynomial, and in the induction step, we show Theorem 4.3 for $p(y) = c + y * q(y)$, assuming the theorem holds true for $q(y)$.

The second simulation relates FloVer's finite-precision double expressions with their translation to CakeML double floating-point code via `floVer2CML`:

Theorem 4.4.
Let e be a real-numbered FloVer expression without any cast operations, A an unverified roundoff error analysis result, and I an input constraint, and env a CakeML runtime environment binding the free variables of e within the constraints from I, then

$$\begin{aligned}
&\texttt{FloVerCheck}(e, A, I) = \top \Rightarrow \\
&\exists v_{\text{double}}\ v_{\text{real}}\ \delta. \\
&\quad (e, \texttt{toFloVerEnv}\ env) \Downarrow^{\text{real}} v_{\text{real}} \wedge \\
&\quad \texttt{evaluate}(\texttt{floVer2CML}\ e, env) = v_{\text{double}} \wedge \\
&\quad A\ [e] = \delta\ \wedge |v_{\text{double}} - v_{\text{real}}| \leq \delta
\end{aligned}$$

Theorem 4.4 restricts the FloVer expression e to not contain cast operations. This is necessary as while FloVer supports mixed-precision computations, at the time of writing the thesis CakeML supports only 64-bit double floating-point arithmetic. Further, Theorem 4.4 assumes that FloVer has validated the analysis result A to establish the roundoff error bound δ. The roundoff error bound δ relates the execution of the CakeML code produced by `fv2CML` with the idealized real-numbered semantics of the FloVer expression e.

To relate the CakeML code with the real-number semantics of FloVer, we use FloVer's relation to IEEE-754 floating-point semantics as the middle ground. FloVer's idealized finite-precision semantics are related to FloVer's IEEE-754 semantics with a simulation proof, and the IEEE-754 semantics are related to CakeML source code floating-point

semantics with a simulation proof as well. Proving the second simulation required some additional caution because of the interaction of evaluation contexts and double rounding. The main complications of the proof arises in the leaf nodes: loading a variable from the execution environment, and rounding a constant.

When loading a variable, FloVer's semantics model the result as the real-numbered value, perturbed by a constant which is smaller than or equal to the machine epsilon. While sound, FloVer's model is inaccurate and thus over-approximates. On the contrary, CakeML models loading a floating-point constant as a lookup of a 64-bit word in the execution environment without any additional error. To fix this discrepancy between FloVer and CakeML, our simulation proof assumes that all values loaded from the environment are already rounded, i.e., their perturbance is 0. This assumption is not a limitation of libmGen as we discharge it later when proving the whole-program specification by instantiating the environment with CakeML's runtime execution environment, where all values must have been rounded anyway.

The key complication when rounding a constant is that HOL4 did not prove an absolute general error bound for denormal numbers. We extended the formalization of IEEE-754 arithmetic in HOL4 with a proof of an absolute error bound for denormal numbers.

4.4. Evaluation

We have explained libmGen's key components and given an overview of the simulation proofs required for libmGen to automatically implement elementary functions with polynomial approximations in CakeML source code. In this section we want to assess the practicality of libmGen by answering two central questions:

1. Is libmGen applicable to different elementary functions?

2. Are libmGen's accuracy bounds useful?

To the best of our knowledge, there is at this point in time no tool that focuses on verifying custom polynomial approximations inside a verified compiler. In contrast to libmGen, Rlibm and other related tools focus on providing global approximations that work for all possible inputs. Hence the approximation is only generated (and verified) once. For libmGen, its use-case lies in producing a custom approximation for an end-user in different settings, so we consider Rlibm and related tools not an appropriate point of comparison. The alternative approach by Appel and Bertot is not fully automated and thus not a useful comparison either.

Function	Input Range	Runtime	Approx. Err.	Roundoff Err.
$\cos(x)$	$[0.1, 1]$	2186	2.710E-4	6.859E-16
$\sin(x)$	$[0.1, 1]$	2832	2.203E-4	6.781E-16
$\tan(x)^{-1}$	$[-0.5, 0.5]$	2551	4.416E-4	3.196E-16

Table 4.1: Results of compiling example functions to CakeML machine code

Instead, in our evaluation we run libmGen on some simple, random degree three polynomials. We measure the overall running time of libmGen for translating the elementary function to CakeML machine code and proving a specification proof including accuracy bounds. This is merely to assess libmGen's performance, in a real-world scenario, a custom polynomial approximation is use-case-specific and thus libmGen would be used to verify parts of a larger program. We further report the accuracy bounds computed by libmGen.

All of our experiments were performed on a computer running Ubuntu 20.04 with an i7-6820HQ CPU, clocked at 2.7 GHz with 16 GB of RAM. The main reason why we limit the evaluation to polynomials of degree three is to make sure that Dandelion can certify the approximation error within HOL4 without having to resort to using its binary. As we discussed in Section 3.6, Dandelion uses a binary extracted with CakeML to speed-up computations, but this binary is not proof-producing. Hence we cannot use it as part of the libmGen pipeline[1].

To answer both questions of our evaluation we compile polynomial approximations of sin, cos, and $\tan(x)^{-1}$ to CakeML code and we give an overview of the results in Table 4.1. The table shows first the function being implemented by libmGen (column "Function"), then the input range on which the function is approximated (column "Input Range"), the overall running time of libmGen in seconds ("Runtime"), the approximation error from implementing the elementary function as a polynomial ("Approx. Err") and the roundoff error from implementing the polynomial in floating-point arithmetic (column "Roundoff Err").

For all of the example functions, libmGen takes between 30 minutes (1800 seconds) and an hour (3600 seconds). This time is still reasonable as while one may generate polynomial approximations frequently during development, verification with libmGen would only have

[1]Technically, we could use Dandelion's binary as an external oracle in HOL4 and tag each theorem appropriately. But this would clutter all our theorems with an additional tag that the binary must correctly implement the Dandelion specification from HOL4.

to be done once, when code gets closer to production stages. We notice that the combined error for all our examples is dominated by the approximation error of polynomial. This suggests that higher degree polynomials can also be compiled and verified in libmGen with reasonable error bounds, as the roundoff error is still negligible. Overall, the performance bottleneck of libmGen is Dandelion's verification and with improvements in Dandelion's verification algorithm it should be possible to compile polynomials of higher degree into CakeML code with libmGen in the future.

4.5. Related Work

We have demonstrated how libmGen automatically generates CakeML source implementations for elementary functions. In this section, we put libmGen into a greater context and compare it with existing approaches to verified generation of mathematical functions. From a foundational point of view, libmGen shares its related work with Dandelion (Section 3.8) and thus we do not repeat it here. Instead, we focus on approaches that are more closely connected to a (verified) compiler.

To generate the roundoff errors, we relied on FloVer [12], however, this was merely for the convenience of the tool being already implemented in HOL4 and the thus straightforward connection to CakeML. Other solutions could be used equally well [120, 37, 111]. The exact same reasoning also applies to Dandelion [11]. For mere convenience, we chose a HOL4 tool over implementations in other theorem provers [82, 52, 39].

One of the key aspects of libmGen is that it provides correctness proofs relating the mathematical function to its source code implementation. To the best of our knowledge, the only recent approach that takes a similar route to libmGen is the one by Appel and Bertot [5]. Their approach has recently been extended to verify implementations of ODE's in CompCert C [73] by Kellison and Appel [64].

If we leave behind machine-checked proofs, approaches like Rlibm [77], the CoreMath project [110], and CRLibm [33] claim to provide correctly rounded, i.e., most accurate, implementations of different elementary functions. While being the most accurate implementations, their code cannot be used straight-forwardly in a verified compiler without the overhead of first proving their accuracy manually.

All of the above tools provide correctly rounded software implementations. Orthogonally, one can also generate hardware implementations for floating-point arithmetic, as is done in, e.g., the FloPoCo tool [38]. While no formal guarantees about FloPoCo are provided by the tool itself, the generated hardware descriptions could potentially be verified using the tooling built around CakeML's verified hardware platform, called Silver [78].

4.6. Discussion

We have presented libmGen, the second contribution of the thesis. LibmGen is a proof-producing compiler from real-numbered elementary functions to floating-point machine code. The key feature of libmGen is that it proves a specification relating the elementary function to its machine-code implementation via the approximation error and the roundoff error. This concludes the first extension of the thesis.

In the next chapter we will look at verified optimization of floating-point arithmetic. To this end, we first revisit how floating-point semantics are implemented in a verified compiler in the first place.

CHAPTER 5

Icing
Supporting Fast-math Style Optimizations
in a Verified Compiler

This chapter is based on our paper titled *Icing - Supporting Fast-math Style Optimizations in a Verified Compiler* [13], which has been published at CAV'19. The work is a collaboration with Eva Darulova, Magnus Myreen, and Zachary Tatlock. The formal development has been done by me with feedback from all co-authors. They also helped with editing the paper and provided feedback on the writing.

- Section 5.1 is a new introduction written for the thesis.
- Section 5.2 is new and gives a more high-level overview of the contributions.
- Section 5.3 and Section 5.4 are the original versions from the CAV'19 paper with minor rewordings.
- Section 5.5 is an updated version of the CAV'19 paper, specifically, minor rewordings have been applied throughout, and the explanation for the `NaN` special-value check has been extended.
- Section 5.6 is the original version from the CAV'19 paper.
- Section 5.7 is new and puts Icing into the greater context of the thesis work.

5.1. Introduction

The second major contribution of the thesis extends CakeML with a floating-point optimizer. Before we can look at the implementation of the optimizer, we must revisit how CakeML, and verified compilers in general, implement floating-point arithmetic.

Both CakeML and CompCert implement floating-point arithmetic very conservatively. They necessarily implement strict IEEE-754 [58] floating-point arithmetic, which requires the compiler to preserve the bit-level accuracy of floating-point code. This requirement limits floating-point optimizations to very few conservative ones like, e.g., replacing `2 * x` with `x + x`. As such, the compilation of floating-point arithmetic in a verified compiler is often said to preserve the *literal meaning* of the program; the program is not altered by the compiler and translated one-to-one into machine code, preserving its bit-level accuracy.

This strict implementation in verified compilers has a striking dissimilarity to the implementation of floating-point arithmetic in unverified compilers like GCC [40] and LLVM [70]. Both GCC and LLVM, when given the `--ffast-math` compiler flag, aggressively optimize floating-point programs to improve their performance, and we call these optimizations *fast-math-style optimizations*. When optimizing floating-point code, GCC and LLVM generally reorder operations, exploit real-valued identities (like $0 * x = 0$), and perform `fma`-introduction, where they replace a multiplication and an addition $((x * y) + z)$ with a more efficient fused-multiply-add (`fma`) hardware instruction. A key fact about all of these fast-math-style optimizations is that they usually do not preserve the bit-level accuracy of programs.

These performance-oriented optimizations used by GCC and LLVM are only one side of the spectrum. On the other end, we have accuracy improving optimizations, e.g., accurate summation formulas [54], and, in general, tools that improve accuracy by restructuring the program [101]. Exactly as for performance-oriented optimizations, these accuracy-oriented optimizations do not preseve the bit-level accuracy of programs, and thus are currently out-of-scope for verified compilers. One possible solution to both of these problems is to base implementations of floating-point arithmetic in verified compilers on a semantics other than IEEE-754.

In general, as floating-point arithmetic necessarily approximates real-number arithmetic, a programmer should always be able to specify a global accuracy bound for a floating-point program. We further argue that if a programmer optimizes their floating-point code with fast-math-style optimizations they do not care about preserving the bit-level accuracy of their program. Now if we are to redesign IEEE-754 floating-point semantics, we must ensure that the semantics gives the compiler enough freedom to perform both accuracy and performance-oriented optimizations, but also allows to preserve bit-level accuracy if required.

To model this choice, we argue that the floating-point semantics in a verified compiler should be non-deterministic, to ensure that all possible optimization choices of the

compiler can be modeled. But, we also want to ensure that the user has control over which optimizations are applied by the compiler. For example, the `fma` instruction may not be available on a target architecture and thus should not be introduced by the compiler. Thus the semantics should also be parametrized by the optimizations that the compiler is allowed to perform.

In this chapter, we present a proof-of-concept language, *Icing*, with a novel non-deterministic floating-point semantics. The Icing floating-point semantics supports fast-math-style optimizations in a verified compiler and provides fine-grained control over which optimizations are applied where. To this end, Icing has syntactic scoping annotations that allow programmers to selectively optimize parts of a program, or disable them completely. Further, the semantics has a global list of allowed optimizations, which makes it easy to determine which optimizations the compiler may apply.

We demonstrate that the Icing language can handle fast-math-style optimization similar to those used by GCC and LLVM by implementing and verifying three reference optimizers: The first optimizer, called the *IEEE-translator*, preserves the IEEE-754 bit-level accuracy, and ensures that no optimizations may be applied by the compiler, providing a safe fallback. The second optimizer, called the *greedy optimizer*, greedily applies a set of optimizations (with a correctness proof), mimicking the behavior observed by end-users of unverified compilers. The third optimizer, called the *conditional optimizer*, demonstrates how additional optimization criteria can be integrated into the global optimization process.

Contributions. In summary, this chapter contributes:

- the proof-of-concept Icing language with a novel non-deterministic floating-point semantics supporting fast-math-style optimizations;
- an implementation and formalization of Icing and its semantics in the HOL4 theorem prover;
- an implementation and formalization of the three reference optimizers.

In the remainder of this chapter, we first discuss the key ideas of Icing on a high-level (Section 5.2), then we define Icing's syntax, the floating-point rewriter, and semantics (Section 5.3). Next, we model the behavior of existing verified and unverified compilers with the IEEE-translator and the greedy optimizer (Section 5.4), and we define our conditional optimizer that incorporates additional criteria (Section 5.5). In Section 5.6, we put the Icing work into a broader context, and we conclude this chapter with a discussion in Section 5.7.

5.2. Key Ideas of the Icing Language

Before diving into the formalization of Icing, we first briefly review its three key ideas: giving the user fine-grained control over where optimizations are applied, using non-determinism to model floating-point optimizations, and lazily evaluating floating-point expressions to carry syntactic structure into the semantics.

Fine-Grained Control. Unverified compilers, like GCC and LLVM, provide fast-math-style optimizations as a global on-off switch. To use the optimizations, the compiler is invoked with the additional parameter `-ffast-math`, which enables fast-math-style optimizations for *all of the program*. There is no way of restricting optimization to a part of the program. While this may be fine for an unverified compiler, we argue that in a verified compiler users must be able to both guarantee IEEE-754 compliant behavior in, e.g., mission-critical manually-verified code, as well as being able to optimize parts of the program, e.g., codes that are susceptible to input errors and are designed to be robust against bounded noise. Therefore, the first key idea of Icing is to extend floating-point arithmetic expressions with a special annotation, `opt:`, that restricts where optimizations can be applied by the optimizer.

Non-deterministic Floating-Point Semantics. Fast-math-style optimizations are easy to define, as they can be expressed as local, directed rewrites. However, the reasoning why they are applied by the compiler can vary between just plain "greedy" optimization strategies and elaborate techniques relying on additional information computed via static analysis. As such, we want our new floating-point semantics to be able to handle different optimization strategies, as well as settings where no optimization is applied at all.

Therefore the Icing semantics optimizes floating-point programs non-deterministically while evaluating. Optimizations are chosen from a global set of allowed optimizations. With this non-determinism, we can model IEEE-754 executions, as well as prove correctness for arbitrary optimization strategies, as long as the strategy uses only optimizations from the global set of allowed optimizations.

Lazy Evaluation of Floating-Point Expressions. Eager evaluation of floating-point expressions immediately turns expressions like `1.0f + x` into a floating-point word based on the value of `x`. A side-effect of such an eager evaluation is that the resulting floating-point word retains no information about the structure of the original expression. To non-deterministically optimize an expression in the semantics, we must make sure that

$$w\text{: 64-bit floating-point word} \qquad x\text{: String} \qquad n \in \mathbb{N} \qquad b \in \{\texttt{true}, \texttt{false}\}$$
$$\diamond \in \{-, \texttt{sqrt}\} \qquad \circ \in \{+, -, *, /\} \qquad \square \in \{<, \leq, =\}$$

$$\begin{aligned} e_1, e_2, e_3 \quad ::= \quad & w \mid x \mid [e_1, \ldots] \mid e_1[n] \mid \diamond\, e_1 \mid e_1 \circ e_2 \mid \texttt{fma}(e_1, e_2, e_3) \mid \texttt{opt} : (e_1) \mid \\ & \texttt{let}\, x = e_1\, \texttt{in}\, e_2 \mid \texttt{if}\, c\, \texttt{then}\, e_1\, \texttt{else}\, e_2 \mid \texttt{Map}(\lambda x.e_1)e_2 \mid \texttt{Fold}(\lambda x\, y.e_1)e_2e_3 \\ c \quad ::= \quad & b \mid \texttt{isNaN}\, e_1 \mid e_1 \square e_2 \mid \texttt{opt} : (c) \end{aligned}$$

Figure 5.1: Syntax of Icing expressions

the structure of floating-point expressions is preserved. Preserving the structure of a floating-point expression allows the semantics to decide whether or not an optimization should be applied. We solve this problem by lazily evaluating floating-point expressions in Icing, representing them as a tree with floating-point words as leafs.

5.3. The Icing Language

We start our discussion of Icing by defining its syntax and semantics to support fast-math-style optimizations in a verified compiler. Icing is a prototype language whose semantics is designed to be extensible and widely applicable instead of focusing on a particular implementation of fast-math-style optimizations[1]. This allows us to provide a stable interface as the implementation of the compiler changes, as well as supporting different optimization choices in the semantics depending on the compilation target.

5.3.1. Syntax

Icing's syntax is shown in Figure 5.1. In addition to arithmetic, let-bindings and conditionals, Icing supports `fma` instructions, lists ($[e_1 \ldots]$), projections ($e_1[n]$), and `Map` and `Fold` as primitives. Conditional guards consist of boolean constants (b), binary comparisons ($e_1 \square e_2$), and an `isNaN` predicate. Predicate `isNaN` e_1 checks whether e_1 is a so-called *Not-a-Number* (`NaN`) special value. Under the IEEE-754 standard, undefined operations (e.g., square root of a negative number) produce `NaN` results, and most operations propagate `NaN` results when passed a `NaN` argument. It is thus common to add checks for `NaN`s at the source or compiler level.

[1] As we show in the next chapter (Chapter 6), we have fully integrated a slightly revised version of the Icing language with the CakeML compiler. Here, we focus on the key conceptual design decisions of the language, which apply to the CakeML version as well.

	Name	Rewrite	Precondition
1	`fma` introduction	`x * y + z` → `fma (x,y,z)`	*application precond.*
2	∘ associative	`(x ∘ y) ∘ z` → `x ∘ (y ∘ z)`	*application precond.*
3	∘ commutative	`x ∘ y` → `y ∘ x`	*application precond.*
4	double negation	`- (- x)` → `x`	*x well-typed*
5	∗ distributive	`x * (y + z)` → `(x * y) + (x * z)`	*no control dependency on optimization result*
6	`NaN` check removal	`isNaN x` → `false`	`x` *is not a NaN*

Table 5.1: Rewrites currently supported in Icing ($\circ \in \{+, *\}$)

We use the `Map` and `Fold` primitives to show that Icing can be used to express programs beyond arithmetic, while keeping the language simple. Language features like function definitions or general loops do not affect floating-point computations with respect to fast-math-style optimizations and are thus orthogonal.

The `opt:` scoping annotation implements one of the key features of Icing: floating-point semantics are relaxed only for expressions under an `opt:` scope. In this way, `opt:` provides fine-grained control both for expressions and conditional guards.

5.3.2. Optimizations as Rewrites

Before we turn to explaining the semantics of Icing, we have to first look at how fast-math-style optimizations are modeled in Icing. Typically, in unverified compilers, fast-math-style optimizations are local and syntactic, i.e., they are so-called peephole rewrites. In Icing, these optimizations are written as directed rewrites $s \to t$ denoting that any subexpression matching pattern s rewrites to t, where the rewriter instantiates pattern variables in t with a substitution obtained from matching with s. The find-and-replace patterns of a rewrite are terms from the following pattern language, which mirrors Icing syntax:

$$p_1, p_2, p_3 ::= w \mid b \mid x \mid \diamond p_1 \mid p_1 \circ p_2 \mid p_1 \,\square\, p_2 \mid \texttt{fma}\,(p_1, p_2, p_3) \mid \texttt{isNaN}\, p_1$$

Table 5.1 shows the set of rewrites currently supported in our development. While this set does not include all of GCC's fast-math optimizations, it does cover the three primary categories:

- performance and precision improving strength reduction which fuses $x * y + z$ into an `fma` instruction (Rewrite 1)

- reordering based on real-valued identities, here commutativity and associativity of $+, *$; double negation; and distributivity of $*$ (Rewrites 2 - 5)
- simplifying computation based on (assumed) real-valued behavior for computations by removing `NaN` error checks (Rewrite 6)

A key feature of Icing's design is that each rewrite can be guarded by a *rewrite precondition*. We distinguish *compiler rewrite preconditions* and *application rewrite preconditions*. Compiler rewrite preconditions must be true for the rewrite to be correct with respect to Icing semantics. Removing a `NaN` check, for example, can change the runtime behavior of a floating-point program: a previously crashing program may terminate or vice-versa. Thus a `NaN`-check can only be removed if the value can never be a `NaN`.

In contrast, an application rewrite precondition guards a rewrite that can always be proven correct against the Icing semantics, but where a user may still want finer-grained control over when the rewrite is applied. By restricting the context where Icing may fire these rewrites, a user can establish end-to-end properities of their application, e.g., a worst-case roundoff error. The crucial difference is that the compiler preconditions must be discharged before the rewrite can be proven correct against the Icing semantics, whereas the application precondition is an additional restriction limiting where the rewrite is applied for a specific application. Putting this into the global context of proving compiler correctness, an application precondition does not have to be proven to verify the compiler, it is merely a guideline for the compiler to ensure that an optimization is only used if it is considered beneficial. On the contrary, a compiler precondition must always be formally proven to prove compiler correctness, as applying the rewrite crucially depends on the precondition being true.

A key benefit of this design is that *rewrite preconditions can serve as an interface to external tools* to apply optimizations conditionally. This feature enables Icing to address limitations that have prevented previous work from proving fast-math-style optimizations in verified compilers [17] since "The only way to exploit these [floating-point] simplifications while preserving semantics would be to apply them conditionally, based on the results of a static analysis (such as FP interval analysis) that can exclude the problematic cases." ([17], p. 21) In our setting, a static analysis tool can be used to establish an application rewrite precondition, while compiler rewrite preconditions can be discharged during (or potentially after) compilation via static analysis or manual proof.

This design choice essentially decouples the floating-point static analyzer from the general-purpose compiler. One motivation for this decoupling is that the compiler is free to perform hardware-specific rewrites, which source-code-based static analyzers

would generally not be aware of. Furthermore, integrating end-to-end verification of these rewrites into a compiler would require the compiler to always run a global static analysis. For this reason, Icing provides the rewrite-preconditions as an interface which communicates only the necessary information.

Rewrites which duplicate matched subexpressions, e.g., distributing multiplication over addition, required careful design in Icing. Such rewrites can lead to unexpected results if different copies of the duplicated expression are optimized differently; this also complicates their Icing correctness proof. We show how preconditions additionally enabled us to address this challenge in Section 5.5.

To optimize a program, Icing folds a list of rewrites `rws` over a program `e`:

```
rewrite ([],e) = e
rewrite ((s → t)::rws, e) =
 let e' = if matches (e, s) then app (s → t, e) else e in
 rewrite (rws, e')
```

For rewrite `s → t` at the head of `rws`, `rewrite (rws, e)` checks if `s` matches `e`, applies the rewrite if so, and recurses. Function `rewrite` is used in our optimizers in a bottom-up traversal of the AST. Icing users can specify which rewrites may be applied under each distinct `opt:` scope in their code or use a default set (shown in Table 5.1).

5.3.3. Semantics

Next, we explain the semantics of Icing, highlighting two distinguishing features. First, values are represented as trees instead of simple floating-point words, thus delaying evaluation of arithmetic expressions. Second, rewrites in the semantics are applied non-deterministically, thus relaxing floating-point evaluation enough to prove fast-math-style optimizations correct.

We define Icing's semantics as a big-step relation $\rightarrow$ in Figure 5.2. Given

$$\texttt{(cfg, E, e)} \rightarrow v$$

`cfg` is the current configuration of the floating-point optimizer consisting of a list of allowed optimizations as rewrites ($s \rightarrow t$), and a flag tracking whether optimizations are allowed. The semantics only allows optimizations below an `opt:` scope (`OptOk`). Parameter `E` is the (runtime) execution environment mapping free variables to values and `e` an Icing expression. The value v is then the result of evaluating `e` under environment `E` while

$$\frac{}{(\texttt{cfg}, \texttt{E}, \texttt{c}) \to \texttt{c}}\ \texttt{Const} \qquad \frac{}{(\texttt{cfg}, \texttt{E}, \texttt{b}) \to \texttt{b}}\ \texttt{Bool} \qquad \frac{\texttt{E(x)} = r}{(\texttt{cfg}, \texttt{E}, \texttt{x}) \to r}\ \texttt{Var}$$

$$\frac{(\texttt{cfg}, \texttt{E}, \texttt{e}) \to v \qquad (\diamond v, \texttt{cfg})\ \texttt{rewritesTo}\ r}{(\texttt{cfg}, \texttt{E}, \diamond\, \texttt{e}) \to r}\ \texttt{Unary} \qquad \frac{(\texttt{cfg}, \texttt{E}, \texttt{e}) \to v \qquad (\texttt{isNaN}\, v, \texttt{cfg})\ \texttt{rewritesTo}\ r}{(\texttt{cfg}, \texttt{E}, \texttt{isNaN e}) \to r}\ \texttt{isNaN}$$

$$\frac{(\texttt{cfg}, \texttt{E}, \texttt{e}_1) \to v_1 \qquad (\texttt{cfg}, \texttt{E}, \texttt{e}_2) \to v_2 \qquad (v_1 \circ v_2, \texttt{cfg})\ \texttt{rewritesTo}\ r}{(\texttt{cfg}, \texttt{E}, \texttt{e}_1\ \circ\ \texttt{e}_2) \to r}\ \texttt{Binary} \qquad \frac{(\texttt{cfg}, \texttt{E}, \texttt{e}_i) \to v_i \quad i \in 1,2,3 \qquad (\texttt{fma}(v_1, v_2, v_3), \texttt{cfg})\ \texttt{rewritesTo}\ r}{(\texttt{cfg}, \texttt{E}, \texttt{fma(e}_1\texttt{, e}_2\texttt{, e}_3\texttt{)} \to \texttt{r})}\ \texttt{fma}$$

$$\frac{(\texttt{cfg}, \texttt{E}, \texttt{e}_1) \to v_1 \qquad (\texttt{cfg}, \texttt{E[x} \mapsto \texttt{v}_1\texttt{]}, \texttt{e}_2) \to v_2}{(\texttt{cfg}, \texttt{E}, \texttt{let x=e}_1\ \texttt{in e}_2) \to v_2}\ \texttt{Let-bind} \qquad \frac{(\texttt{cfg}, \texttt{E}, \texttt{e}) \to vl \qquad \texttt{n} < |vl| \qquad vl[\texttt{n}] = r}{(\texttt{cfg}, \texttt{E}, \texttt{e [n])} \to r}\ \texttt{Ith}$$

$$\frac{(\texttt{cfg}, \texttt{E}, \texttt{e}_1) \to v_1 \qquad (\texttt{cfg}, \texttt{E}, \texttt{e}_2) \to v_2 \qquad (v_1 \square v_2, \texttt{cfg})\ \texttt{rewritesTo}\ r}{(\texttt{cfg}, \texttt{E}, \texttt{e}_1\ \square\ \texttt{e}_2) \to r}\ \texttt{Compare} \qquad \frac{(\texttt{cfg}, \texttt{E}, \texttt{c}) \to cv \qquad \texttt{cTree2IEEE}\, cv = \texttt{b} \qquad (\texttt{cfg}, \texttt{E}, \texttt{e}_\texttt{b}) \to r}{(\texttt{cfg}, \texttt{E}, \texttt{if c then e}_\texttt{T}\ \texttt{else e}_\texttt{F}) \to r}\ \texttt{If}$$

$$\frac{(\texttt{cfg with OptOk := true}, \texttt{E}, \texttt{e}) \to v}{(\texttt{cfg}, \texttt{E}, \texttt{opt:}\ \texttt{e}) \to v}\ \texttt{Scope}$$

Figure 5.2: Non-determinstic Icing semantics

applying optimizations from `cfg`. We start the explanation of the details of the semantics with its two key ideas.

The first key idea of Icing's semantics is that expressions are not evaluated to (64-bit) floating-point words immediately; the semantics instead evaluates them into *value trees* representing their computation result. As an example, if e_1 evaluates to value tree v_1 and e_2 to v_2, the semantics returns the value tree represented as $v_1 + v_2$ instead of the result of the floating-point addition of (flattened) v_1 and v_2. The syntax of value trees is:

$$\begin{aligned} c &::= b \mid \texttt{isNaN}\, v_1 \mid v_1 \square v_2 \mid \texttt{opt:}\, c \\ v_1, v_2, v_3 &::= w \mid \diamond v_1 \mid v_1 \circ v_2 \mid \texttt{fma}(v_1, v_2, v_3) \mid \texttt{opt:}\, v_1 \end{aligned}$$

Constants are again defined as floating-point words and form the leaves of value trees (variables are replaced with value trees from the execution environment `E`). On top of constants, each *syntactic* floating-point operation in Icing has a *semantic* value tree as its counterpart.

The second key idea of our semantics is that it non-deterministically applies rewrites from the configuration `cfg` *while evaluating* expression `e` instead of just returning its value tree. The semantics models non-deterministic choice of an optimization result for value tree v with the relation `rewritesTo`. Relation $(v, \texttt{cfg})$ `rewritesTo` r holds if either `cfg` allows for optimizations to be applied (`OptOk` = `true`), and value tree v can be rewritten into value tree r using rewrites from the configuration `cfg`; or the configuration does not allow for rewrites to be applied (`OptOk` = `false`), and $v = r$. If `OptOk` is `true`, relation `rewritesTo` non-deterministically picks a (potentially empty) subset of the rewrites from the configuration `cfg` and applies them to value tree v. Rewriting on value trees reuses definitions of syntactic rewriting from Section 5.3.2.

Icing's semantics allows optimizations to occur while evaluating arithmetic and comparison operations. The rules `Unary`, `Binary`, `fma`, `isNaN`, and `Compare` first evaluate argument expressions into value trees. The final result is then non-deterministically chosen from the `rewritesTo` relation for the obtained value tree and the current configuration. Evaluation of `Map`, `Fold`, and let-bindings follows standard textbook evaluation semantics and does not apply optimizations.

Rule `Scope` models the fine-grained control over where optimizations are applied in the semantics. The Icing semantics stores in the current configuration `cfg` that optimizations are allowed in the (sub-)expression `e` (`cfg with OptOk := true`).

Evaluation of a conditional (`if` c `then` e_T `else` e_F) first evaluates the conditional guard c to a value tree cv. Based on value tree cv the semantics picks a branch to continue evaluation in. This eager evaluation for conditionals (in contrast to delaying by leaving them as a value tree) is crucial to integrate Icing with CakeML later, which also eagerly evaluates conditionals. As the value tree cv represents a delayed evaluation of a boolean value, the semantics turns it into a boolean constant when selecting the branch to continue evaluation in. This is done using the functions `cTree2IEEE` and `tree2IEEE`. Function `cTree2IEEE (v)` computes the boolean value, and `tree2IEEE (v)` computes the floating-point word represented by the value tree `v` by applying IEEE-754 arithmetic operations and structural recursion.

Example. We illustrate Icing semantics and how optimizations are applied both in syntax and semantics with the example in Figure 5.3. The example first translates the

```
let v1 = Map (λ x. opt:(x + 3.0)) vi in
let vsum = Fold (λ x y. opt:(x ∗ x + y)) 0.0 v1 in sqrt vsum
```

Figure 5.3: A simple Icing program

input list by 3.0 using a `Map`, and then computes the norm of the translated list with `Fold` and `sqrt`.

Our example program is optimized with commutativity of $+$ ($x + y \to y + x$) and `fma` introduction ($x * y + z \to \texttt{fma}(x, y, z)$). Depending on their order, the function `rewrite` produces different results.

If we first apply commutativity of $+$, and then `fma` introduction, all $+$ operations in our example will be commuted, but no `fma` introduced as the `fma` introduction *syntactically* relies on the expression having the structure $x * y + z$ where x, y, z can be arbitrary. In contrast, if we use the opposite order of rewrites, the second line will be replaced by `let vsum = Fold (λ x y. fma (x,x,y)) 0.0 v1` and commutativity is only applied in the first line.

To illustrate how the semantics applies optimizations, we run the program on the 2D unit vector (`vi = [1.0,1.0]`) in a configuration that contains both rewrites. Consequently, the `Map` application can produce `[1.0 + 3.0, 1.0 + 3.0]`, `[3.0 + 1.0, 3.0 + 1.0]`, and also results like `[1.0 + 3.0, 3.0 + 1.0]` where due to non-determinism the optimization is only applied to the second term. The terms `1.0 + 3.0` and `3.0 + 1.0` correspond to the value trees representing the addition of `1.0` and `3.0`.

If we apply the `Fold` operation to this list, there are even more possible optimization results:

```
[(1.0 + 3.0) ∗ (1.0 + 3.0) + (1.0 + 3.0) ∗ (1.0 + 3.0)],
[(3.0 + 1.0) ∗ (3.0 + 1.0) + (3.0 + 1.0) ∗ (3.0 + 1.0)],
[fma ((3.0 + 1.0), (3.0 + 1.0), (3.0 + 1.0) ∗ (3.0 + 1.0))],
[fma ((1.0 + 3.0), (1.0 + 3.0), (3.0 + 1.0) ∗ (1.0 + 3.0))], ...
```

Again, the results of the semantics include more possible values than those that can be produced by syntactic optimization. The first result is the result of evaluating the initial program without any rewrites, the second result corresponds to syntactically optimizing with commutativity of $+$ and then `fma` introduction, and the third corresponds to using the opposite order syntactically. The last result is only a result of semantic optimizations as commutativity and `fma` introduction are applied to some intermediate results of `Map`,

Theorem 5.1.
Let `E` be an environment, `e` an Icing expression, v a value tree, and `cfg` a configuration.
If `(cfg,E,compileIEEE754 e)` $\rightarrow v$ and `cfg.OptOk` = `false`
then `(cfg with [],E,e)` $\rightarrow v$.

Theorem 5.2.
Let `E` be an environment, `e` an Icing expression, v a value tree, and `cfg1` and `cfg2` a configuration.
If `(cfg1,E,compileIEEE754 e)` $\rightarrow v$,
`cfg1.OptOk` = `false`, and `cfg2.OptOk` = `false`
then `(cfg2,E,compileIEEE754 e)` $\rightarrow v$.

Figure 5.4: Correctness Theorems for `compileIEEE754`

but not all. There is no syntactic application of commutativity and `fma` introduction leading to such results.

5.4. Modeling Existing Compilers in Icing

With the syntax and semantics of Icing defined, we turn to implementing and proving correct two optimizers. The first optimizer models the behavior of verified compilers, like CompCert and CakeML, and the second models the behavior of unverified compilers, like GCC and Clang, respectively. First, we implement the former as a translator of Icing expressions which preserves the IEEE-754 strict meaning of its input expression and does not allow for any further optimizations. Then, the latter is implemented as a greedy optimizer that unconditionally optimizes expressions, as observed for GCC and Clang.

5.4.1. An IEEE-754 Preserving Translator

When compiling safety-critical code or after applying some syntactic optimizations, the user may require the compiler to preserve the strict IEEE-754 meaning of an expression. In contrast, the Icing semantics always optimizes non-deterministically based on the current configuration.

To make sure that an expression both exhibits strict IEEE-754 compliant behavior and that its behavior cannot be changed any further by the compiler, Icing provides the "optimizer" `compileIEEE754`. Function `compileIEEE754` essentiallys *disallows optimizations*

by replacing all optimizable expressions `opt: e` with their non-optimizable counterpart `e`. We state the correctness properties for `compileIEEE754` in Figure 5.4. We have proven once and for all that running function `compileIEEE754` on an expression a) disallows optimizations (Theorem 5.1), and b) evaluation is deterministic (Theorem 5.2).

5.4.2. A Greedy Optimizer

Next, we implement and prove correct an optimizer that mimics the (observed) behavior of GCC and Clang as closely as possible. The optimizer applies `fma` introduction (`fma-intro`), associativity (`assoc`) and commutativity (`comm`) greedily. All these rewrites only have an application rewrite precondition which we instantiate to `true` to apply the rewrites unconstrained.

To give an intuition for greedy optimization, recall the example from Figure 5.3. Greedy optimization does not consider whether applying an optimization is beneficial or not. If the optimization is allowed to be applied and it matches some subexpression of an optimizable expression, it is applied. Thus the order of optimizations matters. Applying the greedy optimizer with the rewrites `[comm, fma-intro, assoc]` to the example, we get:

```
let v1 = Map (λ x. opt:(3.0 + x)) vi in
let vsum = Fold (λ x y. opt:(y + x * x)) 0.0 v1 in sqrt vsum
```

Only commutativity has been applied as associativity does not match and the possibility for an `fma` introduction is ruled out by commutativity. If we reverse the list of optimizations into `[assoc, fma-intro, comm]` we obtain:

```
let v1 = Map (λ x. opt:(3.0 + x)) vi in
let vsum = Fold (λ x y. opt:(fma (x,x,y))) 0.0 v1 in sqrt vsum
```

which we consider to be a more efficient version of the program from Figure 5.3.

Greedy optimization is implemented as the function `optimizeGreedy (rws, e)` which applies the rewrites in `rws` in a bottom-up traversal to expression `e`. Our greedy optimizer combined with the fine-grained control of the `opt` annotations allows the end-user to control *where* optimizations can be applied.

We prove correctness of `optimizeGreedy` with respect to the Icing semantics, i.e., we show that optimizing greedily gives the same result as applying the greedy rewrites in the semantics:[2]

[2] As in many verified compilers, Icing's proofs closely follow the structure of optimizations. Achieving this required careful design and many iterations; we consider the simplicity of Icing's proofs to be a strength of this work.

Theorem 5.3 (`optimizeGreedy` is correct)**.**
Let `E` be an environment, e an Icing expression, v a value tree and `cfg` a configuration.
If `(cfg with [],E,optimizeGreedy ([assoc, comm, fma-intro], e))` $\rightarrow v$
then `(cfg with [assoc, comm, fma-intro],E,e)` $\rightarrow v$.

In our formal development, we do not prove Theorem 5.3 directly. A key issue of a direct proof is that it would be very specific to our particular choice of optimizations for `optimizeGreedy`, i.e., the proof requires showing correctness of a single optimization in the presence of other optimizations being applied potentially. Instead, we simplify the proof of Theorem 5.3 into separate correctness proofs about each optimization and chain these together. To this end, we first prove that applications of the function `rewrite` can be chained together in the semantics:

Lemma 5.1 (`rewrite` is compositional)**.**
Let e be an expression, v a value tree, $s \rightarrow t$ a rewrite, and *rws* a set of rewrites. If the rewrite $s \rightarrow t$ can be correctly simulated in the semantics, and list *rws* can be correctly simulated in the semantics, then the list of rewrites $(s \rightarrow t) :: rws$ can be correctly simulated in the semantics.

From Lemma 5.1, we can conclude that it suffices to prove each optimization correct separately and then chain them together for a global correctness proof about function `rewrite`. Theorem 5.3 is proven with an analogous lemma about function `optimizeGreedy`. Using these two simplifying lemmas, adding, removing and reordering of optimizations in `optimizeGreedy` is as simple as changing the list of rewrites, while preserving correctness of the optimizer.

5.5. A Conditional Optimizer

In the previous section, we implemented two optimizers: An IEEE-754 optimizer, which has the same behavior as CompCert and CakeML, and a greedy optimizer with the (observed) behavior of GCC and Clang. For the greedy optimizer to be useful, Icing's fine-grained control through `opt:` annotations is essential. However, we argue in this section that only using `opt:` annotations is often not enough. Instead, we demonstrate how preconditions can be used as additional constraints on where rewrites are applied, and sketch how preconditions serve as an interface between a compiler and external tools, which can discharge these preconditions.

A crucial first observation is that in many cases whether an optimization is acceptable or not can be captured with a precondition *on the optimization itself*. One example for such an optimization is the removal of `NaN` checks, as a check for a `NaN` should only be removed if the check never succeeds.

For the rewrites supported in Icing, we have previously distinguished between two classes of rewrite preconditions: application rewrite preconditions that give more fine-grained control to a user but are not strictly necessary, and compiler rewrite preconditions that model necessary conditions for a rewrite to be applicable. We argue that both classes of preconditions should be discharged by external tools. Many interesting rewrite preconditions depend on a global analysis. However, running a global analysis as part of a compiler is infeasible, as it would require maintaining separate analyses for each rewrite, which is not likely to scale. Our proposed solution is for Icing to expose an *interface to external tools* via the rewrite preconditions.

We implement this idea in the *conditional optimizer* `optimizeCond` that supports three different applications of fast-math optimizations: applying optimizations `rws` unconstrained (`uncond rws`), applying optimizations if precondition `P` is true (`cond P rws`), and applying optimizations under the assumptions generated by function `A` which should be discharged externally (`assume A rws`). When applying a rewrite `cond`, `optimizeCond` checks whether precondition `P` is true before optimizing, whereas for `assume` the propositions returned by `A` are assumed, and should then be discharged separately via a static analysis or a manual proof.

Correctness of `optimizeCond` relates syntactic optimizations to applying optimizations in the semantics. Similar to `optimizeGreedy`, we designed the proof modularly such that it suffices to prove correct each rewrite individually.

Our optimizer `optimizeCond` takes as arguments first a list of rewrite applications using `uncond`, `cond`, and `assume` then an expression `e`. If the list is empty, we have `optimizeCond ([], e) = e`. Otherwise the rewrite is applied in a bottom-up traversal to `e` and optimization continues recursively. For `uncond`, the rewrites are applied if they match; for `cond P rws` the precondition `P` is checked for the expression being optimized and the rewrites `rws` are applied if `P` is true; for `assume A rws`, the function `A` is evaluated on the expression being optimized. If execution of `A` fails, no optimization is applied. Otherwise, `A` returns a list of assumptions which are logged by the compiler and the rewrites are applied[3].

[3] As their name suggests, these assumptions are faithfully assumed by the correctness proof of `optimizeCond`, and the optimization designer must ensure that not contradictory or false statements are introduced.

Using the interface provided by preconditions, one can prove external theorems showing additional properties of a compiler run using application rewrite preconditions, and external theorems showing how to discharge compiler rewrite preconditions with static analysis tools or a manual proof. We call such external theorems *meta theorems*.

In the following we discuss two possible meta theorems, highlighting key steps required for implementing (and proving) them. A complete implementation consists of two connections: (1) from the compiler to rewrite preconditions and (2) from rewrite preconditions to external tools. We implement (1) independently of any particular tool. As an example for (2) we implement a connection to a roundoff error analysis later in Section 6.5 when integrating Icing with CakeML; in general, meta theorems depend on global analyses which are orthogonal to designing Icing, which is the main focus of this chapter. Several external tools already provide functionality that is a close match to our interface and we sketch possible connections below. We note that for these meta theorems, `optimizeCond` should track the context in which an assumption is made and use the context to express assumptions as *local* program properties. Our current `optimizeCond` implementation does not collect this contextual information yet, as this information at least partially depends on the particular meta theorems desired.

5.5.1. A Logging Compiler for NaN Special Value Checks

We show how a meta theorem can be used to discharge a compiler rewrite precondition on the example of removing a `NaN` check. Generally, removing a `NaN` check can be unsound if the check could have succeeded. Inferring statically whether a value can be a `NaN` special value or not requires either a global static analysis, or a manual proof for all possible executions.

Preconditions are our interface to external tools and for `NaN` check removal, we implement a function `removeNaNcheck e` that returns the assumption that no `NaN` special value can be the result of evaluating the argument expression e. Function `removeNaNcheck` can then be used as part of an `assume` rule for `optimizeCond`.

We have reduced correctness of `optimizeCond` to a series of local correctness proofs exactly as we did for `optimizeGreedy`. As such, it suffices to prove a local, strengthened correctness theorem for `NaN` check removal, showing that if the assumption returned by `removeNaNcheck` is discharged externally (i.e., by the end-user or via static analysis), then we can simulate applying `NaN` check removal syntactically in the Icing semantics:

Theorem 5.4 (`NaN` check removal)**.**
Let `E` be an environment, v a value tree, `cfg` a configuration, and `e` an Icing expression.
If `(cfg, E, isNaN e)` $\rightarrow$ `false` and `(cfg, E, rewrite ([roNaNcheck], isNaN e))` $\rightarrow$ v
then `(cfg with [noNaNcheck], E, isNaN e)` $\rightarrow v$.

The theorem has the additional assumption that expression `isNaN e` evaluates to false, i.e., the result of evaluating expression `e` is not a `NaN`. The assumption is additionally returned as the result of `optimizeCond` since it is faithfully assumed when optimizing. Such assumptions can be discharged by static analyzers like Verasco [62], and Gappa [37].

5.5.2. Proving Roundoff Error Improvement

Rewrites like associativity and distributivity change the results of floating-point programs. One way of capturing this behavior for a single expression is to compute the roundoff error, i.e., the difference between an idealized real-valued and a floating-point execution of the expression.

To compute an upper bound on the roundoff error, various formally verified tools have been implemented [120, 12, 111, 37]. A possible meta theorem is thus to show that applying a particular list of optimizations does not increase the roundoff error of the optimized expression but only decreases or preserves it.

The meta theorem for this example would show that a) all the applied syntactic rewrites can be simulated in the semantics and b) the worst-case roundoff error of the optimized expression is smaller than or equal to the error of the input expression. Our development already proves a) and we sketch the steps necessary to show b) below.

We can leverage these roundoff error analysis tools as application preconditions in a `cond` rule, checking whether a rewrite should be applied or not in `optimizeCond`. For a particular expression `e`, an application precondition (`check (s` $\rightarrow$ `t, e)`) would return true if applying rewrite `s` $\rightarrow$ `t` does not increase the roundoff error of `e`.

Theorem 5.5 (`check` decreases roundoff error)**.**
`(cfg, E, optimizeCond ([Cond (`λ `e.  check (s` $\rightarrow$ `t, e))], e))` $\rightarrow$ v $\Rightarrow$
`(cfg with opts :=  cfg.opts` $\cup$ `{s` $\rightarrow$ `t}, E, e)` $\rightarrow$ v $\wedge$
`error (optimizeCond ([Cond (`λ `e. check (s` $\rightarrow$ `t, e))], e))` $\leq$ `error e`

Implementing `check (s → t, e)` requires computing a roundoff error for expression `e` and one for `e` rewritten with `s → t` where `check (s → t, e)` returns true if and only if the roundoff error has not increased by applying the rewrite. Proving the theorem would require giving a real-valued semantics for Icing, connecting Icing's semantics to the semantics of the roundoff error analysis tool, and a global range analysis on the Icing programs, which can be provided by Verasco and Gappa.

5.5.3. Supporting Distributivity in `optimizeCond`

So far, all rewrites we used in both the greedy optimizer and the conditional optimizer only reorder subexpressions, but never change their number of occurrences. In this section we consider rewrites which introduce additional occurrences of subexpressions, which we dub a *duplicative rewrite*. Two commonly known duplicative rewrites are distributivity of $\times$ over $+$ ($x \times (y + x) \leftrightarrow x \times y + x \times z$) and rewriting a single multiplication into multiple additions ($x \times n \leftrightarrow \sum_{i=1}^{n} x$). As an example, we consider distributivity, and how it is proven correct with respect to the Icing semantics[4]. A compiler might want to use this optimization to apply further strength reductions or `fma` introduction.

As the name suggests, the key issue of duplicative rewrites is that they add new occurrences for a matched subexpression. If we apply $(x \times (y + z) \rightarrow x \times y + x \times z)$ to `e1 * (2 + x)`, we get `e1 * 2 + e1 * x`. The key problem of verifying this result with respect to the Icing semantics is that in the non-deterministic semantics the two occurrences of `e1` can be evaluated to different results as optimizations may be applied to only one occurrence.

Because of this phenomenon, any correctness proof for a duplicative rewrite must match up the two potentially different executions of `e1` in the optimized expression (`e1 * 2 + e1 * x`) with the execution of `e1` in the initial expression (`e1 * (2 + x)`). This can only be achieved by finding a common intermediate optimization (resp. evaluation) result shared by both occurrences of `e1` in `e1 * 2 + e1 * x`.

In Icing, existence of such an intermediate result can only be proven for expressions that do not depend on "eager" evaluation, i.e., which consists of let-bindings and arithmetic. We illustrate the problem using a conditional (`if c then e1 else e2`) as an example. In the Icing semantics, the guard `c` is first evaluated to a value tree `cv`. Next, the semantics evaluates `cv` to a boolean value b using function `cTree2IEEE`. Computing b from `cv` loses

[4] As we will demonstrate, the original CAV'19 paper included a very strict criterion required for proving distributivity correct. However, when integrating Icing with CakeML, we learned that this criterion can be dropped. At the time, we could not foresee some of the low-level changes required when integrating Icing with CakeML, thus we keep the original discussion here for completeness.

the structural information encoded in the value tree `cv` by computing the results of previously delayed arithmetic operations. Due to this loss of information, rewrites that previously matched the structure of `cv` no longer apply to b.

Generally, the problem of eager evaluation interfering with non-deterministic optimization is not a bug in the Icing semantics. On the contrary, our semantics makes this issue explicit, while in other languages it can lead to unexpected behavior (e.g., in GCC's support for distributivity under fast-math). CakeML, for example, also eagerly evaluates conditionals and similarly loses structural information about optimizations that otherwise may have been applied. To fix this issue, one may think that conditionals should be evaluated lazily. However, lazy conditionals in general would only "postpone" the issue until eager evaluation of the conditional expression is required, e.g., for a loop.

As a first reaction, one may choose to optimize with duplicative rewrites only if there are no control dependencies on the expression being optimized. However, this approach may be unsatisfactory as it disallows branching on the results of optimized expressions and requires a verified dependency analysis that must be rerun or incrementally updated after every rewrite, and thus could become a bottleneck for fast-math optimizers. Instead, in Icing, we restrict duplicative rewrites to only fire when pattern variables are matched against program variables, e.g., pattern variables a, b, c only match against program variables `x, y, z`. This restriction to only matching let-bound variables is more scalable, as it can easily be checked syntactically, and allows us to loosen the restriction on control-flow dependence by simply let-binding subexpressions as needed.

5.6. Related Work

We have already hinted throughout the chapter at related work performed inside verified compilers. In this section, we review related work in terms of verified compilation, optimization, and analysis of floating-point code.

Verified Compilation of Floating-Point Programs. CompCert [73] uses a constructive formalization of IEEE-754 arithmetic [17] based on Flocq [18] which allows for verified constant propagation, strength reduction optimizations for divisions by powers of 2, and replacing $x \times 2$ by $x + x$. The situation is similar for CakeML [113] whose floating-point semantics is based on HOL's formalization of IEEE-754 [49, 50]. With Icing, we propose a semantics which allows important floating-point rewrites in a verified compiler by allowing users to specify a larger set of possible behaviors for their source programs. The precondition mechanism serves as an interface to external tools. While

Icing is implemented in HOL, our techniques are not specific to higher-order logic or the details of CakeML and we believe that an analog of our "verified fast-math" approach could easily be ported to CompCert.

The Alive framework [79] has been extended to verify floating-point peephole optimizations [86, 98]. While these tools relax some exceptional (`NaN`) cases, most optimizations still need to preserve "bit-for-bit" IEEE-754 behavior, which precludes valuable rewrites like the `fma` introductions Icing supports.

Optimization of Floating-Point Programs. One interesting approach to increase performance of floating-point programs is to decrease precision at the expense of accuracy in so-called "mixed-precision tuning". For instance, mixed-precision tuning may change parts of a program from double to running in single floating-point precision. Current tools [107, 26, 36, 29], ensure that a user-provided error bound remains satisfied either through dynamic or static analysis techniques. In this work, we consider only uniform 64-bit floating-point precision, but Icing's optimizations are equally applicable to other precisions. An extension of Icing to support mixed-precision computations would be straight-forward. However, supporting optimizations in the style of mixed-precision tuning is not straight-forward as it requires a careful redesign of the value tree datatype, the optimization language and the correctness proofs for the optimizer to properly handle precision assignments as a separate layer of complexity.

Spiral [105] uses real-valued linear algebra identities for rewriting at the algorithmic level to choose a layout which provides the best performance for a particular platform, but due to operation reordering it is not IEEE-754 semantics preserving. Herbie [101] optimizes for accuracy, and not for performance by applying rewrites which are mostly based on real-valued identities. The optimizations performed by Spiral and Herbie go beyond what traditional compilers perform, but they fit our view that it is sometimes beneficial to relax the strict IEEE-754 specification, and could be considered in an extended implementation of Icing. On the other hand, STOKE's floating-point superoptimizer [109] for x86 binaries does not preserve real-valued semantics, and only provides approximate correctness using dynamic analysis.

Analysis and Verification of Floating-Point Programs. Static analysis for bounding roundoff errors of finite-precision computations with respect to a real-valued semantics [111, 34, 80, 92, 45, 37] (some with formal certificates in Coq or HOL), are currently limited to short, mostly straight-line functions and require fine-grained domain annotations at the function level. Whole program accuracy can be formally verified with respect

to a real-valued implementation with substantial user interaction and expertise [106]. Verification of elementary function implementations has also recently been automated, but requires substantial compute resources [72].

On the other hand, static analyses aiming to verify the absence of runtime exceptions like division by zero [14, 23, 61, 62] scale to realistic programs. We believe that such tools can be used to satisfy preconditions and thus Icing would serve as an interface between the compiler and such specialized verification techniques.

The KLEE symbolic execution engine [22] has support for floating-point programs [76] through an interface to Z3's floating-point theory [19]. This theory is also based on IEEE-754 and will thus not be able to verify the kind of optimizations that Icing supports.

5.7. Discussion

In this chapter, we have presented the Icing language, the first non-deterministic floating-point semantics that verifies fast-math-style optimizations. With our two reference optimizers, `compileIEEE754` and `optimizeGreedy`, we have demonstrated that Icing can model the observed behavior of existing verified and unverified compilers.

In the next chapter we demonstrate how Icing can be tightly integrated with CakeML to support fast-math-style optimizations of floating-point code. By integrating a verified roundoff error analysis, CakeML follows the design outlined for the conditional optimizer and we relate optimized floating-point machine code to its unoptimized real-valued counterpart.

CHAPTER 6

RealCake
Verified Compilation and Optimization of Floating-Point Programs in CakeML

This chapter is based on our paper titled *Verified Compilation and Optimization of Floating-Point Programs in CakeML*[9], which has been published at ECOOP'22. The work is a collaboration with Robert Rabe, Eva Darulova, Magnus Myreen, Zachary Tatlock, Ramana Kumar, Yong Kiam Tan, and Anthony Fox. The formal development was done mainly by me. Robert Rabe has helped with the design of the optimizer under my supervision during his internship, Yong Kiam Tan and Ramana Kumar have helped with small proof goals in the optimization correctness proofs, and Anthony Fox has helped with the 64-bit floating-point arithmetic implementation, which is part of the original publication but not included in the thesis. Eva Darulova, Magnus Myreen, and Zachary Tatlock have helped with writing the high-level sections of the paper and provided feedback on the technical write-up.

All sections in this chapter are taken from the ECOOP publication, with minor rewordings and edits. The introduction (Section 6.1) is shortened to avoid some repetition. Throughout, I removed the contribution of extending CakeML with IEEE-754 compliant floating-point arithmetic as I did not contribute to this part. Finally, the paragraph discussing distributivity (Section 6.4.1) is new and was not part of the original publication.

6.1. Introduction

State-of-the-art verified compilers like CakeML [113] for Standard ML and CompCert [74] for C still only have very limited support for floating-point programs: CompCert performs only a few conservative optimizations and, prior to this work, CakeML can only preserve the literal meaning of 64-bit double floating-point programs.

This limited support is in stark contrast to the rich set of `fast-math` optimizations suppported by GCC [40] and LLVM [70]. Fast-math optimizations include reassociating arithmetic, e.g., rewriting $x \times (x \times (x \times x)) \rightarrow (x \times x) \times (x \times x)$ to enable common subexpression elimination; fused-multiply-add (`fma`) introduction, i.e., rewriting $x \times y + z \rightarrow \texttt{fma}(x, y, z)$ for strength reduction and to avoid intermediate rounding; as well as branch folding and dead code elimination by assuming special floating-point values like Not-a-Number (NaN) do not arise.

A key reason, which we have encountered throughout the thesis, why fast-math optimizations are not supported by verified compilers up until now is that both CompCert [17] and CakeML strictly preserve IEEE-754 semantics which disallows such optimizations. However, for many applications strict preservation of IEEE-754 semantics is overly constraining and artificial, preventing useful performance optimizations. Numerical applications are typically *designed* implicitly assuming real-number arithmetic and are only later *implemented* in floating-point arithmetic[1].

Previously, we presented the Icing language [13] as a relaxed, non-deterministic semantics for floating-point expressions that allows a limited set of fast-math-style optimizations to be applied. For our proof-of-concept optimizer, we have formally proven that the optimization result is one of those modeled by the semantics of the initial expression.

However, while Icing formalizes what it means to allow fast-math-style optimizations in a verified compiler, Icing does not bound the accuracy of the resulting, fast-math-optimized code. In general, Icing's correctness theorems describe only the optimizations that a verified compiler can apply to a floating-point expression, but not their effect on overall program behavior. Strictly speaking, the Icing optimizer cannot bound changes in the accuracy of the optimized floating-point expression with respect to the real-valued semantics of the unoptimized expression.

We argue that a verified compiler must provide accuracy guarantees to reasonably support fast-math-style optimizations. Applications in domains such as signal processing [27], embedded controllers [84], and neural networks [47], which could be optimized with fast-math-style optimizations, are designed to operate in noisy environments and can thus tolerate a certain amount of floating-point roundoff error by design; however, this noise has to be *bounded*. For example, a real-number version of an embedded controller is typically proven correct (i.e., stable) with respect to *bounded* implementation noise [85]. At the same time, performance is important and so developers are often indifferent to fine-grained floating-point implementation decisions.

[1] Error-free computation with rational or constructive real [15] libraries is often prohibitively expensive.

To support such potentially safety-critical applications in a verified compiler, we introduce a local, more flexible notion of correctness which we call *error refinement*: a floating-point kernel within an application may be optimized (potentially changing its IEEE-754 behavior) as long as its results remain within a user-specified error bound relative to the implicit real-number semantics.

We formalize error refinement inside the CakeML compiler to support fast-math optimizations *end-to-end*. Our extension, which we call *RealCake*, carries source-level guarantees down to fast-math-optimized executable machine code. That is, our final correctness theorem shows that running the *machine code* for a fast-math-optimized floating-point program under strict IEEE-754 semantics produces a result that is within a programmer-provided error bound w.r.t. the *unoptimized program* evaluated under real-number semantics. While our extension is done in the context of the CakeML compiler, we expect it to carry over to other verified compilers like CompCert as well.

Our first key technical contribution is a relaxed floating-point semantics that allows both fast-math-style optimizations as well as *backward simulation* soundness proofs (as CakeML's semantics requires determinism). RealCake's relaxed semantics preserves the core ideas of our Icing semantics Chapter 5 and models non-deterministic application of an arbitrary number of fast-math rewrites, just as Icing does. However, Icing's non-deterministic semantics cannot be directly added to CakeML. Therefore, we design RealCake's semantics to be more tightly integrated with the CakeML source semantics. This new integration is necessary to prove end-to-end error refinement that relates *unoptimized real-valued* CakeML programs and *optimized floating-point machine code*. RealCake's design furthermore supports function calls, I/O and memory beyond (Icing supported) floating-point expressions and can thus prove error refinement for complete applications.

The second technical contribution is to realize error refinement with *translation validation* [103, 108] using an interface to FloVer, our existing proof-producing roundoff error bound analysis (Section 2.4 and Section 4.3). RealCake automatically composes the error bound proofs with its optimizer's correctness theorems to support fast-math optimizations with semantic and accuracy guarantees within a verified compiler for the first time.

RealCake is primarily designed to support *numerical kernels*: straight-line code as it occurs in (safety-critical) embedded controllers and sensor-processing applications[2]. Often such kernels are evaluated in a control loop or process sensor inputs repeatedly. For such

[2]RealCake nevertheless proves error refinement for whole programs, including I/O (Section 6.6).

programs, both correctness as well as performance are important, and an analysis of the straight-line code is sufficient: the correctness (stability) of the overall programs (and loops) can be shown with, e.g., control-theoretic techniques that rely on the straight-line loop body's errors being ibounded [85, 81]. We do not address some of the orthogonal challenges in bounding the floating-point roundoff error for programs with loops and conditional statements, which remain open research problems [35]; state-of-the-art proof-producing error bound analyses only robustly support straight-line numerical kernels [111, 92, 12]. A key aspect of RealCake's design is the loose coupling between the compiler and error analysis. This loose coupling leads to a clean separation of concerns, which we hope will allow us to switch to more general error analysis methods when such are discovered.

We evaluate RealCake by optimizing all kernels from the standard floating-point arithmetic benchmark suite FPBench [31] (Section 6.6) which can be expressed as input to RealCake, for a total of 51 kernels. During our evaluation we found that CakeML was missing a general optimization that is particularly effective for floating-point programs: global constant lifting. RealCake achieves a (geometric) mean performance improvement for fast-math optimizations of 3% and a maximum improvement of 16% on top of improvements from constant lifting with respect to the unoptimized FPBench kernels. Our additional constant lifting optimization achieves a geometric mean performance improvement of 83% across all benchmarks with speedups of up-to 97%. For all optimized kernels, RealCake formally guarantees that the roundoff error remains within a user-specified bound.

Contributions. To summarize, this paper makes the following contributions:

- the concept of error refinement and its formalization within the CakeML verified compiler (Section 6.2);
- an extension of CakeML with a relaxed non-deterministic floating-point semantics (Section 6.3);
- a fast-math optimizer that is effective in improving the performance of floating-point programs (Section 6.4);
- automated proof tools that soundly bound roundoff errors of (optimized and unoptimized) kernels w.r.t. our new real-number semantics for CakeML (Section 6.5).

```
1  (* target error bound: 2^-5,
2     precondition P:
3     0.0 ≤ x1 ≤ 5.0 ∧ −20.0 ≤ x2 ≤ 5.0 *)
4  fun jetEngine(x1:double, x2:double):double =
5   opt: (let
6   val t = (((3.0 * x1) * x1) + (2.0 * x2)) - x1
7   val t2 = (((3.0 * x1) * x1) - (2.0 * x2)) - x1
8   val d = (x1 * x1) + 1.0
9   val s = t / d
10  val s2 = t2 / d
11  in
12    x1 + (((((((((2.0 * x1) * s) * (s - 3.0)) + ((x1 * x1) * ((4.0 * s) - 6.0))) * d) +
13      (((3.0 * x1) * x1) * s)) + ((x1 * x1) * x1)) + x1) + (3.0 * s2))
14  end)
```

Figure 6.1: Example unoptimized CakeML floating-point kernel. The `opt:` annotation (lines 5) allows developers to selectively apply optimizations. Here, we choose to optimize the entire kernel, but a user may place only part of a program under `opt:` and the rest will be compiled preserving IEEE-754 semantics.

6.2. Overview

We start by demonstrating at a high-level how RealCake works using an example before giving an overview of the RealCake toolchain and our major design decisions.

6.2.1. Example

Figure 6.1 shows `jetEngine`, a straight-line nonlinear embedded controller [85] adapted to CakeML syntax. This controller has been proven to be safe for the dynamical system of a jet engine compressor. That is, Martinez and Tabuada show that if the controller is run with inputs (x_1, x_2) within the bounds given by P (on line 2 of Figure 6.1), then the system will always steer the state variables towards the equilibrium point $(0, 0)$, and thus the system remains stable. This so-called stability proof assumes the control expression to be real-valued, but accounts for a certain amount of bounded error, including measurement and implementation errors, and hence the controller is stable as long as the errors remain below this bound.

For the purpose of this example, we choose 2^{-5} as the bound on the roundoff error for implementing the kernel in floating-point arithmetic (on line 1 of Figure 6.1), which would then be used as the noise bound in the stability proof. Going beyond stability proofs, a

```
1  (* guaranteed error bound: 2^-5,
2     precondition P:
3     0.0 ≤ x1 ≤ 5.0 ∧ −20.0 ≤ x2 ≤ 5.0 *)
4  fun jetEngine(x1:double, x2:double):double =
5   noopt: (let
6   val t = fma((x1+x1)+x1, x1, (x2 + x2) - x1)
7   val t2 = fma((x1+x1)+x1, x1, fma(-2.0, x2, -x1))
8   val d = fma(x1, x1, 1.0)
9   val s = t / d
10  val s2 = t2 / d
11  in
12    x1 + fma(x1 * d, fma((s - 3.0) + (s - 3.0), s, x1 * fma(4.0, s, -6.0)),
13      fma(x1 * x1, ((s + s) + s) + x1, x1 + ((s2 + s2) + s2)))
14  end)
```

Figure 6.2: Example optimized CakeML floating-point kernel. The `noopt:` annotation (lines 5) is added by the compiler and disallows further optimizations.

designer of a controller for a resource-constrained embedded systems is also concerned with performance of the produced code. To summarize, an embedded developer designs a controller, such as `jetEngine`, assuming real-valued arithmetic together with an error bound, and requires that the executed finite-precision code A) correctly implements the control expression, B) is as efficient as possible, and C) meets the error bound.

In a real-world scenario, a kernel like `jetEngine` may be part of a safety critical system and we want to ensure that all guarantees of the stability proof still hold true for its executable version. Thus we would compile the kernel using a verified compiler. Unfortunately, no verified compiler today meets the requirements listed above: While both CompCert [17] and CakeML support floating-point arithmetic, and so ensure A), they do not optimize floating-point programs and cannot prove roundoff error bounds[3].

RealCake, our extension of the CakeML compiler, closes this gap. RealCake automatically optimizes the input kernel into the optimized version shown in Figure 6.2, compiles it down to machine code, and proves the end-to-end correctness theorem that captures both 'traditional' compiler correctness as well as accuracy guarantees.

For this example, RealCake prepares the program for optimization by replacing floating-point subtraction by addition of the inverse ($(4.0 \times s) - 6.0 \rightarrow (4.0 \times s) + (-1 \times 6.0)$), and during optimization, RealCake replaces multiplications by additions ($2.0 \times x1 \rightarrow x1 + x1$),

[3]The implementation and proofs about roundoff errors of the previously presented libmGen (Chapter 4) were developed after RealCake, thus they were not (yet) availabe.

```
fun main () = let
  val args = Commandline.arguments ()
  val a = Double.fromString (List.nth args 1)
  val b = Double.fromString (List.nth args 2)
  val r = jetEngine (a, b)
  in
    TextIO.print (Double.toString r)
  end
```

Figure 6.3: Stand-in main function

and introduces `fma` instructions ($x1 \times x1 + 1.0 \rightarrow \texttt{fma}(x1, x1, 1.0)$) that go beyond IEEE-754 semantics. For this example, RealCake compiles the optimized floating-point kernel (Figure 6.2) together with a simple stand-in main function (Figure 6.3) into a verified binary.

On a Raspberry Pi v3, RealCake improves the performance of our example kernel by 95%. This performance improvement comes from both floating-point specific optimizations, as well as global constant lifting that is not specific but particularly effective for floating-points and that CakeML did not support before (Section 6.6). Such a speedup is important for repeatedly run code such as our embedded controller.

RealCake automatically proves the end-to-end correctness theorem that we formally state as:

Theorem 6.1 (*jetEngine* - Whole program correctness)**.**

$$
\begin{aligned}
&\texttt{jetEngineInputsInPrecond}\ ((s_1, s_2),\ (w_1, w_2),\ P)\ \wedge \\
&\texttt{environmentOk}\ ([\mathit{jetEngine};\ s_1;\ s_2], \mathit{fs})\ \Rightarrow \\
&\exists\, w\ r. \\
&\quad \texttt{CakeMLevaluatesAndPrints}\ (\mathit{jetEngineCode}, s_1, s_2, \mathit{fs})\ (\texttt{toString}\ w)\ \wedge \\
&\quad \texttt{initialFPcodeReturns}\ \mathit{jetEngineUnopt}\ (w_1, w_2)\ w\ \wedge \\
&\quad \texttt{realSemanticsReturns}\ \mathit{jetEngineUnopt}\ (w_1, w_2)\ r\ \wedge \\
&\quad \texttt{abs}\ (\texttt{fpToReal}\ w\ -\ r)\ \leq\ 2^{-5}
\end{aligned}
$$

In Theorem 6.1, *jetEngineCode* refers to the overall program consisting of the *jetEngine* kernel, the stand-in *main* function from Figure 6.3, and the glue-code for I/O; *jetEngine* is the name of the produced binary; and *jetEngineUnopt* is the kernel from Figure 6.1.

At a high-level, Theorem 6.1 relates the behavior of the optimized program with the behavior of the real-number semantics of the initial, unoptimized program on the domain specified by precondition P.[4]

Formally, the theorem states that, if the kernel is run on a pair of two arbitrary input strings s_1 and s_2, representing the double word inputs w_1 and w_2 respectively, and the double words satisfy precondition P from Figure 6.1 (assumption `jetEngineInputsInPrecond`), and if the machine code for the `jetEngine` kernel is run with three command line arguments (the name of the binary, and s_1 and s_2) in an environment with filesystem `fs` (assumption `environmentOk`), then there exists a double-precision floating-point word w and a real-number r such that

(a) running the optimized kernel with the command line arguments prints the word w on stdout[5] (conclusion `CakeMLevaluatesAndPrints`)

(b) w is a result of running the unoptimized `jetEngine` kernel on (w_1, w_2) with optimizations applied by our relaxed semantics, (conclusion `initialFPcodeReturns`) and

(c) running the *initial unoptimized* `jetEngine` kernel under real-number semantics on (w_1, w_2) returns r (conclusion `realSemanticsReturns`) such that $|w - r| \leq 2^{-5}$, where 2^{-5} is the user-given error bound from Figure 6.1.

6.2.2. Overview of CakeML

At its core, RealCake is an extension of the CakeML compiler toolchain. Before giving a high-level overview how RealCake proves Theorem 6.1, we give a high-level overview of the CakeML compiler toolchain [113] built around a verified compiler for (a dialect of) Standard ML (SML). CakeML compiles programs written in SML to x86, ARMv7, ARMv8, MIPS, RISC-V and Silver [78] machine code and is implemented completely in the HOL4 theorem prover [67]. Our work mainly focuses on the compiler part of the CakeML ecosystem.

The behavior of a program written in the CakeML dialect of SML is defined in the CakeML source semantics. This semantics is implemented as a deterministic function

[4] The precondition is important, since roundoff errors directly depend on the ranges of (intermediate) values.

[5] We chose printing to standard output as one option for implementing I/O behavior to show how the error bound proof can be related to I/O behavior. In a real-world setting this could be replaced by other I/O functionality.

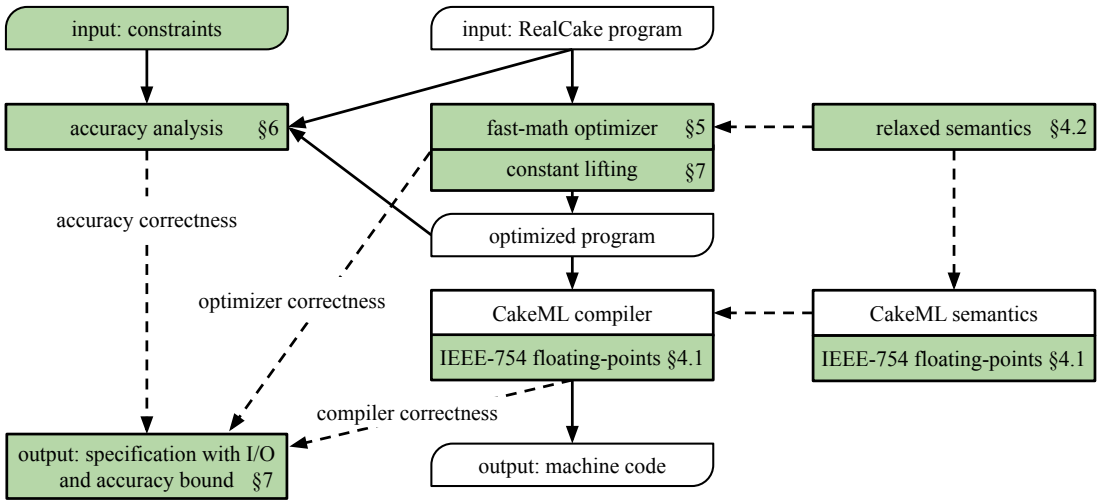

Figure 6.4: Overview of the RealCake toolchain. Boxes with white background are part of the original CakeML toolchain, our RealCake extensions are marked with a green background, dashed lines indicate proof dependencies, solid lines indicate output flows.

in the HOL4 theorem prover, in the style of functional big-step semantics [99]. CakeML programs are turned into machine code using the in-logic compiler, with compiler passes going through various intermediate languages.

The CakeML compiler's correctness theorem states that the compiler preserves observable behaviors of the input program, modulo out-of-memory errors that can occur in the generated machine code [42].

6.2.3. Overview of RealCake

Figure 6.4 illustrates the RealCake toolchain, with the extensions over CakeML marked in green. The RealCake toolchain takes as inputs a program and constraints similar to those in Figure 6.1. In a first step, the fast-math optimizer is run on each floating-point kernel, optimizing it with respect to RealCake's relaxed floating-point semantics, as well as lifting constants with our general constant lifting optimization. As a result we obtain an optimized floating-point kernel, and a proof relating executions of the optimized kernel back to its unoptimized version. Next, the input constraints, and the optimized kernel are run through our accuracy analysis pipeline (left part of Figure 6.4). We have proven once and for all that if the analysis succeeds, the roundoff error between the optimized floating-point kernel and a real-number semantics of its unoptimized version is below

the user specified error bound. This requires non-trivially combining properties of the fast-math optimizer with a simulation proof relating results of the roundoff error analysis to CakeML floating-point programs. Finally, the CakeML compiler compiles the optimized kernel into machine code that can be run on x86-64 and ARMv7 platforms.[6] RealCake automatically combines optimizer correctness with the correctness of the accuracy analysis and the CakeML compiler correctness theorem to prove a theorem about the *I/O behavior* and the *accuracy* of the machine code with respect to the real-number semantics of the unoptimized, initial kernel.

One of the key insights of RealCake is to apply the fast-math optimizations that require reasoning about non-deterministic semantics early; a non-deterministic semantics is integrated more easily with a deterministic verified compiler by resolving the non-determinism before the code enters the compiler itself. We formally prove the correctness of the fast-math optimizer once and for all: if the optimizer turns kernel p_1 into kernel p_2, and evaluating p_2 returns the floating-point word w, then the relaxed floating-point semantics can evaluate and optimize p_1 such that it returns w too.

6.2.4. Error Refinement

Suppose that RealCake would only prove correctness of floating-point optimizations alone, like Icing did. This proof would effectively only capture the machine's point of view, and ignore the programmer's (implicitly) real-valued source semantics. Instead, RealCake proves *error refinement* as RealCake relates the real-numbered, unoptimized program with its fast-math-optimized version. Proving error refinement requires both proving the correctness of our optimizer (i.e., showing that the behavior of the source semantics is preserved), as well as establishing accuracy guarantees using roundoff errors. We chose error refinement as our baseline for correctness proofs as any fast-math optimization necessarily changes the rounding and thus the result value of the floating-point kernel, ruling out alternative bit-wise comparisons.

While the programmer will be indifferent to how exactly the floating-point code is compiled and will accept some roundoff error—or she would not have chosen finite-precision arithmetic in the first place—this roundoff error should not be unduly large

[6] At the time of writing, only the underlying ISA models for x86-64 and ARMv7 support floating-point arithmetic in CakeML.

and make the computed results useless[7]. We argue that a correctness theorem of verified fast-math-optimized floating-point compilation thus needs to capture this error refinement.

To this end, RealCake follows the ideas of libmGen (Chapter 4 and automatically infers verified accuracy bounds via a verified translation from CakeML source to the proof-producing formally verified static analysis tool FloVer (Section 2.4). The key difference between RealCake and libmGen is that libmGen translates FloVer polynomials into CakeML code, while RealCake translates (a subset of) CakeML source into FloVer expressions, i.e., the simulation proofs are performed in different directions. To prove an accuracy bound for RealCake kernels, we combine a simulation proof relating the floating-point semantics of FloVer and CakeML with a proof that all optimizations done by our fast-math optimizer are real-valued identities. RealCake then automatically lifts the roundoff error bound to the complete program and combines it with the general compilation correctness proofs to automatically show the end-to-end correctness theorem for our example (Theorem 6.1). This makes RealCake the first verified compiler for floating-point arithmetic that proves a whole-program specification relating the I/O behavior of optimized floating-point machine code to the real-number semantics of the unoptimized initial program.

We choose to integrate the roundoff error analysis only loosely with CakeML. This gives us a flexible compiler infrastructure that allows us to prove roundoff error bounds on optimized as well as unoptimized floating-point kernels, or to greedily optimize kernels without necessarily proving roundoff error bounds (but still obtaining compiler correctness guarantees). By not tightly integrating the roundoff error analysis into CakeML, we have the option to relatively easily replace FloVer with an extension or another tool in the future.

RealCake's end-to-end correctness theorem only relies on error bounds proven independently for each straight-line kernel instead of a global kernel error bound. Our focus on straight-line kernels is inherited from the current capabilities of verified floating-point error analyses (see Section 2.4 for a more detailed discussion), but can be easily lifted with advances in this area. Per-kernel error analysis, on the other hand, is crucial to maintaining compiler modularity: it is not (nor should it be) the compiler's responsibility to ensure that a program is globally numerically stable—that is a job for the algorithm designer. Rather, the compiler compiles *and optimizes* a program and, in the case of fast-math floating-point optimizations, ensures that it has preserved sufficient (user-provided)

[7]There are programs, such as compensated sum algorithms [55], that explicitly rely on the exact floating-point semantics; such code would not be subject to fast-math-style optimizations and thus not written under an `opt:` scope.

accuracy bounds with respect to a specification over real-number semantics. This can be checked locally.

Similarly, the goal of the accuracy analysis is not necessarily to improve the accuracy of a given kernel, even though introducing `fma`s will generally have this effect, but rather to ensure that the compiler has not introduced unacceptable numerical instability by accident.

Overall, a key challenge of RealCake is proof engineering. RealCake combines a verified roundoff error analysis with the deterministic CakeML compiler and a non-deterministic semantics that supports floating-point optimizations. Specifically, the main proof engineering challenge is getting the different tools to "cooperate". CakeML's source semantics is an integral part of the CakeML ecosystem. Therefore, our integration of the relaxed floating-point semantics must make sure to not break any existing invariants. Further, the semantics of the external roundoff error analysis and the semantics of CakeML source programs must be compatible such that analysis results can be transformed into CakeML source properties. Finally, all of this has to happen while making sure that RealCake optimizes floating-point programs with a non-deterministic relaxed floating-point semantics.

6.3. RealCake's Semantics

Our overall goal for RealCake is to compile and optimize floating-point kernels, establishing verified end-to-end correctness and accuracy guarantees. In this section, we lay the foundations for this work by extending the CakeML compiler with two different semantics: a relaxed floating-point semantics going beyond IEEE-754, and a real-number semantics as a ground truth for bounding errors.

6.3.1. RealCake's Relaxed Floating-Point Semantics

First, we present RealCake's relaxed floating-point semantics. Exactly like our Icing semantics (Chapter 5), the relaxed floating-point semantics applies optimizations during evaluation. In CakeML, we call the process of applying optimizations to floating-point kernels during evaluation *semantic optimization*. Before going into the details of how evaluation and semantic optimization is implemented in the relaxed semantics, we briefly review some necessary details of CakeML's source semantics.

CakeML Source Semantics. The CakeML source semantics is implemented in the style of functional big-step semantics [99]. As such, CakeML source semantics is a pure,

deterministic function, called `evaluate`, in the HOL4 theorem prover and evaluating a CakeML source term has the form

$$\texttt{evaluate}\ (st_1, env, e) = (st_2, r)$$

This result means that evaluating the CakeML source expression e under environment env and global state st_1 results in the global state st_2 and ends with result r. If evaluation succeeded, r is a value, otherwise r is an error. The global states st_1 and st_2 model the state of global references, and interactions with the outside world (e.g., I/O) through the foreign-function-interface (FFI).

We explain the `evaluate` implementation of operator evaluation in more detail, as we extend it with relaxed floating-point operations later. The definition of operator evaluation in CakeML is given in Figure 6.5a. In CakeML source, an operator application is written as `App` *op es*, denoting that operator *op* is applied to the list of expressions *es*.

First, `evaluate` is run on the argument list (line 2 of Figure 6.5a). If evaluation of the argument list fails with an error and a new state, the error and the state are returned (line 3). If evaluation succeeds and returns values *vs*, function `do_app` (line 5) applies operator *op* to the value list *vs* for the current references (st'`.refs`), and the current state of the FFI (st'`.ffi`). Function `do_app` fails if not enough or too many arguments to operator *op* are given in *vs*. Therefore the semantics raises a type error (`Rabort Rtype_error`) if `do_app` fails (line 6). If successful, function `do_app` returns a new state for the global references (*refs*), a new state of the FFI (*ffi*), and a value v. The overall result of the `evaluate` call is then the global state updated with *refs* and *ffi*, and value v (line 7).

Relaxed Floating-Point Semantics. Both the relaxed floating-point semantics and Icing use value trees to represent floating-point values. However, Icing's non-deterministic semantics cannot be directly added to CakeML source, because `evaluate` is a deterministic function. RealCake instead encodes the non-determinism as a deterministic *optimization oracle*. Specifically, RealCake's relaxed floating-point semantics extends the global state with a floating-point optimization oracle:

```
fpState = <| rewrites : optimization list; opts : num → rewriteApp list;
             canOpt : optChoice; choices : num |>
```

In the oracle, `rewrites` stores the currently allowed optimizations; `opts` encodes the oracle decisions of when which optimization is applied; `opts 0` returns all optimizations that are applied next during evaluation of a floating-point expression; `canOpt` records the last optimization scope that has been seen while evaluating and models the fine-grained control.

```
evaluate (st, env, [App op es]) =
 case evaluate (st, env, es) of (st', Rerr v) => (st', Rerr v)
 | (st', Rval vs) =>
  case do_app (st'.refs, st'.ffi, op, vs) of
  None => (st', Rerr (Rabort Rtype_error))
  | Some ((refs, ffi), r) => (updateState (st', refs, ffi), list_result r)
```

(a) Standard operator evaluation

```
evaluate (st, env, [App op es]) =
 case evaluate (st, env, es) of (st', Rerr v) => (st', Rerr v)
 | (st', Rval vs) =>
  case do_app (st'.refs, st'.ffi, op, vs) of
  None => (st', Rerr (Rabort Rtype_error))
  | Some ((refs, ffi), r) =>
  let (st', r_opt) = optimizeIfOk (st', r)
     fp_res = if isFpBool op then toBool r_opt else r_opt
  in (updateState (st', refs, ffi), list_result r)
```

(b) Relaxed floating-point evaluation

```
evaluate (st, env, [FpOptimise ann e]) =
 case evaluate (updateOptFlag st ann, env, [e]) of
 (st', Rerr e) => (resetOptFlag (st', st), Rerr e)
 | (st', Rval vs) => (resetOptFlag (st', st), Rval (addAnnot (ann, vs)))
```

(c) Optimization scope evaluation

```
evaluate (st, env, [App op es]) =
 case evaluate (st, env, es) of (st', Rerr v) => (st', Rerr v)
 | (st', Rval vs) =>
 if ¬realsAllowed st'.fpState then (advanceOracle st', Rerr (Rabort Rtype_error))
 else case do_app (st'.refs, st'.ffi, op, vs) of
  None => (st', Rerr (Rabort Rtype_error))
  | Some ((refs, ffi), r) => (updateState (st', refs, ffi), list_result r)
```

(d) Real-valued operator evaluation

Figure 6.5: HOL4 definitions of operator evaluation in CakeML source (a), relaxed floating-point semantics (b), real-number semantics (d), and evaluation of optimization scopes (c). In (b) and (d) difference to (a) is highlighted in bold font.

The relaxed floating-point semantics optimizes only if `canOpt` is an `opt:` annotation. In `choices` we track the number of optimizations that have been applied. We will use this global counter for integrating the relaxed floating-point semantics with CakeML's proof-producing synthesis.

In principle, RealCake's relaxed floating-point semantics and the Icing semantics model the same set of optimization results, as for each non-deterministic Icing result there exists a deterministic oracle under which RealCake's semantics returns the same value, and vice versa. However, because CakeML source semantics is deterministic and the compiler correctness proofs inherently rely on this fact, supporting floating-point optimizations in CakeML source is only possible with the optimization oracle. Adding the oracle to the global state of the semantics causes the least amount of friction with existing CakeML proofs, while also enabling the non-deterministic simulation proofs from Icing in CakeML via manipulation of the global optimization oracle.

To integrate the relaxed floating-point semantics of RealCake with `evaluate`, we add a separate case for floating-point operators to `evaluate` in Figure 6.5b. As for standard operator evaluation in Figure 6.5a, when evaluating a floating-point operation, `evaluate` first evaluates the arguments, and runs `do_app`. When evaluating a floating-point operation, function `do_app` does not alter the global state (`st'.refs`), and it does not call into the foreign function interface (`st'.ffi`). Function `do_app` simply returns the value tree representing operator `op` applied to the argument values in `vs`. If `do_app` successfully returns value tree `r`, `evaluate` attempts to optimize the value tree. To this end, function `optimizeIfOk` first checks whether the `canOpt` field of the optimization oracle is set to `opt`. If optimizations are allowed, the function performs the optimizations of the oracle (`opts 0`). Then, the optimization oracle is advanced to the next decision, and the global optimization counter `choices` is incremented. Function `optimizeIfOk` returns both the global state updated with the new optimization oracle, and the optimized value tree. If no optimizations are allowed, the function leaves its inputs unchanged. Finally, if `op` is a Boolean comparison of floating-point value trees (`isFpBool op`), `evaluate` turns the resulting value tree into a Boolean constant as CakeML eagerly evaluates control-flow expressions.

For Icing, it was sufficient to eagerly evaluate value trees into floating-point words once a control-flow decision was made (Section 5.3). However, to keep the changes to the CakeML semantics local and manageable, RealCake's relaxed floating-point semantics eagerly evaluates value trees into words as soon as a Boolean comparison is applied to them, even if no control-flow decision is made afterwards.

Figure 6.5c adds the optimization annotations `opt:` and `noopt:` as a separate case to `evaluate`. `FpOptimize annot e` means that expression `e` is evaluated under the optimization scope `annot`, which can either be `opt:` or `noopt:`. Evaluation of an optimization scope replaces the current semantic scope in `canOpt` with the new scope annotation (`updateOptFlag st annot`), before evaluating `e`. Next, the old annotation is recovered by `resetOptFlag(st', st)`. Function `addAnnot(annot, vs)` ensures that all value trees in `vs` are extended with a correct scoping annotation. This is required to ensure that the semantics respects the fine-grained control. If `evaluate` did not add the annotation to the value trees, the semantics could optimize expression `noopt:(x + 2.4)` by first evaluating `x + 2.4` and then optimizing it once the expression is used as part of a larger floating-point expression.

6.3.2. Integrating Relaxed Floating-Point Semantics into the Compiler Toolchain

If we want to fully integrate RealCake's relaxed floating-point semantics with CakeML, we have to also integrate it with the CakeML compiler backend and the tools included in the CakeML compiler toolchain. In the toolchain, a binary implementation of the compiler is obtained by verified bootstrapping [68] of the in-logic compiler using proof-producing synthesis [2]. Furthermore, CakeML source code can be verified using CakeML's program verification tools that rely on characteristic formulae (CF) [46], allowing Hoare-logic like manual proofs (e.g., to verify non-terminating programs [104] and a proof checker for higher-order logic [1]). To prove whole-program specifications (Section 6.5), we integrate RealCake's relaxed floating-point semantics with the compiler backend, the proof-producing synthesis and CF.

CakeML Compiler Backend. A key insight for getting the deterministic compiler proofs to interact nicely with the optimization oracles used in RealCake's relaxed floating-point semantics was to implement the fast-math optimizer as a *source-level* optimization pass, separate from the CakeML compiler backend. The CakeML compiler backend already compiles deterministic 64-bit floating-point kernels to machine code [9] and we reuse this infrastructure by adding a third optimization scope, `strict`, to the relaxed floating-point semantics. Intuitively, the `strict` annotation completely disallows floating-point optimizations in the compiler backend, allowing us to preserve determinism of the source semantics for the correctness proofs.

Any program that is run with the `strict` annotation will never apply optimizations and the program will only perform IEEE-754 correct arithmetic operations. The difference between a `strict` and a `noopt` annotation is that `strict` is "sticky" in the sense that if a program ever enters `strict` mode, evaluation becomes deterministic and cannot escape from it through successive `opt` annotations, while `noopt` and `opt` can be mixed freely; e.g., a program may be under a `noopt` scope, while parts of it are marked with `opt` to selectively apply optimizations.

Proof-Producing Synthesis and CF. The proof-producing synthesis and CF are key components of the CakeML compiler, and required for bootstrapping the compiler. As both crucially depend on how the CakeML source semantics are defined, we have to make sure that the bootstrapping still works, even after adding the relaxed floating-point semantics. Specifically, the synthesis relies on expressions being pure, and thus not altering global state. The crux is that we need the optimization oracle to reside in global state for the backwards simulation proofs. Combining both of these facts, we therefore must ensure that no floating-point optimizations can be applied in code produced by the proof-producing synthesis.

Strictly speaking, we must be able to prove that the code produced by the proof-producing synthesis ignores the state of the floating-point optimizer, i.e., we can always safely replace it by a different optimizer state without changing the result. This restriction effectively limits the proof-producing synthesis to emitting only IEEE-754 correct code that is not subject to fast-math-style optimizations. To prove that the code produced by the proof-producing synthesis has this property, we use the `choices` component of the optimization oracle as it makes optimization attempts by the semantics observable in the global state. We prove a lemma that if optimizations are allowed and the optimization oracle does not change, evaluation cannot have attempted to optimize floating-point code under an `opt:` scope and we can safely replace the oracle with any other floating-point oracle:

Theorem 6.2 (`evaluate` can swap floating-point oracles)**.**

```
evaluate(st, refs, e) = (st', r) ∧
st.fpState.canOpt = opt ∧ st.fpState = st'.fpState ⇒
∀ fp. evaluate(st with fpState = fp, refs, e) =
    (st' with fpState = fp, r)
```

To use this lemma, the synthesis configures the initial optimization oracle to be running under an `opt` scope, with an empty list of optimization choices, essentially allowing us to prove the assumptions of Theorem 6.2. From the conclusion of Theorem 6.2 we can then ultimately establish the invariant of the expression being pure.

We use an optimization counter instead of a Boolean flag, as some of our simulation theorems must combine optimization oracles, while preserving optimization decisions (e.g., when combining oracles for left and right-hand sides of binary operators). In such proofs, the optimization counter gives an exact bound on when the behavior of the oracle must change.

The exact same technique is applied to CF: we make sure that programs reasoned about with CF cannot apply optimizations based on the optimization oracle.

6.3.3. Extending CakeML with Real-Number Arithmetic

The second semantics added to CakeML in RealCake is a real-number semantics used for bounding roundoff errors of floating-point kernels. We extend the CakeML source semantics with support for real numbers and real-number operations by adding a new case to `evaluate`'s operator evaluation in Figure 6.5d. Here, we focus on the real-number semantics. In Section 6.5 we explain how RealCake translates floating-point programs into their real-number counterpart.

Evaluation of real-number operations follows the simple case from Section 6.3.1. The main difference is that we extend the optimization oracle in the global state with an additional flag `real_sem`. Function `realsAllowed st.fpState` checks that the flag is set to true, otherwise evaluation is aborted. The flag disallows real-number operations where necessary, as the real-valued semantics is only used for verification purposes. Further, the compiler does not compile real-valued operations or constants. In the compiler proofs, we rule out real-number operations by assuming that the flag is switched off.

Finally, to preserve invariants of the proof-producing synthesis and CF, the real-number semantics requires a treatment similar to the relaxed-floating-point semantics: When integrating the relaxed floating-point semantics with proof-producing synthesis of CakeML (Section 6.3.2), the global counter `choices` is used to make attempted floating-point optimizations observable, and the global counter is similarly incremented if evaluation of a real-number operation is attempted but fails (function `advanceOracle`).

In this section, we have presented operator evaluation of RealCake as separate functions. In our implementation, when evaluating an `App op es` expression, the CakeML source semantics first does a case split on `op` and chooses whether to apply standard operator

evaluation (Figure 6.5a), relaxed floating-point semantics (Figure 6.5b), or real-number semantics (Figure 6.5d).

6.4. RealCake's Floating-Point Optimizer

Having extended CakeML with relaxed floating-point semantics, we implement a fast-math-style peephole optimizer for RealCake and prove its correctness with respect to the relaxed floating-point semantics. The pseudo-code for our optimizer is given in Figure 6.6. At a high-level, we split optimization into two steps: In step one, function `planOpts` computes which optimizations should be applied to the kernel. We call the list of optimizations returned by `planOpts` the *optimization plan* and refer to this first step as *optimization planning*. In step two, function `applyOpts(plan,e)` applies the optimization plan `plan` to floating-point kernel `e`. Finally, function `noOpts` tags the result with a marker to dissallow further optimizations, which is required to recover the determinism needed by the CakeML compiler proofs. We call this second step *optimization execution*. Instead of verifying optimization planning and optimization execution, we simplify the proof and show only correctness of optimization execution. This approach decouples the implementation of the algorithm that decides which optimizations are applied from the correctness argument.

Optimization Planning. For a floating-point kernel `e`, function `planOpts(e)` returns a list of tuples `(path, opts)`, where the left-hand side `path` is an index into the kernel stating *where* the kernel should be optimized, and the right-hand side `opts` is a list of optimizations stating *how* the kernel should be optimized. The optimization planner `planOpts` is split into the following phases (applied in this order):

- `canonicalForm` puts all floating-point kernels into a canonical shape replacing $x - y$ with $x + ((-1) \times y)$, associating $+, \times$ to the right $((x + y) + z \to x + (y + z))$, and moving constants to right-hand sides with commutativity of $+$ and $\times$.
- `undistribute` replaces expressions like $(x \times y) + (x \times z)$ with $x \times (y + z)$, "undistributing" as much as possible to increase possibilities for `fma` introduction, and reduce the size of the floating-point kernel. The symmetric case of $(y \times x) + (z \times x)$ is ignored by the undistribute phase, as `canonicalForm` rotates all multiplications with commutativity.
- `peepholeOptimize` re-establishes canonical form and applies the optimizations from Figure 6.7.

```
(* Step 1: Which optimizations should be performed *)
def planOpts e = composePlans ([canonicalForm; undistribute; canonicalForm;
                                peepholeOptimize; balanceTrees], e)

(* Step 2: Apply optimizations to kernel *)
def applyOpts (plan,e) = noOpts (fst (optimizeWithPlan (plan,e)))

def optimizeWithPlan (plan, e) =
  case plan of
  | [] => (e, Success)
  | (Expected eOpt):: plan' =>
    if eOpt <> e then (e, Fail) else optimizeWithPlan (plan',e)
  | (path,opts):: plan' =>
  let eOpt = performRewrites (path, opts, e) in
    if eOpt = e then (e, Fail)
    else
      let (eFinal, res) =
         optimizeWithPlan (plan', performRewrites path opts e) in
       if res <> Success and eFinal = eOpt then (e,Fail) else (eFinal, res)

def performRewrites (path, opts, e) =
  if path = Here then rewrite (opts, e)
  else case recurse (path, e) of
      | Some (subpath, subexp) => performRewrites (subpath, opts, subexp)
      | None => e

def rewrite (opts, e) =
  case opts of
  | [] => e
  | (lhs, rhs) :: rws' =>
    if (isPureExp e) then rewrite (opts, apply (lhs,rhs) e) else e
```

Figure 6.6: Pseudo-code for optimization planner and optimization passes

$$x \times 0 \to 0 \qquad x \times 2 \to x + x^* \qquad -(x \times y) \to x \times (-y)^*$$
$$x \times 1 \to x \qquad x \times 3 \to x + (x + x) \qquad x + (-y) \to x - y^*$$
$$x \times -1 \to -x \qquad x + 0 \to x \qquad x \times y + z \to \texttt{fma}(x, y, z)$$
$$x - x \to 0$$

Figure 6.7: Optimizations currently used by the peephole optimization phase, IEEE-754 preserving optimizations are marked with a $*$

- `balanceTrees` reorders sub-expressions in the floating-point kernel by replacing deeply-nested arithmetic expressions like $x_1 + (x_2 + (x_3 + x_4))$ by more shallow versions, such as $(x_1 + x_2) + (x_3 + x_4)$ and similarly for $\times$.[8]

Function `composePlans` concatenates the optimization plans produced by each phase (Figure 6.6, line 2). After each phase, `composePlans` adds an `Expected` annotation into the optimization plan, before the optimizations of the next phase. In the optimization execution phase, when encountering an `Expected eOpt` annotation the optimizer continues optimization only if the current intermediate optimization result is equal to expression `eOpt`. We found this extension very useful when debugging the separate phases as it made the optimizer fail if an incorrect optimization plan was computed.

Optimization Execution. When executing the optimization plan, function `applyOpts` first runs function `optimizeWithPlan` on the plan and its input kernel, where `optimizeWithPlan` applies all elements of a given optimization plan one by one. Function `optimizeWithPlan` optimizes an expression only if it is wrapped under an `opt:` annotation. Further, either all or none of the optimizations in the plan are applied: if optimization fails, then the unoptimized input kernel is returned.

For each element of the plan `(path, opts)`, `optimizeWithPlan` traverses expression `e` following `path` until reaching a sub-expression `e'` (lines 22 and 23 of Figure 6.6) and applies the optimizations `opts` at the end of the path (line 21). Having reached expression `e'` at the end of `path`, function `optimizeWithPlan` calls function `rewrite(e, opts)` (line 26) that applies the optimizations `opts` to the CakeML expression `e'`.

As CakeML source supports stateful features like reference cells, and calls into a foreign-function-interface (FFI), function `rewrite(e, opts)` checks that CakeML expression `e`

[8] We added `balanceTrees` as an optimization pass to simplify register allocations.

Theorem 6.3 (`noOpts` - correctness)**.**

`evaluate` st env [`noOpts e`] $=$ (st_2, r) $\wedge$ st.`fp_state.canOpt` $\neq$ `FPScope Opt` $\wedge$
`isPureExp` e $\Rightarrow$
$\exists$ $choices_2$ r_2.
  `evaluate` (st *with* `fp_state` $:=$ `noOptsApplied` st.`fp_state`) env [e] $=$
  (st_2 *with* `fp_state` $:=$ `noOptsAppliedWithChoices` st.`fp_state` $choices_2, r_2$) $\wedge$
  `noOptSim` r r_2

Theorem 6.4 (`optimizeWithPlan` - correctness)**.**

`evaluate` st_1 env (`optimizeWithPlan` (`planOpts` e) $exps$) $=$ $(st_2,$ `Rval` $r)$ $\wedge$
`allVarsBoundToFPVal` $exps$ env $\wedge$ `flagAndScopeAgree` cfg st_1.`fp_state` $\wedge$
`notInStrictMode` st_1.`fp_state` $\wedge$ `noRealsAllowed` st_1.`fp_state` $\Rightarrow$
$\exists$ $fpOpt$ $choices$ $fpOptR$ $choicesR$.
  `evaluate` (`addOptsAndOracle` st_1 (`getRws` (`planOpts` e)) $fpOpt$ $choices$) env $exps$ $=$
  (`addOptsAndOracle` st_2 (`getRws` (`planOpts` e)) $fpOptR$ $choicesR$, `Rval` r)

Figure 6.8: Correctness theorems for functions `noOpts` and `applyOpts`

is a pure (floating-point) expression. This check, which is implemented as a function `isPureExp e`, effectively rules out optimization of expressions that use any of CakeML's stateful features.

The result of running `optimizeWithPlan` is given to function `noOpts` (line 6 of Figure 6.6). The function performs a bottom-up traversal of expression `e`, replacing any `opt:` annotation with a `noopt:` annotation, disallowing further optimizations and, as a result, making the program's semantics deterministic.

6.4.1. Correctness of the Fast-Math Optimizer

Our optimizer is split into two separate phases, optimization planning, and optimization execution. A key benefit of this split is that we can prove correctness of optimization execution without caring about the exact optimizations contained in the plan. Rather, we verify `applyOpts` for any potential plan generated by our optimization planner. We state the correctness theorem for `applyOpts` in Figure 6.8. At a high-level, we show that the optimizations done by `applyOpts` are correct with respect to the relaxed floating-point semantics, and no further optimizations can be applied afterwards. Accordingly, we split correctness of `applyOpts` into two proofs.

First, we prove in Theorem 6.3 that running the result of `noOpts e` gives the same result as running `e` with an oracle that performs no optimizations. Theorem 6.3 is proven once

and for all via a straightforward structural induction on the argument kernel `e`. The theorem assumes that all expressions given to `noOpts` are pure, however this is not a limitation of the overall result, as the theorem is only used for optimized floating-point kernels which must be pure anyway.

Second, we prove that there is a backwards simulation between the result of `optimizeWithPlan(planOpts e)` and `e`. Theorem 6.4 proves: for the result obtained from evaluating the syntactically optimized kernel, there exists an optimization oracle such that `evaluate` returns the same result when semantically optimizing with the optimizations from the computed plan. The CakeML source semantics is untyped, and thus we assume that all variables are bound to floating-point constants in `exps` (`allVarsBoundToFPVal`). As we did earlier for Icing (Section 5.4.2), instead of proving one global correctness of `optimizeWithPlan` for the overall plan, we reduce the global correctness proof to a series of correctness proofs about the separate phases, and combine them into the overall backwards simulation. Ultimately, the proof boils down to proving a backwards simulation for each rewrite supported by RealCake for function `rewrite`, and composing them to form the overall correctness proof for an arbitrary plan via lifting lemmas.

Distributivity in RealCake. When developing Icing and proving optimizations correct, distributivity was a major challenge because it changes the number of occurrences for a sub-expression (see Section 5.5.3 for a discussion). A key reason for why this was a major challenge was the eager evaluation of conditional expressions in Icing as this forced a "loss of context" in the correctness proof, where optimizations that were applied syntactically could not be applied in the Icing semantics anymore.

For RealCake proving correctness of distributivity turned out to be easier for two reasons: First, we limited the expressions that are syntactically optimized to pure floating-point expressions, i.e., there was no eager execution getting in the way. Second, for the exact setting in RealCake, using relaxed floating-point semantics, we could prove a key lemma that was not provable for Icing, which we state informally below:

> **Theorem 6.5** (`rewrite` hoisting)**.**
> Let `e` be a *pure* CakeML *floating-point* expression and `fpState` a floating-point optimization oracle that allows semantic optimizations (`fpState.canOpt` = `Opt`). Further, evaluating `e` with CakeML semantics returns value tree `fp`.
> Then, there exists an optimization schedule `opts` such that we first evaluate `e` with CakeML semantics using the empty oracle, and then apply the schedule `opts` to the intermediate result to obtain value tree `fp`.

We call Theorem 6.5 "`rewrite` hoisting" as we use it in backwards simulaion proofs to push optimizations to the end of an evaluation, essentially splitting semantic optimization into first evaluating the expression and then applying all optimizations in one go. With this lemma we were able to prove correctness of distributivity with respect to the relaxed floating-point semantics within around 180 LOC.

Extending the Optimizer. Extending the RealCake optimizer requires extending both the implementation of the optimizer and its correctness proof. To add a new peephole optimization, a user adds the optimization to the list of optimization of `peepholeOptimize` and extends the correctness theorem for `peepholeOptimize`. All other theorems need not be changed. We provide a set of lemmas that can be used to reduce the global correctness proof of `peepholeOptimize` to a simple local backwards simulation for the newly-added optimization in terms of the `rewrite` function only. Adding a new phase to `planOpts` is more involved as it requires showing a global correctness theorem for the newly added phase, as well as extending the theorem that splits up correctness of `planOpts` into correctness of its components. The complexity of the first proof depends on the complexity of the phase, whereas splitting up the correctness proof for `planOpts` is a straightforward proof showing that optimizations of the newly added phase are contained in the optimizations applied by `planOpts`.

6.5. Proving Error Refinement with RealCake

CakeML with relaxed floating-point semantics optimizes floating-point kernels and automatically proves a relation between the unoptimized and the optimized kernel. However, we argue that to meaningfully optimize floating-point arithmetic in a verified compiler, the compiler must relate the optimized floating-point program and the unoptimized real-valued program.

To understand why we argue for such an unconventional correctness theorem, we compare floating-point optimizations with classic compiler optimizations. Classic compiler optimizations like constant propagation and dead-code elimination have a clear definition of when they can be applied and one can prove that the optimizations do not change the program result. While floating-point fast-math optimizations obviously also have a clear definition of when they can be applied, they do not follow the intuition of "not changing program results". As an example, we introduce an `fma` instruction in the simple expression `x * 2.9 + 0.05` with relaxed floating-point semantics: `fma(x, 2.9, 0.05)`. The `fma` makes the expression generally faster and locally more accurate, as the result is

only rounded once. Correctness of the fast-math optimizer proves a backwards simulation between the two example expressions, however, the theorem does not capture the change in roundoff errors.

We propose the notion of *error refinement*: the compiler may optimize a floating-point kernel aggressively as long as the results remain within a (given) bound relative to *real-number* semantics. Further, error refinement gives a clear, easy to understand reference semantics for floating-point programs: the idealized real-valued semantics of the unoptimized program. This implicit real-number semantics is closer to what a programmer may have in mind when writing floating-point code, and error refinement respects error bounds that the programmer has already established.

We make this notion of error refinement explicit by implementing a fully automatic pipeline that first computes an upper bound on the roundoff error of a floating-point kernel in CakeML, and then compares it to a user-specified accuracy bound. To implement this pipeline, we use our roundoff error analysis tool FloVer (Section 2.4). We prove the roundoff error bound correct with respect to a run of the original input kernel under an idealized real-number semantics.

6.5.1. Translating RealCake Kernels into FloVer Input

To infer roundoff errors for a RealCake kernel with FloVer, we define a straightforward encoding function `toFloVer e`, translating floating-point kernels with variables, constants, unary and binary floating-point operations, `fma`s, and let bindings into FloVer syntax. Correctness of the translation functions proves once and for all a simulation relating deterministic RealCake floating-point semantics with FloVer's idealized finite-precision semantics. To prove the simulation, our translation function ensures that the kernel is wrapped under a `noopt:` annotation. As roundoff error analysis tools depend on ranges for the input variables, our pipeline also requires a real-number function specifying these input constraints.

RealCake implements a function `isOkError(e, P, err)` that returns true if `err` is a sound upper bound on the worst-case roundoff error for RealCake expression `e` and input constraints `P`. First, the RealCake kernel `e` is translated into FloVer syntax with `toFloVer e`. Function `isOkError` then runs FloVer's unverified inference algorithm (see Section 4.3 for a more detailed discussion) to generate a (untrusted) roundoff error analysis certificate for the FloVer encoding of `e` and input constraints `P`. FloVer's certificate checker automatically checks the certificate, and if the check suceeds, the error bound encoded in

the certificate is correct. Finally, `isOkError` checks that the global upper bound encoded in the certificate is smaller or equal to the user-specified error constraint `err`.

6.5.2. Proving Roundoff Error Bounds for RealCake Kernels

To prove error refinement for an optimized kernel, we connect the soundness theorem of FloVer to RealCake's relaxed floating-point semantics. Together with the idealized real-valued semantics we show once and for all the HOL4 theorem:

> **Theorem 6.6** (CakeML-FloVer roundoff errors)**.**
> Let f a floating-point kernel, P an input constraint for f, err a user-given accuracy bound, $theVars$ the set of free variables of f, vs a list of floating point values, *body* a CakeML expression, and env a CakeML execution environment, then
>
> `isOkError_succeeds` $(f, P, err, theVars, body)$ $\wedge$
> `isPrecondFine` $(theVars,\ vs,\ P)$ $\Rightarrow$
> $\exists\, r\ fp.$
> `realEvals_to` (`realify` *body*, `envWithRealVars` *env* $theVars$ $vs,\ r)$ $\wedge$
> `floatEvals_to` (*body*, `envWithFloatVars` *env* $theVars$ $vs,\ fp)$ $\wedge$
> `abs` (`valueTree2real` $fp\ -\ r) \leq err$

On a high-level, Theorem 6.6 states that if function `isOkError` succeeds, the analyzed function can be run both under floating-point and real-number semantics, and `err` is an upper bound on the roundoff error. The assumptions are: `isOkError` succeeds, and *body* is the function body of the RealCake floating-point kernel f, with the parameters $theVars$ (assumption `isOkError_succeeds`); and the values vs bound to the parameters $theVars$ are within the input constraints P (assumption `isPrecondFine (`$theVars$`, `vs`, `P`)`).

In the conclusion, function `realify` replaces floating-point operations by their real-number counterparts. The theorem then shows that there exists a real number r and a floating-point value tree *fp* such that evaluation of the function under an idealized real-number semantics returns r (`realEvals_to`), evaluation under floating-point semantics returns value tree *fp* (`floatEvals_to`), and err is an upper bound to the roundoff error of kernel f (`abs(valueTree2real` $fp - r) \leq err$).

Error refinement relates the user-given error bound back to a real-number semantics of the initial, unoptimized kernel, but RealCake runs function `isOkError` on the optimized kernel. In addition to Theorem 6.6 we also prove that the optimizations applied by RealCake are real-valued identities. Exactly like we prove correctness of `optimizeWithPlan` in Section 6.4.1, we have proven once and for all a simulation between the real-number

semantics of the optimized kernel and its unoptimized version. The theorem proves that the real-number semantics of optimized floating-point kernel and the unoptimized floating-point kernel are the same. Combining this theorem with Theorem 6.6, we automatically prove error refinement for floating-point kernels.

6.6. Evaluation: Performance and Accuracy Proofs

In this section, we demonstrate RealCake's error refinement proofs, benchmark optimization results, and give an overview of the size of the development. Overall, the RealCake development spans roughly 33k lines of proof-code, composed of the relaxed floating-point semantics and the real-number semantics (∼7k LOC, including proofs), the implementation and correctness proofs for the optimizer (∼20k LOC), and the benchmarks from the evaluation (∼7k LOC).

We evaluate RealCake on 51 benchmarks taken from the standard floating-point benchmark set FPBench [31]. Our evaluation includes all FPBench benchmarks that use floating-point operations that are supported by RealCake and we exclude only those that cannot be expressed in RealCake (for instance we exclude benchmarks with elementary function calls; i.e., functions like sin and cos[9]). We use the preconditions that are already specified in FPBench, but modify them slightly for the `jetEngine` and `n_body` kernels such that FloVer can prove a roundoff error bound and does not report a possible division by zero. Our evaluation shows how RealCake establishes end-to-end correctness proofs, and compares the runtime of the optimized and unoptimized kernels.

6.6.1. Automated End-To-End Proofs

We have translated all 51 FPBench benchmarks into HOL4 script files that are read by RealCake. Each script file defines the original, unoptimized, floating-point kernel, a precondition for the kernel, and a user-provided error bound. For simplicity, our evaluation uses 2^{-5} as the user-provided error bound for all of the benchmarks, though those would be given by the compiler user in a real-world setting.[10]

At the end of each benchmark file, our HOL4 automation fully automatically optimizes the kernel, instantiates Theorem 6.4 for the generated plan, infers a roundoff error bound and compares it to the user-provided error bound. Finally, a whole-program specification

[9] Elementary functions could be added using libmGen (Chapter 4), but this would require some additional machinery beyond error refinement.

[10] If the error bound is choosen too tightly the optimizer may reject every optimization candidate, while a too coarse bound could allow for too aggressive optimizations.

relating the behavior of the machine code for the optimized program to the real-number semantics of the unoptimized program is proven automatically by combining the individual proofs.

RealCake proves the end-to-end correctness theorem (Theorem 6.1) for 45 benchmarks. That is, for these benchmarks it is able to show that the roundoff error of the optimized program is below the specified default error bound of 2^{-5}. For the three `rump` benchmarks and the `test04_dqmom9` benchmark, the computed errors are larger than the user-provided error bound (already for the original unoptimized program), and for the benchmarks `n_bodyXmod` and `n_bodyZmod` FloVer is not able to infer a roundoff error bound as its HOL4 computation becomes stuck, likely due to limitations in the HOL4 real number computations.

We show the errors for the optimized and unoptimized kernels in Table 6.1. "Orig." is the roundoff error for the unoptimized kernel, "fast-math" is the roundoff error for the optimized kernel, and column "Impr." shows the percentage by which the error improved with our fast-math optimizations, i.e., if the number is less than 0% the error has increased, and decreased if it is greater than 0%. We highlight benchmarks where the roundoff error has been decreased by the RealCake optimizer in bold font. While improving the roundoff error is not the goal of our optimizations, `fma` instructions are said to be locally more accurate, and reordering of operations influences roundoff errors too. Hence we evaluate the effect on roundoff errors of our optimization strategy. Overall, we notice that if RealCake can infer a roundoff error, the error of the optimized kernel is usually within the same order of magnitude as the unoptimized version, but in many cases it is actually more accurate.

The benchmarks `delta4`, `delta`, `rigidBody1`, and `rigidBody2` have the largest difference in roundoff errors. By inspecting the generated code we found that in these cases, RealCake has significantly alterted the structure of the kernel. The roundoff error computed for a single kernel is highly influenced by the order of operations, thus we suspect that this large difference is mainly due to operator odering.

6.6.2. Performance Improvements

We compared the performance of unoptimized and RealCake's optimized floating-point kernels. In a first run, we measured wide differences in speedups and slowdowns. By manually inspecting the code, we noticed a missing optimization in CakeML: 64-bit word constants should be pre-allocated (or lifted) to increase performance. Lifting constants is a worthwhile optimization in general, and particularly effective for floating-point programs,

Name	Orig	fast-math	Impr.
bspline3	1.295e-16	1.295e-16	0%
carbonGas	5.688e-08	5.688e-08	0%
cartToPol	2.815e-09	2.463e-09	13%
delta4	4.048e-12	2.028e-13	75%
delta	1.970e-13	2.940e-12	-198%
doppler1	6.534e-13	6.412e-13	2%
doppler2	6.534e-13	1.639e-12	50%
doppler3	1.675e-12	2.680e-13	20%
himmilbeau	3.417e-12	3.003e-12	12%
hypot	2.815e-09	2.463e-09	13%
hypot32	2.815e-09	2.463e-09	13%
i4modified	4.002e-13	4.002e-13	0%
intro_ex	2.220e-10	2.220e-10	0%
jetEngine~	5.209e-08	3.898e-08	25%
kepler0	1.761e-13	1.801e-13	-2%
kepler1	8.397e-13	8.467e-13	-1%
kepler2	4.069e-12	3.973e-12	2%
matDet2	5.107e-12	4.663e-12	9%
matDet	5.107e-12	4.663e-12	9%
n_bodyX~	ERR	ERR	ERR
n_bodyZ~	ERR	ERR	ERR
nonlin1	2.220e-10	2.220e-10	0%
nonlin2	2.657e-09	2.657e-09	0%
pid	7.621e-15	7.727e-15	-1%
predatorPrey	3.395e-16	3.366e-16	1%
rigidBody1	6.565e-11	5.329e-13	80%
rigidBody2	5.579e-13	6.410e-11	-360%
rump_C	4.079e+22	3.859e+22	5%
rump_rev	3.859e+22	3.679e+22	5%
rump_pow	4.079e+22	3.859e+22	5%
runge_kutta4	2.220e-14	2.220e-14	0%
sec4_example	2.657e-09	2.657e-09	0%
sine_newton	7.495e-15	6.275e-15	16%
sineOrder3	1.765e-15	1.765e-15	0%
sine	1.538e-15	1.373e-15	11%
sqroot	1.115e-15	1.059e-15	5%
sqrt_add	1.322e-12	1.322e-12	0%
sum	5.995e-15	5.995e-15	0%
t01_s3	5.995e-15	5.995e-15	0%
t02_s8	9.548e-15	8.438e-15	12%
t03_nl2	4.885e-14	4.885e-14	0%
t04_dqmom9	1.999	1.999	0%
t05_nl1_r4	4.441e-06	4.441e-06	0%
t05_nl1_t2	2.776e-16	2.776e-16	0%
t06_sum41	1.443e-15	1.332e-15	8%
t06_sum42	1.332e-15	1.332e-15	0%
turbine1	1.588e-13	1.541e-13	3%
turbine2	2.213e-13	2.213e-13	0%
turbine3	1.108e-13	1.061e-13	4%
verhulst	8.343e-16	8.343e-16	0%
x_by_xy	2.220e-15	2.220e-15	0%

Table 6.1: Roundoff errors for optimized and unoptimized FPBench benchmarks; benchmarks where we have altered the preconditions to avoid division-by-zero errors are marked with ~; and benchmarks where the roundoff error improves are highlighted in bold font and benchmarks where no end-to-end specification is proven are in italics.

as it is does not change the program's IEEE-754-semantics and floating-point programs usually contain many constants. Thus, we implemented an independent, semantics preserving, global optimization that preallocates 64-bit words as global variables. Our performance evaluation compares three versions of FPBench kernels: the unoptimized version as a baseline, the kernel with preallocated constants, and the kernel after first applying fast-math optimizations and then preallocating constants.

To measure performance, CakeML generates ARMv7 machine code where each numerical kernel is run 10 million times in a loop. Each version of the benchmark, with the core

Name	Orig	Csts	Csts + fast-math	Name	Orig	Csts	Csts + fast-math
bspline3*	18.14	1.75 (91%)	1.75 (91% / 0%)	**rigidBody2**	54.92	5.10 (91%)	4.54 (92% / 11%)
carbonGas *	103.40	3.85 (97%)	3.85 (97% / 0%)	**rump_C**	107.48	6.82 (94%)	6.26 (95% / 9%)
cartToPol	2.05	2.04 (1%)	1.86 (10% / 9%)	**rump_rev**	107.96	6.80 (94%)	6.27 (95% / 8%)
delta4	6.34	6.33 (1%)	6.17 (3% / 3%)	**rump_pow**	112.54	12.34 (90%)	11.57 (90% / 7%)
delta	13.49	13.47 (1%)	11.44 (16% / 16%)	runge_kutta4*	93.46	9.53 (90%)	9.53 (90% / 0%)
doppler1	36.02	3.25 (91%)	3.06 (92% / 6%)	sec4_example*	34.99	2.40 (94%)	2.40 (94% / 0%)
doppler2	36.00	3.25 (91%)	3.06 (92% / 6%)	sineOrder3*	34.86	2.08 (95%)	2.08 (95% / 0%)
doppler3	35.98	3.25 (91%)	3.07 (92% / 6%)	sine_newton*	126.34	10.73 (92%)	10.73 (92% / 0%)
himmilbeau	36.13	3.36 (91%)	3.05 (92% / 10%)	sine*	55.36	6.03 (90%)	6.03 (90% / 0%)
hypot32	2.04	2.04 (1%)	1.86 (10% / 9%)	**sqroot**	87.06	4.85 (95%)	4.65 (95% / 5%)
hypot	2.05	2.05 (1%)	1.86 (10% / 10%)	sqrt_add*	35.21	2.59 (93%)	2.59 (93% / 0%)
i4modified*	1.77	1.78 (0%)	1.78 (0% / 0%)	sum*	3.07	3.07 (1%)	3.07 (1% / 0%)
intro_ex*	17.73	1.32 (93%)	1.32 (93% / 0%)	t01_s3*	3.07	3.08 (0%)	3.08 (0% / 0%)
jetEngine~	195.99	11.89 (94%)	11.12 (95% / 7%)	t02_s8*	3.04	3.05 (0%)	3.05 (0% / 0%)
kepler0	5.32	5.31 (1%)	5.30 (1% / 1%)	t03_nl2*	1.78	1.78 (1%)	1.78 (1% / 0%)
kepler1	8.19	8.20 (0%)	8.16 (1% / 1%)	**t04_dqmom9**	163.82	11.76 (93%)	10.20 (94% / 14%)
kepler2	12.43	12.41 (1%)	12.22 (2% / 2%)	t05_nl1_r4*	34.67	2.06 (95%)	2.06 (95% / 0%)
matDet2	6.38	6.37 (1%)	5.67 (12% / 12%)	t05_nl1_test2*	34.00	1.54 (96%)	1.54 (96% / 0%)
matDet	6.37	6.38 (0%)	5.65 (12% / 12%)	t06_sum41*	1.70	1.70 (0%)	1.70 (0% / 0%)
n_bodyX~	38.46	5.20 (87%)	5.06 (87% / 3%)	t06_sum42*	1.68	1.67 (1%)	1.67 (1% / 0%)
n_bodyZ~	38.40	5.27 (87%)	5.15 (87% / 0%)	turbine1*	121.02	5.29 (96%)	5.29 (96% / 0%)
nonlin1*	17.72	1.31 (93%)	1.31 (93% / 0%)	turbine2*	69.90	3.94 (95%)	3.94 (95% / 0%)
nonlin2*	35.06	2.41 (94%)	2.41 (94% / 0%)	turbine3*	121.29	5.28 (96%)	5.28 (96% / 0%)
pid*	104.11	4.72 (96%)	4.72 (96% / 0%)	verhulst*	51.28	2.27 (96%)	2.27 (96% / 0%)
predatorPrey	52.25	2.81 (95%)	3.08 (95% / -9%)	x_by_xy*	1.51	1.51 (1%)	1.51 (1% / 0%)
rigidBody1*	19.11	2.78 (86%)	2.78 (86% / 0%)				

Table 6.2: Running times for FPBench benchmarks; benchmarks where we have altered the preconditions to avoid division-by-zero errors are marked with ~; and benchmarks where performance improves *with fast-math optimizations* are highlighted in bold

loop running the kernel 10 million times, is run three times on five different sets of inputs, for a total of fifteen runs per benchmark.

We run the ARMv7 code on a Raspberry Pi v3 and summarize the results in Table 6.2. Column "Orig." shows the average running time of the (10 million iterations of the) unoptimized program in seconds. Column "Csts." shows the average running time of the program with preallocated constants in seconds plus the relative speedup in percent. And column "Csts. + fast-math" shows the average running time of the program in seconds when first running RealCake's optimizer and then preallocating constants; the column also shows first the relative speedup in percent with respect to the unoptimized program,

and second the relative speedup with respect to the version with preallocated constants. We mark benchmarks with a performance improvement of more than 1% of the fast-math optimizations with respect to preallocating constants in bold (we identified a difference within ±1% to be noise).

Initially, some benchmarks experienced slowdowns of up to 20%. Via manual inspection, we noticed that the fast-math optimizer created too many instructions. As a simple heuristic to prevent this problem, RealCake sums the arities of the floating-point operators in the program versions, and returns the unoptimized version if the heuristic value of the fast-math optimized program is greater than or equal to the unoptimized program. Even if the heuristic rejects an optimization, RealCake computes roundoff errors for both program versions and proves an end-to-end specification theorem about the optimized program. In total, the heuristic rejects optimizations for 27 benchmarks, and we mark them with a $*$ in Table 6.2.

Overall the evaluation shows that preallocating constants is a valuable optimization for CakeML on its own. On top of this, our fast-math optimizer is able to improve the performance for 20 benchmarks and for 7 of those significantly ($> 10\%$). This is remarkable, since the FPBench benchmarks are carefully hand-written and do not target optimizations specifically and are not representative of, e.g., automatically generated code from tools such as Matlab that would be used in the development of embedded system kernels.

For one benchmark we notice a slowdown of 9% even with our heuristic, and the program versions differ only by a single `fma` instruction. We suspect that this slowdown is due to bad pipelining on the Raspberry Pi.

RealCake's constant preallocation achieves a geometric mean speedup of 83%, and the geometric mean of the speedup for RealCake's optimizer compared with the program with preallocated constants is 3%. The maximum speedup achieved with preallocating constants only is 97%, and we notice no slowdowns. When applying fast-math optimization, the greatest slowdown is -9%, and the maximum speedup is 16%.

In general, benchmarks with higher speed-ups from our optimization strategy usually provide many opportunities to both introduce `fma` instructions, and remove constants. We think that the foundational work in RealCake facilitates exploration of other optimization strategies in the future.

6.7. Related Work

In the previous sections we have already hinted at the immediate related work of RealCake. This section puts the RealCake work into a broader context among three axis: verified compilation of floating-point programs, verification of floating-point programs, and optimization of floating-point programs.

Verified Compilation of Floating-Point Programs. Besides CakeML, CompCert [74] is the other major available verified compiler, compiling imperative C programs. CompCert supports floating-point programs [17] following the strict IEEE-754 semantics [58]. This semantics allows it to perform a few small optimizations that are IEEE-754 compliant such as constant propagation and replacing a multiplication by two by an addition ($x \times 2 \rightarrow x + x$).

RealCake supports additional optimizations based on real-valued identities that are not IEEE-754 compliant. While our implementation is done in the context of CakeML and verified in HOL4, the principles of RealCake are independent of the particular programming language that is being compiled and should thus be portable to CompCert as well.

The Alive framework [79] provides a way to specify and prove correct peephole optimizations for C++ code that can be applied in an LLVM pass. Alive verifies optimizations using SMT solvers and has been extended to bit-precise floating-point optimizations and optimizations involving special values, satisfying the IEEE-754 standard [86, 98]. These optimizations are complementary to RealCake's optimizations. Formal verification of Alive's peephole optimizations is addressed separately by the AliveInLean project [71]. The VELLVM project [124] provides a rigorous semantics for LLVM IR semantics to reason about optimizations and implements IEEE-754-preserving floating-point arithmetic.

Verification of Floating-Point Programs. Besides FloVer, there are several other tools that provide formally verified roundoff error bounds for floating-point arithmetic expressions: FPTaylor [111], real2Float [80], Precisa [92], and Gappa [37]. Each of these can in principle replace FloVer in RealCake and we chose FloVer for convenience as it is implemented in HOL4.

Verification of floating-point programs that go beyond numerical kernels is still relatively limited. The above-mentioned automated tools, for instance, do not consider function calls, and techniques for loops are very restricted [92, 35], and thus require the user to provide range annotations for each function call, as well as loop invariants in general.

Entire programs have been manually formally verified w.r.t. a real-valued specification, but with considerable human effort [106, 16], which is not suitable for a compilation setting. The work by Kellison and Appel [64] is a novel approach to verifying the compilation of ODE's down to CompCert C floating-point code and provides accuracy guarantees just like RealCake. However, their technique still involves manual proofs.

If we only require proofs of the absence of runtime exceptions, resp. absence of special values, then abstract interpretation-based techniques do scale to larger programs [61] and some provide formal verification [62].

Optimization of Floating-Point Programs. Floating-point optimizations have also been considered outside of the traditional compiler context, where most of them focused on performance optimization.

Precimonious [107] performs mixed-precision tuning, by determining which operations can be implemented in a lower or higher precision, while satifying a user-provided error bound. While Precimonious can handle short programs with loops, it cannot guarantee the error bound as it uses a dynamic error analysis. Both FPTuner [26] and Daisy [34] perform mixed-precision tuning while providing accuracy guarantees using a static analysis, but can only handle relatively short straight-line expressions. Mixed-precision tuning requires a global error analysis and is thus not suitable to be performed inside a fundamentally modular compiler. However, the precision-tuned program could be further optimized by a (verified) compiler. While RealCake currently only supports double precision floating-point arithmetic, an extension with (uniform) single precision requires merely some engineering work.

Several tools improve the performance or accuracy of floating-point programs by rewriting with real-valued identities. Spiral [105] rewrites linear algebra kernels to improve their performance on a particular hardware platform. Spiral does not take into account roundoff errors; its rewrites are not IEEE-754-preserving, but it does not quantify the errors. The HELIX project [123] uses Spiral as an external oracle for building a verified optimization pipeline for dataflow optimizations. Optimizations in HELIX are done with respect to real-number semantics which is orthogonal to RealCake's floating-point peephole optimizer.

Herbie [101] aims to improve the accuracy, instead of performance, of floating-point arithmetic expressions but estimates roundoff errors unsoundly using a dynamic analysis. The Salsa [30] tool applies a set of transformation rules to improve performance while soundly tracking roundoff errors. Finally, Daisy [34] first applies rewriting similar to Herbie in order to improve performance gains due to mixed-precision tuning. Still

further away is the tool STOKE [109], which generates small floating-point kernels by superoptimization, but which does not even guarantee real-valued equivalence. We consider these optimizations to be orthogonal to the fast-math optimizations that we consider in RealCake. We note that the scoping mechanism allows RealCake to easily integrate parts of the code that have been heavily optimized, and that thus should not be modified further by the compiler.

CHAPTER 7

Conclusion

We have pointed out two central deficiencies of how floating-point arithmetic is implemented in verified compilers: floating-point machine code is not related to its real-number equivalent; and floating-point machine code is produced by an inflexible one-to-one translation, ruling out interesting optimizations. To remedy these problems, we have presented two complimentary extensions of the CakeML compiler.

The first extension implements support for elementary functions in CakeML. We started by presenting Dandelion, which fully automatically verifies polynomial approximations of elementary functions computed using a Remez-like algorithm. Using CakeML's proof-producing synthesis we have compiled Dandelion into a verified binary that makes certificate checking fast. By combining Dandelion with the floating-point analysis tool Flover and relying on CakeML's implementation of floating-point arithmetic, we then presented libmGen. Our CakeML extension libmGen combines the correctness theorems and implementations of Dandelion, FloVer, and CakeML to compile real-numbered elementary functions into floating-point machine code with an accuracy bound. This accuracy bound accounts for both sources of error when working with floating-point arithmetic: the approximation error due to implementing the elementary function as a real-valued polynomial, and the roundoff error due to implementing the real-valued polynomial in floating-point code.

The second extension adds support for floating-point performance optimization to CakeML. We started by presenting our novel relaxed floating-point semantics called Icing. The Icing semantics is an extension of IEEE-754 floating-point semantics that verifies fast-math-style floating-point optimizations. Icing can model the behavior of both existing verified and unverified compilers and, through its precondition interface, we designed an infrastructure to integrate roundoff error reasoning with the Icing optimizations. We then develop the RealCake compiler, integrating Icing with CakeML. One of the key insights of RealCake is error refinement, which allows RealCake to relate optimized floating-point

machine code with its unoptimized, real-valued counterpart. In our evaluation we have shown that RealCake can achieve significant performance improvements for interesting floating-point kernels while preserving user-given accuracy bounds.

To summarize, the foundational work presented in the thesis greatly improves the support for floating-point arithmetic in the CakeML compiler. If we look back at our initial motivating example, a kernel computing the x-coordinate when converting polar to cartesian coordinates, we can use the tools presented in the thesis to verifiedly implement the kernel in floating-point machine code with accuracy proofs all the way through. These accuracy proofs relate CakeML-generated optimized floating-point machine code to its unoptimized real-number equivalent, including elementary functions. Currently, the two accuracy guarantees provided by libmGen and RealCake are separate. However, to obtain an overall final result combining both extensions of the thesis, we expect that it suffices to provide a good connecting lemma relating libmGen and RealCake.

With the foundational work done in this thesis, an immediate next step is to investigate whether the presented ideas apply to other representations of real-numbers, e.g., fixed-point arithmetic [122]. On the application side, an interesting research question for the presented work is whether the combination of libmGen, RealCake, and the CakeML ecosystem can be used to verify interesting safety-critical programs, and potentially even some machine-learning code. This would extend the work from the accuracy guarantees that relate floating-point code to its real-number equivalent to whole-system-specifications. Such a whole-system-specification may include these accuracy guarantees but should also include guarantees about more application-specific properties like, e.g., liveness, safety, and possibly timing guarantees, depending on the target domain. The tools presented in the thesis could then be used for automatically inferring necessary accuracy guarantees that support the overall specification proof. We hope that such a specification could be proven in terms of the real-number semantics only, abstracting away from the concrete floating-point implementation.

CHAPTER 8

Bibliography

[1] Oskar Abrahamsson. "A Verified Proof Checker for Higher-Order Logic". In: *Journal of Logical and Algebraic Methods in Programming* 112 (2020). DOI: 10.1016/j.jlamp.2020.100530.

[2] Oskar Abrahamsson, Son Ho, Hrutvik Kanabar, Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Yong Kiam Tan. "Proof-Producing Synthesis of CakeML from Monadic HOL Functions". In: *Journal of Automated Reasoning (JAR)* (2020), pp. 1287–1306. DOI: 10.1007/s10817-020-09559-8. URL: https://rdcu.be/b4FrU.

[3] Behzad Akbarpour, Amr Abdel-Hamid, Sofiène Tahar, and John Harrison. "Verifying a Synthesized Implementation of IEEE-754 Floating-Point Exponential Function using HOL". In: *The Computer Journal* 53 (2010), pp. 465–488. DOI: 10.1093/comjnl/bxp023.

[4] Behzad Akbarpour and Lawrence Charles Paulson. "MetiTarski: An Automatic Theorem Prover for Real-Valued Special Functions". In: *Journal of Automated Reasoning (JAR)* 44.3 (2010), pp. 175–205. DOI: 10.1007/s10817-009-9149-2.

[5] Andrew Appel and Yves Bertot. "C Floating-Point Proofs Layered with VST and Flocq". In: *Journal of Formalized Reasoning* 13.1 (2020), pp. 1–16. DOI: 10.6092/issn.1972-5787/11442.

[6] Andrew W Appel. "Verified Software Toolchain". In: *European Symposium on Programming (ESOP)*. 2011.

[7] Andrew W Appel, Sandrine Blazy, Xavier Leroy, and Gordon Stewart. *Program Logics for Certified Compilers*. Cambridge University Press, 2014. ISBN: 978-1-10-704801-0.

[8] Heiko Becker, Pavel Panchekha, Eva Darulova, and Zachary Tatlock. "Combining Tools for Optimization and Analysis of Floating-Point Computations". In: *Symposium on Formal Methods (FM)*. 2018. DOI: 10.1007/978-3-319-95582-7_21.

[9] Heiko Becker, Robert Rabe, Eva Darulova, Magnus O Myreen, Zachary Tatlock, Ramana Kumar, Yong Kiam Tan, and Anthony Fox. "Verified Compilation and Optimization of Floating-Point Programs in CakeML". In: *European Conference on Object-Oriented Programming (ECOOP)*. 2022. DOI: 10.4230/LIPIcs.ECOOP.2022.1.

[10] Heiko Becker, Robert Rabe, Eva Darulova, Magnus O. Myreen, Zachary Tatlock, Ramana Kumar, Yong Kiam Tan, and Anthony Fox. "Verified Compilation and Optimization of Floating-Point Programs in CakeML (Artifact)". In: *Dagstuhl Artifacts Series* 8.2 (2022). DOI: 10.4230/DARTS.8.2.10.

[11] Heiko Becker, Mohit Tekriwal, Eva Darulova, Anastasia Volkova, and Jean-Baptiste Jeannin. "Dandelion: Certified Approximations of Elementary Functions". In: *Conference on Interactive Theorem Proving (ITP)*. 2022. DOI: 10.4230/LIPIcs.ITP.2022.6.

[12] Heiko Becker, Nikita Zyuzin, Raphaël Monat, Eva Darulova, Magnus O Myreen, and Anthony Fox. "A Verified Certificate Checker for Finite-Precision Error Bounds in Coq and HOL4". In: *Conference on Formal Methods in Computer Aided Design (FMCAD)*. 2018. DOI: 10.23919/FMCAD.2018.8603019.

[13] Becker, Heiko and Darulova, Eva and Myreen, Magnus O. and Tatlock, Zachary. "Icing: Supporting Fast-Math Style Optimizations in a Verified Compiler". In: *Conference on Computer Aided Verification (CAV)*. 2019. DOI: 10.1007/978-3-030-25543-5_10.

[14] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. "A Static Analyzer for Large Safety-Critical Software". In: *Conference on Programming Language Design and Implementation (PLDI)*. 2003. DOI: 10.1145/781131.781153.

[15] Hans-J. Boehm. "Towards an API for the Real Numbers". In: *Conference on Programming Language Design and Implementation (PLDI)*. 2020. DOI: 10.1145/3385412.3386037.

[16] Sylvie Boldo, François Clément, Jean-Christophe Filliâtre, Micaela Mayero, Guillaume Melquiond, and Pierre Weis. "Wave Equation Numerical Resolution: A

Comprehensive Mechanized Proof of a C Program". In: *Journal of Automated Reasoning* 50 (2013), pp. 423–456. DOI: 10.1007/s10817-012-9255-4.

[17] Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. "Verified Compilation of Floating-Point Computations". In: *Journal of Automated Reasoning* 54.2 (2015), pp. 135–163. DOI: 10.1007/s10817-014-9317-x.

[18] Boldo, Sylvie and Melquiond, Guillaume. "Flocq: A Unified Library for Proving Floating-Point Algorithms in Coq". In: *Symposium on Computer Arithmetic (ARITH)*. 2011. DOI: 10.1109/ARITH.2011.40.

[19] Martin Brain, Cesare Tinelli, Philipp Ruemmer, and Thomas Wahl. "An Automatable Formal Semantics for IEEE-754 Floating-Point Arithmetic". In: *Symposium on Computer Arithmetic (ARITH)*. 2015. DOI: 10.1109/ARITH.2015.26.

[20] Florent Bréhard, Assia Mahboubi, and Damien Pous. "A Certificate-Based Approach to Formally Verified Approximations". In: *Conference on Interactive Theorem Proving (ITP)*. 2019. DOI: 10.4230/LIPIcs.ITP.2019.8.

[21] Nicolas Brisebarre and Sylvain Chevillard. "Efficient polynomial L-approximations". In: *Symposium on Computer Arithmetic (ARITH)*. 2007. DOI: 10.1109/ARITH.2007.17.

[22] Cristian Cadar, Daniel Dunbar, and Dawson Engler. "KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs". In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2008. URL: https://www.usenix.org/legacy/events/osdi08/tech/full%5C_papers/cadar/cadar.pdf.

[23] Liqian Chen, Antoine Miné, and Patrick Cousot. "A Sound Floating-Point Polyhedra Abstract Domain". In: *Asian Symposium on Programming Languages and Systems (APLAS)*. 2008. DOI: 10.1007/978-3-540-89330-1_2.

[24] Sylvain Chevillard, John Harrison, Mioara Joldeş, and Christoph Lauter. "Efficient and Accurate Computation of Upper Bounds of Approximation Errors". In: *Theoretical Computer Science* 412.16 (2011), pp. 1523–1543. DOI: 10.1016/j.tcs.2010.11.052.

[25] Sylvain Chevillard, Miora Joldeş, and Christoph Lauter. "Sollya: An Environment for the Development of Numerical Codes". In: *Internation Congress on Mathematical Software (ICMS)*. 2010. DOI: 10.1007/978-3-642-15582-6_5.

[26] Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. "Rigorous Floating-Point Mixed-Precision Tuning". In: *Symposium on Principles of Programming Languages (POPL)*. 2017. DOI: 10.1145/3009837.3009846.

[27] George A. Constantinides, Peter Y. K. Cheung, and Wayne Luk. "Wordlength Optimization for Linear Digital Signal Processing". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 22.10 (2003), pp. 1432–1442. DOI: 10.1109/TCAD.2003.818119.

[28] Samuel Coward, Lawrence Paulson, Theo Drane, and Emiliano Morini. "Formal Verification of Transcendental Fixed and Floating Point Algorithms using an Automatic Theorem Prover". In: *Formal Aspects of Computing (in press)* (2022).

[29] Nasrine Damouche and Matthieu Martel. "Mixed Precision Tuning with Salsa". In: *Pervasive and Embedded Computing and Communication Systems (PECCS)*. 2018. DOI: 10.5220/0006915501850194.

[30] Nasrine Damouche, Matthieu Martel, and Alexandre Chapoutot. "Improving the Numerical Accuracy of Programs by Automatic Transformation". In: *International Journal on Software Tools for Technology Transfer* 19 (2017), pp. 427–448. DOI: 10.1007/s10009-016-0435-0.

[31] Nasrine Damouche, Matthieu Martel, Pavel Panchekha, Chen Qiu, Alexander Sanchez-Stern, and Zachary Tatlock. "Toward a Standard Benchmark Format and Suite for Floating-Point Analysis". In: *Workshop on Numerical Software Verification (NSV)*. 2016. DOI: 10.1007/978-3-319-54292-8_6.

[32] Catherine Daramy, David Defour, Florent de Dinechin, and Jean-Michel Muller. "CR-LIBM: a Correctly Rounded Elementary Function Library". In: *Advanced Signal Processing Algorithms, Architectures, and Implementations*. Vol. 5205. International Society for Optics and Photonics. 2003, pp. 458–464.

[33] Catherine Daramy-Loirat, David Defour, Florent De Dinechin, Matthieu Gallet, Nicolas Gast, Christoph Quirin Lauter, and Jean-Michel Muller. *CR-LIBM A Library of Correctly Rounded Elementary Functions in Double-Precision*. Tech. rep. LIP, ENS Lyon, 2006.

[34] Eva Darulova, Anastasiia Izycheva, Fariha Nasir, Fabian Ritter, Heiko Becker, and Robert Bastian. "Daisy - Framework for Analysis and Optimization of Numerical Programs (Tool Paper)". In: *Conference on Tools and Algorithms for the Construc-*

tion and Analysis of Systems (TACAS). 2018. DOI: 10.1007/978-3-319-89960-2_15.

[35] Eva Darulova and Viktor Kuncak. "Towards a Compiler for Reals". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 39.2 (2017). DOI: 10.1145/3014426.

[36] Eva Darulova, Saksham Sharma, and Einar Horn. "Sound Mixed-Precision Optimization with Rewriting". In: *International Conference on Cyber-Physical Systems (ICCPS)*. 2018. DOI: 10.1109/ICCPS.2018.00028.

[37] Florent De Dinechin, Christoph Quirin Lauter, and Guillaume Melquiond. "Assisted Verification of Elementary Functions Using Gappa". In: *Symposium on Applied Computing (SAC)*. 2006. DOI: 10.1145/1141277.1141584.

[38] Florent De Dinechin and Bogdan Pasca. "Designing Custom Arithmetic Data Paths with FloPoCo". In: *IEEE Design & Test of Computers* 28.4 (2011), pp. 18–27. DOI: 10.1109/MDT.2011.44.

[39] Manuel Eberl. "A Decision Procedure for Univariate Real Polynomials in Isabelle/HOL". In: *Conference on Certified Programs and Proofs (CPP)*. 2015. DOI: 10.1145/2676724.2693166.

[40] Free Software Foundation. *The GNU Compiler Collection*. Version 11.2. Apr. 6, 2022. URL: https://gcc.gnu.org/.

[41] Sicun Gao, Soonho Kong, and Edmund M Clarke. "dReal: An SMT solver for nonlinear theories over the reals". In: *Conference on Automated Deduction (CADE)*. 2013. DOI: 10.1007/978-3-642-38574-2_14.

[42] Alejandro Gomez-Londono, Johannes Åman Pohjola, Hira Taqdees Syeda, Magnus O. Myreen, and Yong Kiam Tan. "Do You Have Space for Dessert? A Verified Space Cost Semantics for CakeML Programs". In: *Proceedings of the ACM on Programming Languages (OOPSLA)* (2020). DOI: 10.1145/3428272.

[43] Georges Gonthier et al. "Formal Proof–The Four-Color Theorem". In: *Notices of the AMS* 55.11 (2008), pp. 1382–1393.

[44] Michael JC Gordon and Tom F Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[45] Eric Goubault and Sylvie Putot. "Static Analysis of Finite Precision Computations". In: *Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. 2011. DOI: 10.1007/978-3-642-18275-4_17.

[46] Armaël Guéneau, Magnus O Myreen, Ramana Kumar, and Michael Norrish. "Verified Characteristic Formulae for CakeML". In: *European Symposium on Programming (ESOP)*. 2017. DOI: 10.1007/978-3-662-54434-1_22.

[47] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. "Deep Learning with Limited Numerical Precision". In: *International Conference on Machine Learning (ICML)*. 2015. URL: http://proceedings.mlr.press/v37/gupta15.html.

[48] Thomas Hales, Mark Adams, Gertrud Bauer, Tat Dat Dang, John Harrison, Hoang Le Truong, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Tat Thang Nguyen, et al. "A Formal Proof of the Kepler Conjecture". In: *Forum of Mathematics, Pi*. Vol. 5. 2017. DOI: 10.1017/fmp.2017.1.

[49] John Harrison. "Floating Point Verification in HOL". In: *Workshop on Higher Order Logic Theorem Proving and Its Applications*. 1995. DOI: 10.1007/3-540-60275-5_65.

[50] John Harrison. "Floating-Point Verification". In: *Symposium on Formal Methods (FM)*. 2005. DOI: 10.1007/11526841_35.

[51] John Harrison. "Floating-Point Verification in HOL-Light: The Exponential Function". In: *Conference on Algebraic Methodology and Software Technology (AMAST)*. 1997. DOI: 10.1007/BFb0000475.

[52] John Harrison. "Verifying nonlinear real formulas via sums of squares". In: *Conference on Theorem Proving in Higher Order Logics (TPHOLs)*. 2007. DOI: 10.1007/978-3-540-74591-4_9.

[53] John Harrison. "Verifying the accuracy of polynomial approximations in HOL". In: *Conference on Theorem Proving in Higher Order Logics (TPHOLs)*. 1997. DOI: 10.1007/BFb0028391.

[54] Nicholas J Higham. "The Accuracy of Floating-Point Summation". In: *SIAM Journal on Scientific Computing* 14.4 (1993), pp. 783–799. DOI: 10.1137/0914050.

[55] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, 2002. ISBN: 0898715210. DOI: 10.1137/1.9780898718027.

[56] Johannes Hölzl. "Proving inequalities over reals with computation in Isabelle/HOL". In: *Workshop on Programming Languages for Mechanized Mathematics Systems*. 2009.

[57] Joe Hurd. "First-Order Proof Tactics in Higher-Order Logic Theorem Provers". In: *Design and Application of Strategies/Tactics in Higher Order Logics, number NASA/CP-2003-212448 in NASA Technical Reports*. 2003.

[58] IEEE. "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019). DOI: 10.1109/IEEESTD.2019.8766229.

[59] ISO/IEC. *ISO 9899:2018 - Programming languages - C*. Geneva, Switzerland, 2018.

[60] Anastasiia Izycheva, Eva Darulova, and Helmut Seidl. "Synthesizing Efficient Low-Precision Kernels". In: *Symposium on Automated Technology for Verification and Analysis (ATVA)*. 2019. DOI: 10.1007/978-3-030-31784-3_17.

[61] Bertrand Jeannet and Antoine Miné. "Apron: A Library of Numerical Abstract Domains for Static Analysis". In: *Conference on Computer Aided Verification (CAV)*. 2009. DOI: 10.1007/978-3-642-02658-4_52.

[62] Jacques-Henri Jourdan. "Verasco: a Formally Verified C Static Analyzer". PhD thesis. Université Paris Diderot (Paris 7), 2016.

[63] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. "Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning". In: *ACM SIGPLAN Notices* 50.1 (2015), pp. 637–650.

[64] Ariel E. Kellison and Andrew W. Appel. "Verified Numerical Methods for Ordinary Differential Equations". In: *Workshop on Numerical Software Verification (NSV)*. 2022.

[65] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. "seL4: Formal Verification of an OS Kernel". In: *Symposium on Operating Systems Principles (SOSP)*. 2009. DOI: 10.1145/1629575.1629596.

[66] Gerwin Klein and Tobias Nipkow. "A Machine-Checked Model for a Java-like Language, Virtual Machine, and Compiler". In: *Transactions on Programming Languages and Systems (TOPLAS)* 28.4 (2006), pp. 619–695. DOI: 10.1145/1146811.

[67] Konrad Slind and Michael Norrish. "A Brief Overview of HOL4". In: *Conference on Theorem Proving in Higher Order Logics (TPHOLs)*. 2008. DOI: 10.1007/978-3-540-71067-7_6.

[68] Ramana Kumar. "Self-Compilation and Self-Verification". PhD thesis. University of Cambridge, 2015.

[69] Olga Kupriianova and Christoph Lauter. "Metalibm: A Mathematical Functions Code Generator". In: *International Congress on Mathematical Software (ICMS)*. 2014. DOI: 10.1007/978-3-662-44199-2_106.

[70] Chris Lattner and Vikram S. Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *Symposium on Code Generation and Optimization (CGO)*. 2004. DOI: 10.1109/CGO.2004.1281665.

[71] Juneyoung Lee, Chung-Kil Hur, and Nuno P Lopes. "AliveInLean: a Verified LLVM Peephole Optimization Verifier". In: *Conference on Computer Aided Verification (CAV)*. 2019. DOI: 10.1007/978-3-030-25543-5_25.

[72] Wonyeol Lee, Rahul Sharma, and Alex Aiken. "On Automatically Proving the Correctness of Math.H Implementations". In: *Symposium on Principles of Programming Languages (POPL)*. 2018. DOI: 10.1145/3158135.

[73] Xavier Leroy. "A Formally Verified Compiler Back-End". In: *Journal of Automated Reasoning (JAR)* 43.4 (2009), pp. 363–446. DOI: 10.1007/s10817-009-9155-4.

[74] Xavier Leroy. "Formal Verification of a Realistic Compiler". In: *Communications of the ACM (CACM)* 52.7 (2009). DOI: 10.1145/1538788.1538814.

[75] Wenda Li, Grant Olney Passmore, and Lawrence C Paulson. "Deciding Univariate Polynomial Problems using Untrusted Certificates in Isabelle/HOL". In: *Journal of Automated Reasoning (JAR)* 62.1 (2019), pp. 69–91. DOI: 10.1007/s10817-017-9424-6.

[76] Daniel Liew, Daniel Schemmel, Cristian Cadar, Alastair F Donaldson, Rafael Zähl, and Klaus Wehrle. "Floating-Point Symbolic Execution: A Case Study in N-version Programming". In: *Conference on Automated Software Engineering (ASE)*. 2017. DOI: 10.1109/ASE.2017.8115670.

[77] Jay P Lim and Santosh Nagarakatte. "One Polynomial Approximation to Produce Correctly Rounded Results of an Elementary Function for Multiple Representations and Rounding Modes". In: *Symposium on Principles of Programming Languages (POPL)*. 2022.

[78] Andreas Lööw, Ramana Kumar, Yong Kiam Tan, Magnus O Myreen, Michael Norrish, Oskar Abrahamsson, and Anthony Fox. "Verified Compilation on a Verified Processor". In: *Conference on Programming Language Design and Implementation (PLDI)*. 2019. DOI: 10.1145/3314221.3314622.

[79] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. "Provably Correct Peephole Optimizations with Alive". In: *Conference on Programming Language Design and Implementation (PLDI)*. 2015. DOI: 10.1145/2737924.2737965.

[80] Victor Magron, George Constantinides, and Alastair Donaldson. "Certified Roundoff Error Bounds Using Semidefinite Programming". In: *ACM Transactions on Mathematical Software* 43.4 (2017). DOI: 10.1145/3015465.

[81] Rupak Majumdar, Indranil Saha, and Majid Zamani. "Synthesis of Minimal-Error Control Software". In: *Conference on Embedded Software (EMSOFT)*. 2012. DOI: 10.1145/2380356.2380380.

[82] Érik Martin-Dorel and Guillaume Melquiond. "CoqInterval: A Toolbox for Proving Non-linear Univariate Inequalities in Coq". In: *Conference on Effective Analysis: Foundations, Implementations, Certification (CIRM)*. 2016.

[83] Érik Martin-Dorel and Guillaume Melquiond. "Proving Tight Bounds on Univariate Expressions with Elementary Functions in Coq". In: *Journal of Automated Reasoning (JAR)* 57.3 (2016), pp. 187–217. DOI: 10.1007/s10817-015-9350-4.

[84] Adolfo Anta Martinez, Rupak Majumdar, Indranil Saha, and Paulo Tabuada. "Automatic Verification of Control System Implementations". In: *Conference on Embedded Software (EMSOFT)*. 2010. DOI: 10.1145/1879021.1879024.

[85] Adolfo Anta Martinez and Paulo Tabuada. "To Sample or not to Sample: Self-Triggered Control for Nonlinear Systems". In: *IEEE Transactions on Automatic Control* 55.9 (2010), pp. 2030–2042. DOI: 10.1109/TAC.2010.2042980.

[86] David Menendez, Santosh Nagarakatte, and Aarti Gupta. "Alive-FP: Automated Verification of Floating Point Based Peephole Optimizations in LLVM". In: *Static Analysis Symposium (SAS)*. 2016. DOI: 10.1007/978-3-662-53413-7_16.

[87] Robin Milner. *Logic for Computable Functions. Description of a Machine Implementation*. Artificial Intelligence Laboratory Memo No. AIM-169. Stanford University, 1972.

[88] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML: Revised*. MIT press, 1997. ISBN: 0262631814.

[89] Robin Milner and Richard Weyhrauch. "Proving Compiler Correctness in a Mechanized Logic". In: *Machine Intelligence* 7.3 (1972), pp. 51–70.

[90] J. Strother Moore. "A Mechanically Verified Language Implementation". In: *Journal of Automated Reasoning(JAR)* 5.4 (1989), pp. 461–492. DOI: 10.1007/BF00243133.

[91] Ramon Edgar Moore. *Interval Analysis*. Prentice-Hall, 1966.

[92] Mariano M. Moscato, Laura Titolo, Aaron Dutle, and César A. Muñoz. "Automatic Estimation of Verified Floating-Point Round-Off Errors via Static Analysis". In: *Conference on Computer Safety, Reliability, and Security (SAFECOMP)*. 2017. DOI: 10.1007/978-3-319-66266-4_14.

[93] Leonardo de Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver". In: *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 2008. DOI: 10.1007/978-3-540-78800-3_24.

[94] Muller, Jean-Michel. *Elementary Functions*. Springer, 2006. ISBN: 978-1-4899-7981-0. DOI: 10.1007/978-1-4899-7983-4.

[95] César Muñoz, Anthony Narkawicz, George Hagen, Jason Upchurch, Aaron Dutle, María Consiglio, and James Chamberlain. "DAIDALUS: detect and avoid alerting logic for unmanned systems". In: *Digital Avionics Systems Conference (DASC)*. 2015.

[96] Anthony Narkawicz, Cesar Munoz, and Aaron Dutle. "A Decision Procedure for Univariate Polynomial Systems Based on Root Counting and Interval Subdivision". In: *Journal of Formalized Reasoning* 11.1 (2018), pp. 19–41. DOI: 10.6092/issn.1972-5787/8212.

[97] Anthony Narkawicz, César Munoz, and Aaron Dutle. "Formally-Verified Decision Procedures for Univariate Polynomial Computation Based on Sturm's and Tarski's theorems". In: *Journal of Automated Reasoning (JAR)* 54.4 (2015), pp. 285–326. DOI: 10.1007/s10817-015-9320-x.

[98] Andres Nötzli and Fraser Brown. "LifeJacket: Verifying Precise Floating-Point Optimizations in LLVM". In: *Workshop on State Of the Art in Program Analysis (SOAP)*. 2016. DOI: 10.1145/2931021.2931024.

[99] Scott Owens, Magnus O Myreen, Ramana Kumar, and Yong Kiam Tan. "Functional Big-Step Semantics". In: *European Symposium on Programming (ESOP)*. 2016. DOI: 10.1007/978-3-662-49498-1_23.

[100] Ricardo Pachón and Lloyd N Trefethen. "Barycentric-Remez Algorithms for Best Polynomial Approximation in the Chebfun System". In: *BIT Numerical Mathematics* 49.4 (2009), p. 721.

[101] Pavel Panchekha, Alex Sanchez-Stern, James R Wilcox, and Zachary Tatlock. "Automatically Improving Accuracy for Floating Point Expressions". In: *ACM SIGPLAN Notices* 50.6 (2015), pp. 1–11.

[102] Christine Paulin-Mohring. "Introduction to the Calculus of Inductive Constructions". In: *All about Proofs, Proofs for All*. Vol. 55. Studies in Logic (Mathematical logic and foundations). 2015. URL: https://hal.inria.fr/hal-01094195.

[103] Amir Pnueli, Michael Siegel, and Eli Singerman. "Translation Validation". In: *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 1998. DOI: 10.1007/BFb0054170.

[104] Johannes Åman Pohjola, Henrik Rostedt, and Magnus O. Myreen. "Characteristic Formulae for Liveness Properties of Non-terminating CakeML Programs". In: *Conference on Interactive Theorem Proving (ITP)*. 2019. DOI: 10.4230/LIPIcs.ITP.2019.32.

[105] Markus Püschel, José M F Moura, Bryan Singer, Jianxin Xiong, Jeremy R Johnson, David A Padua, Manuela M Veloso, and Robert W Johnson. "Spiral - A Generator for Platform-Adapted Libraries of Signal Processing Alogorithms". In: *International Journal of High Performance Computing Applications (IJHPCA)* 18 (2004), pp. 21–45. DOI: 10.1177/1094342004041291.

[106] Tahina Ramananandro, Paul Mountcastle, Benoit Meister, and Richard Lethin. "A Unified Coq Framework for Verifying C Programs with Floating-Point Computations". In: *Conference on Certified Programs and Proofs (CPP)*. 2016. DOI: 10.1145/2854065.2854066:.

[107] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H Bailey, Costin Iancu, and David Hough. "Precimonious: Tuning Assistant for Floating-point Precision". In: *Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 2013. DOI: 10.1145/2503210.2503296.

[108] Hanan Samet. "Proving the Correctness of Heuristically Optimized Code". In: *Communications of the ACM* 21.7 (1978). DOI: 10.1145/359545.359569.

[109] Eric Schkufza, Rahul Sharma, and Alex Aiken. "Stochastic Optimization of Floating-point Programs with Tunable Precision". In: *Conference on Programming Language Design and Implementation (PLDI)*. 2014. DOI: 10.1145/2594291.2594302.

[110] Alexei Sibidanov, Paul Zimmermann, and Stéphane Glondu. "The CORE-MATH Project". In: *Symposium on Computer Arithmetic (ARITH)*. 2022.

[111] Alexey Solovyev, Marek S. Baranowski, Ian Briggs, Charles Jacobsen, Zvonimir Rakamaric, and Ganesh Gopalakrishnan. "Rigorous Estimation of Floating-Point Round-Off Errors with Symbolic Taylor Expansions". In: *ACM Transactions on Programming Languages and Systems* 41.1 (2019), 2:1–2:39. DOI: 10.1145/3230733.

[112] Geoff Sutcliffe, Jürgen Zimmer, and Stephan Schulz. "TSTP Data-Exchange Formats for Automated Theorem Proving Tools". In: *Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems* 112 (2004), pp. 201–215.

[113] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. "The Verified CakeML Compiler Backend". In: *Journal of Functional Programming (JFP)* 29 (2019). DOI: 10.1017/S0956796818000229.

[114] Ping-Tak Peter Tang. "Table-driven Implementation of the Exponential Function in IEEE Floating-Point Arithmetic". In: *ACM Transactions on Mathematical Software (TOMS)* 15.2 (1989), pp. 144–157. DOI: 10.1145/63522.214389.

[115] *The Agda Proof Assistant*. 2022. URL: https://wiki.portal.chalmers.se/agda/Main/HomePage.

[116] *The Coq Proof Assistant*. 2022. URL: https://coq.inria.fr.

[117] *The HOL-Light Proof Assistant*. 2022. URL: https://www.cl.cam.ac.uk/~jrh13/hol-light/.

[118] *The HOL4 Proof Assistant*. 2022. URL: https://hol-theorem-prover.org/.

[119] *The Isabelle/HOL Proof Assistant*. 2022. URL: https://isabelle.in.tum.de/.

[120] Laura Titolo, Marco A Feliú, Mariano Moscato, and César A Munoz. "An Abstract Interpretation Framework for the Round-Off Error Analysis of Floating-Point Programs". In: *Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. 2018. DOI: 10.1007/978-3-319-73721-8_24.

[121] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. "Finding and Understanding Bugs in C Compilers". In: *Conference on Programming Language Design and Implementation (PLDI)*. 2011. DOI: 10.1145/1993498.1993532.

[122] Randy Yates. *Fixed Point Arithmetic: an Introduction*. Digital Sound Labs. Mar. 2001. URL: https://www.cpplab.net/web/dsp/fp.pdf.

[123] Vadim Zaliva. "HELIX: From Math to Verified Code". PhD thesis. Carnegie Mellon University, 2020.

[124] Jianzhou Zhao, Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. "Formalizing the LLVM Intermediate Representation for Verified Program Transformations". In: *Symposium on Principles of Programming Languages (POPL)*. 2012. DOI: 10.1145/2103656.2103709.