PHD THESIS

ELECTRONIC AND ELECTRICAL ENGINEERING

TOATIE — FUNCTIONAL HARDWARE DESCRIPTION WITH DEPENDENT TYPES

CRAIG RAMSAY

University of Strathclyde,

Faculty of Engineering,
Software Defined Radio Research Laboratory

SUPERVISED BY

DR LOUISE CROCKETT,
PROF BOB STEWART

NOVEMBER, 2023

This thesis is the result of the author's original research. It has been composed by the author and has not been previously submitted for examination which has led to the award of a degree.

Signed: *Craig Ramsay*
Date: *January 21, 2024*

Dependent types make a lot of people nervous. They make me nervous, but I like being nervous, or at least I find it hard not to be nervous anyway.

— PIGWORKER, 2012

ABSTRACT

Describing *correct* circuits remains a tall order, despite four decades of evolution in Hardware Description Languages (HDLs).

Many enticing circuit architectures require recursive structures or complex compile-time computation — two patterns that prove difficult to capture in traditional HDLs. In a signal processing context, the Fast FIR Algorithm (FFA) structure for efficient parallel filtering proves to be naturally recursive, and most Multiple Constant Multiplication (MCM) blocks decompose multiplications into graphs of simple shifts and adds using demanding compile-time computation. Generalised versions of both remain mostly in academic folklore. The implementations which do exist are often ad hoc circuit generators, written in software languages. These pose challenges for verification and are resistant to composition.

Embedded functional HDLs, that represent circuits as data, allow for these descriptions at the cost of forcing the designer to work at the gate-level. A promising alternative is to use a stand-alone compiler, representing circuits as plain functions, exemplified by the CλaSH HDL. This, however, raises new challenges in capturing a circuit's *staging* — which expressions in the single language should be reduced during compile-time elaboration, and which should remain in the circuit's run-time? To better reflect the physical separation between circuit phases, this work proposes a new functional HDL (representing circuits as functions) with first-class staging constructs.

Orthogonal to this, there are also long-standing challenges in the verification of parameterised circuit families. Industry surveys have consistently reported that only a slim minority of FPGA projects reach production without non-trivial bugs. While a healthy growth in the adoption of automatic formal methods is also reported, the majority of testing remains dynamic — presenting difficulties for testing entire circuit families at once.

This research offers an alternative verification methodology via the combination of dependent types and automatic synthesis of user-defined data types. Given precise enough types for synthesisable data, this environment can be used to develop circuit families with full functional verification in a *correct-by-construction* fashion. This approach allows for verification of entire circuit families (not just one concrete member) and side-steps the state-space explosion of model checking methods. Beyond the existing work, this research offers synthesis of combinatorial circuits — not just a software model of their behaviour. This additional step requires careful consideration of staging, erasure & irrelevance, deriving bit representations of user-defined data types, and a new synthesis scheme.

This thesis contributes steps towards HDLs with sufficient expressivity for awkward, combinatorial signal processing structures, allowing for a correct-by-construction approach, and a prototype compiler for netlist synthesis.

ACKNOWLEDGEMENTS

Firstly, I would like to thank my supervisors, Dr Louise Crockett and Prof Bob Stewart, for the countless opportunities, exposure, support, and direction they've afforded me during this project. I couldn't have asked for more welcoming mentors.

I have been exceptionally fortunate to spend so much of my time with the University of Strathclyde's Software Define Radio Laboratory. Each and every member deserves my thanks for these years together, both in and out of the lab. In particular, thank you, Josh, for making our time at Xilinx Research Labs so memorable.

Also, I'd like to acknowledge Craig Roy's significant influence on this work — both as a close friend and for nudging me towards a topic with which I've formed a deep affection.

Most of all, this work would not have been possible without the support of my partner, Francesca. Thank you for having patience with me during my periods of momentum and, perhaps more so, beginning to run out of it when my priorities go awry. Sei un bicchiere di vino con un panino.

This work was supported by the EPSRC [grant number EP/N509760/1].

CONTENTS

LIST OF FIGURES

1 INTRODUCTION

Describing *correct* digital circuits can be an exercise in balancing uncertainty and pain.

One stage of this process ought to incite anxiety in particular: an implementation that looks plausibly correct, but whose complete intricacies do not reside in the mind of any one designer. This is a common phase, especially for modern high-performance digital circuits. Single subcircuits can often contain such complexity that no single designer can take full responsibility. More so, design reuse encourages implementations of highly parameterised circuit generators (*circuit families*) but this complicates verification efforts, often allowing bugs to lay dormant until an unfortunate parameter set is requested.

Resigning from the verification challenge at this stage can be tempting, but there can be dramatic consequences if something does go wrong. According to a 2022 Wilson Research Group verification study [1], this *if* is essentially a *when*. From their survey of 980 Field Programmable Gate Array (FPGA) projects, only 16% reached deployment with no non-trivial bugs.

Perhaps it is time for more digital designers, the author included, to better embrace their own fatuity. The core motivation behind the work in this thesis is to allow *digital design tooling to carefully "check our working" over our shoulder as we go*, especially when asking silly questions. To make strides towards this style of development, this work proposes that circuit description and circuit verification should happen simultaneously, with progress in one avenue symbiotically guiding the other.

1.1 FIELD PROGRAMMABLE GATE ARRAYS

Throughout this thesis, FPGAs and their design techniques are used as a concrete example of digital electronic design. This focus is adopted in order to provide specifics about implementation details and also make direct comparisons between our own work and the wealth of existing digital design literature and tooling targeting FPGAs (which have been widely adopted in academia). Despite this, most of the contributions presented in Chapters 4 to 6 are also somewhat applicable to topics of Application-Specific Integrated Circuit (ASIC) design, as evidenced by industry's longstanding adoption of FPGAs as a means of prototyping high-performance ASICs before tape-out.

This section offers an introduction to modern FPGA architectures and their applications. This also provides a baseline for Chapter 2's discussion of Hardware Description Languages (HDLs) and verification methodologies for FPGA projects — two aspects which are very much interwoven with FPGA architecture and applications. To briefly touch on *why* these topics are interconnected, let us consider the trends in Xilinx FPGA devices[†] over the last 35 years. Figure 1.1 summarises the growth of these devices over time, highlighting the largest and smallest devices supplied in Xilinx's flagship Virtex series and their predecessors.

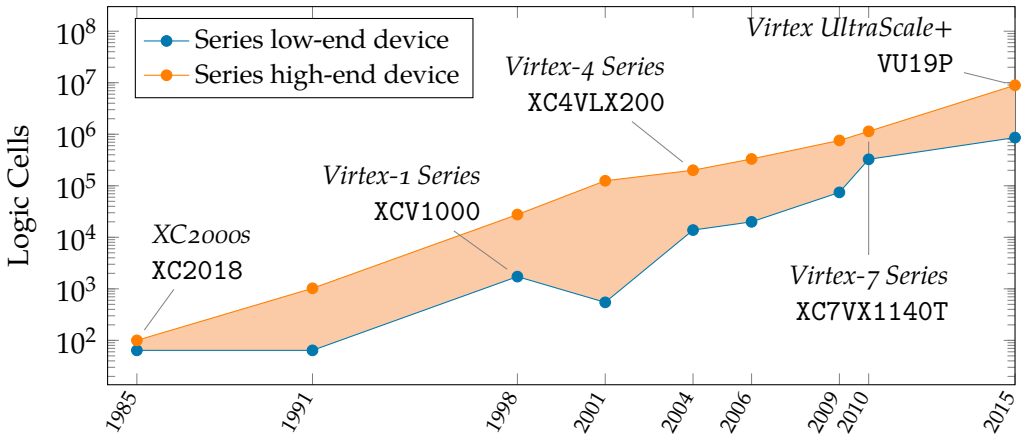

Figure 1.1: Trends in Xilinx FPGA resource density over time [2–10]

The exponential trend in effective logic density is clear. The single-chip density spans nearly 5 orders of magnitude since 1985, and this has not been reciprocated by substantial changes in the most popular HDLs. To substantiate this, we return to the Wilson Research Group's 2022 industry survey of FPGA functional verification. Their reporting is, to the best of our knowledge, the largest biannual survey of the FPGA industry. It is the result of a collaboration between the Wilson Research Group (a market research firm specialising in embedded systems) and Siemens EDA (formerly Mentor Graphics; one of the biggest names in FPGA/IC tooling) which lends some credibility to their findings. [1] claims that VHDL (VHSIC Hardware Description Language) and Verilog appear in the vast majority of *new* FPGA projects. Both languages were:

↪ Conceived in 1983, spanning the entire lifetime of Figure 1.1.

↪ Originally intended for modelling the *simulation* of digital circuits only, supporting synthesis only at a later date.

[†]For posterity, note that Xilinx were acquired by AMD in 2022 and that these products are now branded accordingly.

Clearly, as flagship FPGA logic density increases by nearly $\times 10^5$, digital designers are given the opportunity to realise larger and more complex designs. This is, however, just that: an *opportunity*. To reify such implementations without increasing designer effort by the same magnitude, we must depend on HDLs which are expressive enough to convey and compose these complex architectures concisely. This kind of expressivity can consist of language features such as:

- ↪ Iterative or recursive constructs to generate large structures concisely.
- ↪ Higher-order functions (functions which can have function-valued arguments or return new functions) to help separate circuit *structure* from *behaviour*. E.g. we describe the structure of a tree of binary operations only once (as a higher-order function) but can reuse it easily by specialising its behaviour to an adder tree, a min/max search, etc.
- ↪ Parameterisation of a component by compile-time values. This allows us, in conjunction with the other techniques, to describe a "family" of similar circuits all at once. Since we can specialise this single description to one of many members of the family, there is more opportunity for reuse in larger designs.

As well as increased complexity in the description of circuits, the challenge of verification also increases. This is especially true of circuit families — it may be simple to exhaustively test a single, trivial circuit, but how can we be confident that all possible configurations in the family are well behaved? As will be demonstrated later in this thesis, the choice of features in the description language can quite fundamentally change how we approach the verification challenge. This research investigates a new single language for both circuit description and verification using *dependent types*.

For now, we consider one modern system-on-chip device with an FPGA component as a case study. Although each FPGA series demonstrates variations in architecture and particular use cases, the fundamentals remain consistent. The case studies presented here examine the Xilinx UltraScale+ Radio Frequency System-on-Chip (RFSoC) device family [11, 12], as employed in Chapter 4. Figure 1.2 shows an overview of the RFSoC's FPGA architecture, ignoring the on-chip processor groups and hardened peripheral blocks.

The majority of the device is dedicated to general "logic fabric", consisting of interwoven Configurable Logic Blocks (CLBs) and their surrounding programmable routing. This sea of CLBs can be configured to perform *any* synchronous digital operation. Figure 1.3 shows a simplified CLB architecture for UltraScale devices. In essence, a set of 6-input Look-Up Tables (LUTs) can be configured to realise any combinatorial logic

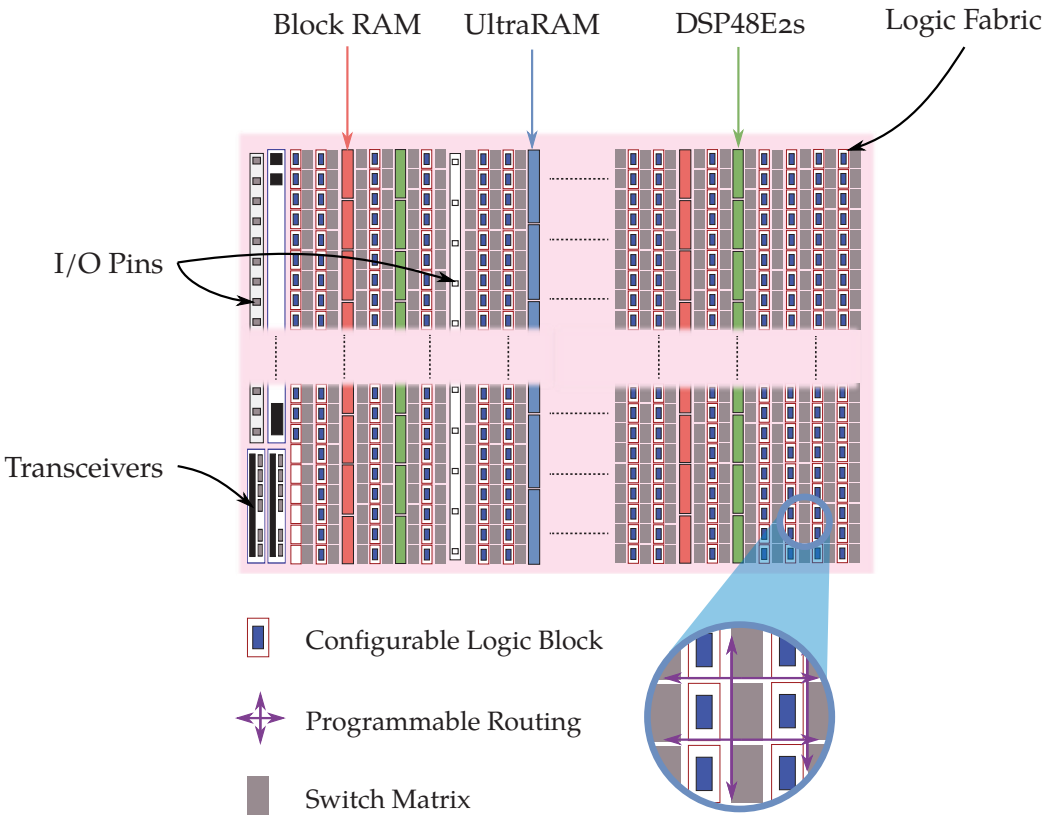

Figure 1.2: Overview of RFSoC's UltraScale+ FPGA architecture

in a user application. These are followed by optional synchronous Flip-Flops (FFs) which can be used to register signals, allowing designs with memory. While an architecture with LUTs and FFs is sufficient to reproduce any digital circuit internally, modern FPGAs provide many specialised blocks for more efficient implementation of common circuit patterns, as well as dedicated I/O features.

For RFSoC devices in particular, these dedicated FPGA elements include:

↪ *Dedicated logic & memories*

- Digital signal processing blocks (`DSP48E2s`) containing a hardware multiplier with optional post/preadders and pipeline registers.
- Block RAMs and Ultra RAMs for much more dense on-chip memory implementation than with generic logic fabric.
- A variant of the CLB slice shown in Figure 1.3 which adds write address and write enable functionality to each LUT, allowing cheap implementation of small, distributed 64-bit memories.
- Dedicated clock routing and management capabilities.

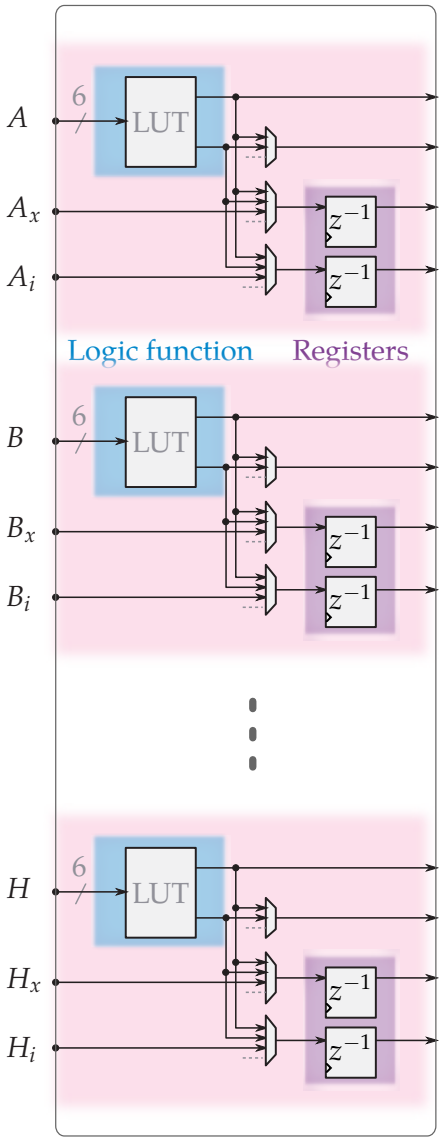

Figure 1.3: Simplified view of UltraScale+ CLB Architecture [13] with the vertical carry chain left implicit

↪ *Input / Output capabilities*

- General purpose I/O blocks to interface with package pins.
- Gigabit Transceivers supporting several standards.
- High-speed RF Data Converters (Analogue to Digital and Digital to Analogue) blocks, sampling at up to 5.9 and 10 Gsps, respectively [11].

For each FPGA architecture, there is a (usually closed source) vendor tool to take a high-level description of the user's circuit and map it efficiently to the given CLB and specialised resource architectures. This is essentially a compiler from the HDLs

explored in Section 2.2 to the specific FPGA's resources. Most of the novel contributions offered in this thesis sit above this level and are device agnostic — leaving the particulars of low-level resources to existing vendor tooling.

The practical use cases for these FPGA devices are numerous, both inside and outside of academia. As alluded to previously, it might be easy to dismiss FPGAs at first glance as a prototyping platform for designs which will eventually be realised as ASIC devices. While this is one application, it is far from the full picture. Note that almost all modern FPGA architectures are *reconfigurable*; indeed the "field programmable" part of the acronym demands that the behaviour is configurable in the field (well after fabrication!). There are three broad categories of use case which are both motivated by this reconfigurability:

1. *Low volume applications:*

   ASIC design and fabrication incurs enormous Non-Recurring Engineering (NRE) costs. The reconfigurability of FPGAs sidesteps these costs but each unit is likely more expensive than its ASIC counterpart. Given the lack of up-front NRE costs, FPGA implementations are more viable than ASICs for low volume. Trimberger suggests that the "crossover" threshold where ASICs become viable is also raising over time [14]. Low volume application are countless — ranging from small commercial products to bespoke control logic for scientific instrumentation.

2. *Future-resistant applications:*

   High performance industries including mobile communications, data centre networking, and video compression require custom digital electronics. These applications in particular are susceptible to a cruel obsolescence due to an important part of their ecosystems — *standards*. FPGA implementations allow in-field updates to support new standards without the costly manufacture respin and full redeployment.

3. *Dynamically reconfigured applications:*

   Perhaps the smallest of the three groups, the reconfigurability of FPGAs can be exploited during the execution of a single application. A device may reconfigure regions of its own fabric to update its behaviour. This can be used to time-share one set of resources between multiple low-performance applications, or to iteratively tweak the behaviour of a single application. For example, a neural network deployment might be highly optimised for inference by specialisation on one set

of weights, but can use dynamic reconfiguration to deploy improved weights on-the-fly.

One theme which links many FPGA applications is the need for safety. Failures in control systems for aerospace engineering or industrial automation, for example, can be catastrophic. In these cases, no matter how challenging, verification of a circuit's functional behaviour is an absolute requirement. Indeed, for designs fully utilising the modern, logic-dense flagship FPGAs seen earlier in Figure 1.1, verification *is* challenging. Returning to the 2022 Wilson Research Group study on FPGA verification:

> *The results shown in* [*Figure 1.4*] *are somewhat disturbing. In 2022, only 16 percent of all FPGA projects were able to achieve no bug escapes into production*[...]
>
> — HARRY FOSTER IN [1]

With this "disturbing" scene set, the next section introduces the programming techniques that could help motivated engineers raise the 16% to a more comforting metric.

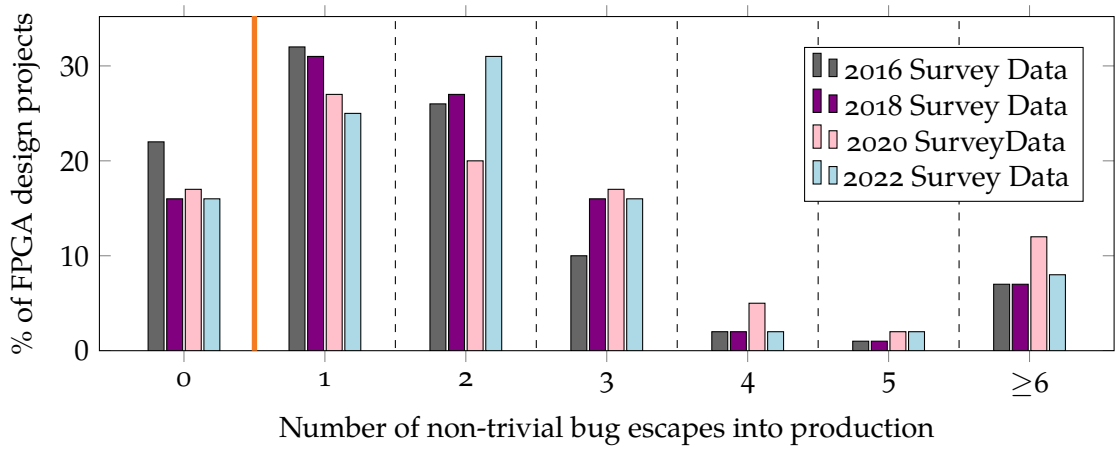

Figure 1.4: Survey trends of FPGA bug escapes into production, based on data from [1]

1.2 FUNCTIONAL PROGRAMMING & VERIFICATION WITH DEPENDENT TYPES

Functional programming is a programming paradigm in opposition to the imperative paradigms that an engineering audience are likely familiar with, such as procedural or object-oriented. Especially for this audience, it is easier to convey the essence of functional programming in terms of negative information — what functional programming does *not* allow.

A functional program consists of *pure functions* which essentially represent mathematical equations. This may appear at first glance as a definition true of functions in imperative languages such as C, but this is not quite the case. A pure function has some extra restrictions:

Immutability: Once a variable has been assigned a value, it cannot be changed. There is no standard assignment operation, only *definitions*.

No side-effects: A more general version of immutability. A pure function can have no effect on the wider system other than simply calculating its result.

To illustrate these differences, Listings 1.1 and 1.2 give two possible implementations that calculate the Fibonacci sequence. The prior is expressed using a pure function with standard mathematical syntax and the later is a valid implementation written in C.

Listing 1.1: Fibonacci function expressed using pure functions

$$\text{fib}(n) = \begin{cases} 0 & \text{, if } n = 0 \\ 1 & \text{, if } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{, otherwise} \end{cases} \tag{1.1}$$

The use of recursion in functional programs is ubiquitous. Since there is no assignment of variables, we are precluded from having constructs such as `for` loops. Also notice that this definition does *nothing* but calculate its result — every single time the developer writes `fib(n)` they will always receive the same value. Contrast this to Listing 1.2's C implementation, written in bad-faith in order to expose some of the common challenges in imperative programming. This particular C implementation does have a better time and space complexity than our pure implementation. This is a concern for many functional programs in software, but it does not always translate as strongly to hardware design since all combinatorial sections are inherently parallel.

The implementation in Listing 1.2 will usually report the expected numerical answer, but there are two nasty properties which both arise due to the presence of side-effects. Line 16 will conditionally produce the unfortunate side-effect of thermonuclear war. More subtly, the mutability of the global variable `n` can also cause problems. Suppose that Listing 1.2 is part of a multi-threaded application. If there are two concurrent calls to `fib`, the later instance will likely clobber the prior's value of `n`, causing it to exit the loop early and return an incorrect value. More insidiously, if these two calls form part of a race condition, we might only encounter this bug extremely rarely.

Listing 1.2: A valid Fibonacci program expressed in C

```c
int n;

int do_fib()
{
  int a=0, b=1, i, swp;

  if (n==0) return a;
  if (n==1) return b;

  for (i=2; i<=n; i++) {
    swp = b;
    b = a + b;
    a = swp;
  }

  if (b==4181) launchMissiles();
  return b;
}

int fib(int new_n)
{
  n = new_n;
  return do_fib();
}
```

By disallowing side-effects, functional programs preclude both of these classes of error. Limiting ourselves like this makes it substantially easier for both the human and the compiler to reason about our programs. This is a contributing factor towards our belief that functional programming is an important foundation for describing correct circuits. The order of evaluation does not matter in pure functional programs. This makes parallel evaluation much more tractable — and this is a boon for circuit descriptions too.

Hughes' seminal paper "Why Functional Programming Matters" [15] provides an excellent, alternative framing of functional programming not as a set of constraints placed upon us, but rather a set of advantages. These contribute towards code modularity, at the level of functions. We are encouraged by the language to implement small, generic functions for simple tasks and are rewarded with tools to very concisely compose them into solutions to complex tasks. One important feature enabling this compositional glue is higher-order functions: functions which themselves have functions as arguments. Higher-order functions come quite naturally to functional programming environments since it is straightforward to treat pure functions, which are free of side-effects, as first-class citizens of the language.

Functional programming often seems to, quite coincidentally, go hand-in-hand with sophisticated *type systems*. A type system being the set of rules the compiler follows to decide whether or not a given program, colloquially, makes sense. These two topics are distinct (many functional languages are completely untyped!), but perhaps the two communities overlap strongly in their reverence for mathematics. We consider a program's types to be extremely valuable; a stance probably not shared by the majority. Types are the only concrete source of information we pass to a compiler that describes our *intent*. A type checker can attempt to match (or infer) a term's type with its implementation. If it succeeds, the implementation is accepted. If it fails, we get feedback raising suspicion of our attempt. Clearly, the more information encoded in a program's types, the more opportunity the compiler has to identify our errors for us.

There are many approaches to typing, each allowing a different level of precision. This research focuses on one of the more radical disciplines called dependent typing. In the extreme, we could imagine writing a function's type so precisely that there is only one possible solution — one which is verified to be correct by the type checker. Dependently typed programming based on Martin-Löf type theory, as popularised by languages including Epigram [16], Agda [17], and Idris [18], enables such precise typing with one fundamental concept. They allow types to mention terms.

1.3 AIMS

Our aim is to explore the design of an HDL that provides absolute confidence in the implementation of an entire circuit family. The thesis statement guiding this exploration is: uniting the *verification* and the *implementation* of circuit families can allay both challenges. More concretely, this work pursues answers for three research questions:

1. To what extent is it possible to exploit dependently typed software programming techniques for describing combinatorial circuits?
2. How can real-world netlists be synthesised in the presence of dependent typing?
3. How does the formal verification effort scale for large Digital Signal Processing (DSP) circuit families?

1.4 CONTRIBUTIONS

Towards answering these three research questions, this thesis designs a language and compiler which:

1. Encodes circuits as plain functions (inspired by CλaSH [19]), rather than data. This typically simplifies circuit descriptions, allowing all language constructs to be used in the *behaviour* of a circuit, not just its elaboration.
2. Allows *meaning* to be ascribed to synthesisable data via its type. This should allow circuit functionality to be written in a *correct-by-construction* fashion, rather than performing verification after the fact.

These two properties provide an interesting, largely unexplored point in what is quite a densely populated field of study. Research of functional HDLs is as mature as that of VHDL and Verilog — two mainstays of digital electronic design. However, to the best of our knowledge, the literature only contains work that actually synthesises circuits with either one of these two properties.

Notably, CλaSH [19] introduces a compiler for a subset of Haskell which represents circuits as plain functions, satisfying property 1. However their host language, Haskell, limits how much meaning we can convey in synthesisable data's types. In practice this is often limited to describing just the *structure* of a composite type: how many elements are in this vector or how deep is this binary tree? There are also a few HDL projects which incorporate dependent types. Π-ware [20] is hosted in a dependently typed software language but does not directly implement either of our desired properties. Still, the host language does permit powerful theorem proving techniques for verification ex post facto, just not in a correct-by-construction style. Sheard's work in [21] explores an encoding of circuits in the general purpose Ωmega with correct-by-construction style, but encodes circuits as a data type. Finally, Brady in [22] outlines why a HDL with both properties would be advantageous, culminating in a correct-by-construction model of a simulated carry-ripple adder family. Although their work presents strong arguments for this style of circuit description, it is only presented on software models of circuits: there are no means of synthesising circuits from these descriptions.

Chapter 4 contributes a case study used to scrutinise CλaSH's development experience for real-world DSP applications.

↪ A novel CλaSH implementation of a DSP architecture for low-cost, high-speed, parallel filters for direct RF sampling. This combines techniques for exploiting sharing between parallel subfilters (FFA) and decomposing the constant coefficient multiplications into graphs of inexpensive additions and bit shifts. The design completely precludes the need for valuable hardware multipliers in front-end stages in RFSoC applications.

↪ Considering the implementation of this circuit structure, the benefits of CλaSH's circuits-as-plain-functions approach are argued. The shortcomings are identified and motivate this work's focus on dependent types.

The work presented in Chapter 5, in essence, enhances the proposal from [22]. While an environment with dependent types can facilitate the full functional verification presented by Brady, Chapter 5 identifies that there are different degrees to which a designer may wish to lean on these type-level features. It contributes use-cases for a dependently typed HDL where circuits are represented as functions:

↪ Even when avoiding interaction with the type system, this environment simplifies circuit descriptions. A single set of language constructs can be used to describe the full lifetime of a circuit family — from type-level programming to model bit growth, circuit elaboration-time, to the circuit's run-time behaviour.

↪ With a stronger adoption of type-level programming, designers can easily track non-functional properties of a circuit family. This is demonstrated using DSP examples with non-trivial bit growth patterns.

↪ Extending Brady's original correct-by-construction approach, Section 5.4 identifies how the method scales to real-world DSP structures. It reveals an encouraging trend — higher-level DSP blocks such as dot products and Fourier transforms often give correctness theorems entirely *for free!*

Finally, Chapter 6 contributes the implementation of this thesis' language and compiler: `toatie`. This is a single new language for both combinatorial circuit description and theorem proving under one roof. In particular, contributions include the following:

↪ Existing language features (used only as optimisations) in software programming can be repurposed to enforce distinction between phases in a circuit's lifetime. `toatie` exploits irrelevance and erasure to help isolate the type checking phase. Staging annotations are then exploited to distinguish between the circuit's elaboration phase and the circuit's run-time.

↪ Since `toatie` represents circuits as plain functions, it also automatically derives bit representations for synthesisable user data types. Section 6.4 demonstrates how this can be done by reusing the unification engine already required for any dependently typed language implementation.

↪ Elaborating a circuit to a netlist largely becomes an issue of normalisation. Again, the type checker for any dependently typed language already has tooling for normalisation.

1.5 OUTPUTS

The following outputs have been produced as a direct result of this work:

[23] **On Applications of Dependent Types to Parameterised Digital Signal Processing Circuits**
C. Ramsay, L. Crockett, and R. Stewart
2021 IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)
Conference paper — 2021
Sources and data sets are available at [24]

[25] **Low-cost, High-speed Parallel FIR Filters for RFSoC Front-Ends Enabled by CλaSH**
C. Ramsay, L. Crockett, and R. Stewart
55th Asilomar Conference on Signals, Systems, and Computers
Conference paper — 2021
Sources and data sets are available at [26]

[27] **Data for `toatie`— A Hardware Description Language With Dependent Types**
C. Ramsay, L. Crockett, and R. Stewart
Self-published
Digital artefact — 2022
Sources and data sets are available at [27]

1.6 THESIS OUTLINE

The rest of this thesis is composed of two background chapters followed by three contribution chapters. Chapter 2 offers relevant background on digital design with FPGA devices and the corresponding languages for description and verification. Here there is also further justification of our aims by exploring the differences between representing circuits as functions vs data, and different verification strategies. Chapter 3 offers

a concise practical background on dependently typed functional programming with examples in the new language presented in this thesis, `toatie`. This introduces concepts such as algebraic data types, irrelevance & erasure, staging, and theorem proving, before we use them in the context of combinatorial circuits.

Chapter 4 is the first contribution chapter. It explores the design and implementation of a novel DSP architecture, addressing a real pain-point of many designs targeting RFSoC devices: high-speed, parallel filtering. We offer a solution in CλaSH exploiting heavy compile-time evaluation mechanisms, combining existing methods for optimised parallel filtering and reduction of constant multiplications to graphs of shifts and additions. This chapter concludes by offering a reflection on our practical experience with the successes and limitations of existing functional HDLs.

The last two chapters concern this thesis' protagonist, the `toatie` language. Chapter 5 focuses on the implementation and verification of circuits in `toatie`. It explores three different degrees of verification, ranging from the minimum requirements for synthesis (fixed sized structures) to fully verified arithmetic meaning. An intriguing mid-point on this spectrum is also presented, where types help guide wordlength arbitrary pruning strategies. This is concluded with an implementation of a radix-2 Decimation-In-Time (DIT) Discrete Fourier Transform (DFT) circuit which uses correct-by-construction techniques to ensure it meets our arithmetic specification. We also supply a separate proof demonstrating that the radix-2 DIT algorithm is indeed equivalent to the more expensive direct DFT implementation.

Chapter 6 details the implementation of the compiler for `toatie`. It begins by formalising the foundation for our language, TinyIdris [28]. The chapter continues by discussing `toatie`'s surface grammar, core language, and its typing rules. Particular attention is paid to the new features of erasure, staging, and our synthesis scheme. This combination of techniques is then consolidated by a selection of concrete examples; stepping through their synthesis process and visualising their intermediate results.

Finally, Chapter 7 concludes with a summary of this thesis' main contributions and offers several promising avenues for future work.

2 CIRCUIT DESCRIPTION AND VERIFICATION

This chapter discusses the literature and relevant work in description and verification of digital circuits for FPGAs. Although this content is spun with a stronger emphasis on functional programming and dependent types, much of the wider context is shared with two publications produced during the project:

1. *L. Crockett, D. Northcote, C. Ramsay, F. Robinson, and R. Stewart in "Exploring Zynq MPSoC: With PYNQ and Machine Learning Applications" [29] — Strathclyde Academic Media*
2. *J. Goldsmith, C. Ramsay, D. Northcote, K. W. Barlee, L. Crockett, and R. Stewart in "Control and Visualisation of a Software Defined Radio System on the Xilinx RFSoC Platform Using the PYNQ Framework" [30] — IEEE Access Journal*

This section offers an introduction to FPGA design by discussing the interwoven topics of HDLs and design verification. The dependency between these topics is a recurring theme throughout this thesis. We first establish a set of common digital circuit verification methodologies. Following this, we explore the landscape of hardware description and verification languages more concretely.

This discussion is initially grounded with reference to industry surveys, focusing on the features typical to modern incarnations of the three most widely used HDLs: VHDL, Verilog, and SystemVerilog. This wider context prepares us for a deeper dive into the most relevant literature HDLs based on functional programming languages — the strongest points of comparison for the contributions made in Chapters 4 to 6. This background investigation pays special attention to aspects of the functional HDL design space:

↪ The representation of circuits — either as functions or as data.

↪ The degree to which an HDL's type system aids verification.

We believe that the contributions of this thesis occupy an interesting, unexplored point within these dimensions.

2.1 VERIFICATION METHODOLOGIES FOR CIRCUITS

It may seem jarring to be confronted with the verification techniques for hardware descriptions before we introduce the implementations themselves, but that is, in essence, this work's entire thesis statement. The verification and description of circuits should not be entirely separate concerns.

As evidenced in Chapter 1, FPGA designs are growing exponentially in complexity. Neither the verification tooling nor design expressivity have managed to keep pace historically, however. In order to help advance the discussion on both topics simultaneously (explored more in Chapter 5), we should first address the status quo for verification techniques. In particular, we are interested the opportunity to verify properties of entire circuit *families* at once.

The abstract methodologies introduced here will be used by concrete implementations throughout Section 2.2. As reported by the Wilson 2022 FPGA verification survey [1], only 16% of industry FPGA projects get to deployment without non-trivial bugs escaping. Within the same survey, over 50% of all design respins are attributed to logic/functional issues. There is also an overlapping ascription to reused Intellectual Property (IP) cores — parameterised descriptions which often prove difficult to test in full. The root causes are split between logical design errors and, perhaps more interestingly, incorrect or incomplete specifications. Clearly, we need an alternative way to perform verification, addressing:

↪ Reusable IP cores, which have a potentially infinite parameter space (*circuit families*).

↪ The design specifications themselves — spotting ill-defined protocols in a machine-checked manner.

In the broadest terms, circuit verification methods assume one of two forms: dynamic or static analyses. Dynamic analysis calls for a case-by-case *simulation* of circuit behaviour. In order for this to provide guarantees of circuit behaviour, the set of examples used would need to be exhaustive. This is, for most real-word circuits, entirely unreasonable.

For example, an arithmetic unit providing an operation over two 32-bit inputs already has a parameter space of 2^{64} pairs of inputs — just over 1.8×10^{19} simulation runs for exhaustive testing! This has historically been a real, practical issue, with the Intel Pentium's `FDIV` floating point division bug [31] in 1994 inducing a complete recall

at an expense in the order of $\$475,000,000$. In particular, this demonstrates the need for *completeness* in testing because of the `FDIV` bug's infrequency (an estimated one in nine billion chance of random inputs provoking significant error) and its severity. This class of bugs was later precluded at Intel by formal, static verification methods including [32].

This search space will compound over each clock cycle in synchronous circuits with long memory or many states. Instead of testing large circuits in this manner, we must admit that completeness with dynamic verification is not achievable and instead rely on subjective metrics to determine when our testing is "good enough".

An obvious metric is *code coverage*, measuring how many lines of source code are exercised by a set of tests. This might seem to indicate completeness, but it does not necessarily exercise every global behaviour — including different combinations of chosen branches. This more complete, but more elusive, metric is known as *functional coverage*. Both must usually be supported by the simulator for measurement, rather than as an intrinsic part of the verification language. A complementary observation to narrow the coverage required is the use of local *assertions*. Here the tester can insert local statements throughout the testbench which raise exceptions when certain conditions are violated. This allows greater confidence in the behaviour of the global design with fewer test runs, since we can more easily detect certain illegal behaviours closer to their source.

A further method to augment these dynamic, example-driven verification techniques is the use of *constrained random* testing. Instead of providing a fixed, finite set of test inputs, the tester instead provides a set of paired details:

1. A property (or local assertion) relating the input to the desired output. This can often include relationships between cycles, as well as combinatorial relationships.
2. A means of generating random, but valid, input sequences.

Armed with these two things, a testing infrastructure can generate and simulate as many inputs as are feasible for a given timescale. Furthermore, randomly generated inputs are more likely to exercise edge cases than a human designer. The element of *constraint* here is important since, especially for stateful circuits, a completely random sequence might not correspond to a possible real-world scenario. We will encounter a high reward, low effort implementation of constrained-random testing in Chapter 4 using the Haskell library `QuickCheck` — SystemVerilog also has some support for this

methodology. A more complete discussion of these topics can be found in [33]. Figure 2.1 shows the actual adoption of each of these dynamic verification techniques, as well as two complementary static techniques.

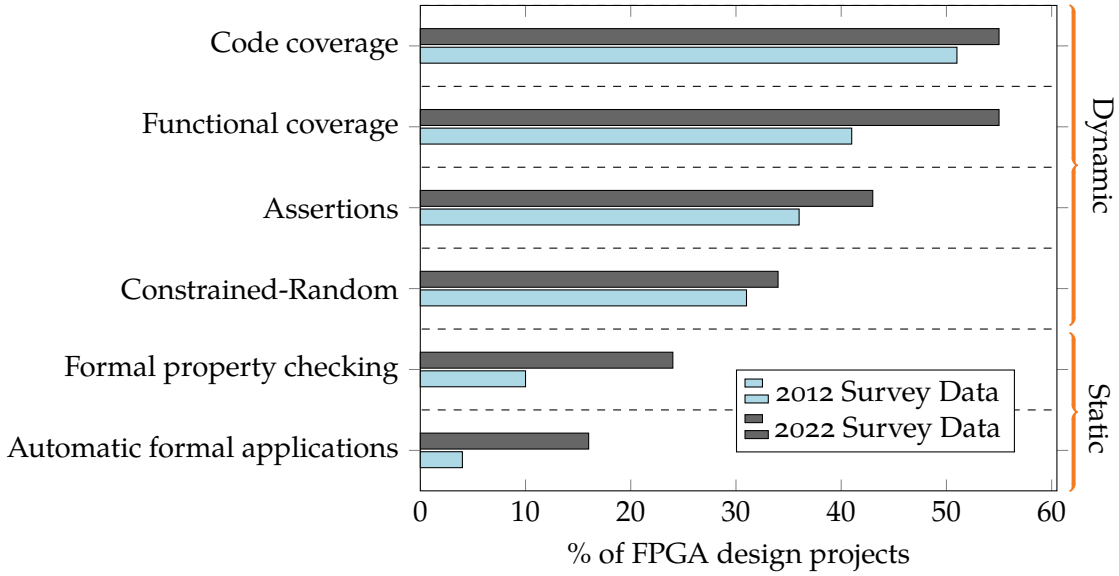

Figure 2.1: Survey trends of FPGA verification technique adoption, based on data from [1]

From the survey in [1] it is clear that dynamic techniques remain the most common method in industry practice. However, it is important to note that the *growth* in adoption is greater in formal, static methods.

Formal, static methods approach the completeness problem from a different perspective — often employing mathematical or symbolic techniques to ensure coverage without predetermined test cases. While these do provide guarantees for completeness, at least within certain constraints, they usually demand more specialised knowledge from the developer or more complex implementation strategies than dynamic methods using incomplete metrics.

Figure 2.1 demonstrates that a major growth area in this field is the use of automatic formal methods. These include exhaustive, automatic model checking and automatic theorem proving techniques. These are appealing since they require very little expertise from the developer. We will encounter such uses adopted in the Lava HDLs in Section 2.4.1. These, however, come with real limitations when faced with the state-space explosion of large circuits. Often it becomes impossible to automatically verify properties of large concrete circuits, let alone circuit families. Instead, this thesis focuses on manual (but often interactive) theorem proving techniques. Chapters 5 and 6 demonstrate how we can use the same description for both implementation and theorem proving while, better yet, letting one inform the other, creating a streamlined,

type-driven development environment.

Before we do so, we must consider the languages used for the verification and description of circuits in industry today.

2.2 INTRODUCTION TO HARDWARE DESCRIPTION LANGUAGES

A likely fundamental aim of any high-level language is productivity. The productivity of a language itself can be attributed to both its expressivity and the ease of verifying a program's correctness. These motivations are as true for hardware description as they are for software programming. The contributions of this work will encourage a selection of language features which ought to promote both of these metrics. These topics include:

1. Representing circuits as pure functions, directly capturing the semantics of combinatorial circuits.
2. Using *staging* to clearly distinguish between elaboration-time evaluation and circuit run-time evaluation.
3. Dependently typed circuit families which demonstrate full functional correctness, in a *correct-by-construction* fashion.

Before we dive more deeply into the literature for these techniques, we return to the Wilson Research Group's 2022 industry survey of FPGA projects [1] to investigate the productivity possible within the industry's status quo. Figure 2.2 shows survey results for use of various HDLs in industry FPGA projects. These values sum to over 100% since it is common for a single project can use multiple source languages, especially when reusing legacy or vendor IP cores.

The vast majority of design projects contain either VHDL or Verilog code, hinting towards a large inertia surrounding real-world tooling. We consider VHDL, Verilog, and its more recent derivative, SystemVerilog, as being representative of traditional (non-functional) HDLs and the real-world baseline. These are introduced in Section 2.3. We also offer an overview of the functional HDLs most relevant to `toatie`, representing the academic literature. Although *all* of the functional HDLs discussed in Section 2.4 are collected in the $\leq$ 5% *Other* of Figure 2.2, functional hardware description is a longstanding research topic. Indeed, Sheeran introduced μFP as early as 1984, making it contemporary to VHDL and Verilog [34].

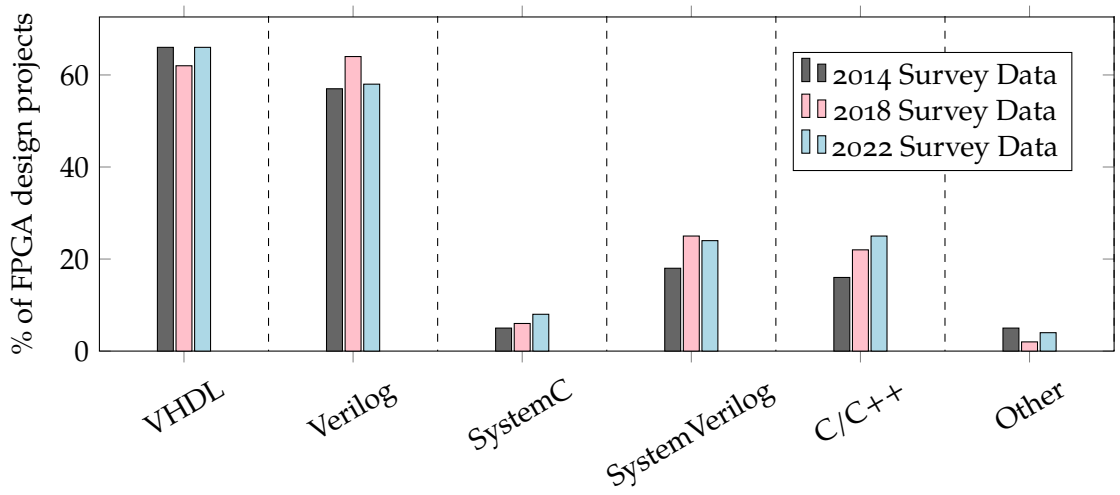

Figure 2.2: Survey trends of HDL adoption, based on data from [1]

There are two remaining languages from Figure 2.2 which are not particularly useful as comparisons for `toatie`: SystemC and C/C++. SystemC is commonly used for modelling at a system-level (orchestrating existing blocks), whereas we are concerned with the realisation of the hardware for each block. This thesis focuses on HDLs which compile with a *structural* view of the source. With a structural view, the developer has full control over how the final netlist will look, as their description directly informs both the area and the timing of a circuit. To afford the developer such control, the tooling necessarily has only explicit pipelining, explicit unrolling of iterative structures, and no automatic derivation of stack-based or soft-processor implementations of recursive algorithms. The structural approach, the topic of this thesis, is dialectic with the High-Level Synthesis (HLS) approach implied by the FPGA projects using C/C++. HLS compilers are often invoked as a back-end for single-source systems languages such as OpenCL and SYCL. HLS tooling aims to take an abstract specification of a circuit's *behaviour* (usually captured in a imperative software language) and automatically generate a circuit architecture to realise the original behaviour. HLS approaches promise productivity by abstracting away circuit-specific concerns such as clocking, pipeline implementation, and the corresponding control logic. This loss of control for a circuit's non-functional properties is also true of Bluespec's rule-based descriptions. Instead, we advocate for productivity directly at the Register-Transfer-Level (RTL) via programming language features including higher-order functions and staging.

The RTL level of abstraction sits nicely in the middle of the road. Here we are explicit about the overall structure of our circuit, while gaining a productive level of abstraction over the combinatorial paths within it. We are free to use choice constructs such as `case` or `if` statements within our combinatorial sections — a freedom granted

when describing the *transfer between registers*. The behavioural approach of HLS languages sits above this level, and their compilers are free to insert registers at will or infer stack-based implementations. The designer only has control over the functionality and is leaving most non-functional properties, such as timing and circuit area, to the discretion of the compiler. In contrast to this, the lowest level of abstraction we consider in this chapter is the gate level. Here the designer has full control over the structure and the composition of combinatorial paths at the expense of productivity. We must explicitly express all combinatorial logic in terms of primitive circuit gates and are disallowed the comfort of native language choice constructs.

Figure 2.3 captures the commonalities and distinctions between most of the approaches to hardware description discussed in the rest of this chapter.

Figure 2.3: Summary of HDL features and styles

	Paradigm	**Abstraction Level**	**Typing Discipline**	**Hosting Style**
				Traditional HDLs
VHDL	Mixed / Synchronous	RTL	Strong Typing	Stand-Alone
Verilog	Mixed / Synchronous	RTL	Weak Typing†	Stand-Alone
SystemVerilog	Mixed / Synchronous	RTL	Strong Typing	Stand-Alone
				High-Level Synthesis Languages
Vivado HLS	Imperative	Behavioural	Strong Typing	Stand-Alone
Intel HLS	Imperative	Behavioural	Strong Typing	Stand-Alone
				Functional HDLs
Bluespec Haskell	Rule-based	Behavioural	Strong Typing	Stand-Alone
Lava	Functional	Gate	Stronger Typing + Hindley–Milner	Embedded (Haskell)
Chisel	Functional	Gate	Strong Typing	Embedded (Scala)
Π-ware	Functional	Gate	Stronger Typing + Dependent Types	Embedded (Adga)
CλaSH	Functional	RTL	Stronger Typing + Hindley–Milner	Stand-Alone
`toatie`	Functional	RTL	Stronger Typing + Dependent Types	Stand-Alone

2.3 TRADITIONAL HDLS

To further introduce the set of *traditional* HDLs, we focus in on VHDL, Verilog, and SystemVerilog. As well as being (or derived from) early contemporary languages, they all share a few base properties in common as well.

The overall structure of any program comprises of blocks described using one of two paradigms:

Concurrent: Using a declarative (nearly functional) approach.
These blocks easily capture the behaviour of *combinatorial* logic, usually limited in order to avoid combinatorial loops. Or...

Sequential: Using synchronous semantics more akin to imperative programming.
These blocks facilitate the *synchronous* logic found in Finite State Machines (FSMs) and other common control circuits.

When used with certain restrictions, these two kinds of code block can express circuit descriptions in a *synchronous* style — a single system described as the co-ordination of logically separate processes. These traditional languages all contain two classes of features; those which can be used in the description of circuits (the *synthesisable* subset) and those which have no circuit semantics (the *non-synthesisable* subset). The latter is afforded mainly for the construction of testing infrastructure for the actual synthesisable circuit descriptions. Even within the synthesisable subset of features, there are some extra restrictions placed on the programmer in order to maintain synthesisability. For example, the concept of signal assignment (by default) slightly deviates from the semantics of imperative software languages such as C in order to avoid introducing race conditions. It is also common that no two sequential blocks may drive a shared signal.

The traditional HDLs also share an important non-technical property: they are all now designed and maintained by committee. In fact, VHDL and (System)Verilog are both guided by the *same* committee — the IEEE Design Automation Standards Committee [35, 36]. Also, due to early success and industry adoption, any improvements to the language are defined as additive extensions which carefully navigate the context of the language's legacy. Neither of these are inherently Bad Things (Haskell was itself designed by committee!) however, it does have a consequence specific to the wider FPGA development ecosystem. The definition of an HDL is necessarily quite separate

[†]Verilog only accepts synthesisable data encoded as one-dimensional bit arrays, and only distinguishes between nets (`wire`) and memory elements (`reg`).

from the implementation of its low-level compiler since the binary format for almost all modern, dense FPGAs is proprietary. Hence each FPGA vendor implements and maintains their own tooling. As we will see in the following subsections, even when a committee defines excellent new language extensions for a traditional HDL, there is often a large inertia to overcome before it is implemented in downstream proprietary tooling — a harsh impedance mismatch in the industry. We also demonstrate that the burden of early success encourages many modern features to be added via extensions in an ad hoc manner, rather than addressing a whole set of concerns with a single, fundamental change.

We continue by looking at concrete examples, and since this thesis is concerned with design productivity, focusing on the abstraction mechanisms implemented in each traditional HDL.

2.3.1 *VHDL*

The language VHDL, as standardised by IEEE in [36], embodies the ideas behind the synchronous programming style and offers reasonably strong typing. Its types include arrays, subtypes, enumerations, and heterogeneous record types, among others. A strong typing discipline is an important consideration in both software and hardware programming languages as it allows the type checker to catch some classes of errors statically at compile-time, rather than leaving them to wreak havoc at run-time — possibly obscuring the original source of misbehaviour in the process. Just how many classes of error can be caught statically (colloquially, its "strength") depends on the expressivity of the language's type system and how the programmer chooses to employ types in their implementation. Although type safety is often simplified to a binary "weak"/"strong" when comparing VHDL and Verilog, the true answer always lies somewhere on a spectrum. This is an important topic for the remainder of the thesis and we explore different points of this spectrum throughout Chapter 5.

In terms of abstractions, VHDL has historically offered three distinct features:

1. *Generate statements:*

   Used to *generate* structures directed by some constant value. The first of these mechanisms is the `for generate` statement, directed by a constant *range* parameter. These correspond to an unrolled version of an iterative body — a simple `for` loop for hardware. The second is `if generate` on constant boolean arguments, offering conditional generation of structures.

2. *Generics:*

   Used to parameterise a design entity by a constant value. Traditional examples include numeric constant generics (e.g. to parameterise the width of a signal) or time constant generics. Generics in VHDL are quite a composable feature — an entity may accept generics as a parameter and propagate them to other entities further down the design hierarchy.

3. *Configurations:*

   Used to direct which circuit implementation (or, an *architecture* in VHDL terminology) is used within a component instance. Configurations can be defined either locally (hidden entirely within the implementation of an entity) or globally (a single definition touching any level hidden within the design hierarchy). This feature can be viewed as a means of parameterising component behaviour but without composability — the choice is either localised to a single entity or for the entire design hierarchy.

Note that we already have two features (generics and configurations) which are really both providing parameterisation, and could be unified into a single language feature. This effect compounds when we consider features added by future iterations of the VHDL specification.

A typical, reasonably good faith VHDL-2002 implementation of an N-word summation is shown in Listing 2.1. Note that before VHDL-2008 (defined ≈ 25 years after the original), generics in VHDL did *not* support non-value parameters such as types, functions, or components. Despite that, Listing 2.1 still makes good use of generics and generate statements, parameterising the input wordlength and the number of input words. We go as far as to intelligently bound the output wordlength (assuming a *linear* chain of adders). However, we do not limit the wordlength of internal signals and instead propagate the worst-case output wordlength throughout. At a top-level, we may choose to use a configuration declaration to specify a particular implementation of the `adder` component, but this must be defined globally.

This implementation does have bugs which will only be revealed with certain sets of parameters. For example, when $N \leq 2$ elaboration will fail even though summing two input words is entirely reasonable. These kinds of error, as edge cases of a circuit family, can be elusive when only using simple testbench testing for larger designs.

Listing 2.1: An implementation of an adder chain using VHDL-2002 with generics and generate statements

```
entity summation is
  generic (N   : integer := 8  ;
           M   : integer := 12);
  port    (xs  : in  std_logic_vector (N*M-1 downto 0);
           sum : out std_logic_vector (N+M-2 downto 0));
end entity;

architecture behavioural of summation is
  component add is
      generic (A : integer);
      port    (
        x : in  std_logic_vector (A-1 downto 0) ;
        y : in  std_logic_vector (A-1 downto 0) ;
        z : out std_logic_vector (A-1 downto 0));
  end component;

  type inter_array is array(0 to N-3) of std_logic_vector(N+M-2 downto 0);
  signal accs : inter_array;

begin

  -- Initial adder
  acc_init : add generic map ( A => N+M-1 )
                 port map (
              x => (N+M-2 downto M => '0') & xs(  M-1 downto 0),
              y => (N+M-2 downto M => '0') & xs(2*M-1 downto M),
              z => accs(0)
  );

  -- Generate intermediate sums
  acc_inters: for i in 1 to N-3 generate
    acc_i : add generic map ( A => N+M-1 )
                port map (
              x => (N+M-2 downto M => '0') & xs((i+2)*M-1 downto (i+1)*M),
              y => accs(i-1),
              z => accs(i)
    );
  end generate;

  -- Final element
  acc_last : add generic map ( A => N+M-1 )
                 port map (
              x => (N+M-2 downto M => '0') & xs(N*M-1 downto (N-1)*M),
              y => accs(N-3),
              z => sum
  );
end behavioural;

```

While this is a reasonable attempt using VHDL-2002, there are a few aspects which are less parameterised than one may like. For example:

↪ We have encoded our input as a single `std_logic_vector` but we would like to, more accurately, describe it as an array of N M-bit logic vectors. As it stands, we need to perform error-prone mental gymnastics to construct our indices. However, the tools surprisingly do not allow us to index both the array and the `std_logic_vector` elements with generics simultaneously.

↪ The caller cannot locally specify which adder implementation should be used for an application. This is required to easily make context-sensitive choices balancing between area and performance (e.g. a filter structure on the critical path vs. a large but latency-tolerant adder structure).

↪ We cannot fully parameterise the type of the input words. This limits our understanding of the input words too. For example, what numeric interpretation of the raw input bits should be used? Unsigned, signed, one-hot, or perhaps canonical signed digit?

↪ We cannot easily parameterise the circuit *structure* more complexly. We might want a single generic component to handle linear adder chains *and* adder trees. We could also quite reasonably want to sum a collection of signals with heterogeneous wordlengths.

To partially address these limitations, VHDL-2008 was defined with extra features for abstraction; including type and function generics. These alleviate the first two difficulties, however the latter two remain troublesome. It is also important to put the definition of the VHDL language in its proper context for FPGA development: support available in vendor tooling.

Although Xilinx's Vivado design suite has had partial support for VHDL-2008 since Vivado 2016.1, the type and function generics have only been supported since 2019.1 — nearly 11 years after the language definition. The Quartus Prime EDA software from Intel (formerly Altera) now also has support for VHDL-2008. However, at the time of writing, they have opted for the surprising strategy of placing their implementation of these language features behind a pay-wall [37].

Due to a combination of vendor support and the volume of pre-2008 material circulating throughout the industry, it remains a rare occurrence to see many of these new features used in earnest. The interested reader can find details of all VHDL constructs (including VHDL-2008) in Ashenden's excellent resources [38].

2.3.2 (*System*)*Verilog*

As well as standardising VHDL, the IEEE Design Automation Standards Committee also acts as the steward of (System)Verilog [35]. However, in contrast to VHDL, Verilog has quite a weak typing discipline. Every synthesisable signal is viewed simply as a one-dimensional array of bits. While this does allow the designer to omit explicit conversions, it disallows the designer from conveying anything more about the *meaning* of their data to the compiler or other designers via its type. Whereas, in VHDL, we often encounter signals with meaningful types — we treat unsigned numeric representations differently from signed representations, and very differently compared to a record structure.

Verilog takes inspiration from popular imperative languages, seducing C programmers with a familiar syntax, but this choice introduced a deadly issue early in the project's history. The blocking assignment semantics introduced non-determinism between concurrent statements [39] which is, unfortunately, the basis for most digital design! Beyond its type system and its syntax, many of Verilog's fundamental ideas are similar to those we have covered for VHDL. We still use the synchronous style, interleaving concurrent and sequential code blocks. We also have abstraction mechanisms which map nearly 1 : 1 onto those found in pre-2008 VHDL. For instance, we have:

Parameters: which abstract over constant values (not including types or functions). These correspond to VHDL's generics, but can be additionally associated with tasks and functions.

Configurations: analogous to the global configuration declarations of VHDL.

Generate blocks: for iterative and conditional design generation, mirroring VHDL's generate statements.

SystemVerilog is a substantial extension of the Verilog language which introduces many features, including enhancements paralleling the abstraction features discussed for VHDL-2008. Verilog has now been consumed by the SystemVerilog banner, leaving only one standard [35]. Listing 2.2 shows an improved version of our summations example (also parameterising over the adder function) in SystemVerilog. Note that an equivalent is also possible in VHDL-2008, subject to tooling support.

Listing 2.2: A SystemVerilog adder chain with parameterised adder function, synthesised in Vivado 2022.1

```
// A specific adder implementation
interface i_default_adder
  #( parameter W=32 );
  task automatic add
    (input logic [W-1:0] x, input logic [W-1:0] y, output logic [W-1:0] z);
    z = x + y;
  endtask
endinterface

// Summation module generic in word type and adder function
module sum
  #(parameter N, parameter type WORD_TY                        )
   (interface f, input WORD_TY [N-1:0] xs, output WORD_TY sum);

  WORD_TY [N-3:0] accs;

  // Initial adder
  always_comb
      f.add(xs[0], xs[1], accs[0]);

  // Generate intermediate sums
  genvar i;
  generate
    for (i = 1; i <= N-3; i+=1)
      always_comb
        f.add(xs[i+1], accs[i-1], accs[i]);
  endgenerate

  // Final adder
  always_comb
    f.add(xs[N-1], accs[N-3], sum);

endmodule

// Example top-level summation
typedef logic [7:0] T_WORD;
module top
  (input T_WORD [3:0] xs, output T_WORD sum);

  i_default_adder #(.W(8)) adder_arch();
  sum #(.N(4), .WORD_TY(T_WORD))
      sum_inst (adder_arch, xs, sum);
endmodule

```

As well as adding type parameters and function parameters (via the *interfaces* construct), SystemVerilog also contributes two major topics: more expressive types and a substantial language for verification-only code. The weak typing of Verilog is extended to include multidimensional arrays, enumerations, structures, and unions. The non-synthesisable subset of the language used for verification has been dramatically reworked, demonstrating the industry appetite for improved verification tooling.

The verification language supports the object-oriented programming style, defining stateful classes and allowing inheritance. Alongside this productivity boost comes built-in support for generating constrained-random inputs for verification (a vital building block for property-based testing), boolean and temporal assertions, and collection of coverage statistics. For our own narrative, SystemVerilog offers two insights: parameterisation is important (designers will navigate multiple different features in order to implement it) and verification is acknowledged by the industry as a pain point in VHDL/Verilog.

2.4 FUNCTIONAL HDLS

Recall that functions are treated as first-class citizens in functional programming. We are free to pass pure functions as arguments, and to return new functions as results. Functions which do so are called higher-order functions. As in software programming, this offers a powerful way of composing abstractions. Depending on the type system of the language, such functions are often enough to replace *every* abstraction feature in the extensive libraries of VHDL-2008 and SystemVerilog. This simplifies both the language implementation (which was a topic of much inertia for traditional HDLs) and the designer's burden — one fundamental feature can be employed for many purposes.

As mentioned previously, the subject of functional HDLs is a well studied one. The interested reader is referred to surveys such as [40]. Although neglected in our own summary, the functionally inclined digital designer may also wish to explore the seminal work of μFP [41], or the more recent BlueSpec Verilog [42] and Chisel [43]. We will continue with a small summary of the functional HDLs most relevant to this thesis. We pay particular attention to the choice of hosting style and how the type system applies to hardware descriptions. The hosting style impacts the implementation effort but also the representation of circuits: either *embedding* circuits as data in an existing host language, or representing circuits as functions with a *stand-alone* compiler.

2.4.1 *Lava Languages*

A key observation behind Lava is that we already have a platform with *all* of the powers of abstraction needed for describing circuits quite well: pure functional software languages. The family of Lava languages (including [44–46]) offers clear insights into the use of hardware description libraries *embedded* in a standard software language — called an Embedded Domain-Specific Language (EDSL). Lava is hosted in the functional software programming language Haskell. Haskell offers an environment where

functions are treated as first-class citizens and there is an expressive (although not dependently typed) type system offering:

↪ Algebraic Data Types (ADTs)

↪ Parameterisation by type (via type variables, type classes, etc.)

↪ Parameterisation by function (via higher-order functions)

↪ Opportunity for much stronger type safety than traditional HDLs

↪ Powerful type inference — function type definitions are usually not explicitly required while we still maintain the advantages of strong typing

This is just a brief list of the highlights — a full appreciation for the features of Haskell can be gained from [47]. Looking at Lava in particular, we are quickly confronted with the EDSL style of hardware description. This approach is implemented as a library within an existing software language. It offers the programmer two main sets of tools. The first is a collection of primitive gates which typically correspond to the dedicated hardware units in an FPGA's architecture — e.g. LUTs, FFs, and perhaps even Block RAMs. The EDSL also provides us with a set of basic *combinators* which are functions to combine subcircuits in a variety of ways. Digital designers can write Haskell/Lava code using these combinators to compose the circuit primitives in interesting ways. When their program is evaluated, a circuit description is elaborated to circuit composed entirely of the primitives. One perspective is that the host language offers an extremely rich, Turing-complete[†] version of VHDL's simple `generate` statements — a completely programmable elaboration process. The advantage to this approach falls largely in favour of the implementer. It is then the *software evaluation* of a Haskell program which produces a circuit netlist. Relying on the Haskell host means reusing Haskell's entire machinery for free, including its type checker, evaluator, parser, Integrated Development Environment (IDE) support, etc. The main disadvantage is, as we will soon encounter, that the embedded language is completely distinct and secondary to the host language's own constructs.

Listing 2.3 shows a binary adder from first principles described in Xilinx-Lava [48, 49]. Here `xor2`, `xorcy` and `muxcy` are primitives supplied by Lava and we use standard Haskell code to compose them into a binary adder structure. In particular, notice that we are building our circuit as a data structure rather than representing it directly as a function. Here `(Out a)` is a Lava data type representing a netlist which outputs a signal

[†]A Turing-complete system is one which, if given enough time and memory, can compute *anything* that is computable. I.e. a tempting metric for a language's completeness.

Listing 2.3: An example of an adder from first principles in Xilinx-Lava

```
1  oneBitAdder :: (Bit, (Bit, Bit)) -> Out (Bit, Bit)
2  oneBitAdder (cin, (a, b))
3    = do part_sum <- xor2 (a, b)
4         sum      <- xorcy (part_sum, cin)
5         cout     <- muxcy (part_sum, (a, cin))
6         return (sum, cout)
7
8  adderWithCarry (cin, (a,b))
9    = col oneBitAdder (cin, zip a b)
10
11 adder :: ([Bit], [Bit]) -> Out [Bit]
12 adder (a,b)
13   = do (c, _) <- adderWithCarry (zero, (a,b))
14        return c
```

of type `(a)`. Towards building our adder circuit, also note the use of the higher-order `col` function. This is our first introduction into the important concept of decoupling a circuit's *structure* from its *behaviour*. `col` replicates a subcircuit and connects them together as shown in Figure 2.4. These functions for capturing common structural patterns can be developed once and reused for many applications — resulting in concise, meaningful descriptions for a developer familiar with the base set of combinators. We show full definitions here for demonstration, but note that these particular functions are usually supplied as part of the standard library. Although the Haskell type system is very expressive, in this scenario, we are transforming dynamically sized lists of bits. At the lowest level, this actually results in type safety *worse* than an equivalent Verilog description! We will encounter a stronger use of Haskell's type system during our discussion of CλaSH.

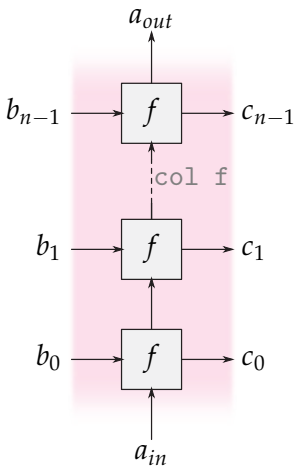

Figure 2.4: The `col` combinator, connecting subcircuits `f`

Working with such low-level gates sounds unproductive, but functional languages allow us to quickly build abstractions on top of these structures, implementing increasingly complex structures and exploring the design space. As a brief example, Listing 2.4 shows two summation circuits described via higher-order functions: one linear adder chain and one adder tree. A feature unique to the Xilinx-Lava implementation is that the combinators, including `col`, `mapPair`, and `>=>` also capture layout semantics:

`adder` uses '`col oneBitAdder`' to infer a *vertical* chain of *N*-full adders, mapping well to the vertical specialised carry-chain hardware of many FPGAs.

`foldChain` uses '`f >=> rec`' to iteratively build a chain of `f` subcircuits placed *horizontally* with vertical centring.

`foldTree` uses `mapPair` to build a single layer of independent adders stacked vertically, with each layer sequenced by `>=>` in adjacent horizontal slices with vertical centring.

These combinators with geometry semantics can be a effective tool for FPGA power-users, capturing hand-crafted floorplanning information for regular structures without much extra effort beyond the standard digital design. We can reuse these structural helpers with similar ease for much more complicated designs too. Instead of composing adder circuits, we could similarly be composing entire FFTs, sorter circuits, FSMs, or complete processors.

Listing 2.4: A two summation circuits in Xilinx-Lava using higher-order functions for structure

```
1  foldTree :: ((a, a) -> Out a) -> [a] -> Out a
2  foldTree f [a] = return a
3  foldTree f as  = (mapPair f >=> foldTree f) as
4
5  foldChain :: ((a, b) -> Out a) -> (a, [b]) -> Out a
6  foldChain f (a, [b])
7    = f (a,b)
8  foldChain f (a, b:bs)
9    = (f >=> rec) (a,b)
10   where rec c = foldChain f (c, bs)
11
12 sumTree  = foldTree  adder
13 sumChain = foldChain adder
```

Abstraction via higher-order functions is great for quickly combining subcircuits; capturing a *structural* view of hardware description. In Lava's case each subcircuit is described at a gate level. Some applications, however, are more naturally described using an RTL style, where we have more freedom when expressing the behaviour of

each combinatorial section. Would this also be possible in Lava? It is relatively simple to implement combinators for parts of the choice constructs we are familiar with: think `case`, or `if`. For example, one can construct a simple case construct as a function accepting a value to scrutinise and a set of alternatives paired with the corresponding scrutinee values. This might have the type:

```
casE :: Eq a => Out a -> [(Maybe a, Out b)] -> Out b
```

However, without substantial type-level shenanigans and language extensions, we will often lose the ability to check our coverage in `case` constructs (i.e. do we actually cover all valid choices or do we infer a latch?). More fundamentally, we necessarily introduce a new name and syntax for these constructs since it is unusual for an EDSL to be able to hook into the host syntax. This means the designer will have to juggle two sets of choice constructs: one in the host language and one in the embedded language with reduced syntactic expressivity. There are also arguments to be made about how feature-complete these embedded choice constructs can be, as per Section 2.3.2 of [19].

Our Lava examples so far have only been combinatorial but the language can also encode synchronous logic. Although it is internally a data structure describing a netlist, developers may consider a value of `Out a` to represent an infinite stream of `a`-typed values. Element at index N in this stream represents the output value which is stable on the N^{th} clock cycle. The flip-flop primitives, including `fd` and `fdce`, provide a means to permute these streams in valid ways — e.g. a delay corresponds to prepending a constant to the start of a stream.

Circuits in Lava may also be interpreted in various other ways: as data for a concrete simulation, as data for netlist synthesis, or as data for (external) symbolic verification methods. As such, a common approach to verification in Lava is to adopt a property-based strategy relying on external model checkers or automatic theorem provers [46]. Here, properties are captured using the same language as the circuits themselves; a symbolic version of the property is passed to external tools for checking. Since these are either bounded model checkers or *automatic* theorem provers, a circuit family must first be completely specialised — i.e. we verify a single circuit rather than a circuit generator itself. We would struggle to generate guarantees about the behaviour of *all* possible multiplier circuits from a (potentially infinite or particularly complex) circuit family but we can automatically gain confidence in a hand-picked finite subset of them. Chapter 5 demonstrates our own dependently typed environment for hardware description which does not suffer this restriction.

In summary, the family of Lava languages provides interesting insights into how to capture circuit semantics with functional descriptions. It helps sing the praises of higher-order functions as they apply to digital circuit design. Once we consider RTL circuit descriptions, however, we do begin to feel some resistance. To address this we consider stand-alone functional HDLs, rather than EDSLs such as Lava. These generally require more upfront engineering effort but, as CλaSH demonstrates in the following section, it can be worthwhile. Nonetheless, Lava certainly has its own kind of elegance, delivering a beautiful implementation of an interesting observation — the *evaluation* of a functional program can generate circuit netlists well, even with geometry information.

2.4.2 *CλaSH*

CλaSH is a functional HDL which is a standalone compiler for Haskell, rather than hosted as an EDSL [19, 50]. Here the developer writes (nearly) vanilla Haskell functions to describe the circuit structure *and* run-time behaviour. The CλaSH compiler then transforms this description into a synthesisable circuit. As we will see shortly, this allows us to directly utilise Haskell's rich choice constructs and user-defined data types directly in our circuit's run-time descriptions — not just at elaboration-time.

CλaSH supports descriptions which are higher-order and polymorphic. These are two common demands on fully parameterised circuits and, as demonstrated back in Section 2.3, are not supported by pre-2008 VHDL or Verilog. While these parameterisation techniques are better supported by VHDL-2008 and SystemVerilog, FPGA developers are limited by the subset of features implemented in vendor tooling. This can often lag a decade behind the language definitions for good support.

Another consequence of being a standalone compiler is that CλaSH represents circuits as plain functions, instead of directly as a data structure. This necessitates a slightly more complex compilation step but it is the fundamental choice that permits the developer to enjoy the same set of Haskell syntax for both circuit generation and circuit run-time behaviour. In particular, the compilation uses a term-rewrite system defined in Chapter 4 of [19] to normalise the circuit description, eliminating intermediate non-synthesisable terms. Like most synchronous functional HDLs, CλaSH models synchronous signals as infinite streams. Only "safe" operations on these streams are exposed to the developer, such as the `delay` function. Listing 2.5 shows an introductory CλaSH implementation of an up/down counter which is entirely polymorphic in terms of its numeric type.

Listing 2.5: An up/down counter in CλaSH, representing circuits as functions over user-defined data types

```
1  data Mode n
2    = IncrUp   n
3    | IncrDown n
4    | Hold
5
6  getIncr :: Num a
7          => Mode a -> a
8  getIncr (IncrUp   n) = n
9  getIncr (IncrDown n) = negate n
10 getIncr  Hold        = 0
11
12 counter :: (Num a, NFDataX a, HiddenClockResetEnable dom)
13         => Signal dom (Mode a) -> Signal dom a
14 counter mode = current
15   where
16     current = delay 0 next
17     next    = current + (getIncr <$> mode)
```

Notice that we define our own Algebraic Data Type (ADT), `Mode`, to describe the circuit's control input. We are free to use this type in our circuit description and perform all of the pattern matching, case expressions, and guards one might expect from Haskell. Indeed, `getIncr` is defined by pattern matching on a `Mode` type argument. The compiler will automatically derive a bit representation for our type with a statically known length and infer multiplexer structures for each alternative branch in our circuit description. This extra compiler support assists us in directly describing the run-time behaviour of a combinatorial circuit without just composing gate-level primitives. This vanilla Haskell style lets us convey much more about the *meaning* of our data than when working directly with collections of bits.

Although we are essentially writing vanilla Haskell, in order to ensure that a circuit really is synthesisable by CλaSH, there are a few restrictions placed on the top-level function:

↪ It is monomorphic and first order.

↪ Its arguments and return value are *representable* (i.e. have a finite length, statically known bit representation).

Note that these restrictions only exist for the top-level — we can make use of higher-order or polymorphic function deeper in the hierarchy. This is akin to VHDL's restriction requiring all generics to be given a value by the top-level entity. There are some secondary considerations hidden within the need for representable arguments. Most severely, this precludes recursive data types and functions defined via data-dependent

recursion! The need for these is somewhat dampened by a rich set of library primitives with recursive patterns hard-coded over vectors and trees. At the time of writing there is only experimental support for Generalized Algebraic Data Types (GADTs) too [51].

As evidence towards CλaSH's synthesis restrictions being quite reasonable, there are a surprising number of ubiquitous Haskell libraries that "just work" in synthesisable CλaSH descriptions. One excellent example of a CλaSH design which makes good use of the existing Haskell ecosystem is Érdi's implementation of a processor for the language *Brainfuck* [52, 53]. We reproduce its high-level control logic in Listing 2.6 in order to investigate the mixing of plain Haskell libraries (created without any consideration for CλaSH) and CλaSH-specific code in the wild. Here, the `step` function captures the combinatorial logic coordinating the control for a single clock cycle of the entire processor. The `CPUIn` input combines all of the input pins to the processor and we compose our control system in the custom type `CPU ()` — a monad which threads CPU state between sub-operations and arbitrates driving of output pins. While the reader can skip the finer details of this implementation, we wish to draw attention to a few techniques in particular:

↪ We are free to use monad abstractions to describe the control circuit's run-time behaviour.

  This nearly-imperative style is often a very concise way of expressing control logic, especially when reusing the large set of helper functions over monads from the standard library, `Control.Monad`. Since this is quite a simple processor architecture, we mostly see use of the fundamental `(>>=)` combinator to compose computations in the `CPU` monad. While this is a familiar abstraction pattern for most Haskell programmers, it is perhaps surprising that it works out-of-the-box for hardware description — offering a powerful imperative-style tool to capture subcircuits which are inherently sequential.

↪ We can also use the ubiquitous `Control.Lens` library, substantially enhancing the functionality of record types.

  Again, this is common practice in industrial Haskell projects. We might make use of deeply nested data structures when describing the state of a more complex processor and *lenses* help us work with these concisely. This accounts for the `(%=)`, `(.=)` combinators in Érdi's example.

↪ Behind the scenes, the `barbies` library is also at play.

This offers us general purpose abstractions to parameterise data structures by a functor. In this setting, Érdi employs `barbies` to implicitly generate default values for the CPU output pins, selecting the correct output when there are multiple drivers, and appealing to CλaSH's bundling/unbundling of signal groups.

Listing 2.6: Example of control logic for the Brainfuck processor from [53] in CλaSH

```
step :: Pure CPUIn -> CPU ()
step CPUIn{..} = use phase >>= \case
    Halt -> return ()
    Init -> phase .= Exec
    Skip depth -> fetch >>= \case
        '[' -> phase .= Skip (depth + 1)
        ']' -> phase .= maybe Exec Skip (predIdx depth)
        _ -> return ()
    Exec -> fetch >>= \case
        '>' -> ptr %= nextIdx
        '<' -> ptr %= prevIdx
        '+' -> writeCell $ nextIdx ramRead
        '-' -> writeCell $ prevIdx ramRead
        '.' -> outputCell ramRead
        ',' -> startInput
        '[' -> if ramRead /= 0 then pushPC else phase .= Skip 0
        ']' -> popPC
        '\0' -> phase .= Halt
        _ -> return ()
    WaitWrite  -> phase .= Exec
    WaitOutput -> when outputAck $ phase .= Exec
    WaitInput  -> traverse_ writeCell input
```

All the reuse of Haskell libraries to describe a circuit's run-time behaviour is a direct consequence of siding with a standalone implementation, rather than an EDSL. While one can reuse similar libraries with Lava, they are only useful in the circuit generators, not for directly describing circuit run-time behaviour. Beyond libraries, both approaches also get access to the entire Haskell ecosystem including IDEs and package management.

While the example in Listing 2.6 presents as very idiomatic Haskell, this is easier to achieve for combinatorial circuits than synchronous ones. Chapter 14 of [52] presents a reasonably simple address decoder circuit which cannot be synthesised by CλaSH; compile-time evaluation mechanisms in CλaSH remain, at the time of writing, "very poor" [51].

Next, we discuss the aspects of the CλaSH compiler with room for improvement. One difficulty is the lack of *staging*. It is sometimes difficult to reason about *when* part of our description will be evaluated — will an expression be reduced during the compilation process or will it survive and appear in our final circuit? This challenge is, in

part, due to the choice of representing circuits as plain functions. The staging is quite clear in Lava, where all Haskell code is evaluated during elaboration-time, and anything within the final netlist data type is present at circuit run-time. Since CλaSH uses a single language for both, and relies on partial evaluation strategies, there can easily be ambiguity. This staging concern appears most often when thinking about recursively defined circuit generators which is a topic central the contributions in Chapter 4. Some circuit architectures are simply recursive by nature, including prevailing FFT and parallel filtering structures studied in this thesis. If the recursion in those generators is not eliminated at compile-time, we end up with a non-synthesisable circuit. While partial evaluation at compile-time remains a work in progress for the CλaSH project, developers can resort to metaprogramming techniques such as Template Haskell (TH). TH is a language extension allowing developers to write Haskell code which manipulates the abstract syntax tree representing other Haskell code. It vitally allows forcing evaluation of expressions at compile-time — allowing us to force the flattening of a recursively defined structure. TH as a tool does come with its own restrictions too, since these sorts of metaprogramming techniques were never a first-class element of the language. These include limitations on where definitions with different staging requirements can appear and can impact type safety; discussed more in Section 4.6.2.

Another common pattern found in CλaSH programs is the use of `Vec` types. We can think of these as lists whose length is part of its type — a value of type `Vec 3 Bit` is a collection of three elements, each of type `Bit`. Programming with `Vecs` is especially important in hardware description since we need our synthesisable signals to have a statically known length. Dynamically sized lists are not synthesisable in CλaSH, so we instead use `Vecs` and enjoy the extra type safety that they infer. These length-indexed structures are introduced later in Chapter 3 where we demonstrate that they are quite natural to capture in dependently typed languages. Haskell, however, is not dependently typed, so many of the `clash-prelude` standard functions operate over types parameterised by natural numbers in this way: `Vec` helpers, binary tree helpers, sized numeric types, and bit vectors. The way these structures are implemented in a system without dependent types is an orchestration of language extensions and special tricks well explored in [54], whose closing remark reads "The best thing about banging your head off a brick wall is *stopping*".

As a small demonstration of these extra considerations required by Haskell's lack of dependent types, let us revisit our adder chain example. This time, armed with a more expressive type system than traditional HDLs, we attempt to model the bit-growth along each part of the chain. Assuming an adder function `add` which performs our desired bit-growth, we might expect that a simple `fold add` would implement the adder chain. This is not the case since the accumulated value changes type after each

subsequent adder stage — the output is growing. Most standard definitions of `fold` do not allow for this. We instead must use several different parts of type-level machinery from Haskell extensions in order to convince it that this growth is OK. Listing 2.7 shows this example in full. Again, the reader is not necessarily expected to appreciate the full details yet, but we wish to highlight the number of different constructs required to implement something reasonably simple in CλaSH's type system.

Listing 2.7: Faking dependently typed folds (directed by `Nats` only) in CλaSH to model an adder chain with bit-growth

```
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE AllowAmbiguousTypes   #-}
module Adders where

import Clash.Prelude
import Data.Singletons

-- Type class describing the types for each step of an incremental sum with
-- bit-growth
class ExtendingSum a where
  type StepResult a (n :: Nat)
  stepAdd     :: SNat n -> a -> StepResult a n -> StepResult a (n+1)
  toFirstStep :: a -> StepResult a 0

-- An implementation of `ExtendingSum` for Unsigned numbers
instance (KnownNat n) => ExtendingSum (Unsigned n) where
  type StepResult (Unsigned n) m = Unsigned (n + m)
  stepAdd SNat = add
  toFirstStep = id

-- An implementation of `ExtendingSum` for Signed numbers
instance (KnownNat n) => ExtendingSum (Signed n) where
  type StepResult (Signed n) m = Signed (n + m)
  stepAdd SNat = add
  toFirstStep = id

-- A type level function ("motive") describing how the type of our
-- accumulator changes for each step, `n`.
data SumMotive (a :: Type) (f :: TyFun Nat Type) :: Type
type instance Apply (SumMotive a) n = StepResult a n

-- The actual sum circuit for any `ExtendingSum`, using Clash's `dfold`
summation :: forall n a . (KnownNat n, ExtendingSum a)
          => a -> Vec n a -> StepResult a n
summation x xs = dfold (Proxy @ (SumMotive a)) stepAdd (toFirstStep x) xs
```

For this single fold over signed/unsigned binary numbers, the developer needs to confront some of the more daunting features (and extensions) of Haskell. These techniques include singleton natural numbers (linking term and type-level representations), proxies, type families, type-level functions, `DataKinds`, and knowing which extension incantations are required for a given task. No dependently typed language would make such a scenario as challenging! Since dependently typed languages share

one syntax between the term and type levels, we only need standard data types and standard functions over those data types. Chapters 5 and 6 work towards our own dependently typed HDL, `toatie`, which demonstrates this simplification. In one sense, we wish to continue a trends towards *generality*: just as the several abstraction features of traditional HDLs are condensed into one feature by most functional HDLs, we aim to condense the several language features required here for type-safe circuits into one with `toatie`. As well as this generalisation, dependent types will offer us interesting avenues for formal verification and theorem proving with our circuit families.

In terms of verification, CλaSH users can enjoy access to Haskell's entire testing ecosystem. Since circuits in CλaSH can be simulated just by evaluating a function (they are represented as plain Haskell functions, after all) we can perform verification on circuit generators and their resulting circuits alike. A common choice is to use a property-based testing library such as QuickCheck [55]. These approaches provide excellent infrastructure for constrained-random testing, often making it easier to write such tests than less thorough example-based testbenches. Beyond the constrained-random testing present in SystemVerilog, QuickCheck also employs a shrinking technique to report failing test cases in their simplest, and often most insightful, form. While there is no built-in support for model checkers or theorem provers, one could configure this oneself, offering an environment with similar restrictions to Lava's formal verification.

CλaSH descriptions also eliminate whole classes of bugs statically by offering much better type safety than is encouraged by Lava. Programming with types which are indexed by their length, such as `Vec`, goes a long way towards eliminating common bugs allowable by Lava. Here the type checker can ensure statically that a function manipulates data soundly, at least in terms of its size. For example, the compiler catches bugs such as accessing an element from a (potentially) empty vector, or combining two vectors with (potentially) different lengths. We propose that a move towards a dependently typed HDL will simplify the implementation of these circuit descriptions, and open the doors for theorem proving directly in the source language.

2.4.3 Π-*ware*

Π-ware is an EDSL for hardware description embedded in the dependently typed software language, Agda [20]. Here, in the same style as Lava, circuits are encoded as data structures and the developer is offered a set of primitive gates and combinators to construct larger circuits. Since the previous two sections have discussed a comparison of the EDSL and standalone approaches to functional HDLs, we instead focus on Π-ware's unique feature: its choice of a dependently typed host language.

The fundamental advantage of moving to a dependently typed host is that it offers a single roof under which to describe, simulate, synthesise, and formally verify hardware descriptions without handing-off to external model checkers. In particular, the manual theorem proving facilitated by dependent types allows us to verify properties of entire circuit families at once, whereas most model checking techniques work well over only one concrete circuit at a time. These formal foundations are also demonstrated well by the implementation of Π-ware itself. Each semantic interpretation of a netlist (simulation, synthesis, ...) is defined as an algebra type which is folded over a given netlist.

Π-ware takes a more general perspective on circuit description that Lava. Instead of assuming that every circuit operates over a collection of bits and a library of FPGA-friendly gates, a netlist is parameterised by:

↪ The type of data represented by one logical wire, called an atom. This could be a boolean, a larger arithmetic type, a finite enumeration, etc.

↪ A set of primitive gates that operate over those atoms.

Instead of working with bits directly, we are free to raise the abstraction level to something like n-bit signed numbers, offering primitive gates for adders, subtractors, and negation. It is important to note that *all* wires must carry the same atom type — there are no Π-ware language features for splitting or inspecting atom values other than the black-box gates. We imagine most circuits likely opt for bit-level representations and construct larger structures through composition. For verification, it is *assumed* that the function supplied by the developer to simulate each gate's behaviour is correct. If these implementations do not match the physical gate behaviour then we will be able to construct bogus proofs of the real circuit's behaviour.

Writing descriptions at such a low-level can be a challenge, as highlighted by Pizani Flor in [20]. Furthermore, Π-ware does not provide named variable bindings within circuit descriptions which forces the programmer to use a *point-free* style, composing gates and combinators without reference to argument names. Many (human!) readers find this more difficult to parse for complex designs than a *pointed* description with bound variable names. The advantage of the point-free style is that the programmer can *never* accidentally construct circuits with certain mistakes such as combinatorial loops. Later work developing $\lambda\pi$-ware [56] does allow named variable binding in circuits for sharing and loops, helping improve legibility.

While we are free to use Agda's full-spectrum dependent types in our circuit generators, the circuit descriptions themselves are more restrictive. Each netlist type is

indexed by the number of input atoms (usually bits) and the number of output atoms. As an example, an adder circuit might be given the following type where $\mathbb{C}$ is Π-ware's netlist type:

$$\texttt{addN} : \forall\{n\} \to \mathbb{C}\ (n + n)\ (1 + n)$$

Here, two n-bit words are concatenated to form the input signal and the output signal is extended by one bit. This approach gives some type safety for the dimensions of subcircuits we connect together – i.e. chaining `addN` circuits together is only possible if we grow the input wordlengths for each stage or otherwise truncate them. However, beyond these dimensions, we struggle to encode any more information in the circuit's type. In a dependently typed language it really should be possible to write the type of `addN` such that it also encodes the arithmetic meaning of the output. Any implementations which pass type checking are then guaranteed to be correct without further verification effort — they would be correct-by-construction.

Later work for $\lambda\pi$-ware retains most of Π-ware's properties but does make special-case allowances for simple structures built of atoms, including vectors, products, and coproducts. These each come with complementary case-style constructs for elimination. Having product types in the circuit descriptions could improve our `addN` example, making the structure of the two input words more explicit. Ideally a compiler could automatically synthesise these features from suitable user-defined data types, as in CλaSH and `toatie`, rather than having ad hoc support for these three structures.

We are encouraged to think of circuit development in stages using Π-ware: first an implementation, then simulation, then ad hoc testing, and then formal verification. This is a valid approach and the dependently typed host language allows us to really nicely attack the formal verification aspect. We can 1) prove theorems about entire circuit families at once, and 2) do so under the same host language as the circuit's implementation. Π-ware also has an elegant solution for automatically and exhaustively checking equivalence between a (*small*) circuit and a reference function. This is analogous to Lava's toolchain using external model checkers. We still need manual proofs for any reasonably large circuits or over entire circuit generators.

However, we believe a limitation of Π-ware lies in the need for these distinct development phases. It is common practice in dependently typed programming to encode more about your data's *meaning* in its type. This not only gives us better type safety (and implementations which are more likely to be correct) but the types can often help inform our implementations interactively. If our types are specific enough, we can even have circuits whose functional behaviour is entirely *correct-by-construction*.

The proof of functional correctness and the implementation are developed simultaneously, and one helps inform the other symbiotically. Brady explores this concept thoroughly in [57], framing the process of writing software as an iterative conversation with the type checker rather than treating the machine as an adversary. Since Π-ware's structure somewhat limits our ability to use this approach outside generators, we advocate for a standalone implementation (CλaSH-style) with dependent types as an alternative. Chapter 5 builds towards a DFT circuit family in `toatie` where the *correct-by-construction* approach gives us a proof of arithmetic behaviour entirely for free, given a base set of arithmetic building blocks.

2.4.4 *Proposed circuits in Ωmega and Idris*

Here we discuss two pieces of related work towards theorem proving for circuit models. Our own work is strongly influenced by Brady's observations in [22]. He suggests an approach using a dependently typed language to verify properties of entire circuit families. This is achieved by encoding our data's complete meaning in it's type — for unsigned binary words, this includes its length and the natural number it represents. His approach offers two main advantages:

- ↪ Encourages circuits which are functionally *correct-by-construction*. The verification of these properties and the implementation of the circuit are interwoven in such a way that can actually assist its development — progress made in the proofs can inform parts of the implementation, or vice versa.
- ↪ It does not rely on external model checkers. This also sidesteps the state-space explosion present in automatic model checking. Not only can his approach be used to reason about very large circuit structures, it can reason about (potentially infinite!) circuit families.

The disadvantage is that, when model checking is applicable, it provides results *automatically*. This theorem proving approach is very much a manual process, although languages such as Idris have good support for theorem proving interactively (in conversation with the type checker). The work presented in [22] only simulates circuit behaviour and does not consider synthesis. This thesis extends upon these ideas and builds a system which also enables synthesis of circuits, exploring language features including erasure and staging in support of our synthesis process.

Contemporary to Brady's work in [22], is Sheard's similar contribution towards static verification of circuits [21]. Here, insights are offered into how circuits could be

encoded as a data type in the (software) language Ωmega. The central ideas are common between the two: perhaps we ought to be using types to verify circuit descriptions under one language. Sheard's perspective does differ in three important aspects:

↪ Circuits should be encoded as *data*, rather than normal *functions*. Hence, we count Ωmega implementations in our group of EDSLs, most closely resembling Π-ware. Sheard argues that a representation as data is more natural for circuits since a single netlist structure might be interpreted in a number of different ways: synthesis, simulation, or symbolically for verification. While this is true, our discussion has already highlighted that this EDSL approach makes describing choice structures inside a circuit's run-time much more awkward than in a language such as CλaSH that represents circuits as normal functions.

↪ Types can just as naturally be used for tracking non-functional properties. For example, one may wish to encode a metric representing a circuit's area or power consumption in its type. This allows the developer to reason about resource boundedness statically.

↪ Ωmega does not have full-spectrum dependent types. It instead provides a very consistent syntax for programming at any (potentially polymorphic) level — functions and data look much the same at the term-level, type-level, or in higher kinds. Singleton data structures are then used heavily to convey information between each level. It is interesting to note that this approach should be enough to encode every proof presented in Chapter 5. However, we prefer to explore dependent types in order to simplify this process, and to allow circuit generators unrestricted use of full-spectrum dependent types.

Although `toatie` was largely developed in isolation from the ideas of Ωmega, they both use multi-staged programming. Sheard's proposal uses staging only to optimise the software simulation of circuits. This sort of specialisation as optimisation is perhaps the most common use of multi-stage programming. However, `toatie` requires staging to ensure the circuit descriptions are, in a sense, causal. Since both our circuit descriptions and our circuit generators are encoded as normal functions, we need to ensure we can pass information about our variables forwards in time (from the circuit generator to the circuit's run-time) but not backwards (using information only knowable during the circuit's run-time during circuit elaboration).

2.4.5 *Proto-Quipper-D for Quantum Circuits*

Proto-Quipper-D has also heavily inspired the work in this thesis, despite being domain-specific for quantum circuits rather than classical circuits [58]. The focus on quantum circuits comes with a different set of restrictions for synthesisability. For example, one cannot straightforwardly fork a signal into two paths due to the "no-cloning" property of quantum mechanics. To help enforce this property, linearity is carefully considered in their type system — presented alongside a rigorous analysis of its categorical structure. Underneath these differences, we do share challenges including the staging distinction between circuit generator/circuit run-times and our ability to express entire circuit families effectively in a type-safe way.

To this end, Proto-Quipper-D chooses to distinguish between circuit generator values and circuit run-time values by maintaining two separate kinds of types. These are called *parameter* and *simple* types respectively. This is a concept retained in `toatie`. Proto-Quipper-D also supports dependent types. However, dependently typed data structures are only possible for *simple* (synthesisable) types which come with their own restrictions. This dependent typing of *simple* data is only really intended to encode the *structure* of collections of qubits — the equivalent of bits for quantum circuits.

> *Semantically, simple types corresponds to states. Syntactically, a simple type can uniquely determine the size and the constructors of its data. The type checker will check whether a simple data type declaration is well-defined.*
>
> — PENG FU IN [59]

One important subtlety here is that simple types must *unambiguously* determine their constructor. This allows length-indexed collections such as vectors and perfect binary trees since each type only has one possible constructor. As in Listing 2.8, a vector of length zero always corresponds to a `VNil` constructor and a vector of non-zero length always corresponds to a `VCons` constructor.

Listing 2.8: A synthesisable (valid) `Vec` type and a non-synthesisable (invalid) *Maybe* type in Proto-Quipper-D

```
-- Valid simple type
simple Vec a : Nat → Type where
  Vec a Z     = VNil
  Vec a (S n) = VCons a (Vec a n)
```

```
-- Invalid simple type
simple Maybe a : Type where
  Maybe a = Nothing
  Maybe a = Just a
```

A programmer may expect to be able to encode the familiar `Maybe a` type, either holding an element of type `a` or nothing at all. However, this is an illegal simple type in Proto-Quipper-D since there is ambiguity between two possible constructors.

The approach taken by CλaSH and `toatie` does allow for representation of these ambiguities, as any branches will synthesise down to a mux-controlled set of alternatives. Indeed this is the core feature which allows developers to encode circuits as idiomatic, plain functions!

Proto-Quipper-D somewhat blurs the lines between our notions of an EDSL and a standalone implementation. While it is not hosted in another language per se, they do provide black-box primitives for state types and the corresponding gate libraries. Developers can then represent quantum circuits as plain (linear) functions between simple types, using the primitive gates. The language also supplies *boxing* and *unboxing* operations which can convert these circuits between representations as functions and data. In a boxed/data form, we can better inspect and reason about a circuit's structure without supplying concrete inputs (Proto-Quipper-D even provides visualisation tools for this). In an unboxed/function form, we can very easily simulate a circuit's behaviour by treating it just like any other function. This is an interesting feature which we have not yet included in `toatie`, instead relying on external tools to perform visualisations given a synthesised netlist.

Verification also proves to be an interesting middle-ground in Proto-Quipper-D as well. While it is not considered a language "for general theorem proving" [59], it does have a type system strong enough to encode Leibniz equalities between terms. If we wanted to follow this concept to its extreme, one could adopt a similar methodology to Π-ware circuit verification (sans productivity features offered by Π-ware's host language). Here formal verification is treated very much separately to the implementation of the circuit description. Circuit generators are first written and *then* reasoned about. We instead advocate for adopting a correct-by-construction, type-driven approach. Since the qubits are a black-box primitives, there is no opportunity to index them by parameters precisely enough to ensure functional correctness in our circuit types alone.

2.5 SUMMARY

In this chapter, we have solidified the context for this thesis' later contributions towards circuit description and verification techniques.

Section 2.1 began with a brief summary of abstract circuit verification techniques, including the divide between dynamic/static verification, approaches towards completeness of testing, and the tensions between model checking and theorem proving. We continued by drawing case studies from both traditional HDLs (representing the industry status quo in Section 2.3) and the functional HDLs that are most relevant to the ideas presented in `toatie` (in Section 2.4). We note that most traditional HDLs have, over their histories, accrued multiple distinct features for abstraction and parameterisation which are usually condensed down to their most general forms by functional HDLs.

Within the landscape of functional HDLs in particular, there are two clear divisions between projects:

↪ Encoding circuits as data vs encoding circuits as plain functions.

↪ Treating verification separately to circuit description vs enabling correct-by-construction designs.

To the best of the author's knowledge, there has never been a project which generates synthesisable circuits by representing circuits as functions *and* encouraging full functional verification in a correct-by-construction manner. Implementing such a system is the goal of the remainder of this thesis.

3 AN ENGINEER'S INTRODUCTION TO DEPENDENTLY TYPED PROGRAMMING

A brief introduction to programming with dependent types, `toatie`'s syntax and semantics. We cover language features including pattern matching, irrelevance, staging, theorem proving, and GADTs, before we entertain their application to describing circuits in particular.

3.1 INTRODUCTION

A large part of this thesis is investigating concepts from academic computer science and promoting their enjoyment by the digital designer. A basic understanding of both fields is a prerequisite for this discussion — so this chapter offers an uncomfortable crash course in dependently typed programming for the straw man digital designer. All code examples in this chapter are provided in our `toatie` language, although it should look and feel extremely familiar for Idris programmers, and be largely understood by Haskell programmers.

3.2 BASIC FUNCTIONS AND DATA TYPES

Every declaration in `toatie` is either a function declaration or a data declaration. There are no built-in primitive data types (such as `int`,`float`, or `array`), so let's start by defining a simple arithmetic data type — the `Nat`ural numbers.

In the absence of any other arithmetic types, we can define our `Nat` type inductively in a form called Peano numerals[60]. We could say, in prose, a valid natural number is either zero, or one greater than another natural number (called the "successor" function). This definition translates quite directly to the way data types are defined in `toatie` — Generalised Algebraic Data Types (GADTs).

Listing 3.1: Data type for natural numbers

```
1 data Nat : Type where
2   Z :        Nat
3   S : Nat → Nat
```

Line 1 in Listing 3.1 introduces `Nat` as a new type (a "type constructor"), while lines 2 & 3 describe the two ways we can construct a `Nat` (the "data constructors"). The data constructor for zero, `Z`, takes no arguments and represents a valid member of the `Nat` type. The "successor" data constructor, `S`, takes exactly one argument which is of type `Nat` and represents another valid member of the `Nat` type. Some examples of `Nat`-typed values include:

$0 \Rightarrow$ `Z`

$1 \Rightarrow$ `S Z`

$3 \Rightarrow$ `S (S (S Z))`

$9 \Rightarrow$ `S (S (S (S (S (S (S (S (S Z))))))))`

`Nat`ural numbers are used heavily in later examples, so `toatie` will automatically translate/desugar numeric literals to this Peano representation behind the scenes.

With the `Z` and `S` data constructors, we have managed to precisely capture the *infinite set* of natural numbers. There are no values of type `Nat` which do not represent a natural number (e.g. no sneaky quirks such as `NULL` pointers) and no natural numbers without an equivalent `Nat` value. The inductive nature our definition also has some pleasing consequences, despite being wildly space inefficient. We will soon see when writing functions or proofs with `Nat`s that we often only need to handle two cases: the terminating/base case, and an inductive case.

This GADTs-style of defining data types may feel foreign to most digital designers, but basic analogies can be made to concepts present in C. Each data constructor has a (possibly empty) set of arguments. Each of these can be thought of as a `struct` in C; a heterogeneous collection of fields. A `Nat` value can be made via any *one* of the data constructors — picking one option from N constructors. This selection is analogous to a tagged union in C. An unidiomatic but structurally equivalent definition of `Nat` using C structures is shown in Listing 3.2.

Listing 3.2: A structural analogue for our `Nat` type in C

```
1  typedef struct Nat {
2    int tag;
3    union {
4      struct {              } Z;
5      struct {struct Nat* k;} S;
6    } data_cons;
7  };
```

Where this comparison begins to fail is with the level of assistance provided by the type checker. In our C example, the programmer is fully responsible for maintaining coherence between the `tag` value and corresponding interpretation of the union, ensuring that cases for all constructors are covered whenever we use the structure, as well as standard checks and memory management for the pointers. For `toatie`, and most other languages with algebraic data types, all of these concerns are alleviated from the programmer's mind and, instead, enforced by the type checker.

Now let us introduce function declarations by considering an operation over `Natural` numbers — the `plus` function shown in Listing 3.3. Every function is presented in two parts: the function's type (in line 1), followed by a set of pattern matching clauses. Our pattern matching clauses compile into a tree of `case` expressions.

Listing 3.3: Addition of `Natural` numbers

```
1 plus : Nat → Nat → Nat
2 pat y ⇒
3   plus Z y = y
4 pat x, y ⇒
5   plus (S x) y = S (plus x y)
```

The type of `plus` can be read as "a function that takes two `Nat` arguments and returns a `Nat`", where the output type is always the last identifier in the "→" separated list. In our example, we choose to pattern-match on the first argument only. The first clause (lines 2 & 3) handles the case where the first argument is zero, and the second clause (lines 4 & 5) handles the case where the first argument is the successor of another `Nat`, `x`. Together, these two clauses define addition of natural numbers recursively, written mathematically as:

$$\text{plus}(x, y) = \begin{cases} y & \text{if } x = 0 \\ 1 + \text{plus}(x', y) & \text{if } x = 1 + x' \end{cases} \tag{3.1}$$

Note that lines 2 & 4 are only to introduce pattern names used in the following clause's left-hand side (LHS), and to optionally specify their type. By default, `toatie` will attempt to check that every function is *covering* — i.e. it tries to ensure that all valid combinations of input values have been handled by the clauses, even when the arguments influence each other via dependent types. On that note, these examples have been simply typed so far. Let us now look at dependent types in `toatie` and the basic ways we can benefit from them.

3.3 DEPENDENT TYPES

Brady et al. eloquently introduce the motivation for dependent types as follows:

> *Dependent type theory provides programmers with more than an integrated logic for reasoning about program correctness. It allows more precise types for programs and data in the first place, strengthening the type checker's language of guarantees. We have richer function types $\forall x : S.T$ which adapt their return types to each argument; we also have richer data structures which do not just contain but* explain *data, exposing and enforcing their properties.*
>
> — BRADY ET AL. IN [61]

These claims feel quite profound but are readily demonstrable. Perhaps the most surprising thing about these claims is that they are *all* a consequence of one reasonably small design choice in the core language: allowing values (or "terms") to appear in types.

We will introduce these concepts with the obligatory basic examples, but Chapter 5 will demonstrate similar benefits with more complex structures and an eye towards circuit description. First, consider a homogeneous list. In `toatie`, we could describe a list as follows:

Listing 3.4: Definition of the `List` type

```
1 data List : Type → Type where
2   Nil  : (a : Type)                              → List a
3   Cons : (a : Type) → (x : a) → (xs : List a) → List a
```

Here we see dependent types being used to the same effect as polymorphism. An argument term, `'a'` (describing the type of our list's elements), appears in the type of later arguments, as well as the `List a` return type. Note that line 1 now shows that the `List` type constructor only returns `Type` after being applied to one argument — the type of our list's elements. As with our `Nat` example, there are two data constructors used inductively; one for the empty list, and one for an element appended to the head of an existing list. Below are some examples of list terms to help foster an intuition:

```
    [] for Nat ⇒ Nil Nat
[True] for Bool ⇒ Cons Bool True (Nil Bool)
  [3,1] for Nat ⇒ Cons Nat (S (S (S Z))) (Cons Nat (S Z) (Nil Nat))
```

As per this definition, we might sometimes get ourselves into trouble when programming with covering or total functions. For example, we try to implement a function to return the first element (the "head") of a given list in Listing 3.5.

Listing 3.5: A partial definition of `head` for a `List`

```
head : (a : Type) → List a → a
pat a ⇒
  head a (Nil a) = _
pat a, x, xs ⇒
  head a (Cons a x xs) = x
```

All is well for the `Cons` case, but what should the function return when the input list is empty? For a covering function, we do need to return *something* of type '`a`'. Without knowing exactly what type '`a`' will be ahead of time, we simply will not be able to guarantee that we return something correct in all situations. Many would be tempted just to throw a run-time error at this point, hopefully noting the behaviour in the documentation. One other solution is to add a new argument to `head`, acting as a default value to return if we identify an empty list at run-time. However, this run-time check will still impact performance.

Often the programmer will (correctly or otherwise) believe that their particular list is special — it cannot possibly be empty because of some reasoning made outside of the programming language — and thus could skip any of the run-time checks. Making this leap unassisted is usually a dangerous move, but we can actually have the type checker aid us if we make the type of `List` more precise. Listing 3.6 shows a version of `head` on `Vect`ors, a version of lists that track their length in their type.

Listing 3.6: A covering definition of `head` for a `Vect`

```
simple Vect : Nat → Type → Type where
  VNil  : (a : Type)                                  → Vect Z     a
  VCons : (a : Type) → (k : Nat) → a → Vect k a → Vect (S k) a

head : (a : Type) → (n : Nat) → Vect (S n) a → a
pat a ⇒
  head a Z [] impossible
pat a, n, x, xs ⇒
  head a n (VCons a n x xs) = x
```

In this example, empty vectors with zero length can only be constructed with `VNil`, and appending an element to an existing vector with `VCons` will always increase the length by one. For the implementation of `head`, the caller must pass in a vector with length '`S n`' — i.e. any non-zero `Nat`. Now we can dismiss the clause handling empty vectors as `impossible`, or simply omit it entirely. The type checker will ensure that all

impossible clauses have a contradiction in the types, providing certainty that they can be safely excluded.

As an example, let's see how `toatie`'s Read–Eval–Print Loop (REPL) reacts when we try to pass some `Vect` terms to the `head` function. Note that we can opt to leave arguments implicit by replacing them with an underscore. When there is a unique solution to this hole, `toatie` will attempt to find it using unification process similar to that of Idris [18].

Listing 3.7: Examples of calling our type-safe `head`

```
> head Nat _ (VCons _ _ 5 (VNil _))
Type: Nat
Evaluated: 5

> head Nat _ (VCons _ _ 9 (VCons _ _ 0 (VNil _)))
Type: Nat
Evaluated: 9

> head Nat _ (VNil _)
Type mismatch: 0 and (S ?{_:3})
```

Indeed the first two attempts work as hoped but the third attempt, trying to sneak in an empty vector argument, fails with a type error. The type checker claims there is a mismatch between `0` (our vector's length) and `(S ?{_:3})` (where the function is expecting a non-zero length). A crucial aspect is that we encounter this error statically *at compile-time*, rather than deferring this to a run-time check which would incur an overhead and can prove difficult to test for seldom visited branches.

We have experienced a small taste, in its simplest form, of how dependent types can help refine our function types and data declarations. This example uses dependent types to help restrict the domain of an argument via a simple constructor application, `S`. We could use them in a similar way to adapt the function's return type based on each/every argument. Further, we are not restricted to just a simple constructor application — we can use any valid term (other functions, choice constructs, etc.) at the type-level, refining our types arbitrarily.

Considering our current `Vect` implementation, we might start to question its run-time efficiency. We have introduced extra terms to please the type checker, so how does this impact space-requirements or circuit area?

3.4 IRRELEVANCE AND ERASURE

In most statically typed languages without dependent types, there is a reasonably clear separation between term-level expressions and type-level expressions. The type annotations may, broadly speaking, be discarded after type checking (or "erased") and will not be present in a compiled binary. However, in the presence of dependent types, this separation no longer clearly falls on the boundary of terms vs. types since the two are permitted to mingle. There is still a split between compile-time and run-time requirements, but this must be guided by some extra features: in our case, irrelevance and erasure.

In Section 3.3, we introduced a `Vect` type which is parameterised (or "indexed") by its element type and its length. Doing so allows the type checker to better assist us in writing and using functions correctly. The type of a function can restrict an input's domain (e.g. restricting the length of a vector) or precisely model the length of a returned vector (e.g. appending two vectors with length n and m will yield an output of length $n + m$). As it stands, these indices will remain in our program's run-time and have a substantial impact on the memory requirements of a `Vect`. We would perhaps expect the run-time representation of a `Vect` to be a linked-list of elements of a single type. However, we end up with something more akin to a linked-list of triples: the element type (replicated in every single element!), the length of the vector's tail as a `Nat`, and the element term itself.

To step towards a linked-list representation at run-time, we can annotate the vector lengths and element types as "irrelevant"; they are required during type checking but are totally irrelevant at run-time. In `toatie`, we do this by surrounding these arguments with curly braces (not to be confused with Idris 2's use of the same syntax). Our vector example from Listing 3.6 becomes:

Listing 3.8: A `Vect` example with irrelevant element type and length indices

```
1 simple Vect : Nat → Type → Type where
2   VNil  : {a : Type}                                  → Vect Z     a
3   VCons : {a : Type} → {k : Nat} → a → Vect k a → Vect (S k) a
4
5 head : {a : Type} → {n : Nat} → Vect (S n) a → a
6 pat a ⇒
7   head {a} {Z} [] impossible
8 pat a, n, x, xs ⇒
9   head {a} {n} (VCons a n x xs) = x
```

These irrelevance annotations are sufficient to ensure that the vector length and element type will only be used during type checking, and then erased from the final

program/circuit description. Of course, there some extra restrictions on how we are allowed to use irrelevant terms since they are no longer available at run-time. Our implementation uses ICC*, with all of their typing rules presented in [62]. As we are just trying to gain an intuition of how to program in our language, an informal definition of these rules will suffice:

↪ Relevant/explicit terms may appear in *any* position. In other words, run-time terms can be used at run-time *and* during type checking.

↪ Irrelevant/implicit terms may *only* appear in implicit positions. Equivalently, once a term is marked as for type checking use only, it must never be required at run-time again.

There is one other subtlety which concerns pattern matching on irrelevant terms. Since our pattern matching definitions get reduced to a series of run-time `case` statements, should the programmer be allowed to pattern-match on irrelevant terms which will be erased before run-time? In general, no, we cannot make decisions based on information which has already been discarded. There is, however, a special case which we will often exploit; called "inaccessible patterns".

Again, an informal description serves our purposes here, while the interested reader may refer to [63] for a more formal handling of inaccessible patterns. We allow pattern matching to refine an irrelevant term, if and only if the pattern is uniquely identified by the other patterns in the clause. In other words, if the given pattern is the only possible solution, we can use it regardless of its run-time availability. As an example of where using inaccessible patterns is necessary, let's consider a function to append two vectors.

Listing 3.9: Append two `Vects` using inaccessible patterns for length

```
1 append : {a : Type} → {n : Nat} → {m : Nat} →
2          Vect n a → Vect m a → Vect (plus n m) a
3 pat a, m, ys ⇒
4   append {a} {Z} {m} (VNil {_}) ys = ys
5 pat a, n, m, x, xs, ys ⇒
6   append {a} {S n} {m} (VCons {_} {_} x xs) ys =
7     VCons {_} {_} x (append {_} {_} {_} xs ys)
```

Listing 3.9 does type-check as-is, but only because our clauses can use inaccessible patterns on the (irrelevant) argument, `{n}`. For example, if the first clause did not refine `{n}` to `{Z}`, we encounter a type error: `Type mismatch: n and 0`. This is because the type of `append` guarantees that the output length with be `(plus n m)` but we return `ys` with length `m` — leaving the type checker unconvinced that `(plus n m)` = `m`,

unless we also know that `n=0`. Remember that this is a valid inaccessible pattern since `Z` is the *only* possible form that `n` can take given that the first vector must match `VNil`.

3.5 STAGING

Staging techniques can prove useful for code generation, having the type checker assist us in deciding *when* an expression can be evaluated. This is subtly different from Section 3.4's discussion of irrelevance. Instead of limiting the use of certain terms to guarantee their erasure from a program/circuit description, we now want to limit the use of certain terms to guarantee that we can fully evaluate parts of our structure at certain times.

Although we do not focus on circuit description until Chapter 5, our main use case of this feature is to separate what should be evaluated during the (software) elaboration of a circuit and what should be evaluated during the elaborated circuit's run-time. We can think of this feature as allowing for arbitrary, user-defined circuit elaboration — a powerful generalisation of the `generic` and `for generate` constructs in VHDL. The staging rules we introduce help us keep track of *when* (in which "stage") variables are introduced, and maintain causality when we share variables between different stages. For example, a value at the input of a circuit cannot be predicted until the circuit is running, so should not be available during the elaboration of the circuit. If we fail to maintain this distinction, we will not be able to fully synthesise our circuit to a netlist.

We introduce four annotations to enable staging in `toatie` — identical to the software constructs in [64], an extension of the three in [65], and those theoretically applied to circuit description in [66]. These four annotations are:

- $[\![\ldots]\!]$ A quote defers the evaluation of a given term to a later stage.
- $\langle\ldots\rangle$ A `Code` type represents the type of a quoted term.
- $^{\sim}(\ldots)$ An escape splices a quoted term into a lower level, forcing its evaluation sooner.
- $!(\ldots)$ An evaluation is similar to an escape but specifically for splicing a quoted term into stage zero. The inner term must contain no free variables.

To demonstrate the use of these annotations without the baggage of circuit description applications, let us consider the standard literature example of staging: the `power` function, x^i. Listing 3.10 shows one version of this function without staging constructs and another with staging.

Listing 3.10: The `pow` functions without (left) and with (right) staging annotations

```
pow : Nat → Nat → Nat
pat x ⇒
  pow Z x = 1
pat i, x ⇒
  pow (S i) x =
    mul x (pow i x)
```

```
pow : Nat → ⟨Nat⟩ → ⟨Nat⟩
pat x ⇒
  pow Z x = ⟦ 1 ⟧
pat i x ⇒
  pow (S i) x =
    ⟦ mul ~x ~(pow i x) ⟧
```

The introduction of the staging annotations is not particularly intrusive: the structure of the function is identical, each with one recursive call. The benefits we gain are that the recursive call is *explicitly* unrolled (e.g. we expand x^3 to $x \times x \times x \times 1$ immediately) and our use of the quoted argument, `x`, is controlled to ensure that we can always perform this elaboration without inspecting `x`. Without the staging annotations, we cannot guarantee that our intended elaboration process will complete fully, possibly returning a non-synthesisable circuit.

The type checker ensures that all variables bound in stage x are only used in stage y if $y \geq x$, maintaining causality between the stages in our program. The pattern matching definitions in `toatie` provoke one final restriction: we can only use inaccessible patterns for quoted arguments. Informally, pattern matching on a quoted term in general is equivalent to "using" that term at an earlier stage without the safety of the escape/evaluate constructs, and is disallowed. Chapter 5 gives examples, including a binary adder, which require the use of inaccessible patterns on quoted terms.

3.6 THEOREM PROVING

We have previously hinted at encoding proofs in languages with dependent types. This is likely the concept, thus far, that seems the most alien to a digital designer. Towards theorem proving, we must convince ourselves that there are fundamental symmetries between formal logic and programs. While side-stepping much of the nuance, there is a set of observations known as the Curry-Howard correspondence which tells us that logical propositions are equivalent to types and that their proofs are equivalent to programs of that type [67].

We have already encountered scenarios where we rely on our type checker to ensure some property is true — e.g. the input vector for `head` has a non-zero length. Proofs are similar; a function with type that is specific enough to guarantee some property. To avoid trespassing too far into philosophy during our practical introduction, let's ground ourselves by considering what equality really means in a language like `toatie`.

The *only* way to introduce an equality is through reflexivity; stating $x = x$. For this definition, we can describe equality just like any other data type:

Listing 3.11: A definition of propositional equality

```
data Equal : (A : Type) → A → A → Type where
  Refl : {A : Type} → {x : A} → Equal A x x
```

In true functional style, we can also equip ourselves with methods of composing and applying equalities. Our four main functions, with types shown in Listing 3.12, adhere to the naming conventions presented in [62].

Listing 3.12: Common helper functions for equalities

Symmetry:
If $x = y$ then $y = x$

```
eqSym  : {A : Type} →
         {x : A} →
         {y : A} →
         (p : Equal A x y) →
         Equal A y x
```

Transitivity:
If $x = y$ and $y = z$ then $x = z$

```
eqTrans : {A : Type} →
          {x,y,z : A} →
          (l : Equal A x y) →
          (r : Equal A y z) →
          Equal A x z
```

Congruence:
If $x = y$ then $f(x) = f(y)$

```
eqCong  : {A,B : Type} →
          {f : A → B} →
          {x, y : A} →
          (p : Equal A x y) →
          Equal B (f x) (f y)
```

Induction:
If $x = y$, we can rewrite $P(x)$ as $P(y)$

```
eqInd2  : {A : Type} →
          {x,y : A} →
          {p : Equal A x y} →
          {P : (A → Type)} →
          (val : P x) →
          P y
```

Let's state a few simple propositions and try to develop valid proofs for them *interactively*, in conversation with the type checker. Along this short journey, we'll gain an intuition for what `toatie` will be able to infer automatically, when we need to give it a nudge by refining our pattern matching, and when we need to offer more explicit direction. First consider a simple, concrete example without any unknowns: $2 + 2 = 4$. We can encode this proposition using the `Equal` type constructor from Listing 3.11. If we leave the right-hand side (RHS) of our clause implicit (an underscore) `toatie` will report the type it is expecting on the RHS, alongside the type of all local variables in scope.

Here, the type checker claims it expects a value of type `(Equal Nat 4 4)`: it has normalised the expression `(plus 2 2)` to 4 by itself. We can always expect this when an expression has no unknown/"free" variables since there is an evaluation step during type checking. We only need to demonstrate that $4 = 4$ and we can do so simply with the `Refl` constructor from Listing 3.11. Substituting `Refl {_} {_}` for our implicit RHS

Listing 3.13: Interrogating `toatie` about our proposition $2 + 2 = 4$

```
twoPlusTwo : Equal Nat (plus 2 2) 4
twoPlusTwo = _
```

```
Unresolved holes in clause twoPlusTwo = ?{_:958}

Holes:
--------------------------------
{_:958} : (Equal Nat 4 4)
```

completes the definition and the example passes type checking — we have provided a proof that $2 + 2 = 4$, as verified by the type checker!

Moving on to another arithmetic proof, we can explore propositions with universal quantification (a "for all" or $\forall$). As a simple example, let's try to demonstrate that $0 + x = x$. We want to show that this proposition is true for all values of x. As before, let's encode this as a function type in `toatie` and ask to see what it expects of the proof.

Listing 3.14: Interrogating `toatie` about our proposition $0 + x = x$

```
plusZeroLeftNeutral : (x : Nat) → Equal Nat (plus 0 x) x
pat x ⇒
  plusZeroLeftNeutral x = _
```

```
Unresolved holes in clause (plusZeroLeftNeutral x[0]) = ?{_:962}

Holes:
x:_0 Nat
--------------------------------
{_:962} : (Equal Nat x[0] x[0])
```

We again only need to demonstrate an equality through reflexivity: $x = x$. The definition of `plus` has been normalised away again, in this case, reducing `(plus 0 x)` to `x`. A really important point is that normalisation only manages to reduce the expression due to the way we have defined `plus` (back in Listing 3.3) via recursion on the first argument, and without any pattern matching on the second argument. If the second argument was inspected in the definition of `plus`, the normalisation process would get stuck and we would need to do a little more work to satisfy the type checker. To demonstrate such a scenario, let's look at a subtly different proposition: $x + 0 = x$. Although this feels obvious given our previous proof, keep in mind that we haven't proved anything about the commutativity of `plus` yet, so we'll continue with caution.

We finally have a scenario where `toatie`'s normalisation alone is not enough to reduce our problem to a trivial solution. Here, the definition of `plus` immediately tries to pattern-match on its first argument, `x`. Since the value of `x` is unknown at this time,

Listing 3.15: A first interrogation of `toatie` about our proposition $x + 0 = x$

```
plusZeroRightNeutral : (x : Nat) → Equal Nat (plus x 0) x
pat x ⇒
  plusZeroRightNeutral x = _
```

```
Unresolved holes in clause (plusZeroRightNeutral x[0]) = ?{_:967}

Holes:
x:_0 Nat
--------------------------------
{_:967} : (Equal Nat (plus x[0] 0) x[0])
```

the normalisation does not continue any further. We can still complete a proof of our proposition if we can demonstrate that it holds true for *every* valid data constructor of `x`. Let's refine our example by introducing pattern matches for `x`'s two data constructors: `Z`, and `S`.

Listing 3.16: A case-split interrogation of `toatie` about our proposition $x + 0 = x$

```
plusZeroRightNeutral : (x : Nat) → Equal Nat (plus x 0) x
plusZeroRightNeutral Z = _
pat x ⇒
  plusZeroRightNeutral (S x) = _
```

```
Unresolved holes in clause (plusZeroRightNeutral 0) = ?{_:967}

Holes:
--------------------------------
{_:967} : (Equal Nat 0 0)

Unresolved holes in clause (plusZeroRightNeutral (S x[0])) = ?{_:971}

Holes:
x:_0 Nat
--------------------------------
{_:971} : (Equal Nat (S (plus x[0] 0)) (S x[0]))
```

Now there are two holes for us to complete, but the types of each have been slightly reduced by normalisation. For the case where $x = 0$, we're only required to provide a simple `Refl {_} {_}` to satisfy our target, $0 = 0$. The second case is a little more interesting. Notice that our target type *is* reduced, but only quite subtly — from `(plus (S x) (S x))` to `(S (plus x 0) (S x))`. Here we enjoy the benefits of defining our `Nat` numerals inductively; we can write our proofs inductively too! Let's recurse on our proof with a `Nat` argument one smaller than the one we've been presented and see how its type compares to our target.

We can see from the compiler output in Listing 3.17 that our inductive hypothesis

Listing 3.17: A final interrogation of `toatie` about our proposition $x + 0 = x$

```
plusZeroRightNeutral : (x : Nat) → Equal Nat (plus x 0) x
plusZeroRightNeutral Z = Refl {_} {_}
pat x ⇒
  plusZeroRightNeutral (S x) =
    let rec = plusZeroRightNeutral x in _
```

```
Unresolved holes in clause (plusZeroRightNeutral (S x[0])) = let rec :_0 (
    Equal Nat (plus x[0] 0) x[0]) = (plusZeroRightNeutral x[0]) in ?{_:971}

Holes:
x:_0 Nat
rec:_0 (Equal Nat (plus x[0] 0) x[0])
--------------------------------
{_:971} : (Equal Nat (S (plus x[1] 0)) (S x[1]))
```

has the type `Equal Nat (plus x 0) (x)`, which structurally similar to our target of `Equal Nat (S (plus x 0)) (S x)`. All we need to do now is use our congruence rule (`eqCong` from Listing 3.12) to apply `S` to both sides of the equality. Listing 3.18 shows our complete proof of $x + 0 = x$.

Listing 3.18: A complete proof of our proposition $x + 0 = x$

```
plusZeroRightNeutral : (x : Nat) → Equal Nat (plus x 0) x
plusZeroRightNeutral Z = Refl {_} {_}
pat x ⇒
  plusZeroRightNeutral (S x) =
    let rec = plusZeroRightNeutral x
    in eqCong {_} {_} {S} {_} {_} rec
```

While it is completely valid to make recursive calls to our proofs, care must be taken to ensure we keep the function *total*. We must have a *structurally smaller argument*, ensuring that the function will return in finite time and not become stuck in an infinite loop. While languages like Idris 2 explicitly check for totality (to keep type checking decidable), `toatie` only checks that a function is covering, so do be careful when making recursive calls. Making an infinite loop by recursively calling the same function with the same arguments is analogous to the playground circular reasoning of "x is true because x is true".

Through this short introduction to theorem proving, we have seen that our dependently typed setting allows:

↪ Encoding propositional equality as a data type

↪ Encoding proofs of properties as normal functions with a precise enough type

↪ Automatically "solving" or reducing equalities via normalisation when pattern-matched values are known a priori.

↪ Completing "stuck" proofs by pattern matching, rewriting with equality helper functions, and, very often, induction.

↪ Writing functions/proofs *interactively* by leaving implicit "holes" in the definition and letting the type checker report its expectations alongside a summary of the names in scope.

Although not an exhaustive exploration by any means, hopefully the mystery surrounding constructing formal proofs as dependently typed programs has been dissipated somewhat — thanks to the Curry-Howard correspondence [67], we are just reusing the same features we met while writing more traditional programs.

3.7 SUMMARY

This chapter has provided a concise introduction to the central software programming features and themes of `toatie`, shared with most dependently typed languages, aimed at the curious digital designer. While it still may require further reading and a some experience of dependently typed programming in anger to fully appreciate, this is hopefully sufficient to arm the reader with the software programming techniques that are foundational to the rest of this thesis.

The interested engineer is directed to [57] for thorough treatment of dependently typed programming in Idris, [68] for an excellent exploration into theorem proving with Coq, and [69] to be led on a charming amble through the woods of dependent types, pausing to take in only the most beautiful sights.

4 EXPLORING PARALLEL FIR FILTERS FOR RFSOC APPLICATIONS WITH CλASH

We present a new family of low-cost, high-speed, parallel FIR filters targeting direct Radio Frequency (RF) sampling applications with the Xilinx Zynq UltraScale+ RF System on Chip (RFSoC). This chapter is an extended version of this project's conference paper "Low-cost, High-speed Parallel FIR Filters for RFSoC Front-Ends Enabled by CλaSH" [25], and a continuation of the themes from the IEEE Access journal paper "Control and Visualisation of a Software Defined Radio System on the Xilinx RFSoC Platform Using the PYNQ Framework" [30].

4.1 INTRODUCTION

We investigate the successes and limitations of circuit description in CλaSH through the lens of a challenging case study. We present a new low-cost, high-speed parallel Finite Impulse Response (FIR) filter generator targeting RFSoC and direct RF sampling applications. We compose two existing approaches in a novel hierarchy:

↪ Efficient parallelism with Fast FIR Algorithm (FFA) structures [70].

↪ Efficient multiplierless subfilter implementations with H_{cub} [71], which intelligently decomposes a set of constant multiplications into a graph of simple adders.

The cumulative resource usage advantages (in both area and type) are compared with similar output from the canonical approach, exemplified by vendor tools, as well as the H_{cub}-based filters without the FFA optimisation. Although these techniques are well studied individually in the literature, they have not enjoyed mainstream use as their structural complexity proves awkward to capture with traditional HDLs. This work continues a discussion of the use of functional programming techniques in hardware description, highlighting the need for easily composable, staged circuit families. These insights offer a dual purpose:

1. Exploring a novel, open access, implementation of a circuit family for a real-world application. We improve substantially on the footprint of the vendor tools, both in terms of area and the type of resources required.

2. A practical insight into the successes and limitations of existing tooling. The application of parallel, multiplierless FIR filtering proves particularly challenging for description because of the complexity of its *compile-time* computations and demand for recursion. This helps inform the design of our own language, discussed in Chapters 5 and 6.

Before looking at this design, we provide its context and justify its motivation. The RFSoC family of devices (among other direct RF sampling devices) demand fast filtering stages which operate over a number of samples in *parallel*, since the multi-GHz sampling clock is necessarily higher than the fabric clock of the internal FPGA. Figure 4.1 shows an overview of the XCZU28DR RFSoC's FPGA and RF capabilities, highlighting the relative scarcity of hardened DSP48E2 resources and the number/throughput of the RF Analogue–Digital Converters (ADCs) and Digital–Analogue Converters (DACs).

Specific RFSoC use cases for parallel FIR architectures are plentiful. This is exemplified by the vendor support for a set of parallel, or "Super-Sample Rate" (SSR), circuit generators [72]. Example use cases include:

↪ Instrumentation applications that demand processing of the full available spectrum, including arrays for radio astronomy [73] and quantum computing readouts [74].

↪ Channelisation of millimetre wave Intermediate Frequency (IF) signals, including 5G NR (FR2) [75]. Such a signal can contain multiple baseband channels, occupying the whole 4 GHz spectrum provided by the RFSoC and demanding parallel filtering architectures for channel (de)multiplexing.

↪ Custom Digital Up/Down Conversion (DUC/DDC) as a front-end of *any* radio application. Especially useful when the characteristics of any available hardened DUC/DDCs [76] do not meet the application's requirements.

The demand for sample-parallelism and the multi-channel nature of the RFSoC device amplifies the effects of filter resource usage, making optimal filter implementation a renewed battle in the context of RFSoC systems. These optimisations are the concrete focus of our presented work. However, an equally important theme is the reflection on our practical implementation experience, and musing about language features which could better facilitate similar circuits. Our open source example in CλaSH highlights the realisable benefits of optimisations whose theory is well studied but whose implementations remain, for the most part, in academic folklore.

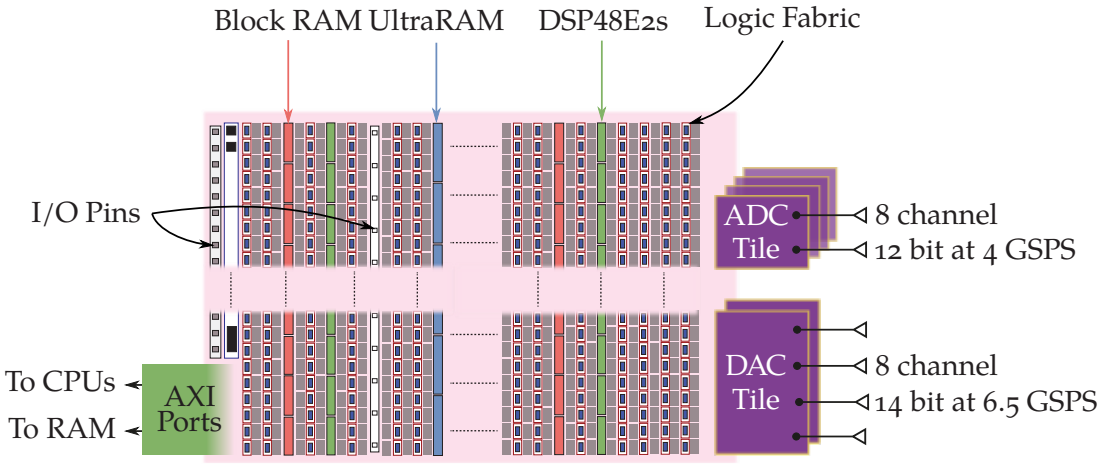

Figure 4.1: Overview of RFSoC's FPGA and RF Data Converters

4.2 BACKGROUND ON DIGITAL, FINITE IMPULSE RESPONSE FILTERING

Before considering optimised filter designs, let's first look at the foundations of digital filtering. Most filter designs aim to change the frequency content of an input signal. Different applications will motivate different changes to the frequency content — these choices are well motivated in introductory texts, such as [77].

Mathematically, this filtering behaviour is achieved by computing a weighted average over a finite window of the input signal. More formally, given a set of weights, w_0 to w_{N-1}, and a time-varying input signal, x, we would like to implement:

$$y_{[k]} = \sum_{i=0}^{N-1} w_i \cdot x_{[k-i]} \tag{4.1}$$

This is essentially just a dot product operation, which is used as a running example throughout Chapter 5. It may not be obvious why this operation actually achieves a filtering effect, even with a carefully chosen set of coefficients. For a basic intuition, think of the desired change in frequency content as a function of frequency — a filter's *frequency response*. The filtering process just multiplies the input's frequency components with the filter's frequency response. The Fourier transform tells us that this *multiplication* in the frequency domain translates to the time domain *convolution* from Equation (4.1). Lyons offers a full introduction to these topics in [77].

4.2.1 *Filter specification*

The behaviour of a filter is designed through careful choice of the coefficients (or "weights"). Figure 4.2 visualises one possible set of weights and the resulting filter's performance. The magnitude response in Figure 4.2 confirms that the example is a half-band filter — one designed to pass the lower half of the frequency content while blocking the upper half.

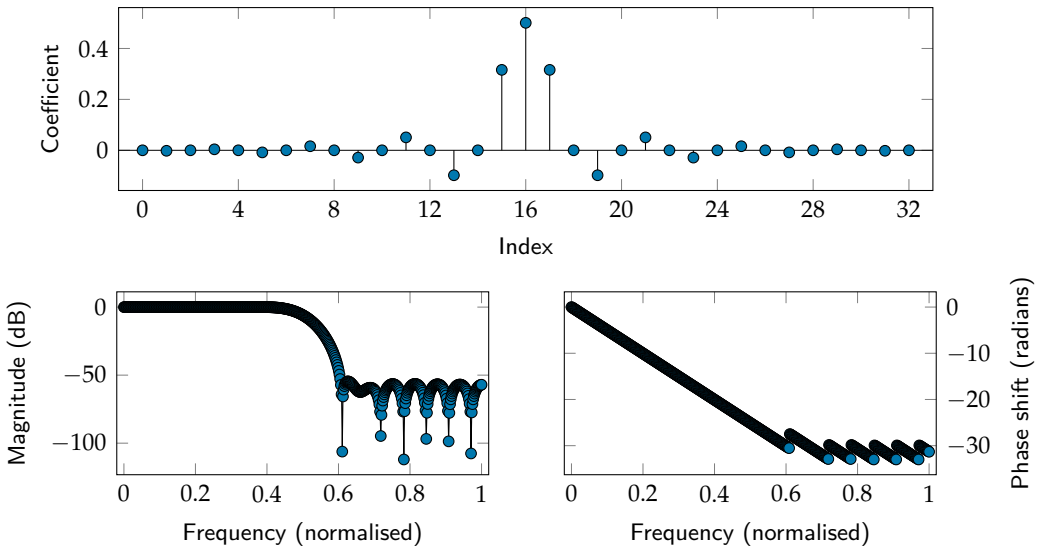

Figure 4.2: Specifications of an example half-band FIR filter, featuring its coefficients (top) with the resulting magnitude response (left) and phase response (right)

Another important property of the filter is its phase response. Figure 4.2 shows a phase response which is linear throughout the filter's pass-band ($0.0 \rightarrow 0.5$), and piecewise linear across the remainder of the spectrum. The linearity of the phase response is directly linked to the symmetry of the chosen weights. Achieving a linear phase response is an extremely common requirement for digital communications. Many modulation schemes encode useful information in a signal's phase — and so a filter should not distort the phase content. For the applications covered in this chapter, we expect the chosen coefficients to be:

$\hookrightarrow$ Symmetric (or antisymmetric)

$\hookrightarrow$ Constants (the filter behaviour is fixed)

The optimisations presented in this chapter will exploit both properties.

4.2.2 *Filter implementation*

The implementation of the filter itself can be as simple as mapping Equation (4.1) directly to hardware resources. A delay line is used to prepare the $x_{[k]}$ to $x_{[k-N-1]}$ window, and arithmetic blocks implement the dot product. This design, shown in Figure 4.3, is called *direct form* (or *standard form*). These direct form filters are often sufficient, but they have a long critical path through the entire adder chain (highlighted in purple in Figure 4.3), limiting the circuit's maximum clock frequency.

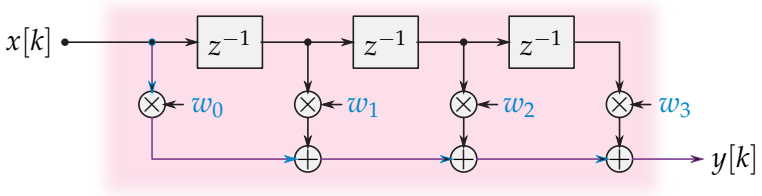

Figure 4.3: A four-weight direct form FIR filter

Most high-speed designs are based on one of the two alternative structures shown in Figure 4.4. The direct form's long critical path can be split by inserting registers in a balanced fashion, at the cost of additional filter latency. In the extreme, this results in the *systolic form* filter (noting the delayed $y_{[k-3]}$ output in Figure 4.4). The *transpose form* offers a solution without additional latency by using flow graph reversal (noting the reversed coefficients in Figure 4.4).

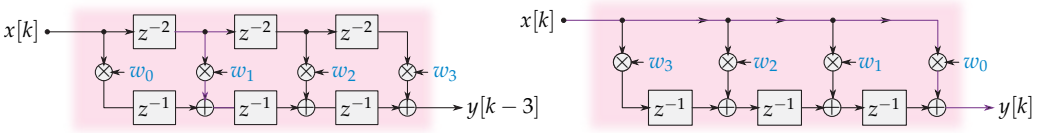

Figure 4.4: A four-weight systolic form (left) and transpose form (right) FIR filters

4.2.3 *Sample-parallel filtering*

So far, we have only considered filtering a stream of samples serially. For RFSoC devices, it would be valuable to process multiple samples concurrently instead — recall that its maximum sample rate greatly exceeds the maximum FPGA clock frequency.

Consider a parallel filtering structure that accepts two samples every cycle ($x_{[2k]}$ and $x_{[2k+1]}$) and also outputs two samples ($y_{[2k]}$ and $y_{[2k+1]}$). Our circuit clock is operating at half the sample-rate, and has a period of $2k$. The challenge here lies in how

we recover the *odd* offsets into x, using only $2k$ increments. The game is to rearrange Equation (4.1) to be in terms of even-valued delays on the input samples (i.e. $x_{[2k-2d]}$ and $x_{[2k+1-2d]}$). We can do this for $y_{[2k+1]}$ by splitting the summation into two terms:

$$y_{[2k+1]} = \sum_{i=0}^{(N-1)/2} w_{2i} \cdot x_{[2k+1-2i]} + \sum_{i=0}^{(N-1)/2} w_{2i+1} \cdot x_{[2k-2i]} \tag{4.2}$$

Each of these two terms are essentially sub-filters that are now amenable to our clock period of $2k$. The first only uses the even coefficients, and the second only uses the odd coefficients. These subfilters are commonly referred to as $H0$ and $H1$, respectively. A similar expression can be derived for $y_{[2k]}$:

$$y_{[2k]} = \sum_{i=0}^{(N-1)/2} w_{2i} \cdot x_{[2k-2i]} + \sum_{i=0}^{(N-1)/2} w_{2i+1} \cdot x_{[2k+1-2(i+1)]} \tag{4.3}$$

Note the additional one-cycle delay hidden in the second subfilter term. The full implementation of this 2-by-2 parallel filter requires a total of four subfilters; each with half the original number of coefficients. This particular architecture will be visualised shortly in Section 4.3.1 but these structures are, more generally, called *polyphase* filters. Harris provides excellent further reading for multi-rate, polyphase filtering in [78].

4.3 PROPOSED FILTER ARCHITECTURE

To improve upon the traditional parallel filter architecture, exemplified by the System Generator SSR blockset [72] and LogiCORE FIR Compiler [79], we employ two well studied but seldom implemented techniques. We compose these techniques into a novel hierarchy; one optimisation for the general structure of the filter's parallelism, and another optimisation for the multiplications required within each subfilter. These techniques have enjoyed wide discussion in the literature [71, 80–82] but are less often seen as practical implementations since they both prove to be extremely awkward to describe, at least in a general form, with traditional HDLs.

The following subsections describes the evolution of this structure from the traditional architecture, to a new multiplierless polyphase structure, and finally to our proposed multiplierless Fast FIR Algorithm (FFA) implementation.

4.3.1 *Traditional Architecture*

The traditional architecture for parallel FIR filters is a polyphase structure with systolic subfilters, mapped to specialised DSP48E2 resources. This approach is exemplified by the LogiCORE FIR Compiler [79] and is often used via System Generator's SSR blockset [72].

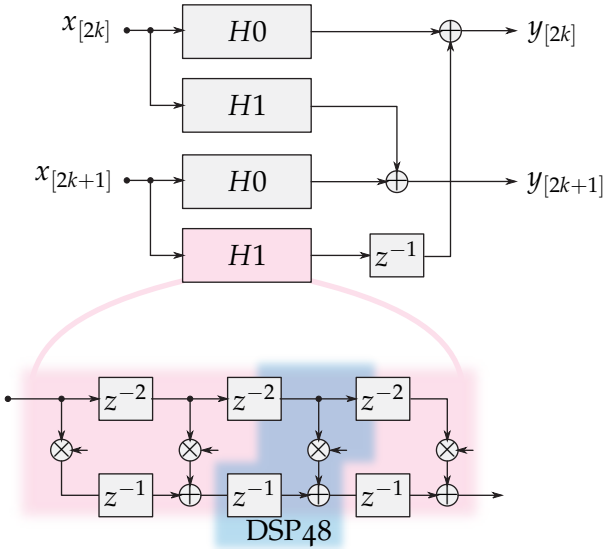

Figure 4.5: Example SSR implementation (Polyphase with systolic subfilters) for 8 non-symmetric weights

These SSR structures can exploit coefficient symmetry, although we only visualise the non-symmetric subfilter architecture in Figure 4.5. Figure 4.6 shows that the footprint scales approximately linearly with the level of parallelism (noting that each subfilter halves in length). The exception to this linear scaling is the introduction of extra adders and registers for phase recombination.

The following section begins to extend this SSR structure, introducing multiplier-less subfilters and exploiting resource sharing between subfilters.

4.3.2 *Polyphase Filter with Shared Multiple Constant Multiplication Subfilters*

As our first step, consider one of the subfilters in isolation. Figure 4.7 shows the systolic form being replaced with a transpose form. All of the multiplications now share the input as a common operand — a property which we will exploit despite the higher fan-out* of the input signal. The shared input gives us an opportunity to share resources

*Being the number of input pins driven by a given output pin. Extremely high fan-out can impact the routability of a design.

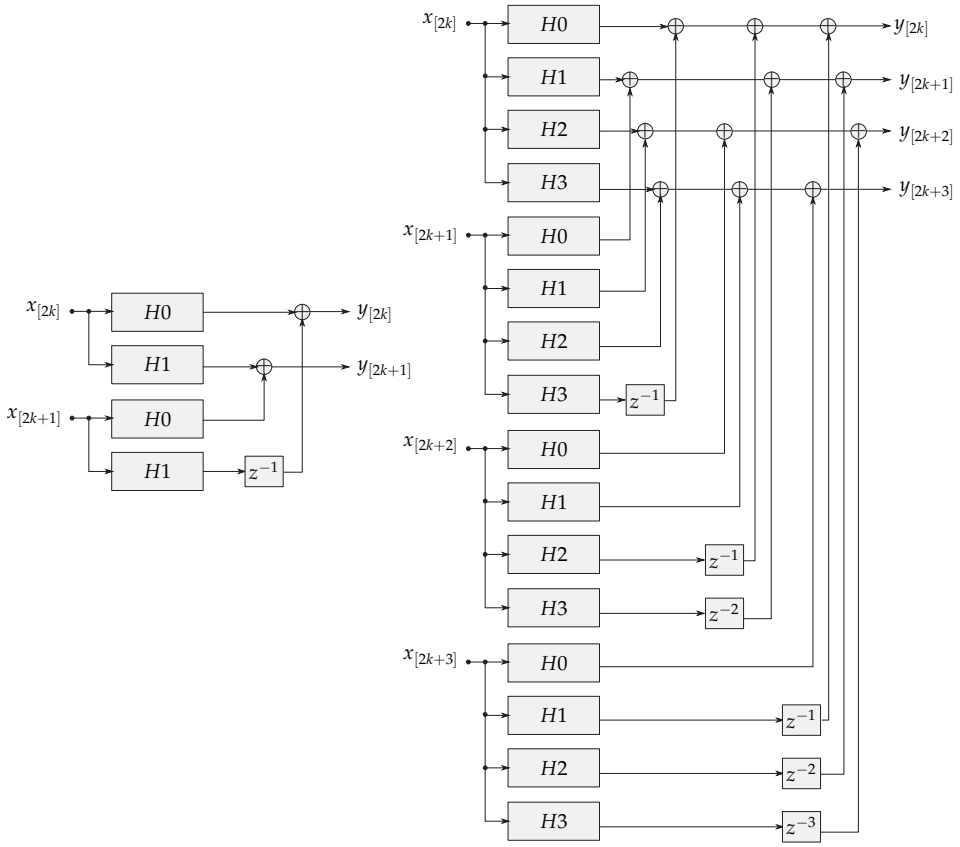

Figure 4.6: Scaling of polyphase structures from $\times 2 \rightarrow \times 4$ parallelism

between each of our constant multiplications. The last step in Figure 4.7 shows the inclusion of a Multiple Constant Multiplication (MCM) block to perform this optimisation.

Many existing MCM algorithms have been presented in the literature, with the most general approaches being graph-based algorithms. These aim to decompose an expensive set of multiplications into a graph of inexpensive additions and bit shifts (offered for free in the routing), precluding the need for any specialised DSP48E2s. The topology of these MCM circuits will change profoundly and quite unpredictably depending on the exact set of coefficient values. Most examples will try not only to minimise the graph for each multiplication in isolation, but also optimise for the MCM block as a whole.

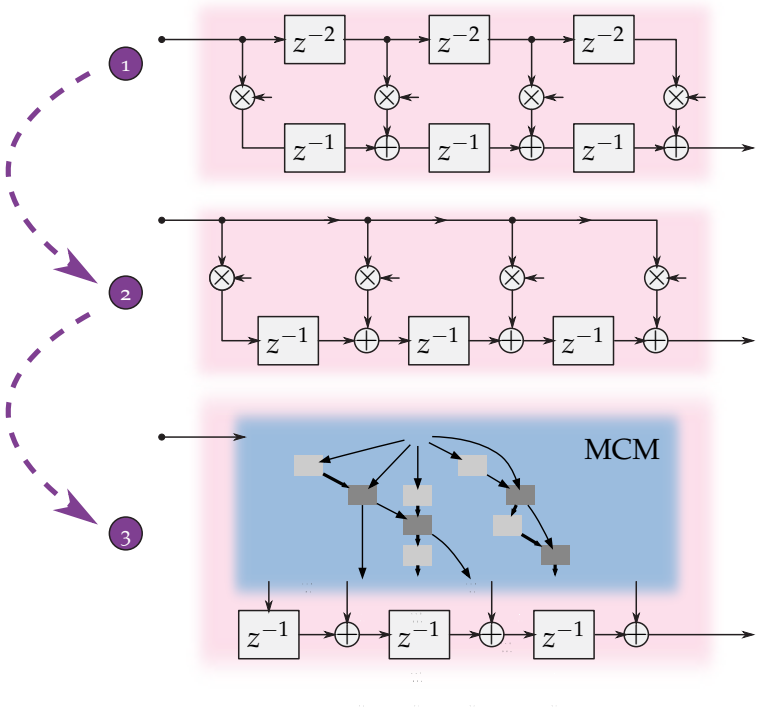

Figure 4.7: From systolic FIR form to MCM-based transpose form

We implement the H_{cub} [71], RSG [81], and RAG-n [80] algorithms in [26] using CλaSH, recommending the use of an H_{cub} variant which limits the graph depth at the expense of the number of adders. This will generally result in smaller FPGA areas for fully pipelined MCM blocks due to the predetermined ratio of look-up tables to registers (1:2 for the RFSoC's architecture); an effect explored further in [81].

Figure 4.8 shows an MCM graph generated using the H_{cub} variant. It realises an example coefficient set — the 15 tap half-band filter (`fir0`) present in the first generation RFSoC's DDC. Here we can implement all 15 multiplications with only 5 pipelined adders and 7 pipeline registers, rather than 15 DSP48E2s. This also helps to demonstrate that patterns in the coefficient sets can be readily exploited. We only need to implement multiplications for *unique, odd, positive coefficients*; even-valued coefficients can be recovered through bit shifts and negative coefficients can simply infer a subtractor in the filter's adder chain.

Due to these duplications, as well as more subtle commonality between coefficients identified by the MCM algorithm, implementing fewer but larger MCM blocks will always encourage more resource sharing, resulting in a more area-efficient circuit. Figure 4.9 shows how we can apply this principle to polyphase filters, combining the MCM blocks common to each input sample. Since each shared MCM will implement the

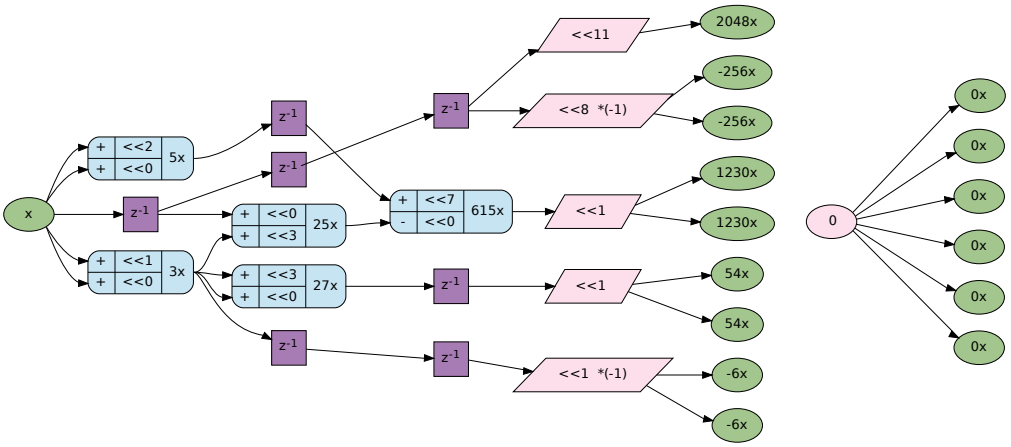

Figure 4.8: MCM Graph for `fir0` using an H_{cub} variant

full impulse response, H, we will directly exploit both symmetry and antisymmetry in the coefficients. Antisymmetry in this context implies that the upper half is a negated reflection of the lower half (i.e. $w_n = -w_{(N-n)}$ for $0 \leq n \leq \frac{N}{2}$).

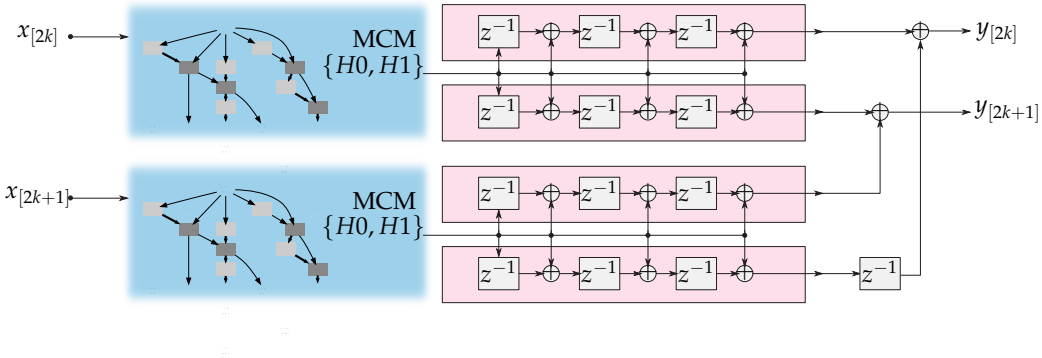

Figure 4.9: Sharing MCMs in polyphase filters

The polyphase structure with shared MCM-based filters is one structure we will present implementation results for, but our final optimisation step considers a more complex parallel structure in place of polyphase.

4.3.3 *Fast FIR Algorithm Filter with MCM subfilters*

We propose the use of FFA for the overall parallel structure of the filter, as opposed to the more common polyphase decomposition. FFA identifies extra resource sharing opportunities and generally requires fewer subfilters, at the expense of extra pre/-postadders and increased coefficient wordlengths in some subfilters. An example of a 2-parallel FFA structure is shown in Figure 4.10. Although further specialisation can

be made for higher parallelisms, we will nest successive 2-parallel FFA structures in order to implement any required power-of-two level of parallelism.

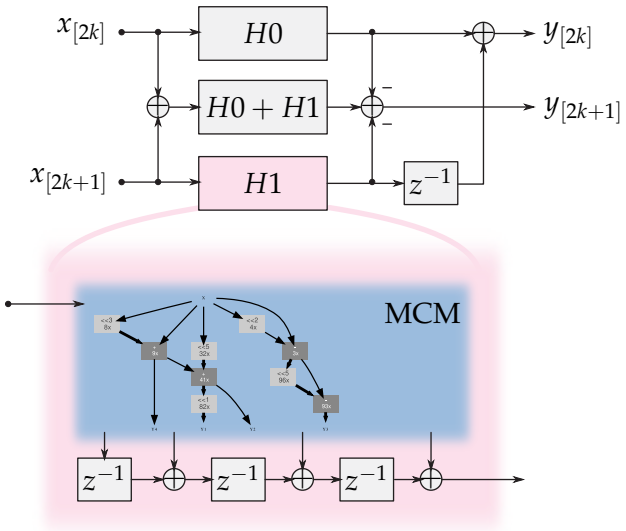

Figure 4.10: Proposed 2-parallel filter with 8 weights

This form of FFA is recursive in nature and can be difficult to represent and parameterise with VHDL. If we can overcome this, as with the difficulty in implementing MCM algorithms, there are clear resource savings to be discovered. We would like to take a moment to support our claim that the FFA and (at least the more complex of the) MCM algorithms still lurk largely in academic folklore. In a survey of four journal and transaction papers analysing the use of FFA structures under various coefficient symmetries [83–86] (i.e. under the practical application for most filtering) only one of three authors offer *any* experimental implementation results. For the one author who does [83, 84], there are (non-open access) Verilog implementations of only a few specialisations of these structures. This helps evidence our claim that many useful DSP structures prove too complex to describe in a fully generalised way in traditional HDLs.

Although even the 2-parallel structure shows a reduction in multiplier count of 25%, this reduction grows with the level of parallelism. Consider a filter architecture which processes 2^p samples in parallel. Its polyphase realisation will require a number of multipliers proportional to 4^p, while our nested FFA realisation is proportional to only 3^p. This scaling behaviour is shown in Figure 4.11.

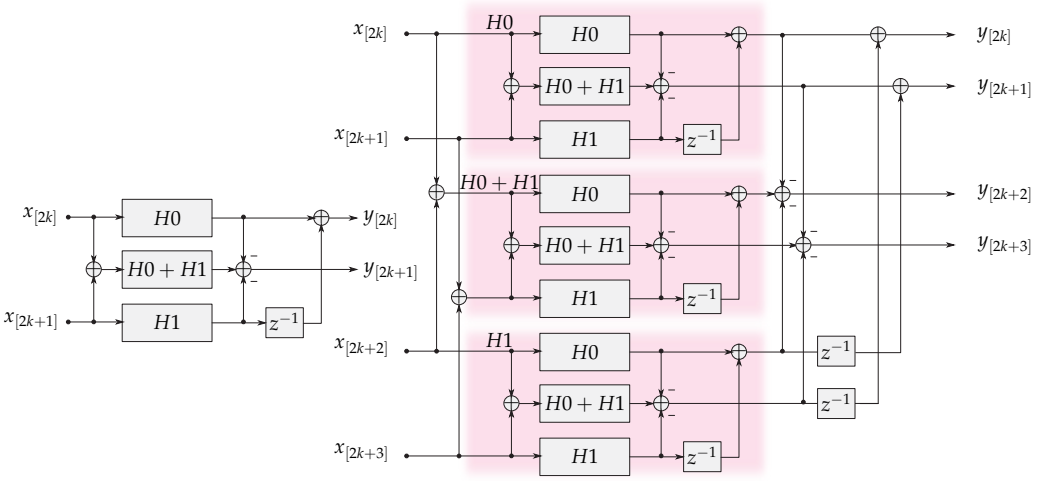

Figure 4.11: Scaling of nested 2-parallel FFA filters for $\times 2 \rightarrow \times 4$ parallelism

The main trade-off with this optimisation is FFA's ability to exploit coefficient symmetry. Each subfilter now has a unique input, precluding our shared MCM approach used with polyphase filters. So, although FFA appears extremely promising in the most general case, there is opportunity for a polyphase equivalent to perform favourably under real-world coefficient sets with symmetry or duplication. Section 4.4 offers an analysis of how each structure will perform under the six most common coefficient patterns.

4.4 MULTIPLIER COUNTS UNDER COEFFICIENT SYMMETRY

Although FFA reduces the number of multiplications in general, the preadders and $H_0 + H_1$ response can cause less favourable performance under coefficient symmetry. Since symmetry is prevalent in real-world impulse responses, we quantify the effect in this section.

From our MCM-based polyphase architecture in Figure 4.9, we can see that a 2^p-parallel structure can be realised with 2^p MCM blocks, each implementing the full impulse response (H). Because each MCM block contains every coefficient, we can always exploit any symmetry/antisymmetry present in the impulse response. This is not the case for FFA structures since each recursive step in the algorithm demands subfiltering for three new permutations of the full impulse response — the even-phased coefficients $(H0)$, the odd-phased coefficients $(H1)$, and the pairwise sum of even and odd-phased coefficients $(H0 + H1)$. These permutations are mixed again during every recursive step during its structure, resulting in some non-trivial patterns of sharing.

The general version of this effect has been studied in the literature: [83, 84] offers alternative FFA architectures for 2×2 and 3×3 filters for even and odd symmetries respectively, [85] proposes an extension to capture *multirate* FFA structures for symmetries, and [86] offers similar work for polyphase structures. Our contribution to this analysis is twofold:

1. An application-driven, quantitative analysis of the traditional FFA structure. This is guided by the particular parameters demanded by real-world RFSoC applications.
2. A qualitative analysis of how coefficient symmetries are further augmented by the *composition* of FFA *and* MCM approaches.

We select four classes of impulse response types, representative of real-world DSP designs. Each of these patterns is extended with zero padding to reach an integer multiple of 2^p, making it suitable for our recursive FFA structure. The common patterns we consider are:

Nonlinear phase: Each coefficient is unique:
$[w_0, w_1, \ldots, w_{n-1}]$

Type-II/IV: Even taps with symmetry/antisymmetry:
$[w_0, w_1, \ldots, w_{\frac{n}{2}-1}, w_{\frac{n}{2}-1}, \ldots, w_1, w_0]$ and
$[w_0, w_1, \ldots, w_{\frac{n}{2}-1}, -w_{\frac{n}{2}-1}, \ldots, -w_1, -w_0]$

Padded Type-I/III: Odd taps with symmetry/antisymmetry, then padded with one zero:
$[w_0, \ldots, w_{\frac{n}{2}-2}, w_{\frac{n}{2}-1}, w_{\frac{n}{2}-2}, \ldots, w_0, 0]$
$[w_0, \ldots, w_{\frac{n}{2}-2}, w_{\frac{n}{2}-1}, -w_{\frac{n}{2}-2}, \ldots, -w_0, 0]$

Half-band: Odd symmetric taps where every second tap is zero except the centre tap, padded with one zero:
$[w_0, 0, w_1, 0, \ldots, w_{\frac{n}{4}-1}, w_{\frac{n}{4}}, w_{\frac{n}{4}-1}, \ldots, 0, w_1, 0, w_0, 0]$

Since we target direct RF sampling applications with the RFSoC, we are only considering power-of-two parallelisms (1, 2, 4, 8, and 16 in particular) as these are directly supported by the RFSoC's front-end data converters. This also justifies our choice of cascading 2×2 FFA structures only, as we can easily synthesise any power-of-two parallelism. Table 4.1 shows non-recursive equations derived for the required multiplier

Table 4.1: Multiplier count under symmetries for 2^p-parallelism and 2^pN coefficients

Structure	Impulse Response	Multiplications
Polyphase	*Nonlinear Phase*	4^pN
	Padded Type-I & III	$2^p\left\lceil\frac{2^pN}{2}\right\rceil$
	Type-II & IV	$2^p\left\lfloor\frac{2^pN}{2}\right\rfloor$
	Half-band	$2^p\left\lceil\frac{2^pN}{4}\right\rceil$
FFA	*Nonlinear Phase*	3^pN
	Padded Type-I	$N(2+\sum_{k=1}^{p-1}3^k+2\sum_{i=0}^{p-2}\sum_{j=0}^{i}3^j)+(p-1)\left\lceil\frac{N}{2}\right\rceil$
	Padded Type-III	$N(2+\sum_{k=1}^{p-1}3^k+2\sum_{i=0}^{p-2}\sum_{j=0}^{i}3^j)+(p-1)\left\lfloor\frac{N}{2}\right\rfloor$
	Type-II	$\left\lceil\frac{N}{2}\right\rceil+2N\sum_{i=0}^{p-1}3^i$
	Type-IV	$\left\lfloor\frac{N}{2}\right\rfloor+2N\sum_{i=0}^{p-1}3^i$
	Half-band	$1+2^{p-1}+N+4N\sum_{i=0}^{p-2}3^i$

count for 2^p-parallel filters with 2^pN weights. The equations for FFA structures have been verified with the assistance of symbolic programming, identifying the number of symbolically unique, absolute coefficients per subfilter. The source for this verification is available at [26].

While we expect these forms to be difficult to interpret, there are two immediate takeaways. The FFA equations for Type-II and Type-IV are slightly different — for odd N, we expect Type-IV to exploit the cancellation of $w_{\frac{n}{2}-1}$ with $-w_{\frac{n}{2}-1}$. This effect is also visible through the padded Type-III's decomposition including one Type-IV subfilter. The second point is that, while the polyphase equations are simply described in this form, the structure of the FFA equations are noticeably more complex. This actually

serves as evidence for how difficult a generic FFA structure can be to describe without use of recursion — precluding descriptions in both VHDL and Verilog. To juxtapose these complex expressions, we demonstrate how structurally simple each of the FFA equations are in a recursive form.

$$H_{\text{NLP}}(p) = \begin{cases} n = N, & p = 0 \\ 3H_{\text{NLP}}(p-1), & \text{otherwise} \end{cases} \tag{4.4}$$

$$H_{\text{Type-II}}(p) = \begin{cases} \lceil \frac{N}{2} \rceil, & p = 0 \\ H_{\text{Type-II}}(p-1) + 2H_{\text{NLP}}(p-1), & \text{otherwise} \end{cases} \tag{4.5}$$

$$H_{\text{Type-IV}}(p) = \begin{cases} \lfloor \frac{N}{2} \rfloor, & p = 0 \\ H_{\text{Type-IV}}(p-1) + 2H_{\text{NLP}}(p-1), & \text{otherwise} \end{cases} \tag{4.6}$$

$$H_{\text{Type-I}}(p) = \begin{cases} \lfloor \frac{N}{2} \rfloor, & p = 0 \\ H_{\text{Type-I}}(p-1) + H_{\text{Type-II}}(p-1) + H_{\text{NLP}}(p-1), & \text{otherwise} \end{cases} \tag{4.7}$$

$$H_{\text{Type-III}}(p) = \begin{cases} \lceil \frac{N}{2} \rceil, & p = 0 \\ H_{\text{Type-III}}(p-1) + H_{\text{Type-IV}}(p-1) + H_{\text{NLP}}(p-1), & \text{otherwise} \end{cases} \tag{4.8}$$

$$H_{\text{HB}}(p) = \begin{cases} \lceil \frac{N}{4} \rceil, & p = 0 \\ H_{\text{single}}(p-1) + H_{\text{Type-II}}(p-1) + H_{\text{Type-II}'}(p-1), & \text{otherwise} \end{cases} \tag{4.9}$$

$$\text{where } H_{\text{single}}(p) = \begin{cases} 1, & p = 0 \\ 2H_{\text{single}}(p-1), & \text{otherwise} \end{cases}$$

$$H_{\text{Type-II}'}(p) = \begin{cases} 1 + \lfloor \frac{N}{4} \rfloor, & p = 0 \\ 2H_{\text{NLP}}(p-1) + H_{\text{Type-II}'}(p-1), & \text{otherwise} \end{cases}$$

Each of these expressions is either a base case or a simple combination of smaller FFA equations for common impulse responses. The only exception to this pattern is the decomposition of the halfband filter — we encounter one subfilter with a response of only one non-zero coefficient and another that is equivalent to a Type-II with one non-symmetric coefficient pair.

Let's take a moment to consider the properties of coefficient symmetry that are unique to our combination of FFA and MCM-based subfilters. While the $H0 + H1$ response can often create difficulty for implementations hoping to exploit symmetry (as explored in Table 4.1), there is an argument in favour of our combination of algorithms. Under coefficient symmetry, the $H0 + H1$ response will, no matter the length of the full impulse response, always introduce some common factors between coefficients —

even when the symmetry of the full response is not preserved. For a DSP48-based implementation this is not a useful property, nor is it particularly useful for implementations with simple MCM algorithms. The driving principle behind H_{cub}, however, is optimising for some *cumulative benefit* (hence "CUB"). The goal of this principle is to exploit the common factors shared between coefficients, and this application ought to have an unusually large number of common factors.

The analysis provided in Table 4.1 is simply a worst-case for our algorithm, with a strong indication of better performance due to the structure of our H_{cub} MCM blocks. While only a qualitative analysis, we will see this effect in action when discussing our experimental circuit usage results in Section 4.5.1.

Although our equations for quantitative analysis (ignoring the advantages of an H_{cub} MCM) are useful for numerical evaluation of the algorithms, we appreciate that the visualisation in Figure 4.12 provides a clearer insight into the behaviour. The half-band analysis is for singe-rate filters only; the down-sampling step as included in Section 4.5 would introduce its own effects in FFA, varying with the level of parallelism.

In particular, note that the required multiplication count for FFA is dramatically lower than polyphase for high levels of parallelism ($\times 8$ and $\times 16$). The extreme results for low levels of parallelism expose some subtleties in our consideration of real-world filter weights. For $\times 2$ parallel, type-II filters, FFA actually requires *more* multiplications than the simpler polyphase structure since the H_0 and H_1 responses break the symmetry in a worse-case manner. Here, the designer should opt to either adopt polyphase, or convert to a type-I impulse response.

These rules of thumb only regard the number of multiplications required in the filter structure and neglect any of the differences in additional adders, registers, and wordlengths which arise from the full filter implementation. The following section addresses these factors by presenting implementation results for each filter structure and impulse response type.

4.5 IMPLEMENTATION RESULTS

Figures 4.13 and 4.14 summarise resource utilisation and timing results for a set of filters with 16 bit inputs and coefficients, using a set of realistic impulse responses. These results are generated from a CλaSH implementation of the filter architecture (discussed more in Section 4.6) and Vivado 2020.1, targeting the ZCU111 development board. The raw dataset and (and an environment to reproduce it) is available at [26].

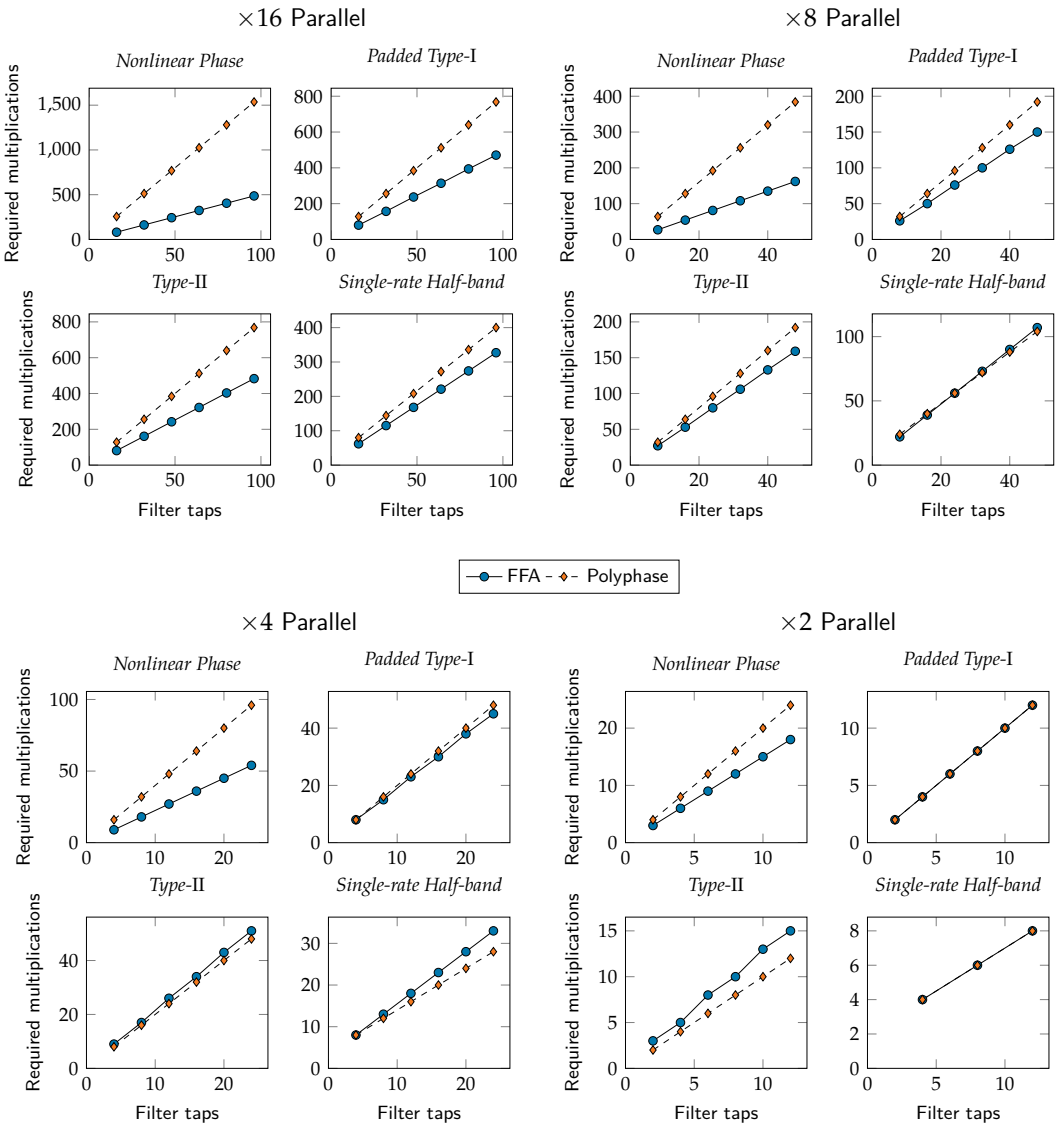

Figure 4.12: Number of multiplications synthesised under symmetries

4.5.1 Utilisation Results

Figure 4.13 shows the Configurable Logic Block (CLB) and DSP48E2 usage for the FFA, polyphase with shared MCM-block subfilters, and SSR structures. Results are generated using out-of-context implementation [87]. We sweep over degrees of parallelism, number of taps, filter structures, and impulse response types. The SSR results are split into two resource types — one line for DSP48E2 usage (the systolic subfilter logic) and another for CLBs (likely for overheads in phase recombination). Our two proposed filter structures only use CLB resources, so the DSP lines are omitted.

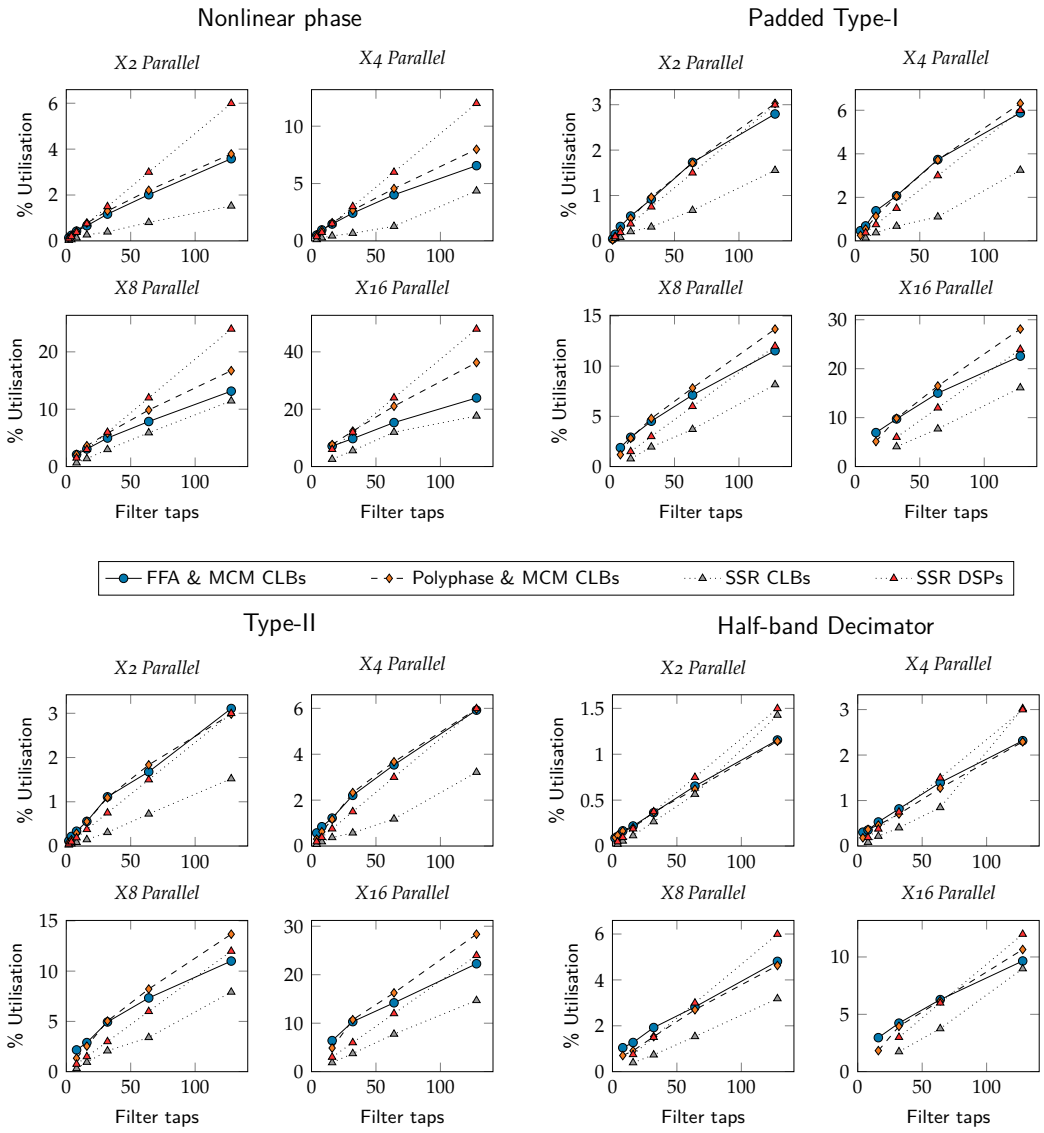

Figure 4.13: Implementation utilisation results

As a general rule, our polyphase and FFA implementations have a percentage CLB usage that is bounded by the percentage DSP usage of the traditional SSR implementation. From this, we can think of the proposed designs as a means of trading off a percentage DSP usage for a similar or smaller percentage of the more general CLB fabric. A stronger assertion is that our total FFA CLB percentage tends towards approximately 50% of the traditional DSP percentage for nonlinear phase responses, 90% for type-I/II responses, and 80% for half-bands.

So far, we have neglected the CLB overhead incurred with the SSR implementation. The overhead is often comparable to the *total* CLB area required by our FFA implementation — especially for high levels of parallelism. Indeed, some of the extreme results

for half-band filtering show that the FFA CLB area is actually smaller than just the CLB overhead incurred with SSR; not to mention the additional DSP usage!

The comparison between FFA and polyphase is more subtle. As predicted in Section 4.4, FFA consistently outperforms polyphase for any nonlinear phase impulse responses — trending towards 65% for $\times 16$ parallelism. For the remaining response types, the two architectures perform quite similarly for low levels of parallelism. For higher levels of parallelism, the FFA's advantages depend on the length of the subfilters. Small subfilters result in MCM blocks without much opportunity for resource sharing, limiting any optimisation. Since the polyphase structure shares larger MCM blocks between subfilters, the effect is less pronounced. In general, subfilters with only one or two weights are better suited to polyphase implementations, while FFA performs better for longer subfilters; tending towards 80% of the area for large $\times 16$ type-I/II filters.

4.5.2 *Timing Results*

This section aims to estimate the maximum achievable clock frequency, $f_{\max}$, for each of the filter structures by implementing half-band decimators with $\times 8$ parallelism and various filter lengths. Each architecture is implemented as a small loopback design (no longer using out-of-context implementation). The $f_{\max}$ metric is defined as the fastest clock rate achieved over 6 iterations of a binary search, directed by the previous run's achieved timing estimate. Source code for this process is available at [26].

Figure 4.14 shows the results for our MCM-based FFA and polyphase filters, as well as the SSR architecture. This work is conducted in the context of front-end digital filtering for the RFSoC and we should aim to support the ADC block's full data rate. Given the clocking resource's physical limit of 775 MHz [11] , the lowest possible level of parallelism we can use is $\times 8$ with a clock speed of $\geq$ 500 MHz. This target frequency is annotated as a red line in Figure 4.14 — any implementation above this line is sufficient for processing at the full ADC data rates.

All three implementations remain above the 500 MHz target for up to at least 128 taps. The SSR half-band decimators are the clear winner in terms of $f_{\max}$, maintaining the physical maximum — but anything above our red line target is acceptable for all front-end applications. The MCM-based polyphase structure behaves comparably but does start to dip below the chip's maximum frequency between 64 $\rightarrow$ 128 taps, possibly due to high fan-out from our shared MCM blocks. Finally, the FFA architecture displays a similar trend with filter length but with smaller absolute frequencies. This reduction is due to our pipelining strategy attempting to better balance utilisation

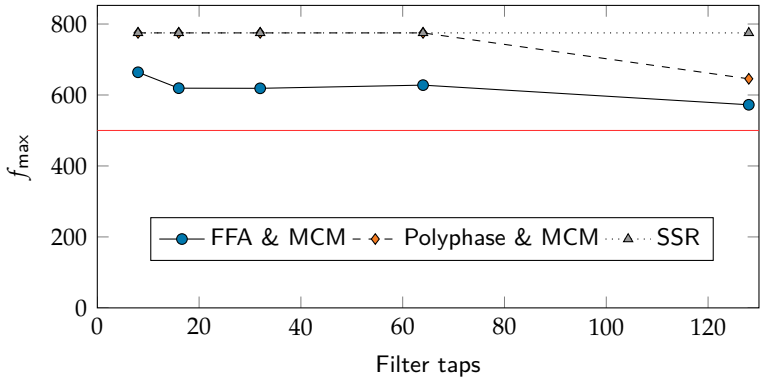

Figure 4.14: Maximum frequency results for $\times 8$ half-band decimators

and f_{max}. The phase recombination step introduces critical paths including two adders between registers, but this can be easily further pipelined when timing closure is an issue.

4.6 PRACTICAL HARDWARE DESCRIPTION

This chapter has, so far, only discussed the advantages offered by the FFA with MCM-based subfilters architecture. While we do offer some new analysis of how these two techniques interact in Section 4.4, and publish open source implementations of both, the fundamental idea is simple — compose two existing, complementary algorithms from the literature. An interesting question to reflect on is: "Why has this (to the best of our knowledge) not been implemented already?".

We propose that the main factor is due to traditional HDLs and practices. Both FFA structures and MCM blocks have proven awkward to describe with traditional HDLs for two different reasons:

↪ FFA lends itself to descriptions with structural recursion. This is not typically supported by traditional HDLs, perhaps because support for general recursion in a (structural) HDL releases the floodgates to many non-synthesisable descriptions.

↪ MCM blocks can demand a huge amount of computation *at compile-time* in order to infer the circuit topology. The exact set of coefficient values will have a profound effect on the structure of the MCM block's shifts and adds — and this *must* be evaluated during compile-time, rather than during circuit run-time. This level of meta-programming is rarely seen in the more traditional HDLs.

These two properties of our architecture result in a challenging implementation which we can use as a probing tool to test the limits of modern HDLs in anger. Furthermore, the *composition* of these two circuits proves especially troublesome since each is traditionally realised by ad hoc circuit generators, written in a separate software language.

Historically, a designer can choose to either handcraft a circuit for one set of parameters, or turn to software programming in order to implement an ad hoc circuit generator for their algorithm of choice (see the implementation of H_{cub} [71] as a well regarded example). Implementations of the latter are used as atomic black boxes, producing specialised HDL output given a set of input parameters. This approach has known challenges, including being resistant to circuit verification techniques. These difficulties arise, in part, from the wide range of technologies at play with no unifying type checker or other assistance, as well as an effect similar to the required "semantic domain crossings" described in [88]. Our addition to this discussion is that ad hoc circuit generators also come with fundamental challenges for the *composition* of complementary techniques, discouraging demonstrably useful classes of circuit such as our *FFA with MCM-based subfilters* architecture.

Figure 4.15 visualises the main additional steps needed to compose two (hypothetical) ad hoc circuit generators. In prose, these steps include splicing new sections into each generator, working with the non-standard intermediate representation of each circuit, passing these representations between generators in a language agnostic way, and merging the HDL generation sections of the two codebases.

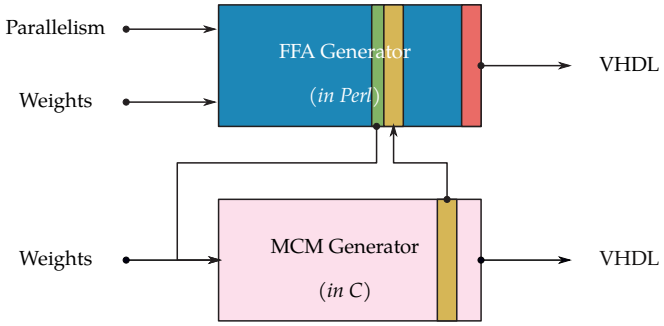

Figure 4.15: Composing ad hoc circuit generators

We instead advocate for modern functional HDLs which are expressive enough to directly encode our algorithms and their staging. Our implementation demonstrates that CλaSH can accommodate these features under a single source language, type checker, simulator, and compiler. This massively simplifies verification efforts and enables the designer to have some confidence in the final circuit output for any combination of parameters, not just a select few example test cases.

4.6.1 The successes of our CλaSH implementation

Functional approaches like CλaSH directly encourage composition in particular by allowing subcircuits to be passed to and from circuit generators. This allows exploration of the design space for many different algorithm combinations with little extra effort — as has been the focus of this chapter. As a concrete example, each of our parallel filter structures has been implemented as a generator parameterised by its subfilter architecture. There is then almost no hurdle to testing composite architectures with different combinations of parallelisms and MCM algorithms, which facilitated our exploration of implementation results in Section 4.5. We will see in Section 4.6.2 that this advantage can become slightly tangled with our reliance on Template Haskell for metaprogramming.

Another advantage demonstrated by this case study concerns the use of Algebraic Data Types (ADTs). Programming directly with well structured data, quite precisely capturing the the meaning of our data, is a boon both at the circuit generator and circuit run-time stages. Enjoying these data types fully for descriptions of circuit run-time behaviour is more natural for languages which encode circuits as plain functions, as both CλaSH and `toatie` do. Unfortunately, since this filtering example is all about compile-time complexity in order to realise a directed acyclic graph of arithmetic operations, all of our circuit's run-time data structures are just simple numeric types. Regardless, let's consider the structured data types used during our circuit elaboration. In particular, these quite directly capture the formal mathematical specifications given in MCM literature.

The work in [71] towards a shared framework for all graph-based MCM algorithms defines an MCM algorithm as simply a means of selecting successive fundamental operations (coined as $\mathcal{A}$-operations) to add to an accumulated graph. Each of these $\mathcal{A}$-operations are relative to a configuration, $c = (l_1, l_2, r, s)$, where:

$$\mathcal{A}_c(u, v) = |(u \ll l_1) + (-1)^s (v \ll l_2)| \gg r \tag{4.10}$$

for two previously synthesised positive coefficients (or "fundamentals"), u and v. This is one operation providing a configurable add/subtract with optional shifts on both operands and the output. The use of the absolute value can be avoided in all circuit realisations of an $\mathcal{A}$-operation — it just simplifies the equational definition by only allowing subtraction in one direction.

During our implementation of each MCM algorithm, our only goal is to iteratively construct a graph of these $\mathcal{A}$-configurations until we have nodes which synthesise to

every coefficient in our target set. Haskell, and therefore CλaSH, encourage the designer to take a structured approach to this. For example, Listing 4.1 shows the data types we define to direct this generalised, graph-based MCM framework.

Listing 4.1: CλaSH data types for (combinatorial) MCM algorithms

```
-- | Aop configuration, `c`, as in  Equation (4.10)
data AConf = AConf
  { l1 :: Shift
  , l2 :: Shift
  , r  :: Shift
  , s  :: Sign
  }

-- | A data type for a fully applied AOp
data AOp = AOp { c::AConf, u::Fundamental, v::Fundamental}

-- | A node in an MCM graph
data Node = Node {op :: Maybe AOp, depth :: Int}

-- | A combinatorial MCM graph: a collection of annotated AOps
newtype Graph = Graph {unGraph :: Map Fundamental Node}
```

This structured approach, along with helper functions for manipulating `Graphs`, allows the implementation of each algorithm to focus only on what is important — the choice of the next $\mathcal{A}$-operation(s) to insert. Each algorithm is notionally just a function from some configuration structure to a graph description of the MCM topology: e.g. `hcub :: McmConfig -> Graph`. In practice, this definition is slightly more involved in order to allow I/O access to precomputed lookup tables.

The MCM graph data type can then be refined into a second set of data types, facilitating the clean separation (and hand-off between) the pure mathematical operation of an MCM algorithm and the practical description of an equivalent *pipelined* circuit. When considering a hardware implementation we need to handle nodes that describe not only $\mathcal{A}$-operations, but also registers for pipeline synchronisation between stages, shifts to reconstruct even coefficients from our odd fundamentals, etc. The description of these nodes becomes more complex, but is captured quite concisely as the ADT "`PNode`" in Listing 4.2.

Since we can directly use the `PNode` data type in our synthesisable descriptions in CλaSH, the actual circuit description from this point is trivial — just $\approx$ 30 lines in [26].

4.6.2 *The limitations of our CλaSH implementation*

In reality, the implementation presented in this chapter does brush up against some limits of what is possible with many modern functional HDLs. One point to emphasise

Listing 4.2: CλaSH data types for (pipelined) MCM algorithms

```
-- A node for a fully explicit pipelined MCM graph
data PNode = PNodeIn
           | PNodeZero
           | PNodeAOp   AOp
           | PNodePipe  Fundamental
           | PNodeShift Fundamental Shift Negation

-- A pipelined MCM graph, with nodes grouped by pipeline stage
newtype PGraph = PG {unPG :: Map Stage (Map Fundamental PNode)}
```

is that we lean heavily on Template Haskell — a rather heady language extension, allowing the manipulation of a Haskell program's abstract syntax tree and the splicing of the results back into the program. We rely on this for two related meta-programming purposes: staging of our circuit generator, and flattening the structural recursion.

We have already seen that the structure of our nested FFA parallelism is inherently recursive, and it proves difficult to describe with iterative constructs alone. Listing 4.3 shows part of an FFA structure, captured in CλaSH with basic Template Haskell. The Template Haskell annotations should look similar to those introduced back in Section 3.5.

Listing 4.3: A partial example of an FFA structure in CλaSH with Template Haskell

```
-- An FFA structure where the whole filtering stage is left as a parameter
genFFA :: SNat n -> Q Exp
genFFA n =
  [| \f -> bundle . $(reorder n) . $(post n) . f . $(pre n) . unbundle |]
  where
    pre  1 = [| id                     |]
    pre  n = [| … $(pre  (n `div` 2)) |]
    swap 2 = [| id                     |]
    swap n = [| … $(swap (n `div` 2)) |]
    post 2 = [| …                      |]
    post n = [| … $(post (n `div` 2)) |]
    …
```

Note the type of `genFFA` suggests that, given a singleton natural number for the FFA parallelism, it will return (a computation providing) an untyped "expression". We define each of the `pre`, `swap`, and `post` stages recursively, also as untyped expressions. This is, in some ways, quite a dangerous use of Template Haskell. The loss of static typing information means:

1. We do not know if a circuit will type-check until we supply a value of n and splice the result into our circuit. We no longer have type safety for our entire

circuit family. Indeed, Listing 4.3 *will* error for some values of n, since the caller can supply a non-power-of-two, or 1 to break the `swap` and `post` functions.

2. We lose the circuit family's best form of documentation. It becomes difficult to statically reason about how the FFA inputs and outputs change shape as n increases.

More recently, there is also support for a typed use of Template Haskell. This does overcome most of the type safety issues but still comes with a set of constraints. As a formal example, [89] discusses the interaction of constraints, polymorphism, and staging with typed Template Haskell after using it in anger. Template Haskell's use of the `Q` monad can very easily start to pollute a codebase (such as our MCM algorithms and their verification, in Section 4.7). Some careful navigation of the Glasgow Haskell Compiler's stage restrictions is also required. This impacts the way we can supply our coefficients and perform our testing. Work on supporting existing language features, such as type classes, has continued to as recently as [90] from 2022.

We propose that an implementation of the staging required for circuit description could be better facilitated in a language with first-class staging constructs (rather than extensions) and dependent types (one feature which subsumes many others, such as type polymorphism and some constraints).

There is one other area that relies on Template Haskell where it may be undesirable to do so. Our MCM algorithms (before the `PGraph` representation in Listing 4.2) use dynamically sized collections, since encoding the computations for their sizes at the type level is challenging without dependent types. In order to synthesise a circuit from these dynamically sized collections, at some point after the `PGraph` representation, we need to convert these to bit representable collections (such as CλaSH's `Vect` type). We use Template Haskell to do so — perhaps innocently enough, as long as there are no infinite structures. This particular Template Haskell splice can also be thought of in a more positive light as an important staging annotation. It marks the MCM decompositions as being evaluated at compile time, and only the resulting `PNode` operations must be performed at circuit run time.

4.7 PRACTICAL VERIFICATION

The most ubiquitous approach to circuit verification is example-driven testbenches. These are easy to write in both VHDL and Verilog, presenting a (usually small) handcrafted set of inputs to one circuit, already specialised for a particular set of `generics`. We will use an adder as a demonstrator for our techniques here, and while it is easy

to loop exhaustively over *all* possible operands in VHDL for this case, a more complex design (such as our full filter architecture) will tempt designers into using only a small set of hand-crafted inputs.

An adder circuit under example-based testing might be presented with the following four hand-crafted scenarios, with `assert` statements checking the output against the expected value:

$$0000_2 + 0000_2 = 0000_2 \tag{4.11}$$
$$0111_2 + 1000_2 = 1111_2 \tag{4.12}$$
$$0001_2 + 1001_2 = 1010_2 \tag{4.13}$$
$$1001_2 + 0001_2 = 1010_2 \tag{4.14}$$

Although there are clear omissions, this set of examples has some reasonable justifications. We test the lower extreme (the addition of zeros), one interpretation of the upper extreme (the largest output we can encode), and two complementary examples (suggesting that the addition might be commutative). Clearly, there are many incorrect implementations that would still satisfy these test cases. For example, there is no example describing the desired overflow behaviour, or there might be a hard-coded rule to return zero if *either* operand is zero. Perhaps a more serious omission is that we test our generic adder structure only for one size of input (two four-bit operands). We have not tested *any* other adder structures from this family, let alone the common edge cases. These concerns can scale with the complexity of the design under test.

For our CλaSH implementation of MCM-based parallel filters, we use a different approach to verification: property-based testing. Using a well known Haskell library for this, Quick Check [91], gives a high-reward with very low developer effort. As an introduction, let's revisit our adder example through the lens of property-based testing. We express our test cases as *properties* which should be true for our adder's behaviour. In this case, our properties should always hold true, but Quick Check makes it possible to guard these by preconditions also. One obvious property we may want to test for our adder is:

$$\forall n \in \mathbb{Z}, \forall x, y \in \{0, 1, \ldots, 2^n - 1\}.\ x + y = \text{int}(\text{adder}_n(\text{bin}_n(x), \text{bin}_n(y))) \tag{4.15}$$

This first property appears equivalent to a VHDL testbench which iterates through all operands (between 0 and $2^n - 1$) with an assert statement ensuring that the cast

output matches the in-built implementation of the + operator for integers. However, we are now sampling from the entire circuit family — it applies to adders of any length, not one concrete value of n. This also ensures that there is no overflow in the output. While defining our property in terms of a built-in + operator is valid in this scenario, we will not always have a such an obvious reference. In these cases, we look to construct a set of different properties which, when combined, fully describe the intended behaviour. There are, perhaps surprisingly, only a few properties that we need to test for addition before we can be certain that the design under test definitely implements addition. These are commutativity, associativity, and the neutrality of addition with zero:

$$\forall n \in \mathbb{Z}, \forall x. \quad \text{adder}_n(x, 0) = x \tag{4.16}$$

$$\forall n \in \mathbb{Z}, \forall x, y. \quad \text{adder}_n(x, y) = \text{adder}_n(y, x) \tag{4.17}$$

$$\forall n \in \mathbb{Z}, \forall x, y, z. \quad \text{adder}_{n+1}(x, \text{adder}_n(y, z)) = \text{adder}_{n+1}(\text{adder}_n(x, y), z) \tag{4.18}$$

These properties can also be extended to account for latency and ramp-up times in real synchronous circuits. For our FIR circuits, we can aim to judge their behaviour via two properties:

↪ The impulse response: how does the circuit react to a unit impulse? We expect a time-delayed copy of the test's coefficients.

↪ The frequency response: how does the circuit transform inputs of different frequencies? The expected result can be statically derived from the test's coefficients in terms of both phase and magnitude.

Once we have used these properties to establish one good reference implementation, we can continue by comparing our more complex implementations directly with this reference. This approach to testing is facilitated by Quick Check which gives us three important features. Firstly, it allows us to define random generators for terms of a given type. Secondly, for all of the variables bound by '∀' in our properties, Quick Check will generate many random values via their generators and check that the property holds true in each case. Finally, if a set of arguments does cause a property to fail, it will attempt to reduce the failing case to the simplest possible form — exposing a useful, simple failing example to the developer rather than the arbitrary original values.

Unfortunately for this chapter's filtering implementation, our reliance on Template Haskell is somewhat at odds with the structure of standard Quick Check testing. We

use Template Haskell to flatten all FFA structures at compile time. However, we would ideally like to write properties of FFA structures for *any* parallelism. Within the Quick Check framework, the specialisation of the FFA structure would need to happen at run time but Template Haskell does this at compile time. This is not a restriction true of multistage programming in general, but is a consequence of the strict constraints in Template Haskell. Although far from ideal, this is forgivable in the context of RFSoC applications since we will only encounter the fixed parallelisms of 1,2,4,8, and 16 — we can hard-code separate sets of properties for each.

Without the restrictions imposed by the interaction of Template Haskell and Quick Check, this appears to be a reasonably effective and low-effort means of verification. So, what is missing in this approach? Although we do gain reasonable evidence of correctness, we do not have a *proof* that the output is always functionally correct. For our MCM-based filtering, [71] provides formal proofs of H_{cub}'s (without heuristic stage) optimality, its complexity, and its termination. With the appropriate tooling, we could encode these proofs and use them to verify such properties for our particular implementation; not just the abstract algorithm. This theorem proving approach *is* facilitated by the `toatie` language, discussed in Chapters 5 and 6.

As a final comparison for circuit verification in particular, [91] uses circuit descriptions in Lava as a Quick Check case study. They identify Quick Check as a means of catching two classes of errors:

↪ Logical errors that *would* be otherwise caught by Lava's external theorem prover. Since a call to the external theorem prover is a heavyweight task, they propose Quick Check could be used to catch these errors more readily.

↪ Errors in only *some* members of the circuit family. These might not be caught with Lava's external tools since they require that the circuit has already been specialised to a first order description. These errors usually stem from unhandled clauses in accidentally non-covering descriptions (such as addition of two *differently* sized operands).

A comforting thought is that `toatie` can also catch both of these classes of error, with more formal guarantees than property-based testing. We preclude non-covering descriptions with our coverage checker and offer the means for theorem proving over entire circuit families.

4.8 SUMMARY

We have demonstrated that, in the context of RFSoC applications, a combination of FFA parallelism and MCM-based subfilters can generate area-efficient and high-speed parallel filters. These filters quite consistently exchange the traditional architecture's DSP usage for a similar percentage of the generic fabric resources (CLBs) — or for nonlinear phase filters, often under half of the equivalent DSP usage. This is ignoring the CLB "overhead" incurred by the traditional architecture as well — there are (somewhat extreme) scenarios where our *full* implementation has a smaller CLB area than the traditional implementation's *overhead* alone.

There are some interesting edge-cases for small filters with low parallelism where our polyphase structure with shared MCM blocks will often outperform an FFA equivalent, due to better exploitation of the coefficient symmetry explored in Section 4.4. Both implementations were presented in this chapter, and are available under open source licences.

We also reported on our practical experiences with hardware description, identifying some limitations of traditional methods and how these discourage exploration of circuits with similarly complex structures. After identifying how modern functional HDLs, such as CλaSH, help alleviate some of these difficulties, we also proposes language extensions which could improve the experience for future designers.

CλaSH does facilitate our implementation with clear advantages over ad hoc circuit generators, but this work has also encouraged us to entertain new language features as will be explored in Chapters 5 and 6. Our additions include support for dependent types. This enables a host of quality-of-life improvements for the designer, including:

- ↪ Concise, type-safe minimum wordlength tracking, especially for DSP with constant coefficients. For readability, our CλaSH implementation manually resizes inputs to the worst-case wordlength and relies on vendor Electronic Design Automation (EDA) tools to prune uninhabited bits. This is demonstrated later in Section 5.3.
- ↪ Encoding proofs of circuit behaviour in the source language, wherever formal verification is demanded. This is explored further in Section 5.4.

Our reliance on meta-programming techniques also highlights the need for multi-stage programming, which `toatie` includes as a first class element of its source language. This enforces a clear split between compile-time computation and circuit runtime computation in a type safe way. Not only can this address some common criticism

of Template Haskell, it even alleviates the pressure on the compiler for unrolling primitive recursion and partial evaluation; we can write unrolling functions in the source language explicitly and evaluate them at compile-time. This is in opposition to CλaSH's set of templated VHDL code for recursively defined library functions. These ideas have already been explored in [22, 23, 64] but we go on to showcase them in a practical light, giving examples in our open source language and compiler, `toatie`. We hope that this will further encourage modern, verifiable DSP solutions for high-speed RFSoC applications and beyond.

5 ON APPLICATIONS OF DEPENDENT TYPES TO DSP CIRCUIT FAMILIES

This chapter is an extension of the ideas presented in the paper "On applications of dependent types to parameterised DSP circuits" [23]. It is adapted to our `toatie` *language and now includes exploration of circuit families with full functional verification in a correct-by-construction fashion.*

5.1 INTRODUCTION

This chapter presents combinatorial circuit examples in `toatie`, a new HDL whose implementation will be detailed in Chapter 6. This environment is dependently typed and enables faithful descriptions of DSP circuit families, where various circuit properties can be statically verified by the type checker. There are a few different degrees to which we might want to lean on our type system in a practical context, balancing productivity and the completeness of verification. To demonstrate this circuit description methodology, this chapter threads a combinatorial dot product example across three different levels of type-level verification:

↪ A introductory implementation, concerning standard, worst-case wordlengths only.

↪ Improved wordlengths with bounded integer ranges, where the types ensure that each word is exactly as long as it needs to be — not any longer or shorter than strictly necessary.

↪ Types which encode a circuit family's precise arithmetic meaning. This explores the full functional verification of circuit families in a correct-by-construction fashion.

Beyond the interests of the original developer, using the type checker to verify these properties provides a contract to the user of a 3rd party's circuit family; the developer must generate well-formed circuits for all possible parameter sets (i.e. every circuit within the family), and the developer is given a clear, computer-checked guarantee of its behaviour. Although not dwelled on in this chapter, as well as tracking functional

aspects of our circuits in their type, we can also track non-functional properties such as abstract resource usage. This could offer a direct way of reasoning about circuit area, etc., in the source language, rather than forcing the designer to interpret results after a lengthy external elaboration with industry EDA tools.

We will pause along the way, sightseeing the likes of speculation on synchronous circuits in Section 5.5.1, and proving the equivalence between a direct DFT implementation and a substantially optimised form in Section 5.4.4. We will clarify where our circuit families otherwise prove difficult to describe in traditional HDLs, and how our static verification compares to that of similar example-driven testbenches or model checking.

5.2 MINIMAL TYPE-LEVEL GUARANTEES: TOWARDS A COMBINATORIAL DOT PRODUCT

This section approaches circuit design with `toatie` in its most simple form. We will not lean heavily into the verification aspects of the language, but will instead see how naturally we can describe circuits with recursive structures and how we can compose subcircuits together, quickly realising a non-trivial dot product circuit. Note that we have adopted unsigned arithmetic for the purposes of illustrating `toatie` concepts, acknowledging that signed arithmetic would be preferred for the likes of FIR filter implementation. The interested reader can access our full source, including a signed variant, at [24] and [27].

To begin thinking about our simple arithmetic building blocks, consider the direct form FIR filter shown in Figure 5.1. All wordlengths have been annotated with the worst-case for each unsigned arithmetic operation in isolation.

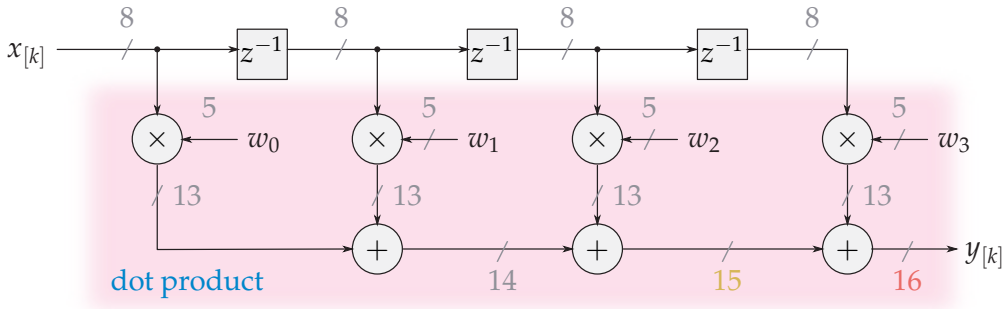

Figure 5.1: A direct form FIR filter with worst-case growth along the adder chain

For this introductory example, the input is 8_b (an 8-bit word), the coefficients are all 5_b, and the adder chain ends as 16_b. In this section we consider each arithmetic op-

eration in isolation and only account for the immediate input wordlengths, regardless of any other information about the signal's range that could be inferred. For example, given n_b and m_b unsigned words, multiplication with worst-case bit growth is represented by $(n + m)_b$, and addition gives $(\max(n, m) + 1)_b$.

This sort of growth can be captured by VHDL designs using generics, functions, and `for generate` statements, but encoding heterogeneous collections of these signals for an adder chain is challenging. As an introduction to using `toatie` for circuit description, let's spend some time implementing our arithmetic functions and introducing a data type for our binary numbers.

5.2.1 *An unsigned adder circuit*

Listing 5.1 defines two data types, `Bit` and `Bin`, where a value of type `Bit` represents a single bit and `Bin n` represents a collection of n bits. Notice that we introduce both types with the `simple` keyword, rather than the `data` keyword seen in Chapter 3. This marks the data type as being *synthesisable*. We will see that all top-level synthesisable circuits must be a quoted function of `simple` arguments. Informally, a `simple` type must have a statically known bit representation. To satisfy this requirement, `Bin` is essentially defined as a fixed-length vector of bits, noting that a dynamically sized *list* of bits would not have a single, known bit representation based on the type alone. The length index in the `BinCons` data constructor is marked as irrelevant and will be erased — and important annotation since this unbounded natural number is not synthesisable.

Listing 5.1: A binary word indexed by its wordlength

```
1 simple Bit : Type where
2   O : Bit
3   I : Bit
4
5 simple Bin : Nat → Type where
6   BinNil : Bin 0
7   BinCons : {n : Nat} → Bin n → Bit → Bin (S n)
```

Note that we opt to construct `Bin`s by appending bits to the least significant side. This very slightly simplifies our implementation of addition and multiplication, but appending to the most significant side also results in reasonable circuit descriptions.

As a first step, let's consider a single-bit full adder as presented in Listing 5.2. This implementation shows how simple pattern matching clauses can quite naturally describe a truth table or lookup-table. Note that we introduce a pair, or "tuple", type to encode the two output bits: `cout` and `sum`. We could have also chosen to implement

the full adder as a series of `xor`/`or`/`and` gates, while being explicit about any resource sharing with `let` bindings.

Listing 5.2: A single-bit full adder

```
-- A pair of values, a tuple.
simple Pair : Type → Type → Type where
  MkP : {a,b : Type} → a → b → Pair a b

-- Single bit full adder
addBit : Bit → Bit → Bit → Pair Bit Bit
addBit O O O = MkP {_} {_} O O
addBit O O I = MkP {_} {_} O I
addBit O I O = MkP {_} {_} O I
addBit O I I = MkP {_} {_} I O
addBit I O O = MkP {_} {_} O I
addBit I O I = MkP {_} {_} I O
addBit I I O = MkP {_} {_} I O
addBit I I I = MkP {_} {_} I I
```

It should now be straightforward to compose multiple instances of our full adder circuit into an n-bit unsigned adder. Figure 5.2 shows how we can cascade full adders to achieve an n-bit adder. Since we define our binary words inductively, we should also take a moment to identify the structure of our base case (for empty inputs) and our inductive case (defining an $(n+1)$-bit adder in terms of an n-bit adder). The base case is trivial: we just return the c_{in} input. The inductive case requires us to pass the least significant bit of our two operands to a full adder, along with c_{in}. We can then recurse on our adder structure for the remaining most significant bits, threading along the c_{out} generated by the full adder.

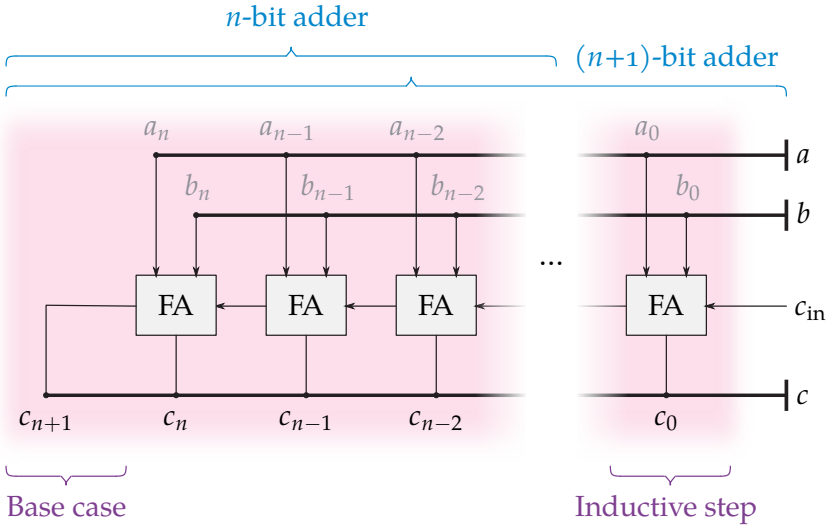

Figure 5.2: Structure of an unsigned adder circuit

The translation to `toatie` in Listing 5.3 matches our prose quite closely. We will now make use of the staging constructs (introduced in Section 3.5) to ensure that our circuit can always be fully elaborated once the explicit non-synthesisable arguments are supplied. This ensures causality between elaboration time and circuit run-time stages at the circuit's top level.

Listing 5.3: An unsigned adder for `Bin n`

```
1  -- An `n` bit adder
2  addBin' : (n : Nat) → ⟨Bit⟩ → ⟨Bin n⟩ → ⟨Bin n⟩ → ⟨Bin (S n)⟩
3  -- Base case for empty inputs
4  pat cin ⇒
5    addBin' Z cin ⟦ BinNil ⟧ ⟦ BinNil ⟧
6      = ⟦ BinCons {_} BinNil ~cin ⟧
7  -- Inductive case for non-empty inputs
8  pat n, cin, x, xs, y, ys ⇒
9    addBin' (S n) cin ⟦ BinCons {_} xs x ⟧ ⟦ BinCons {_} ys y ⟧
10     = ⟦ case addBit ~cin x y of
11           pat cin', lsb ⇒
12             MkP {_} {_} cin' lsb ⟹
13               BinCons {_} ~(addBin' _ ⟦ cin' ⟧ ⟦ xs ⟧ ⟦ ys ⟧) lsb
14       ⟧
```

There are a few features in this example that will become very familiar by the end of the chapter. We see three of our staging annotations being sprinkled throughout the definition. Since these are largely to assist the elaboration process, the reader may choose to ignore them during their first pass of each example. For an intuition about their meaning, each annotation can be read as:

Circuit type : `⟨ty⟩` ⇒ an argument of type '`ty`' that will only be available during the elaborated circuit's run-time (not during elaboration). For example, our two binary words and the carry-in input are marked as only being available during the circuit's run-time.

Quote : `⟦ tm ⟧` ⇒ defers the evaluation of a term '`tm`' until circuit run-time. For example, the RHS of both clauses are wrapped with quotes, deferring the addition until circuit run-time by default. Quotes can also appear on the LHS of a clause as part of an inaccessible pattern — our `⟦ BinNil ⟧` and `⟦ BinCons {_} xs x ⟧` matches, for example.

Escape : `~tm` ⇒ force the evaluation of a term immediately. For example, we unroll the recursive call to `addBin'` by explicitly evaluating it with an escape. This is akin to flattening a nested circuit hierarchy into a flat, wide structure.

Another important aspect is the relevance/irrelevance of the length argument, `n`. One may be tempted to mark `n` as irrelevant since it is only used in irrelevant positions in the definition of `Bin` (see Listing 5.1). For `Bin`'s data constructors, we needed this irrelevance since `Nat`s are not synthesisable. However, we *must* keep the adder's length argument relevant here since it is required during the elaboration of an adder circuit — i.e. we need access to the length when deciding how many times to unroll our generic adder structure. More formally, an explicit length argument is needed to permit our inaccessible patterns which statically identify when our input words are empty (⟦ `BinNil` ⟧) or non-empty (⟦ `BinCons {_} xs x` ⟧).

The final feature of interest is our use of an in-line case statement on line 10 of Listing 5.3. This lets us perform simply typed pattern matching on an intermediate value. In this case, we use it to help us decompose the `Pair` of `Bits` returned by the full adder and give each of these `Bits` names we can reference later. A more verbose option for our decomposition would have been to introduce an entire helper function and use the standard pattern matching techniques. As an aside, the latter approach is necessary when we want a matched clause to refine the types of existing names in our scope, since we do not implement the `with` rule present in Idris 2.

Let us now generalise our adder circuit family to accept two words of different lengths. Back in Listing 5.3, the type of `addBin'` accepts two `Bin`ary words of length n and returns a word of length $1 + n$. For convenience we would like a wrapper that accepts an n-bit word and an m-bit word, returning a word with length $1 + \texttt{max}(n, m)$. Listing 5.4 demonstrates such a wrapper, `addBin`. We resize each input to $\texttt{max}(n, m)$ by extending their Most Significant Bits (MSBs) with `0`s and then call `addBin'` as normal.

Listing 5.4: An unsigned adder for heterogeneous wordlengths

```
1 addBin : (n, m : Nat) → ⟨Bit⟩ → ⟨Bin n⟩ → ⟨Bin m⟩ → ⟨Bin (S (max n m))⟩
2 pat n, m, cin, xs, ys ⇒
3   addBin n m cin xs ys = addBin' _ cin (resizeBin _ (max n m) xs)
4                                        (resizeBin _ (max n m) ys)
```

We have omitted the implementation of the helper function `resizeBin` for brevity. Suffice to say, '`resize j k bits`' takes a quoted `Bin`ary number of length j and resizes it to length k by either truncating or padding zeros to the MSB. Using this version of the adder circuit family should incur no overhead in the elaborated circuit when both operands happen to be of equal length. Note that the return type, ⟨`Bin (S (max n m))`⟩, contains a call to a user-defined function, `max`. This is a consequence of dependent types — we can put *any* term in the types, including calls to arbitrarily complex functions.

To conclude this section on our first adder circuit, let's ground the discussion by looking at the netlist that our description actually produces. Figure 5.3 shows an annotated schematic generated by passing `toatie`'s netlist output through the Yosys Open Synthesis Suite [92].

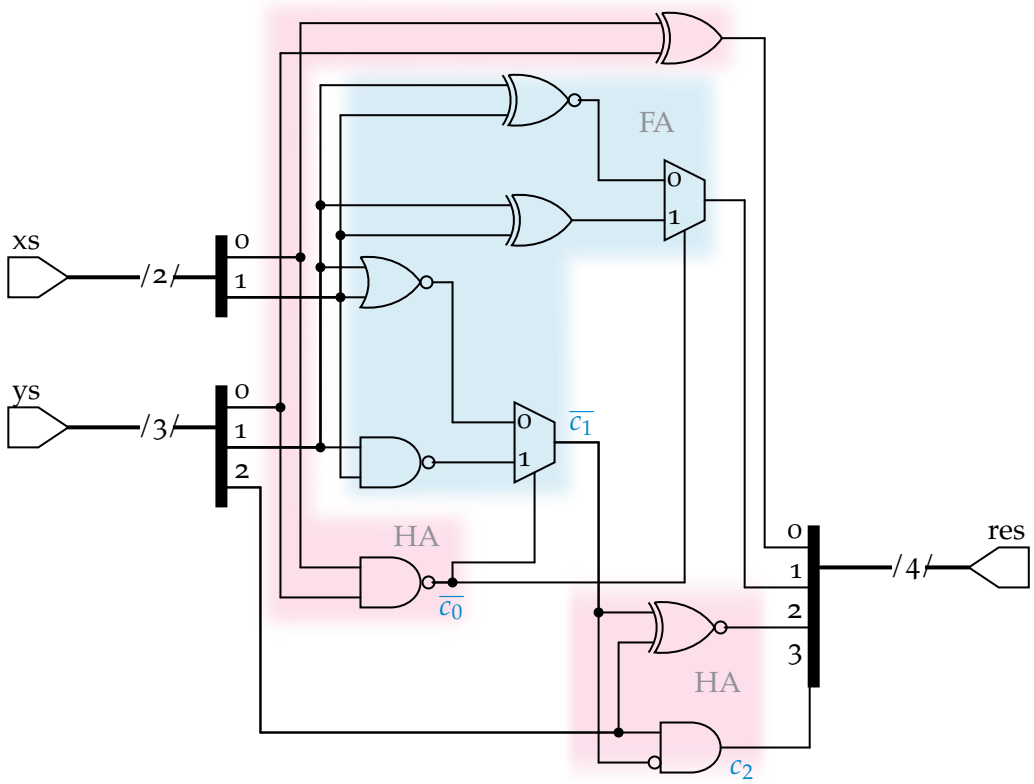

Figure 5.3: Schematic for `addBin` with 2-bit and 3-bit inputs, and a constant '0' carry input

We can identify the adder's composition in terms of half-adders and full-adders (annotated as HA and FA respectively). This example performs addition on a 2-bit input and a 3-bit input, with the carry input tied to a constant zero. As we may expect, the first output bit comes from a half-adder (since there is no carry input to handle). The second stage appears as a full-adder, while the final stage is another half-adder since the corresponding bit derived from the *xs* input is a constant zero. The only oddity after synthesis is that the intermediate carry signals are negated, although this is compensated for later in the schematic at no intrinsic cost.

Although our steps into hardware description thus far are quite reserved, we have fully demonstrated its overall methodology in microcosm: describing suitable data types, defining structures over those types, and generating a netlist.

5.2.2 *An unsigned multiplier*

Now that we have introduced the basic patterns for describing circuits over the `Bin` type, we can quite quickly start to realise more complex designs. The implementation of a hardware multiplier for `Bin`s should require a modest number of lines once we have a feel for its inductive structure.

Figure 5.4 shows one architecture for an array multiplier. It is clear that the approximate structure is similar to a cascade of n, m-bit adders (like the one introduced in Figure 5.2), but the exact inductive pattern might merit some extra inspection. We can imagine such an array multiplier being defined by recursion only on its first operand (labelled as 'a' in Figure 5.4) while passing through 'b' untouched. Each inductive step generates a single new output bit and routes it to the current Least Significant Bit (LSB) position — bits $c_0 \rightarrow c_3$ are examples of this. All other intermediate accumulated bits (shown as blue lines in Figure 5.4) are routed through to a recursive call. Our base case, when there are no remaining bits in the first operand, is to simply return any accumulated bits (see bits $c_4 \rightarrow c_7$, for example).

This description can be quite directly translated to `toatie`, noting that we expect an $(n + m)$-bit output for n-bit and m-bit inputs. We will also include an m-bit accumulator argument, although this will usually be set to zeros by the caller. An implementation of this in Listing 5.5 reinforces many of the characteristics we have been growing accustomed to. These include the use of staging annotations to safely elaborate our recursive structure, the use of inaccessible patterns to safely refine the shape of circuit run-time variables (e.g. our `⟦ BinNil ⟧` and `⟦ BinCons _ xs x ⟧` patterns), and the use of an inline case statement to conveniently decompose an intermediate result. Perhaps the *only* new construct we introduce for the multiplier is the difference between our two uses of case statements. The case on line 13 of Listing 5.5 incurs no extra logic in our final circuit. The type checker is satisfied that the scrutinee will *always* match the `BinCons` constructor, so this check will not have to be performed again at circuit run-time. For the case on line 10 of the same listing, there are multiple valid choices and we will not be able to tell which branch to take until the circuit's run-time. This implies that both branches should appear in the netlist, and a multiplexer will select the correct branch at circuit run-time. Indeed, omitting either of these branches (lines 11 and 12) from our definition of `mulBin` will result in a compiler error — all choice constructs should be covering, and since we encode circuits as plain functions, the type checker can easily help enforce this.

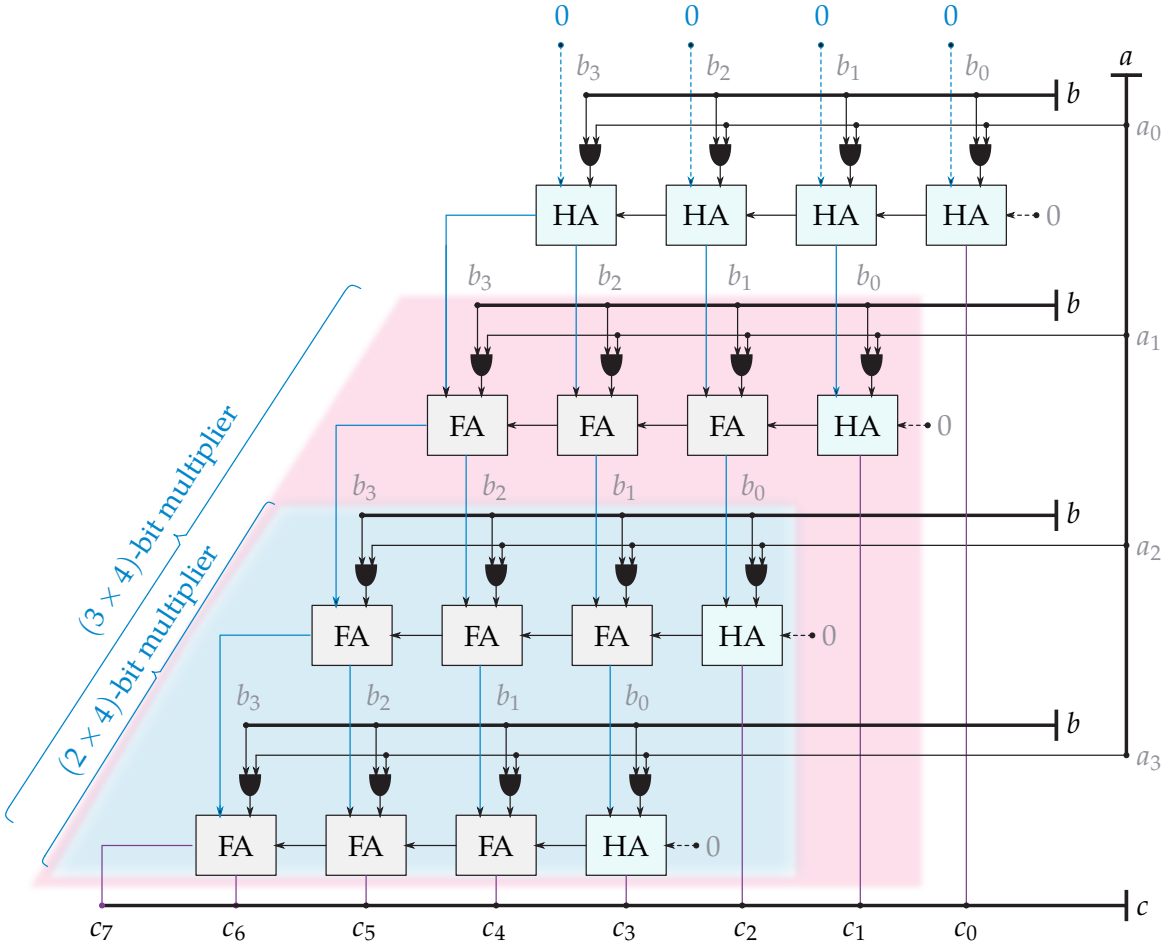

Figure 5.4: Structure of a 4×4 array multiplier

5.2.3 *A dot product and structure with higher-order functions*

To conclude our circuit family implementations for this section, we want to compose our multipliers and adders into a dot product. This is in part to realise a common, non-trivial building block of many DSP applications, but also to introduce the use of higher-order functions to abstract the structure of a circuit. Since functions are a first class element of `toatie`, we are free to pass function-valued arguments to and from other functions. This allows us to separate the *structure* of a circuit with a regular pattern from the *behaviour* of the repeated subcircuit. This separation allows us to try radically different structures very easily, encouraging a thorough exploration of the design-space.

Figure 5.5 introduces 3 common examples of higher-order functions. In each case, the behaviour of the subcircuit, `f`, is left as a parameter and only the structure of the full circuit is described by the higher-order function.

Listing 5.5: An unsigned multiplier

```
mulBin : (n, m : Nat) → (acc : ⟨Bin n⟩) → ⟨Bin n⟩ → ⟨Bin m⟩ →
         ⟨Bin (plus n m)⟩
-- Base case: return accumulator
pat m, acc, ys ⇒
  mulBin Z m acc ⟦ BinNil ⟧ ys = acc
-- Inductive case: add gated row to accumulator,
--                 return LSB, and then recurse
pat n, m, acc, xs, x, ys ⇒
  mulBin (S n) m acc ⟦ BinCons {_} xs x ⟧ ys
    = ⟦ let row : Bin m = case x of
              I ⟹ ~ys
              O ⟹ ~(zeroBin _)
        in case ~(addBin' _ ⟦ O ⟧ ⟦ row ⟧ acc) of
             pat zs, z ⇒
               (BinCons {_} zs z) ⟹
                  BinCons {_} ~(mulBin _ _ ⟦ zs ⟧ ⟦ xs ⟧ ys) z
      ⟧
```

We consider versions of Figure 5.5's `map`, `zipWith`, and `foldr` functions defined over the `Vect` type (first introduced in Listing 3.6):

`map` applies a function to every element of a `Vect`.

`zipWith` combines two `Vects`, element-wise, with a given function.

`foldr` accumulates an output by repeated application of a function with every element of a `Vect`.

A dot product operation ($\sum_{i=0}^{n-1} a_i b_i$, also visualised in Figure 5.1) can be considered as a sequence of a `zipWith` to multiply element pairs, followed by an adder chain realised with `foldr`. There is, however, one inconspicuous challenge that arises with the adder chain. Indeed, this is likely the first actually challenging typing scenario that a programmer familiar with Haskell-inspired HDLs (such as CλaSH) will encounter. When we try to construct a `foldr` to replicate our adder, there is a type error: the standard definition of `foldr` expects the accumulated value between each addition to have the same type. Our adder explicitly grows the wordlength, with the inconvenient consequence of ensuring that the output type will *always* be different from its operand types. We must fall back on dependent types here, allowing us to parameterise the accumulator type by the fold's current stage (a function called the "motive").

This challenge, although subtle, feels inevitable when working with data types indexed by their length. Especially so when our basic arithmetic building blocks impact these lengths. While this is addressed in CλaSH with a special purpose function using singleton natural numbers (see discussion of `dfold` in [51]), our dependently typed

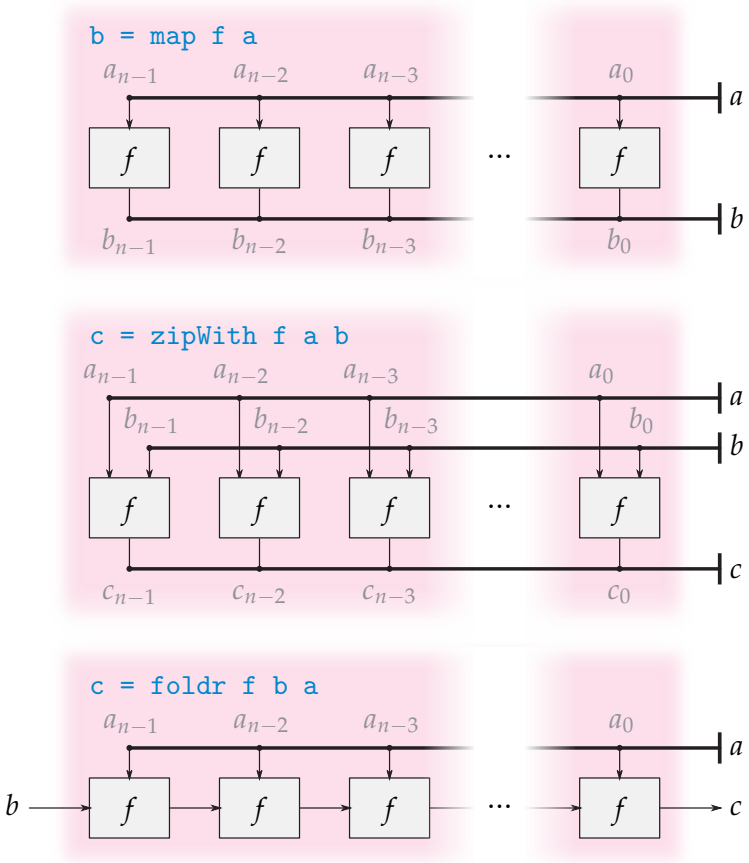

Figure 5.5: Structure of three common higher-order functions: `map`, `zipWith`, and `foldr`

Listing 5.6: Type definitions of our simply typed (left) and dependently typed (right) `foldr`

```
vfoldr :
  (n : Nat) → {a,b : Type} →

  (f : ⟨a⟩ → ⟨b⟩ → ⟨b⟩)    →

  --^ Step function
  (init : ⟨b⟩)              →
  ⟨Vect n a⟩ → ⟨b⟩
```

```
vdfoldr :
  (n : Nat) → {a : Type}   →
  (p : Nat → Type)         →
  --^ Motive
  (f : (i : Nat) → ⟨a⟩     →
       ⟨p i⟩ → ⟨p (S i)⟩) →
  --^ Step function
  (init : ⟨p 0⟩)           →
  ⟨Vect n a⟩ → ⟨p n⟩
```

language permits such a function to be user-defined without special compiler support or restriction to singleton natural numbers.

Listing 5.7 shows an implementation of a dot product circuit, whose structure is defined by element-wise multiplication via `zipWith` and an adder chain via a dependently typed `vdfoldr`.

Listing 5.7: A dot product implementation for `Bin`

```
-- Dot product using higher order functions on vect
dotProdBin : (i,n,m : Nat) →
             ⟨Vect i (Bin n)⟩ →
             ⟨Vect i (Bin m)⟩ →
             ⟨Bin (plus i (plus n m))⟩
pat i, n, m, xs, ys ⇒
  dotProdBin i n m xs ys
    = ⟦ let muls = ~(vzipWith _ {_} {_} {_} (mulBin _ _ (zeroBin m)) xs ys)
        in ~(vdfoldr _ {_}
                (λi ⇒ Bin (plus i (plus n m))) -- The `motive`
                (λi ⇒ λx ⇒ λy ⇒ addBin' (plus i (plus n m)) ⟦ 0 ⟧
                                        (resizeBin (plus n m) _ x) y)
                (zeroBin _) ⟦ muls ⟧
             )
      ⟧
```

The start of this section claimed that the use of higher-order functions can aid design-space exploration. As evidence, it is trivial for us to replace the linear adder chain in the previous example with an adder tree structure. This should reduce the number of adders required and better reflect the worst-case output wordlength for any `Vect` of words with a power-of-two length. For completeness, Listing 5.8 shows such an implementation, along with the type definition for the binary tree version of `foldr`. All of these higher-order functions for `Vect` would be supplied as part of a standard library.

Listing 5.8: An alternative dot product with an adder tree

```
-- Type for a dependently typed fold for a binary tree
vdtfoldr : (n : Nat) → {a : Type} →
           (p : Nat → Type) →
           (a → p 0) →
           (f : (i : Nat) → ⟨p i⟩ → ⟨p i⟩ → ⟨p (S i)⟩) →
           ⟨Vect (pow 2 n) a⟩ → ⟨p n⟩

-- An alternative dot product with an adder tree structure
dotProdTreeBin : (i,n,m : Nat) →
                 ⟨Vect (pow 2 i) (Bin n)⟩ →
                 ⟨Vect (pow 2 i) (Bin m)⟩ →
                 ⟨Bin (plus i (plus n m))⟩
pat i, n, m, xs, ys ⇒
  dotProdTreeBin i n m xs ys
    = ⟦ let muls = ~(vzipWith _ {_} {_} {_} (mulBin _ _ (zeroBin m)) xs ys)
        in ~(vdtfoldr _ {_}
                (λi ⇒ Bin (plus i (plus n m)))
                (λx ⇒ x)
                (λi ⇒ addBin' _ ⟦ 0 ⟧ )
                ⟦ muls ⟧ )
      ⟧
```

5.2.4 *Summary for examples with minimal type-level guarantees*

To summarise Section 5.2, we have introduced the fundamentals of describing combinatorial circuits in a multi-stage language with dependent types, such as `toatie`. This was presented in the practical context of constructing a dot product circuit from first principles — starting without a definition of bits, nor unsigned addition.

Ignoring the specific *features* in the surface language for a moment, we have met some, perhaps unusual, general *approaches* to hardware description. For example, we introduce many circuit families via recursion. Although this may feel alien to many digital designers, many real-world structures are elegantly described this way (such as our FFT example introduced in Section 5.4.4) but are quite cumbersome to capture with imperative '`for generate`' statements. Such descriptions can otherwise be impossible for an algorithm's most general case. Alongside this, we have briefly seen use of the completely user-defined elaboration of circuits. Not only can we elaborate recursive calls to a circuit function, we can write higher order functions to elaborate an arbitrary pattern of subcircuits. Capturing our descriptions in this way offers the possibility of radically changing a circuit's structure with relative ease. This lowers the barrier for structural changes to a circuit and encourages a more thorough exploration of a circuit's design-space.

At this point, we do not make a detailed comparison to traditional and existing HDLs since most of our circuits thus far are possible to describe in such tools. Our limited application of dependent types has just support circuit synthesis, rather than functional verification. We have tracked the width of a bit vector at the type-level, ensuring that `toatie` can synthesise a finite circuit representation for our data types; we cannot synthesise a collection of unknown/dynamic length to a fixed bit representation.

However, we do still uncover some early advantages in `toatie`'s approach. We can define computations over these type-level widths using standard, plain functions. Programmatic control of even just wordlength growth in these early examples enables us to avoid:

1. Awkward circuit family patterns in VHDL or Verilog, which instead encourage the designer to write an ad hoc description of *one* specialised circuit. For example, difficulty in encoding the heterogeneous collection required to properly model the accumulator widths along the dot product adder chain.

2. Dynamically sized circuit families, without the confidence provided by a static type checker, as in Lava descriptions.

3. The complex ecosystem of language extensions and special syntaxes required to equip Haskell with a subset of dependent types. CλaSH does, however, provide an excellent templated standard library to facilitate many common circuit patterns (including all of our examples so far).

Our encounter with `vdfoldr` when implementing the dot product's adder chain was one particularly clean-cut demonstration of the need for dependent types when working with type-safe circuit description. Again, this is something provided in a standard library for projects such as CλaSH, but dependent types allow the developer to describe such structures directly in the source language.

Moving forward, we investigate how a designer can choose to lean more heavily on the type system. This is very much a *choice*: the methodology that was presented in this section is often a sensible approach for simple designs with no high-assurance requirements. Beyond this, when appropriate we can choose to exploit our type system for either verification purposes (Section 5.4) or to improve a circuit family's non-functional properties, such as its area, by more precisely encoding their data's *meaning* in its type (Section 5.3).

5.3 GUARANTEEING MINIMUM WORDLENGTHS: EXPLORING A CIRCUIT FAMILY'S NON-FUNCTIONAL PROPERTIES

One important benefit of dependent types is the ability to encode more information about some data's *meaning* in its type. This section demonstrates such a use of dependent types by refining our unsigned binary data type, indexing it by its length *and the known upper bound* of its `Natural` number encoding. We will see how we can better model wordlength growth in our circuit families before external elaboration, synthesis, or placement and routing. This either lets us improve on a routed circuit's area, or at least gives us a better environment to reason about non-functional properties.

Using the FIR filter example presented back in Figure 5.1, an new circuit with improved wordlengths could be described for two reasons:

1. Each arithmetic operation was considered in isolation. Repeated additions will accumulate quantisation effects when the range of a number does not align with powers of 2. For example, y in Figure 5.1 will only inhabit values within the range $[\![0, 2^{15} - 1]\!]$, despite its 16_b annotation.

2. The coefficients will often be constants. In this case, the bit growth due to multiplication will vary with the numerical value of each constant coefficient.

The latter is particularly relevant, as it clearly demands a language with dependent types — a *term*-level value (a coefficient) must be used to compute a *type* (the output wordlength).

Better bit growth is facilitated by types that also track the numerical range each signal can inhabit, rather than immediately rounding to the required number of bits (i.e. tracking a range, $0 \to r$, rather than $0 \to \lceil log_2(1+r) \rceil$). Our `Bounded` type in Listing 5.9 implements this, where a number of type '`Bounded n w`' is a w-bit unsigned binary number, encoding a value in the closed interval $[\![0, n]\!]$ (i.e. any value between 0 and n, inclusive). Notice that the only data constructor for `Bounded` requires an argument which is a proof that $w = \lceil log_2(1+n) \rceil$. This argument is marked as irrelevant and will be erased from the circuit after type checking. It does, however, provide evidence that every single `Bounded` value we construct will have *exactly* the minimum number of bits that its range requires. Trying to declare a `Bounded` value with extra leading bits will result in a type error (since we cannot generate a valid proof), and so will constructing a value with fewer bits than needed to encode the full range. We will most often encounter words whose width is definitionally $\lceil log_2(1+n) \rceil$, so the function `Bounded'` provides this as a type alias.

Listing 5.9: A unsigned binary data type indexed by its upper bound and wordlength

```
1 simple Bounded : Nat → Nat → Type where
2 MkB : {n, w : Nat} → {prf : Equal Nat w (clog2 (S n))} →
3       Bin w → Bounded n w
4
5 Bounded' : (n : Nat) → Type
6 pat n ⇒
7   Bounded' n = Bounded n (clog2 (S n))
```

Now we can define some arithmetic building blocks quite simply, noting that our `Bounded` type is essentially a wrapper around `Bin` with an extra proof relating its upper bound to its wordlength. Listing 5.10 defines functions for addition of two `Boundeds` and multiplication of a constant and a `Bounded`. Each of these is a wrapper around the respective function on the more general `Bin` type, resizing the outputs to the required length. Note that the use of the resizing operation is not guaranteed to maintain the arithmetic encoding of its argument; the developer is choosing to rely on their own reasoning outside of `toatie` in this case. We address a scenario where the full arithmetic meaning is modelled in the language throughout Section 5.4. The reader can find a discussion of a similar constant multiplication circuit there too — we omit the definition of `mulConstBin` here for brevity.

The interesting part of the above functions is their type definitions. For our adder, `addB`, the output can inhabit the range $[\![0, n+m]\!]$ for any inputs with ranges $[\![0, n]\!]$ and

Listing 5.10: Minimum bit growth for `Bounded` arithmetic functions

```
addB : (n, m : Nat) → ⟨Bounded' n⟩ → ⟨Bounded' m⟩ → ⟨Bounded' (plus n m)⟩
pat n, m, xs, ys ⇒
  addB n m ⟦ MkB {n} {_} {_} xs ⟧ ⟦ MkB {m} {_} {_} ys ⟧
    = ⟦ let ans = ~(addBin _ _ ⟦ 0 ⟧ ⟦ xs ⟧ ⟦ ys ⟧)
        in MkB {_} {_} {Refl {_} {_}} ~(resizeBin _ _ ⟦ ans ⟧)
      ⟧

mulConstB : (n, c : Nat) → ⟨Bounded' n⟩ → ⟨Bounded' (mul c n)⟩
pat n, c, xs ⇒
  mulConstB n c ⟦ MkB {n} {_} {_} xs ⟧
    = ⟦ let ans = ~(mulConstBin _ c ⟦ xs ⟧)
        in MkB {_} {_} {Refl {_} {_}} ~(resizeBin _ _ ⟦ ans ⟧)
      ⟧
```

$[\![0, m]\!]$. This often infers fewer bits than our wordlength-directed implementation from Section 5.2. An important aspect is that the type-level range will propagate to successive function applications, for example, further reducing wordlengths along an adder chain. As a more extreme example, `multconstB` will multiply a `Bounded` by a natural number coefficient. It is the *value* of this coefficient that directs our output range and wordlength, dramatically improving our analysis of wordlengths in FIR filters with constant coefficients.

Our next step is to compose these arithmetic building blocks into a dot product circuit with minimal wordlengths. Let's consider the dot product equation and attempt to construct a precise output type, capturing our expected range. Equation (5.1) shows a dot product of j coefficients (w) and j elements of a vector (x).

$$y_{[k]} = \sum_{i=0}^{j-1} w_i \cdot x_{[k-i]} \tag{5.1}$$

For our output type, let's consider the worst-case magnitude of each term in Equation (5.1) — restricting all x inputs to the same worst-case range. This is not too restrictive in our context, since this will always be the case for dot product based FIR examples (we have no a priori knowledge of the input signal). Besides, we do encounter an FFT example in Section 5.4.4 which does discuss descriptions with heterogeneous collections of ranges/wordlengths. In our case, the range of $x_{[k]}$ is constant for all k, so the output type can become:

$$|y_{[k]}| = |x_{[k]}| \sum_{i=0}^{j-1} |w_i| \tag{5.2}$$

From this, we can deduce that a valid type for the dot product function is `Bounded' (mul n (sum j ws))`, given a collection of j coefficients, `Vect' j Nat` called `ws`, and a collection of j samples of x, `Vect j (Bounded' n)`. Note that `sum` is an ordinary function and, once more, it is a consequence of dependent types that we can use it to construct a type for the dot product output.

Listing 5.11 gives one interpretation of this dot product. We could have generalised this with a new dependently typed fold (with intermediate types indexed by the preceding portion of the vector) but instead opt for basic recursion for simplicity.

Listing 5.11: A dot product with minimum wordlengths

```
dotProd : (j,n : Nat) → (ws : Vect' j Nat) →
          ⟨Vect j (Bounded' n)⟩ →
          ⟨Bounded' (mul n (sum j ws))⟩
pat n, xs ⇒
  dotProd Z n (VNil' {_}) xs = ⟦ MkB {_} {_} {Refl {_} {_}} ~(zeroBin _) ⟧
pat j, n, w, ws, x, xs ⇒
  dotProd (S j) n (VCons' {_} {_} w ws) ⟦ VCons {_} {_} x xs ⟧
    = ⟦ let y = ~(addB _ _ (mulConstB _ w ⟦ x ⟧ )
                           (dotProd _ _ ws ⟦ xs ⟧ ))
        in eqInd2 {_} {_} {_} {lemmaDotProd n w (sum j ws)}
                              {λh ⇒ Bounded' h} y
      ⟧
```

We address the distinction between `Vect` and `Vect'` first of all. We have already met the synthesisable, "`simple`" type, `Vect`. We use a variant of this, `Vect'`, for our coefficients which is not synthesisable. This is, unfortunately, a requirement since *parameter* types and *simple* types share one namespace, and `simple` types cannot appear in a dependently typed setting: i.e. a circuit's shape cannot depend on something synthesisable. Other than the type and data constructors having an apostrophe appended to their names, they are equivalent in practice.

The second point of note is that we construct our output, given the name '`y`', before rewriting its type with some proof, `lemmaDotProd`. To understand why this proof is necessary, Listing 5.12 demonstrates the pertinent part of the type error thrown in the absence of this coercion.

Listing 5.12: An excerpt of the type error given by `dotProd` without our proof

```
Constraints:
  (plus (mul w[4] n[5]) (mul n[5] (sum j[6] ws[3])))
  ~~~
  (mul n[5] (plus w[4] (sum j[6] ws[3])))
```

This constraint tells the developer that the types *might* align, but we need to demonstrate that $w \times n + n \times \sum ws$ is equivalent to $n \times (w + \sum ws)$. Essentially we need to

provide evidence to the type checker that the multiplication of `Nats` is distributive and commutative. This arises because, although *we* can intuit that the two equations are equivalent, the structure of our definition is slightly different from the expectation set by our types. The type of the dot product is defined as in Equation (5.2), and while our recursive implementation is mathematically equivalent, it does have a subtly different structure, as shown in Eq. 5.3 for the s^{th} recursive step.

$$|y_{[k]}| = |w_s \cdot x_{[k]}| + |x_{[k]}| \sum_{i=s+1}^{j-1} |w_i| \tag{5.3}$$

The goal of our rewrite rule, `lemmaDotProd`, is to demonstrate this equivalence: Equation (5.2) ≡ Equation (5.3). We use two helper functions on proofs (`eqInd2` and `eqSym`) introduced back in Section 3.6 to demonstrate this equality. We start from the reflexivity of $w \times n + n \times \sum ws$. Then we apply two other rules, defined in our standard library without any special treatment, to demonstrate the commutativity property of $w \times n$, and then the distributive property of $n \times (w + \sum ws)$. It is important to note that both of these sub-proofs, called `mulCommutative` and `mulDistributesOverPlusRight`, can be defined by the developer in our source language — the full source for each can be found at [27].

Listing 5.13: The rewrite rule for our dot product

```
lemmaDotProd : (n, c, cs : Nat) →
               Equal Nat (plus (mul c n) (mul n cs))
                         (mul n (plus c cs))
pat n, c, cs ⇒
  lemmaDotProd n c cs
    = let -- Start from (c*n)+(n*cs) = (c*n)+(n*cs)
          h0 = Refl {_} {plus (mul c n) (mul n cs)}
          -- Rewrite c*n to n*c in RHS
          h1 = eqInd2 {_} {_} {_} {mulCommutative c n}
                 {λh ⇒ Equal Nat (plus (mul c n) (mul n cs))
                                 (plus h         (mul n cs))}
                 h0
          -- Rewrite (n*c)+(n*cs) to n*(c+cs) in LHS
          h2 = eqInd2 {_} {_} {_}
                 {eqSym {_} {_} {_} (mulDistributesOverPlusRight n c cs)}
                 {λh ⇒ Equal Nat (plus (mul c n) (mul n cs)) h}
                 h1
      in h2
```

This step in our implementation is worth lingering on for a little while. It is the first instance of us using a dependently typed language to reason about the equivalence of two expressions. Here, it is only used to demonstrate that our desired description for wordlengths is met by our circuit's definition (which has subtle structural differences). At this point it may feel like an extra chore, satisfying the tooling but not benefiting

ourselves, but later we can use the same techniques to reason about more interesting properties for verification. For example:

1. Demonstrating that the arithmetic meaning of a circuit's output *always* meets a certain specification, e.g. an adder circuit always produces a binary word which encodes the sum of its two operands and the carry input bit (Section 5.4.1).
2. Demonstrating equivalence between two arbitrary functions. For example, Section 5.4.4 implements an optimised (radix-2 Cooley–Tukey) FFT circuit and then proves that it matches the equation for the optimised FFT structure using bullet point 1). We then continue by proving that this optimised equation is totally equivalent to the more straightforward, but computationally intense, direct FFT implementation. In essence, reasoning about equivalence between two algorithms, not just between our circuit family and its expected behaviour.

In summary, we have significantly tightened the ideas in Section 5.2 by leaning more heavily on the type system without much extra effort — ensuring that we always implement the absolute minimum wordlengths throughout our dot product. If an implementation failed to do so for *any* possible set of parameters, it will raise a type error, seen both by the IP's designer and the IP's user. This gives a strong and clear contract between the IP's designer and the IP's user. Errors for edge-case circuits within the family are a common bug in traditional circuit generators, but we can identify such errors statically. For example, we ensure that we handle zero coefficients or elimination of complementary pairs as per the contract in Equation (5.2). We know exactly what has been statically verified (the wordlength of the output given by certain inputs) and, by inference, what has not been statically verified (the full arithmetic meaning of the output in this case). Perhaps there will be more evidence of testing provided out-of-band, but these should be taken with some trust placed in the original developer.

It is important to appreciate that tracking the wordlengths within a circuit is just one of many non-functional properties of a circuit that can be tracked in a dependently typed environment. Even just within the context of circuit area, we may choose to model the resource usage more directly for a certain FPGA family. A dependently typed language gives us the tools to encode a simple model for how addition can be packed into 6-bit LUTs, as one example. We are also free to encode abstract cost models while evaluating different algorithms, without going through the lengthy *place and route* process for each example circuit.

5.3.1 *Brief comparison to VHDL and Lava alternatives*

Compare this dependently typed, minimum bit growth FIR filter to the implementations possible in other HDLs. A typical VHDL FIR filter can be parameterised in terms of its coefficient values, the wordlength of the coefficients, and the input wordlength. However, the bit growth is likely to be worse than even the scenario presented in Figure 5.1. Because of the lack of type inference or type-level generate statements in VHDL, a common approach is to simply resize all arithmetic stages to match the worst-case output width — heavily relying on synthesis tools to remove unused nets. We offer an environment to reason about such non-functional properties in the source language itself, rather than after the elaboration process in vendor EDA tools.

Although leaving this bit pruning to downstream tools is a valid design choice when considering the filter in isolation, it presents practical difficulties for real designs. The filter will usually be just one part of a larger chain of DSP circuits. At several points along the data path, the full precision signals will be shortened to constrain resource usage.

In this case, the designer may employ two strategies:

↪ Truncation/rounding of the LSBs.

↪ Removing uninhabited MSBs identified by Eq. 5.2.

The second option should be appealing as it can reduce wordlengths without loss in precision, but it requires extra manual effort (for each coefficient set!) just to emulate a static property of our `toatie` implementation.

There are also clear benefits above other modern functional HDLs, such as Lava [93]. In Lava, a similar circuit can be described using dynamically sized lists of bits to represent each word. It is then the execution of a (software) Haskell program that generates the circuit, since statically sized structures are required for most structural hardware descriptions. In this case, the output circuit *might* be equivalent to the `toatie` implementation, but there are no guarantees about wordlengths checked by the compiler — this is what we have addressed with dependent types. Similar benefits are demonstrated in an adjacent domain; the language Proto-Quipper-D uses dependent types to ensure properties about the structure of entire families of parameterised quantum circuits [58]. Without these compiler checks, there is a large burden on the developer to provide good evidence of testing.

5.4 FORMAL VERIFICATION OF A CIRCUIT FAMILY'S ARITHMETIC MEANING

The approaches to circuit description explored in Sections 5.2 and 5.3 are often satisfactory, employing dependent types to naturally describe circuit structures precisely. However, in a dependently typed HDL, the designer could also choose to tackle a circuit family's implementation and its formal verification simultaneously.

This section demonstrates the extreme of a developer's reliance on the type system — ensuring full functional correctness. We use a correct-by-construction approach where the fact that an implementation successfully type checks is evidence that its functional behaviour matches our specification. This proof will often come "for free" in this section's later examples, once we have established a few arithmetic building blocks. In the cases where manual theorem proving *is* required, we will see that more designer effort is often required than alternative model checking techniques. However, the theorem proving approach enables us to reason about entire, potentially infinite, circuit families at once. We sidestep the state-space explosion encountered by traditional model checking methods.

The rest of this section follows the correct-by-construction development of a signed dot product and a (radix-2 decimation-in-time) FFT circuit. The FFT example also provides a proof showing the equivalence between the radix-2 Decimation-In-Time (DIT) algorithm and the Discrete Fourier Transform (DFT).

5.4.1 *A verified unsigned adder*

We begin by considering an unsigned adder. This builds upon Brady's work in [22] by presenting a synthesisable example (with the required erasure and staging) rather than just a model of the desired circuit behaviour. The remaining subsections extend these ideas and apply them to larger DSP challenges.

Beginning the journey towards a verified unsigned adder circuit, we must introduce some more precise data types for bits and unsigned words. In Section 5.3 we saw an unsigned binary type indexed by its wordlength and its (potentially stricter) range. In order for our circuits to be correct-by-construction, we want to index an unsigned word by its wordlength and the exact natural number encoded by its bits.

Listing 5.14 presents a type for `Bit`, indexed by a `Natural` number encoding of its value, and a type for `Unsigned` binary words, indexed by its wordlength and its `Natural` number encoding. This is an extreme case of working with precise types — an `Unsigned`'s indices uniquely identifies its value! That is to say, there is exactly one com-

Listing 5.14: A binary word indexed by its wordlength and encoded Nat

```
1 simple Bit : Nat → Type where
2   O : Bit 0
3   I : Bit 1
4
5 simple Unsigned : Nat → Nat → Type where
6   UNil  : Unsigned 0 0
7   UCons : {width, val, b : Nat} →
8           Unsigned width val → Bit b →
9           Unsigned (S width) (plus b (double val))
```

bination of n bits that encodes a particular number (within the interval $[\![0, 2^n - 1]\!]$). As a consequence, if we can satisfy the type checker when implementing a function over `Unsigned` types, we can have total confidence in the arithmetic performed by that function.

Note that we append `Bit`s to the least significant side of a `Unsigned` word. This choice is quite arbitrary, but does result in a slightly simpler inductive description of our `Nat` encoding. Appending a bit (b) to the *left* of a word (x) is equivalent to a shift-and-add rule ($b + 2x$), whereas appending to the right would also depend on the word's current width. Also note that `toatie` will happily synthesise descriptions using `Unsigned` types where the `val` argument is unknown (left as an irrelevant argument) since, even without `val`, we can directly infer a fixed bit representation from the `width` index alone. This is the common case — it would be quite rare to know the exact value encoded by a run-time input a priori!

Let's revisit our full adder implementation. In contrast to Listing 5.2, `addBit` is now dependently typed, precisely guaranteeing that the pair of output bits will represent the sum of the two operands and the carry input.

Like before, we describe the full adder in look-up table style. The main difference is that we return a `BitPair` which now includes a proof that the encoding of our bit pair ($B_a B_b$) is exactly $b + 2a$. This proof will be used when constructing the adder for `Unsigned` operands shortly. For now, notice that it is marked as irrelevant (and will be erased before the netlist is generated) and we can just use reflexivity to demonstrate this proof for all cases in the single-bit full adder. With this implementation, changing any of the output bits erroneously will result in a type error, since the output is no longer equivalent to $c + (x + y)$. This is what we mean by the term "correct by construction" — a function with incorrect behaviour is guaranteed to fail type checking.

Let's compose these full adders into an adder for w-bit `Unsigned` words. Listing 5.16 shows a partial implementation of this function, leaving a hole for the final output. The

Listing 5.15: A verified full adder

```
simple BitPair : Nat → Type where
  MkBitPair : {a,b,c : Nat} →
              {prf : Equal Nat (plus b (double a)) c} →
              Bit a → Bit b → BitPair c

addBit : {c,x,y : Nat} → Bit c → Bit x → Bit y →
         BitPair (plus c (plus x y))
addBit {_} {_} {_} O O O = MkBitPair {0} {0} {0} {Refl {_} {_}} O O
addBit {_} {_} {_} O O I = MkBitPair {0} {1} {1} {Refl {_} {_}} O I
addBit {_} {_} {_} O I O = MkBitPair {0} {1} {1} {Refl {_} {_}} O I
addBit {_} {_} {_} O I I = MkBitPair {1} {0} {2} {Refl {_} {_}} I O
addBit {_} {_} {_} I O O = MkBitPair {0} {1} {1} {Refl {_} {_}} O I
addBit {_} {_} {_} I O I = MkBitPair {1} {0} {2} {Refl {_} {_}} I O
addBit {_} {_} {_} I I O = MkBitPair {1} {0} {2} {Refl {_} {_}} I O
addBit {_} {_} {_} I I I = MkBitPair {1} {1} {3} {Refl {_} {_}} I I
```

structure of this function is identical to the adder introduced back in Section 5.2.1, but we have slightly more work ahead of us here.

Listing 5.16: A partial implementation of a verified unsigned adder family

```
addU : (w : Nat) → {x,y,c : Nat} →
       ⟨Unsigned w x⟩ → ⟨Unsigned w y⟩ → ⟨Bit c⟩ →
       ⟨Unsigned (S w) (plus c (plus x y))⟩

pat c, cin ⇒
  addU 0 {0} {0} {c} ⟦ UNil ⟧ ⟦ UNil ⟧ cin
    = ⟦ UCons {_} {0} {c} UNil ~cin ⟧

pat w, c, xsn, xn, xbs, xb, ysn, yn, ybs, yb, cin ⇒
  addU (S w) {_} {_} {c} ⟦ UCons {w} {xsn} {xn} xbs xb ⟧
                         ⟦ UCons {w} {ysn} {yn} ybs yb ⟧ cin
    = ⟦ case (addBit {_} {_} {_} ~cin xb yb) of
          pat a, b, prf, cin', lsb
          ⇒ (MkBitPair {a} {b} {_} {prf} cin' lsb) ⟹
               let rec = ~(addU _ {_} {_} {_} ⟦ xbs ⟧ ⟦ ybs ⟧ ⟦ cin' ⟧)
                   ans = UCons {_} {_} {_} rec lsb
               in _
      ⟧
```

Given our past experience implementing adders, we might expect `ans` to be accepted as "correct". However, looking at `toatie`'s output upon encountering the hole, the type of `ans` does not align with the type that our function expects to return.

The challenge we are left with is to demonstrate to the type checker that our attempt's `Nat` encoding, similar to $b + 2(a + (xsn + ysn))$ is equivalent to the expected form, similar to $c + ((xn + 2xsn) + (yn + 2ysn))$. In isolation, this seems impossible but we also have the proof supplied with our `BitPair` in scope, justifying the relationship between a, b, xn, and yn. Before we dive into the specifics of this proof in `toatie`,

Listing 5.17: An excerpt of `toatie`'s output for `addU`

```
Holes:
w   :0 Nat
c   :0 Nat
xsn :0 Nat
xn  :0 Nat
ysn :0 Nat
yn  :0 Nat
a   :1 Nat
b   :1 Nat
prf :1 (Equal Nat (plus b[0]  (plus a[1] a[1]))
                  (plus c[11] (plus xn[9] yn[5])))
ans :1 (Unsigned (S (S w[16]))
          (plus b[4] (plus (plus a[5] (plus xsn[14] ysn[10]))
                           (plus a[5] (plus xsn[14] ysn[10])))))
--------------------------------
{_:1310} : (Unsigned (S (S w[17]))
              (plus c[16] (plus (plus xn[14] (plus xsn[15] xsn[15]))
                                (plus yn[10] (plus ysn[11] ysn[11])))))
```

it may be helpful to attempt the proof with pen-and-paper. We offer one solution below, only assuming the associativity of addition and a lemma to commute pairs of additions.

Lemma 5.4.1. $\forall i, j, k \in \mathbb{N}$ *the expression* $i + (j + k)$ *is equal to* $(i + j) + k$.

Lemma 5.4.2. $\forall i, j, k, l \in \mathbb{N}$ *the expression* $(i + j) + (k + l)$ *is equal to* $(i + k) + (j + l)$.

Proof. Let $a, b, c, xsn, xn, ysn, yn \in \mathbb{N}$ where $b + (a + a) = c + (xn + yn)$. So,

$$
\begin{aligned}
& b+((a+(xsn+ysn))+(a+(xsn+ysn))) && \text{(Lemma 5.4.2)} \\
&= b+((a+a)+((xsn+ysn)+(xsn+ysn))) && \text{(Lemma 5.4.1)} \\
&= (b+(a+a))+((xsn+ysn)+(xsn+ysn)) && \text{(substituting variables)} \\
&= (c+(xn+yn))+((xsn+ysn)+(xsn+ysn)) && \text{(Lemma 5.4.2)} \\
&= (c+(xn+yn))+((xsn+xsn)+(ysn+ysn)) && \text{(Lemma 5.4.1)} \\
&= c+((xn+yn)+((xsn+xsn)+(ysn+ysn))) && \text{(Lemma 5.4.2)} \\
&= c+((xn+(xsn+xsn))+(yn+(ysn+ysn)))
\end{aligned}
$$

□

Assuming that we already have functions proving these two lemmas (`toatie`'s standard library does), we can directly translate this proof to a definition in `toatie`. We can start from the reflexivity of the LHS, and then for every step in the proof above, we use `eqInd2` to incrementally rewrite part of our expression until we reach the RHS. While verbose, Listing 5.18 shows the full implementation of this proof in `toatie`.

Listing 5.18: Lemma for our unsigned adder

```
lemmaAddU : (c, xb, yb, a, b, xsb, ysb : Nat) →
            (prf : Equal Nat (plus b (double a)) (plus c (plus xb yb)) ) →
            (Equal Nat
             (plus b (double (plus a (plus xsb ysb))))
             (plus c (plus (plus xb (double xsb)) (plus yb (double ysb))))
            )
pat c, xb, yb, a, b, xsb, ysb, prf ⇒
  lemmaAddU c xb yb a b xsb ysb prf
    = let h0 = Refl {_} {plus b (double (plus a (plus xsb ysb)))}
          h1 = eqInd2 {_} {_} {_}
                 {plusPairsCommutative a (plus xsb ysb) a (plus xsb ysb)}
                 {λn ⇒ Equal Nat (plus b (double (plus a (plus xsb ysb))))
                                 (plus b n)
                 } h0
          h2 = eqInd2 {_} {_} {_}
                 {plusAssociative b (double a) (double (plus xsb ysb))}
                 {λn ⇒ Equal Nat (plus b (double (plus a (plus xsb ysb))))
                                 (n)
                 } h1
          h3 = eqInd2 {_} {_} {_}
                 {prf}
                 {λn ⇒ Equal Nat (plus b (double (plus a (plus xsb ysb))))
                                 (plus n (double (plus xsb ysb)))
                 } h2
          h4 = eqInd2 {_} {_} {_}
                 {plusPairsCommutative xsb ysb xsb ysb}
                 {λn ⇒ Equal Nat (plus b (double (plus a (plus xsb ysb))))
                                 (plus (plus c (plus xb yb)) n)
                 } h3
          h5 = eqInd2 {_} {_} {_}
                 {eqSym {_} {_} {_} (plusAssociative c (plus xb yb)
                                        (plus (double xsb) (double ysb)))}
                 {λn ⇒ Equal Nat (plus b (double (plus a (plus xsb ysb))))
                                 (n)
                 } h4
          h6 = eqInd2 {_} {_} {_}
                 {plusPairsCommutative xb yb (double xsb) (double ysb)}
                 {λn ⇒ Equal Nat (plus b (double (plus a (plus xsb ysb))))
                                 (plus c n)
                 } h5
      in h6
```

The developer is free (and encouraged) to construct such proofs interactively with the assistance of the type checker. Instead of trying to come up with a complete proof atomically, Listing 5.18 was constructed one step at a time, leaving the output as a hole each time. This lets us see a summary of every name available in our scope and their normalised types while incrementally working towards our goal. The type checker's normalisation process can often simplify proof obligations through evaluation too.

We can now complete the implementation of the adder by using `eqInd2` and this lemma to rewrite the type of `ans`. Our types ensure the behaviour matches the defini-

tion of `plus` for natural numbers. Here we have a stronger guarantee than traditional dynamic verification using testbenches — we prove correct behaviour *completely*, not just for a finite number of test cases. Our evidence is also stronger than static model checking techniques since it applies to *every* width of adder within our circuit family.

Now that we've completed the definition of a simple circuit family, we continue by exploring more complex circuits while only highlighting each of their unique challenges. Full source code for each example is available at [27].

5.4.2 *Signed arithmetic*

Our second stop on the tour of verified arithmetic circuits explores the differences encountered when implementing a *signed* adder. The overall methodology remains the same, but the data types are slightly different and there are some interesting edge cases which we encounter. The latter, in particular, helps expand our tool belt of theorem proving strategies.

If we wish to index a signed binary word by the numeric value it encodes, we must move away from the comfort of `Natural` numbers and towards an integer type. We present this inductive integer type as `ZZ` (a typographical pun on $\mathbb{Z}$) in Listing 5.19.

Listing 5.19: A definition for our integer data type

```
1 data ZZ : Type where
2   -- A positive Nat
3   Pos  : Nat → ZZ
4   -- The negated successor of a Nat
5   NegS : Nat → ZZ
```

We define an integer as, in essence, a `Nat` or a negated `Nat`. The only nuance is that we actually define the latter option as the negation of the successor of a `Nat` — precluding the embarrassing case of having two valid representations of zero. Functions and proofs using `ZZ` terms are defined very similarly to those with `Nats`, usually only needing cases for zero, the positive successor, and the negative successor.

We can define a signed binary word data type, indexed by its integer encoding. Listing 5.20 gives an example of this, introducing a new data constructor, `SMsb`, which enforces the negative weighting associated with the most significant bit in the 2's complement scheme. Note that the data constructor for appending a bit to the left of an existing signed word, `SCons`, has a type which ensures that we can never accidentally begin a signed word with a positively weighted bit on the right.

Listing 5.20: A definition for our signed binary word type

```
simple Signed : Nat → ZZ → Type where
  -- Empty word
  SNil  : Signed 0 (Pos 0)
  -- Single MSB with a negative weighting
  SMsb  : {b : Nat} → Bit b → Signed 1 (negateZ (Pos b))
  -- Append LSB with positive weighting
  SCons : {width, b : Nat} → {val : ZZ} →
          Signed (S width) val → Bit b →
          Signed (S (S width)) (plusZ (Pos b) (doubleZ val))
```

Perhaps the usefulness of representing empty words (via `SNil`) is questionable, but we retain it for completeness. We can then properly handle situations such as encoding the result of a multiplication with a zero constant — commonplace in the half-band filters from Chapter 4.

We conclude by recounting two interesting verification challenges exposed by the signed adder. First of all, we are immediately confronted with a consequence of allowing empty signed words! While an adder for empty inputs was our base case for unsigned addition, consider the type we might attempt to write for a 0-bit signed adder:

```
emptyAddS : {x,y : ZZ} → {c : Nat} → ⟨Signed 0 x⟩ → ⟨Signed 0 y⟩ →
            ⟨Bit c⟩ → ⟨Signed 1 (plusZ (Pos c) (plusZ x y))⟩
```

Our output must be a 1-bit signed number that encodes $c + x + y$. We can deduce that x and y must both be zero since they are representations of 0-bit words. When the carry input is zero, we could construct a valid output (`SMsb {_} 0`) but what should the implementation return when the carry input is set high? We simply cannot encode the correct result (1) as a 1-bit signed word. Our solution is to only allow addition/subtraction for non-empty arguments at this stage. The specialised functions for addition and subtraction are free to allow empty arguments *after* the carry input has been provided.

The second verification challenge appears on the MSB side, in the structure of the sign extension and discard of the final carry output. Figure 5.6 shows the structure of the two most significant output bits in a signed adder (s_2 and s_1). Note that x and y are the MSBs of each operand, c_0 is the carry input driven by the previous full adder stage.

Although this is likely a familiar structure, the reader may find it non-trivial to justify from first principles. We might be convinced that the sign extension (by repeating the MSBs x and y) has no effect on the numerical meaning of the inputs, but what about our right to discard c_2, for example? If we work though the equations for the

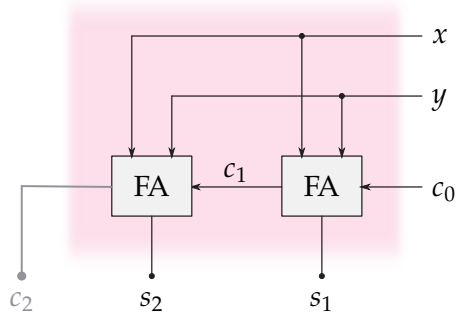

Figure 5.6: Structure for the two most significant output bits in a signed adder

full adder (Listing 5.15), or indeed let `toatie` report them automatically, we quickly encounter a constraint. Noting that x, y, and s_2 should be negatively weighted since they are the MSBs of 2's complement numbers:

Theorem 5.4.3. *The signed adder structure in Figure 5.6 is correct if:*

$$\begin{aligned} s_1 - 2s_2 &= c_0 + (-x) + (-y) \\ \textit{where } s_1 + 2c_1 &= c_0 + x + y \\ s_2 + 2c_2 &= c_1 + x + y \end{aligned}$$

Theorem 5.4.3 *might* be true, if we can demonstrate that $c_1 = c_2$. The equality of two successive carry outputs might seem like a strange proposition to make, but it turns out this *is* justifiable. We provide a formal proof in [27] while, informally, we can look at the truth table for the full adder. We want to analyse if and when the carry output might differ from the carry input. There are only two out of eight cases which can cause a change between carry bits. Following these two cases across both full-bit adders in Figure 5.6, we discover that there are only three possibilities since x and y are common to both full-adders:

$$c_2 = \begin{cases} c_1 & \text{when } c_0 = c_1 \\ 1 & \text{when } c_0 = 0,\ c_1 = 1,\ x = y = 1 \\ 0 & \text{when } c_0 = 1,\ c_1 = 0,\ x = y = 0 \end{cases} \tag{5.4}$$

In all cases, we conclude that $c_1 = c_2$ really is true. The implementation of this in `toatie` dismisses the other impossible cases with assistance from the coverage checker. Concluding our experience with the signed adder, having a conversation like this with a type checker can expose some cracks in the designer's own understanding of fundamental DSP constructs — even the signed adder hides some nuance which we seldom confront so explicitly!

5.4.3 *A verified, signed dot product*

Let's now consider our recurring example — the combinatorial dot product. This is a particularly interesting example of the correct-by-construction methodology because, as we will see, its verification requires no extra theorem proving! In fact, many DSP structures are quite direct compositions of fundamental arithmetic blocks so, once we have implementations of verified arithmetic primitives, the developer enjoys either little or no extra proof obligations. We first need to briefly introduce two prerequisites: our verified constant multiplication, and *heterogeneous* collections of signed words.

We choose to implement constant multiplication with a simple shift-and-add algorithm. This is a simple alternative to the constant multiplication algorithms presented in Chapter 4. Our implementation is guided not just by a numerical constant, but a more useful structural *view* of that constant. For a `Natural` coefficient, we opt to view it as a series of divisions of two, essentially revealing the binary structure of the `Nat`. This directly informs the structure of a shift-and-add multiplier, as seen in Figure 5.7.

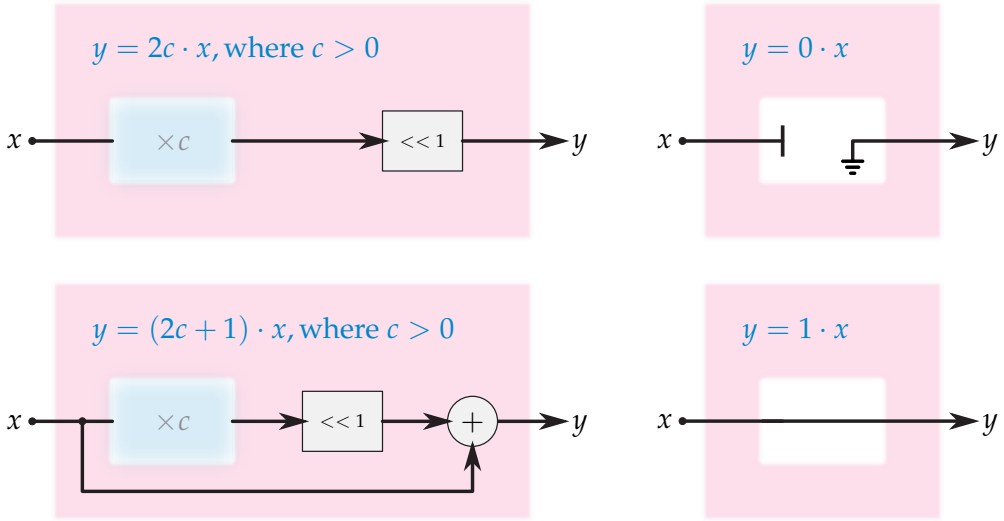

Figure 5.7: The structure of our shift-and-add multiplier given a recursive halving view of a natural coefficient

We infer an arithmetic shift left by one bit when the coefficient can be halved without remainder, while we shift *and* add the input when there is a remainder. There are simple base cases for coefficients of zero and one. The data type used to capture this view of `Nats` is shown in Listing 5.21.

Listing 5.21: A view for recursively dividing a Nat in half

```
data HalfRec : Nat → Type where
  HalfRecZ : HalfRec 0
  HalfRec1 : HalfRec 1
  HalfRecEven : (n : Nat) → HalfRec (S n) → HalfRec (plus (S n) (S n))
  HalfRecOdd  : (n : Nat) → HalfRec (S n) → HalfRec (S (plus (S n) (S n)))
```

We direct the reader to [27] for the full listing of the constant multiplication between an integer coefficient and a `Signed` word, `mulConstS`, since it does not introduce any new concepts. Note that our implementation does not guarantee absolute minimum wordlengths (see Section 5.3), instead favouring *simple* descriptions which are provably correct. As with our adder circuit, we do need to provide some extra proofs to satisfy the type checker. These are still fairly trivial, only requiring arithmetic fundamentals such as the distributivity of multiplication and the neutrality of multiplication by one. The two main proof obligations we have are to justify the recursive structures shown in Figure 5.7. This boils down to proving the two following theorems:

Theorem 5.4.4. *For input and coefficient,* $x, c \in \mathbb{Z}$ *then* $0 + (c \times x + c \times x) = (c + c) \times x$

Theorem 5.4.5. *For input and coefficient,* $x, c \in \mathbb{Z}$ *then* $x + (c + c) \times x = (1 + c + c) \times x$

Now we can tackle the full dot product structure. Let's consider the type of the circuit family. It might feel natural to attempt encoding the collection of `Signed` inputs as a `Vect`, however, we would encounter a type error even for a simple case. Two signed words, for example `Signed w x` and `Signed w y`, demand two different types unless they encode the same integer value. Their width indices may also differ — and likely *should* after an optimised constant multiplication with different coefficients. This poses an issue since our definition of *Vect* enforces the property that all elements have the same type. A good solution for this is a heterogeneous vector; one which is indexed by a *list* of element types. We instead introduce a specialised version of this for heterogeneous collections of `Signed` elements in Listing 5.22. Such an approach lets us very easily reason about the width and encoded value indices as their own collections, discussed further in Section 5.4.4.

Listing 5.22: A heterogeneous collection of `Signed` words

```
simple HWords : (n: Nat) → Vect n Nat → Vect n ZZ → Type where
  HNil : HWords 0 [] []
  HCons : {n, w : Nat} → {ws   : Vect n Nat} →
          {val  : ZZ } → {vals : Vect n ZZ } →
          Signed w val → HWords n ws vals →
          HWords (S n) (VCons {Nat} {n} w ws) (VCons {ZZ} {n} val vals)
```

Just as a `Vect n a` is a collection of elements indexed by its length and element type, `HWords n ws vals` is a collection of `Signed` elements indexed by its length, and each element's wordlength and each element's integer encoding via two type-level `Vects`. With `HWords`, we can precisely describe the dot product's set of input words in a way suitable for our correct-by-construction approach. The full listing for the verified dot product circuit family is shown in Listing 5.23.

Listing 5.23: A definition for our signed, verified dot product

```
-- Description of wordlengths along our dot product's adder chain
dotProdBits : (j : Nat) → (ws : Vect j Nat) → (cs : Vect j ZZ) → Nat
pat j, ws, cs ⇒
  dotProdBits j ws cs
    = foldr j {_} {_} (λw : Nat ⇒ λacc : Nat ⇒ S (max w acc)) 0
        (zipWith j {_} {_} {_} mulConstBitsS cs ws)

-- A verified dot product with heterogeneous wordlengths
dotProd : (j : Nat) → (ws : Vect j Nat) → {vals : Vect j ZZ} →
          (cs : Vect j ZZ) →
          ⟨HWords j ws vals⟩ →
          ⟨Signed (dotProdBits j ws cs)
                  (sumZ j (zipWith j {_} {_} {_} multZ cs vals))⟩
dotProd Z [] {[]} [] ⟦ HNil ⟧ = ⟦ SNil ⟧
pat j, w, ws, val, vals, c, cs, x, xs ⇒
  dotProd (S j) (VCons {_} {_} w   ws)
                {VCons {_} {_} val vals}
                (VCons {_} {_} c   cs)
                ⟦ HCons {_} {_} {_} {_} {_} x xs ⟧
    = addS' _ _ {_} {_} (mulConstS _ c {_} ⟦ x ⟧ )
                        ⟦ ~(dotProd _ _ {_} cs ⟦ xs ⟧ ) ⟧
```

The type of this implementation *guarantees* that output of every single specialised dot product circuit will always encode the result $\sum_{i=0}^{j-1}(cs_i \times vals_i)$ for coefficients cs, and inputs encoding $vals$. This property is formally verified, machine checked, and applies to every member of the circuit family. Best of all we get this property "for free" in terms of developer effort. Once the fundamental arithmetic circuits are provided, this often happens for *direct* implementations of DSP circuits — the property enforced on the integer indices will often have an identical structure to that of the circuit's implementation. In this case, we do not need to construct any additional proofs to satisfy the type checker but we still enjoy its full functional verification.

This is, of course, the best case scenario. There are many applications which place a larger burden on the developer when proving properties about a circuit family. There are also other ways in which we might want to use these proof mechanisms for circuit description. The following section marks the end of our examples, culminating in an optimised FFT circuit, and highlights a subtly different use of theorem proving in `toatie`.

5.4.4 *FFT*

Thus far, we have been concerned with guaranteeing full functional correctness of a circuit family relative to a reference function defined over `Nats` or `ZZs`. This section demonstrates a slight twist on this theme — using theorem proving to justify a non-trivial reference function. We present an *optimised* (radix-2 DIT) DFT, enjoying a correct-by-construction approach, and then use `toatie` to prove that the optimised radix-2 DIT algorithm really is equivalent to the direct DFT definition.

The standard definition of the DFT for a length (N) is shown in Equation (5.5). It requires a constant *twiddle factor* (W_N) defined as $e^{-j2\pi/N}$, and a series of time-domain inputs (x).

$$X(k) = \sum_{n=0}^{N-1} x(n) \cdot W_N^{kn}, \qquad \text{for } k \in \{0 \ldots N-1\} \tag{5.5}$$

Note that the twiddle factor is raised to various different powers, which are generally all precomputed (called the *twiddle factors*). These powers exhibit some interesting properties, due to the symmetries of complex exponentials. These patterns are exactly what the radix-2 DIT algorithm exploits in order to reduce the DFT's computational cost. Defined as an extended version of [94], the twiddle factor properties include:

$$W_N^0 = 1 \tag{5.6}$$

$$W_N^N = 1 \tag{5.7}$$

$$W_N^{N/2} = -1 \tag{5.8}$$

$$W_n^k \cdot W_n^m = W_n^{k+m}, \qquad \text{for } 0 < n \leq N, \text{ and } k \leq N \tag{5.9}$$

$$W_n^k = W_{2n}^{2k}, \qquad \text{for } 0 < n \leq N, \text{ and } k \leq N \tag{5.10}$$

The radix-2 DIT algorithm can use Equations (5.6) to (5.10) to rewrite the DFT equation (Equation (5.5)) into an expression in terms of just two $N/2$-length DFTs. From there we can use recursion to realise each $N/2$-length subtransform, assuming N is a power of two. We begin by splitting the summation into two: one for even components and one for odd.

$$X(k) = \sum_{m=0}^{N/2-1} x(2m) \cdot W_N^{(2m)k} + \sum_{m=0}^{N/2-1} x(2m+1) \cdot W_N^{(2m+1)k}, \qquad \text{using 5.9} \tag{5.11}$$

$$X(k) = \sum_{m=0}^{N/2-1} x(2m) \cdot W_N^{(2m)k} + W_N^k \sum_{m=0}^{N/2-1} x(2m+1) \cdot W_N^{(2m)k} \tag{5.12}$$

Let's continue by considering each half of X independently. For the lower half, where $k < \frac{N}{2}$, we can use the twiddle factor property (5.10) to reduce the equation to two $N/2$-size DFTs. We label the "even" and "odd" halves as E_k and O_k respectively.

$$X(k) = \sum_{m=0}^{N/2-1} x(2m) \cdot W_{N/2}^{mk} + W_N^k \sum_{m=0}^{N/2-1} x(2m+1) \cdot W_{N/2}^{mk}, \quad \text{where } k < \frac{N}{2} \tag{5.13}$$

$$X(k) = E_k + W_N^k \cdot O_k \tag{5.14}$$

For the upper half, where we use indices of the form $k + \frac{N}{2}$, we employ twiddle factor properties from Equations (5.7) to (5.10) to refine the expression into a very similar structure. Applying these properties step-by-step gives:

$$X(k+\frac{N}{2}) = \sum_{m=0}^{N/2-1} x(2m) \cdot W_N^{(2m)(k+N/2)} + W_N^{k+N/2} \sum_{m=0}^{N/2-1} x(2m+1) \cdot W_N^{(2m)(k+N/2)} \tag{5.15}$$

$$X(k+\frac{N}{2}) = \sum_{m=0}^{N/2-1} x(2m) \cdot W_{N/2}^{mk} \cdot W_N^{mN} + W_N^{k+N/2} \sum_{m=0}^{N/2-1} x(2m+1) \cdot W_{N/2}^{mk} \cdot W_N^{mN} \tag{5.16}$$

$$X(k+\frac{N}{2}) = \sum_{m=0}^{N/2-1} x(2m) \cdot W_{N/2}^{mk} \cdot 1 + W_N^k \cdot W_N^{N/2} \sum_{m=0}^{N/2-1} x(2m+1) \cdot W_{N/2}^{mk} \cdot 1 \tag{5.17}$$

$$X(k+\frac{N}{2}) = \sum_{m=0}^{N/2-1} x(2m) \cdot W_{N/2}^{mk} + W_N^k \cdot (-1) \sum_{m=0}^{N/2-1} x(2m+1) \cdot W_{N/2}^{mk} \tag{5.18}$$

$$X(k+\frac{N}{2}) = E_k - W_N^k \cdot O_k \tag{5.19}$$

Now both the lower and upper halves can share the subtransforms, E_k and O_k, each of which can be implemented themselves as a radix-2 DIT structure. Identifying this resource sharing opportunity is the advantage of the optimised DIT structure. This existing derivation from first principles is afforded so much emphasis here because this is *exactly* the process that one must translate to `toatie` when proving the equivalence between the direct DFT and radix-2 DIT algorithms. Armed with the twiddle factor properties, the above derivation, and the theorem proving mechanisms we have already seen, we can construct a proof in `toatie`. The only extra effort required is to defend the *structural* changes we made to the expressions during our derivation. For example, we must demonstrate that it is indeed OK to reorder and split a summation

into its even and odd elements, and demonstrate that we can build an equality between vectors given different subproofs for the lower and upper halves.

The full implementation is available at [27], but we list the type of this proof in Listing 5.24 to introduce some new concepts. This is an example for a complete FFT *structure* but we choose to work only over integer values. While a fixed-point complex number representation is required for the general case, valid integer FFTs can still be described where $N \leq 2$. Our implementation does not exploit this restriction in a meaningful way, and could be used in full with drop-in replacements for our integer arithmetic functions.

Listing 5.24: The type showing equivalence between radix-2 DIT and direct DFT algorithms

```
1 eqDit : {n : Nat} → {f : Nat → Nat → ZZ} → (isPow2 : Pow2 n) →
2         (tw : Twiddles f n) → (xs : Vect n ZZ) →
3         Equal (Vect n ZZ) (dft  n  {f} tw        xs)
4                           (dit {n} {f} tw isPow2 xs)
```

Listing 5.24 lays out a proposition saying that, for all valid twiddle factors, power of two lengths, and possible integer inputs, the result of the `dft` and `dit` transforms are exactly equal. If we implement a circuit family that conforms to the behaviour of the `dit` function, we can say that its behaviour is also equivalent to the `dft` function.

You may also notice two new types used in Listing 5.24. `Pow2 n` is a "witness" that n is a power of two — it only has constructors which return `Pow2` indexed by `1`, or double that of another `Pow2`. This lets us restrict the valid values of the argument n to satisfy the radix-2 DIT structure's preconditions. The other new type is `Twiddles f n`. This is also, in essence, a witness that f is a valid source of twiddle factors. Its only constructor expects 5 proof arguments; one proof for each of the twiddle factor properties in Equations (5.6) to (5.10). Listing 5.25 shows these witnesses in action, used during the implementation of our radix-2 DIT the circuit family.

Although Listing 5.25 references many helper function whose definitions are omitted, we can quite clearly see two things:

↪ The overall structure of the DIT; recursing on the even and odd input halves, scaling by the twiddle factors, and recombining the halves.

↪ There are no additional proofs required to demonstrate that this implementation matches the arithmetic specified by the `dit` function! This is simply because the structures of the DIT over integers and the DIT over signed binary words are identical. We also reuse the correct-by-construction arithmetic circuits implemented throughout this chapter.

Listing 5.25: A verified circuit family for radix-2 DIT DFTs

```
circDIT : {n : Nat} → {f : Nat → Nat → ZZ} → (tw : Twiddles f n) →
          (pow : Pow2 n) → (ws : Vect n Nat) → {xs : Vect n ZZ}    →
          ⟨HWords n ws xs⟩ →
          ⟨HWords n (dftWidth {n} {f} tw pow ws) (dit {n} {f} tw pow xs)⟩
pat f, tw, w, x,  bs ⇒
  circDIT {1} {f} tw POne [w] {[x]} bs = bs
pat n, f, tw, prec, ws, xs, bs ⇒
  circDIT {double' n} {f} tw (PDouble n prec) ws {xs} bs
    = let -- Recurse on N/2 DFTs
          es = circDIT {n} {f} (halfTwiddles {f} n tw) prec _ {_}
                 (evensH n {ws} {xs} bs)
          os = circDIT {n} {f} (halfTwiddles {f} n tw) prec _ {_}
                 ( oddsH n {ws} {xs} bs)

          -- Scale odds by twiddle factors
          os' = zipWithConstH n {ZZ} _ {_} {_} {_}
                  (λw ⇒ λ{v} ⇒ λc ⇒ λb ⇒ mulConstS w c {v} b)
                  (twiddleRow {f} (double' n) tw 1 n)
                  os

          -- Combine halves
          outEs = zipWithH n _ _ {_} {_} {_} {_} addS' es os'
          outOs = zipWithH n _ _ {_} {_} {_} {_} subS' es os'
          outs = appendHalvesH n {_} {_} {_} {_} outEs outOs
      in outs
```

With standard library functions provided (such as `zipWithH`, `evensH`, and `oddsH`), it takes ≈ 25 lines of code to define this realistic DFT circuit. Moreover, the description is totally generic in terms of transform length and wordlengths of *each* and every input sample. The integer value encoded at the output is *guaranteed* to correspond to our software DIT implementation. In balance, there is a lot of developer effort hidden under the other top-level functions — but most of these are generic, reusable blocks, easily composed in a functional language. Clearly, after the initial effort required to capture some correct-by-construction DSP fundamentals, it can be a relatively cheap exercise to explore their composition for larger, realistic applications.

5.5 FURTHER WORK

The most pressing avenue for further work at the language-level is the support for synchronous logic. The fact that `toatie` only synthesises combinatorial logic is its most limiting factor. However, there is a wealth of existing literature concerning how to encode synchronous signals in functional HDLs. These encodings ought to be applicable to `toatie` with only additional engineering effort so, although this thesis is largely restricted to combinatorial circuits, its contributions should still be broadly valuable. Moreover, the choice to defer handling of synchronous signals is justified by the *volume*

of engineering effort it induces in the compiler back-end for a flexible implementation. Every synchronous operation carries the baggage of its associated clock, clock enable, and reset signal. Allowing properly for different target technologies, the compiler and simulator would also need to be parameterised by enable & reset polarity, and reset synchronicity. Instead, this work prioritises efforts on the *unsolved* challenges of synthesising circuits from a dependently typed environment.

Many projects including [41, 44, 48] represent synchronous signals as infinite streams, at least for simulation. More relevantly, CλaSH in [19] does this while representing synchronous circuits as plain functions over streams. The author believes that a similar approach should be suitable for `toatie`, with additional support for non-strict semantics.

Perhaps the more interesting research aspect regards how developing circuits as functions over streams will impact theorem proving. Totality is vital when working with proofs in dependently typed languages. All descriptions presented in Chapter 5 have been *terminating* — given finite time (possibly an extremely long time) they will terminate. This is not always the case for an infinite (*coinductive*) stream type. Fortunately, we can still perform theorem proving on the sort of stream suitable for hardware description. All recursively defined circuits using streams will be *productive*. That is, whenever we use streams recursively, we will always add a new element to the head of a list — an analogy for inserting a one-cycle delay. There is precedent for encoding productive coinductive types in Idris (see Chapter 11 of [57]). Here it is possible to prove equality between productive functions over streams. The best way to represent this in the context of circuit description is left for future work, as is the check for productivity. It would ideally not just maintain our combinatorial verification techniques, but also *extend* them in order to exploit literature on software verification with dependent types. In particular, there is extremely relevant work in using dependent types to statically ensure that a state machine conforms to a given protocol [95]. This would be a great boon for any synchronous HDL.

Although `toatie` does yet not model or synthesise synchronous logic, the following section does offer an initial investigation into how these future circuit descriptions could assist DSP applications.

5.5.1 *Speculation on synchronous DSP circuits*

While all other sections in this chapter offer examples which are synthesisable to circuits in `toatie`, this subsection discusses *speculative* uses of this minimum wordlength methodology for synchronous circuits. It is speculative since our compiler does not

yet support synchronous circuits — only combinatorial circuits. This is not a fundamental restriction however, and the CλaSH project demonstrates how a compiler for lazy functional languages can cleanly support synchronous logic via quite a small extension.

Now let's take a look at how the methodologies we have encountered so far in our synthesisable applications could, in future, benefit synchronous DSP circuits as well.

A direct form FIR filter

Using our combinatorial dot product as a starting point, we are only a small step away from realising a synchronous, transpose-form FIR filter. If `toatie` had synchronous streams with an applicative interface, it would be trivial to lift an arbitrary combinatorial function up into a synchronous signal. This is analogous to taking some combinatorial expression in VHDL and restructuring it into a clocked process.

Listing 5.26 demonstrates how this FIR implementation might look. The function `liftA` applies a combinatorial function to a given stream, returning the transformed stream. The helper function `window` creates our delay line from a stream — returning the most recent '`j`' samples.

Listing 5.26: Speculation on lifting a combinatorial dot product up to a synchronous domain with an applicative interface

```
1 fir : (j,n : Nat) → (ws : Vect' j Nat) →
2       ⟨Stream (Bounded' n)⟩ →
3       ⟨Stream (Bounded' (mul n (sum j ws)))⟩
4 pat j, n, ws, x ⇒
5   fir j n ws x = liftA {_} {_} (dotProd _ _ ws) (window {_} j (zeroB _) x)
```

This is the crux of how synchronous logic could be described in future versions of `toatie`, as supported by similar approaches already used in Lava and CλaSH. We encode synchronous signals as infinite streams via a new primitive in the language, and encode delays as appending an element to the head of the stream. We then provide an applicative interface to lift combinatorial functions up to the synchronous domain, and to compose operations on streams.

Pruning in CIC Interpolators/Decimators

Moving away from the FIR example now, an interesting tangent to consider is the Cascaded Integrator-Comb (CIC) decimator/interpolator. Unlike our FIR and dot product examples, implementations of CIC filters with pruning *cannot* rely on synthesis tools to mask imprecise descriptions from traditional HDLs.

CIC decimation filters are often used as a very low-resource means of resampling — composed of a chain of N integrators, followed by a $\frac{1}{R}$ downsampler, followed by N comb filters with a differential delay of M. Figure 5.8 shows an example CIC decimator with $R = 8$, $N = 3$ and $M = 1$.

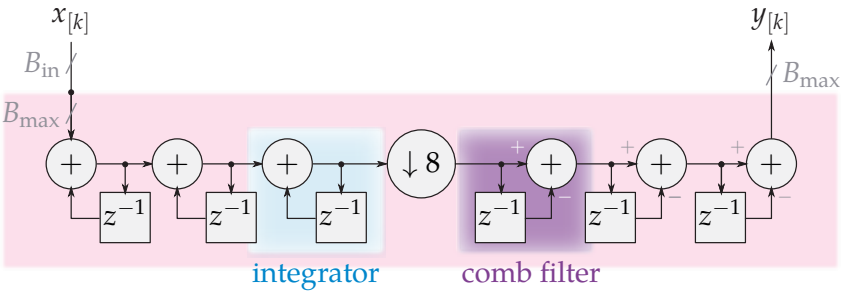

Figure 5.8: A CIC decimator without pruning. ($R = 8$, $N = 3$ and $M = 1$)

Hogenauer introduced a register pruning technique for CIC filters [96], deriving equations for the mean error and variance introduced by truncation at each stage. It is suggested that, given a desired output wordlength, a legitimate design choice is to prune the wordlengths of all previous stages maximally without accumulating an error greater than that introduced by the final rounding/truncation. This choice results in Equation (5.20), describing the number of LSBs to discard at the j^{th} stage.

$$B_j = \left\lfloor -\log_2 F_j + \log_2 \sigma_{T_{2N+1}} + \frac{1}{2}\log_2 \frac{6}{N} \right\rfloor \tag{5.20}$$

where F_j is the variance error gain for the j^{th} stage, $\sigma_{T_{2N+1}}$ is the total variance at the output due to truncation, and N is the number of stages. Note that the first two terms have complicated definitions of their own, including cases, sums, exponentials, and binomial coefficients [96].

As the pruned bits *do* contain information, synthesis tools cannot perform an equivalent optimisation given a non-pruned description. We theorise that the same techniques as shown in Section 5.5.1 could be used to implement and reason about a fully parameterised, pruned CIC decimator in an extended `toatie`. The extensions would need to include support for synchronous signals and the floating point primitives required in the pruning equations. This application quite clearly demonstrates the benefits of having a rich, *single language* that can be used at both the term-level and the type-levels. All language constructs can be used to implement Equation (5.20), and this can then be used to direct the type of each stage in a CIC implementation.

Although this suggestion is currently only speculative, we can still explore the post-layout results for this CIC pruning algorithm using an ad hoc Verilog/VHDL generator. Figure 5.9 presents the LUT usages for different CIC filter parameters, highlighting the resource savings attained by Hogenauer pruning. While these results are generated with a CIC utility from the KiwiSDR project [97] (a C program that returns Verilog code), the outputs should be structurally equivalent to our theorised implementation — only without the safety of our dependently typed properties.

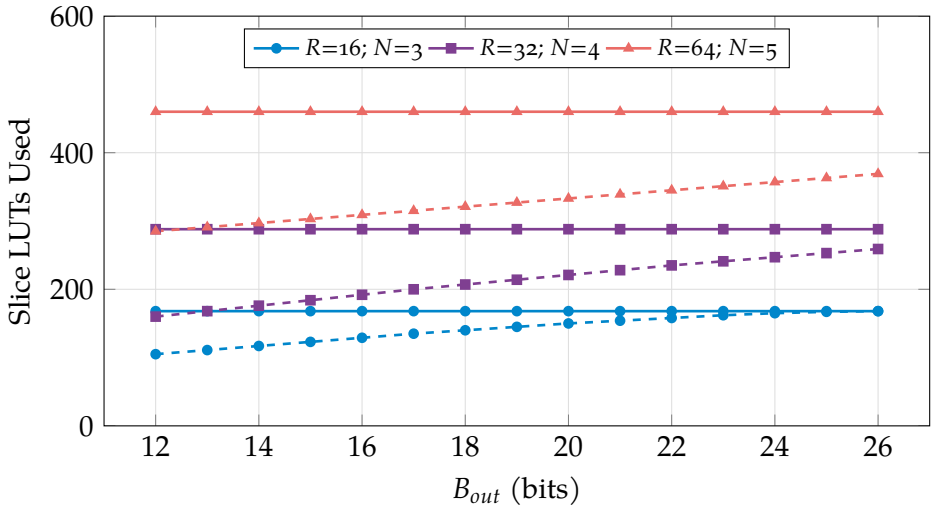

Figure 5.9: Post-layout results for ad hoc CIC filter generation with Hogenauer pruning (dashed lines) and without (solid lines). $M = 1$ and $B_{in} = 16$ for all lines.

Note that the approach to circuit generation in [97] is a prime example of some common issues that we are trying to address with dependent types. There is no verified contract between the IP designer and its user for the range of valid parameters, their effect on the generated circuit, and no clear evidence of testing. One such bug found while generating Fig 5.9 is that any B_{out} which requires bit extension at the final stage, rather than truncation, will silently generate syntactically invalid Verilog.

A dependently typed solution gives an environment in which we can play with wordlength pruning design choices easily — even if this pruning is derived from reasonably complex equations. This sort of lossy optimisation cannot be performed by traditional synthesis tools since it *does* involve discarding information; but it *is* one valid design choice, as highlighted by Hogenauer.

In comparison to HDLs embedded in Haskell, such as Lava, a similar structure may be technically realisable with type-level functions and singletons. However, Haskell does not offer the luxury of one rich language for both the term and type levels. Because of this, implementing the complex, type-level function required to represent Equation (5.20) could prove to be a particularly challenging excursion.

A note on synchronous control systems

Thus far we have been focused only on idealised DSP applications but there many real-world circuits, even within the realm of DSP, that require synchronous control systems. In general, we can capture the behaviour of synchronous control systems as Finite State Machines (FSMs). It is quite straightforward in most functional languages to implement a Mealy machine based on a pure, combinatorial function mapping the current state and input to the new state and output. The memory elements required to hold the state (and optionally register the outputs) can be introduced by lifting the combinatorial transfer function to the synchronous domain and specifying an initial state.

An implementation of an FSM in a functional HDL without dependent types can be well captured by a `State` monad (an abstraction over computations common enough to often merit its own special syntax). As an example, consider a simple state machine controlling the read channel of an AXI4 slave [98]. We present a state diagram in Figure 5.10, and an excerpt of the type definitions for functions encoding each transition in Listing 5.27.

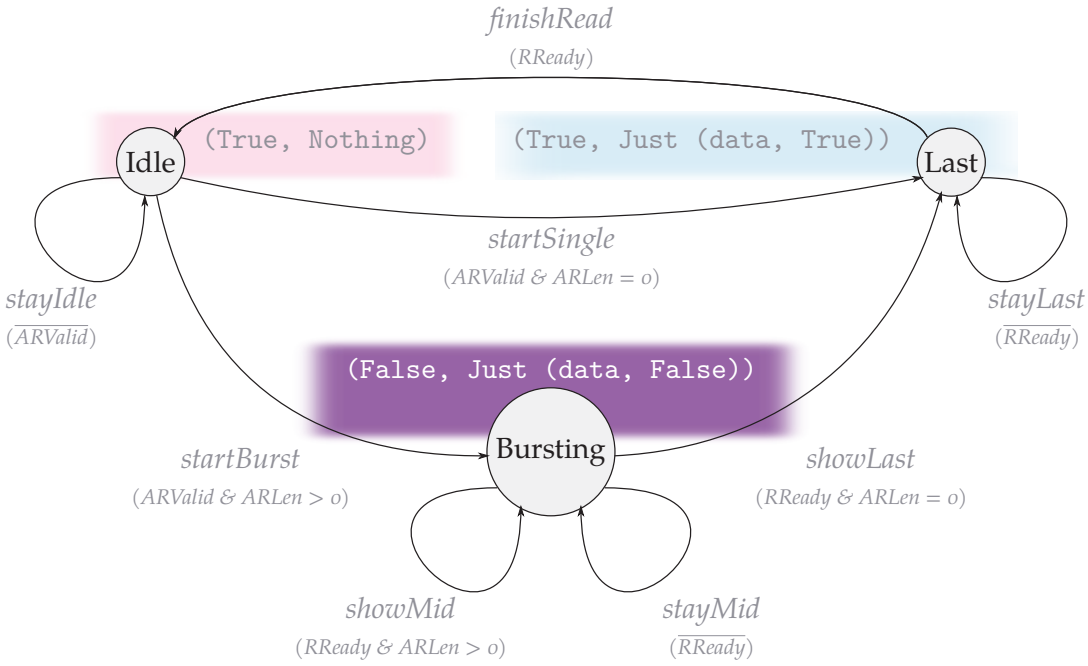

Figure 5.10: A state machine for a simple AXI4 slave read channel

Now, the main state machine function which glues together the different transitions must pattern-match on the state machine inputs and our current state. Even this simply typed example uses GADTs, `toatie`'s data type definitions, in two ways to improve on a more traditional implementation:

Listing 5.27: The data types and transition types for a simple AXI4 slave read channel

```
ROut : Type
ROut = Pair ARReady (Maybe (Pair RData RLast))

data RState : Type where
  Idle     : Mem                      → RState
  Bursting : Mem → ARAddr → ARLen →  RState
  Last     : Mem → ARAddr           → RState

startSingle : ARAddr           → State RState ROut
startBurst  : ARAddr → ARLen → State RState ROut
showMid     : ARAddr → ARLen → State RState ROut
showLast    : ARAddr           → State RState ROut
finishRead  :                    State RState ROut
stayIdle    :                    State RState ROut
stayMid     : ARAddr           → State RState ROut
stayLast    : ARAddr           → State RState ROut
```

1. We can encode inputs and outputs tagged with a "valid" signal as the common data type `Maybe`. This ensures that we do not have access to a piece of data unless it is flagged as valid, letting the type system preclude the temptation to reach for invalid inputs to satisfy an incorrect implementation.

2. We carry different fields with each of the `RState` constructors. Again, this helps us limit our access to only what should be necessary at each step, and diminishes the temptation to reuse stale state values rather than construct them properly from valid inputs.

Taking this idea further, an interesting avenue for future research considers how to best apply dependent types to enhance the description of hardware FSMs. Just like the three approaches we present in Sections 5.2 to 5.4, there are different extents to which a designer may choose to embrace type-level shenanigans.

While a basic Mealy machine implementation is often described via a `State` monad (as in [19]), a designer may choose to graduate to *indexed* monads, where the type of the output state may be of different from the type of the input state. This lets the type checker better enforce *when* a certain transition is a valid option — sometimes only between two specific states, and optionally with specific arguments. For our AXI example, an implementation with indexed monads could have the type checker ensure that each transition is only used to/from the expected states, and that the state is properly updated: e.g. `showMid` decrements the burst length and `showLast` can only be used once the burst length reaches zero. This is a technique which can be encoded in Haskell through use of type families. Brady takes this idea further in a dependently typed context, allowing the next state to depend on the *run-time* outputs of the pre-

vious stage [95]. In essence, we can encode transition types with possible run-time failure or other conditionals. For example, the type checker can enforce the fact that we check an authentication result *before* entering a secure state, or else fall back into an insecure mode.

Both of these extended implementations highlight one new challenge for our hardware description: `toatie` cannot currently mention the value of a synthesisable data type as part of a dependent type. We currently do not allow a (possibly) run-time value to influence the "shape" of a circuit during elaboration. This restriction is presented more formally in Section 6.4.2. Both FSM techniques (indexed monads and encoding possibly failing transitions) require the state to be used in dependent typing rules in the most general form. Future work could consider how these patterns might be adapted to fit within our framework, or how these restrictions might be relaxed in order to proliferate the literature for software state machine implementations.

5.6 SUMMARY

This chapter has offered an investigation into three main methodologies for digital design and verification when armed with a dependently typed, multistage, functional HDL. This thesis presents, to the best of the author's knowledge, the first set of *synthesisable* combinatorial DSP circuits encoded as plain functions in such a language.

Section 5.2 demonstrates that `toatie` can be used quite similarly to the combinatorial subsets of functional HDLs such as Lava or CλaSH. Section 5.3 builds on this by suggesting that, with stronger reliance on the type checker, we can use dependent types to more faithfully capture challenging DSP patterns. This was demonstrated with a focus on modelling wordlengths of intermediate signals. Finally, Section 5.4 explores a powerful *correct-by-construction* approach to the complete function verification of a circuit family. Although there is a burden of manual theorem proving when constructing our fundamental arithmetic circuits from first principles, we did begin to enjoy many theorems "for free" at the level of dot products and DFTs. This approach mixes the concerns of circuit implementation and circuit verification, and is quite uniquely facilitated by representing circuits as plain functions and ascribing precise meaning to our circuit run-time data via its type.

6 TOATIE — A MULTISTAGE HARDWARE DESCRIPTION LANGUAGE WITH DEPENDENT TYPES

This chapter discusses the design and implementation of the new HDL, `toatie`. The source code for the compiler, example designs, and testing infrastructure are all available as an open access artefact at [27].

6.1 INTRODUCTION

This chapter is concerned with detailing the concrete implementation of `toatie`'s core language, type system, and netlist generation. While Chapter 5 offered practical circuit descriptions in `toatie` and explored the merits of dependently typed HDLs more generally, we now change focus to the-behind-the-scenes machinery responsible for compiling these descriptions into synthesisable netlists.

While `toatie` is a language and compiler in its own right, we make no secret of its existing heritage — indeed, the reuse of existing codebases is one of its strengths. The core language and type checker of `toatie` is based on top of the existing dependently typed (software) language, TinyIdris. This chapter will formally introduce TinyIdris before detailing our own additions towards representing and compiling circuit descriptions. Maintaining this separation of concerns should help promote an understanding of which challenges are unique to hardware description, as well as clarifying exactly where this thesis' contributions begin.

TinyIdris was introduced at the 2020 Scottish Summer School on Programming Languages and Verification [28, 99] by Edwin Brady as pedagogical tool for exploring the implementation and ideas of his (much richer) language, Idris 2. Its small but complete core makes it an appealing project on which to base dependently typed research projects, such as `toatie`. TinyIdris eliminates many of Idris 2's ammenities and peripheral features, but its structure (data structures, algorithms, and even file structure) match Idris 2 *exactly* wherever this is feasible. Of course, this was likely a guiding principle due to its use as a teaching tool, but it lends some credence to the idea that `toatie`'s features could be transplanted onto a full Idris 2 base with relative ease. In this case, we would gain all of the niceties of a full-featured language and its development tools for "free".

We offer a formalisation of the TinyIdris language (extending the presentation at [28]) in Section 6.2. We cover the grammar of the surface language, its elaboration to the core language, and the core's type checking. We will see how a type checker in the presence of dependent types becomes entangled with the topics of unification, normalisation, and evaluation. Section 6.3 discusses the extensions to the language we need to consider in order to capture synthesisable hardware descriptions. We go on to discuss what restrictions we place on the surface language in an attempt to ensure that a circuit family is indeed synthesisable and then describe our (currently very simple) pipeline for generating netlists for downstream EDA tooling.

Before we explore the internals of TinyIdris, we first offer a brief background on the standard notation used in this chapter. The two most pertinent sets of notation are for describing *language syntax*, demonstrated with a simply typed λ-calculus, and its *typing judgements*.

6.1.1 The lambda calculus

Our introductory example takes the form of a simply typed lambda calculus ($\lambda^{\rightarrow}$) with primitive support for integer values and their addition. Lambda calculi are at the heart of functional programming, so we will briefly introduce its main concepts for the digital engineer and see the notation used in the rest of this chapter.

Figure 6.1 shows the syntax of our small example language using a common notation — the Backus–Naur Form (BNF). This introduces formal descriptions for two different constructs in the language: term-level *expressions* and their *types*. There are multiple valid forms for each description and we separate each option with a vertical bar (|).

$e,\ u ::=$	Expressions
$\quad x$	$\quad$ Variables
$\quad \mid \lambda x : \tau.\ e$	$\quad$ Function abstraction
$\quad \mid e\ u$	$\quad$ Function application
$\quad \mid n$	$\quad$ Integer literal
$\quad \mid e + u$	$\quad$ Addition
$\tau,\ \sigma ::=$	Types
$\quad \texttt{Int}$	$\quad$ Type of integers
$\quad \mid \tau \rightarrow \sigma$	$\quad$ Type of functions

Figure 6.1: A syntax for $\lambda^{\rightarrow}$

There are a limited number of ways we can construct our expressions, e. Our definition of $\lambda^{\rightarrow}$ has hard-coded support (or "primitives") for integer literals and addition. The other three constructs (variables, function abstraction, and function application) are the core of the lambda calculus. While deceptive in their simplicity, these three constructs are enough to encode data structures such as booleans, encode conditionals, and even recursion (in the untyped case). In prose these three constructs are:

x | Variables let us reference a named argument. These variable names are only ever introduced by function abstraction — x does not appear anywhere else on the RHS in Figure 6.1.

$\lambda x : \tau.\ e$ | Function abstraction introduces a new function. Here the argument is given the name x, whose type is τ, and the body of the function is the expression e. It is important to note that e can indeed include references to x. The function itself is *anonymous* — it is not given an explicit name.

$e\ u$ | Function application describes applying an expression u as the argument of a function expression e.

Programs written in $\lambda^{\rightarrow}$ can be evaluated very simply — exhaustively reducing the expression using a small set of rules. We need only two rules: one rule describing how to reduce addition of two *integer* values, and one rule describing how to apply an argument to a function (called β-reduction). Our addition rule reduces an addition expression with integer arguments to a single integer result, while preserving the arithmetic meaning:

$$n_x + n_y \Longrightarrow n_{(x+y)}$$

.

The β-reduction rule substitutes the concrete expression of an argument in place of a function's formal named parameter. Using the names from our previous application examples, this means that every occurrence of x in the body e is replaced by the argument u. The literature often depicts the β-reduction rule as:

$$(\lambda x : \tau.\ e)\ u \Longrightarrow e[u/x]$$

where the square brackets are notation for substitution: the variable on the right of the slash is replaced by the expression to left of the slash.

We have now explored the use of BNF notation to describe the syntax of a simple functional language, and some of the mechanics for evaluating expressions in such a language. However, the keen reader may be feeling uneasy with way we handle function applications in Figure 6.1. It seems that we allow *any* expression to appear on the left of an application but then only provide a reduction rule for when this expression really *is* a function abstraction. How should we evaluate an application of a non-function expression? In fact, we should *not* attempt to evaluate such an ill-formed expression — it simply does not make sense in the semantics of $\lambda^{\rightarrow}$. In statically typed functional languages, we can ensure that a program is well-formed (and does not contain any of these issues) by looking at its type. The following section introduces this process of type checking and its accompanying notation.

6.1.2 *Typing judgements*

Although we have yet to give much consideration to the types present in $\lambda^{\rightarrow}$, Figure 6.1 demonstrates their simplicity. Every valid expression either has the type `Int` or it has a function-type (with potentially different types for its input and output). The type checker's job is to inspect a program written in $\lambda^{\rightarrow}$ and decide whether it represents a well-typed/well-formed expression or not. If it is ill-typed, we know that we should not continue with its evaluation or compilation.

The type checker performs this analysis, again, through a set of typing judgements. Each judgement can have zero or more *premises* which must be met in order apply the typing rule. All premises are written above a horizontal bar. Below the bar is the rule's *conclusion*; the new type we can infer using the rule. All of these typing judgements are made relative to a *context* (written as Γ), which is simply a collection associating each variable currently in scope to their type. Figure 6.2 shows all of the typing rules for a $\lambda^{\rightarrow}$ type checker.

$$\frac{}{\Gamma \vdash n : \texttt{Int}}\ (\text{int}) \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ (\text{var})$$

$$\frac{\Gamma \vdash e : \texttt{Int} \qquad \Gamma \vdash u : \texttt{Int}}{\Gamma \vdash e + u : \texttt{Int}}\ (\text{plus})$$

$$\frac{\Gamma; x : \sigma \vdash e : \tau}{\Gamma \vdash (\lambda x : \sigma.\, e) : \sigma \rightarrow \tau}\ (\text{intro}) \qquad \frac{\Gamma \vdash e : \sigma \rightarrow \tau \qquad \Gamma \vdash u : \sigma}{\Gamma \vdash e\ u : \tau}\ (\text{app})$$

Figure 6.2: Typing judgements for $\lambda^{\rightarrow}$

We will now consider each of these judgements in prose. This aims to encourage the reader's intuition for reading more complex type judgements later in this chapter as well as reinforcing *why* it is convention to use this notation for typing judgements — it is both more concise and more precise than English language descriptions of the same rules.

(`int`) Typing integer literals.
This rule has no premises above the horizontal line, so it can be called upon at any time. The conclusion states that, in any situation, we can be sure that an integer literal (n from Figure 6.1) will have the type `Int`. This precludes the interpretation of integer literals as, for example, function types.

(`var`) Typing variable references.
This rule's one premise is that the context must associate a variable name, x, with the type τ. The conclusion is that a reference to x in the current expression also has type τ. Note that the name x must appear in our context — we cannot infer types for unknown variable names!

(`plus`) Typing addition of expressions.
Our two premises are that, relative to our context, the expressions e and u both have type `Int`. If so, we can conclude that the addition of e and u will also have type `Int`. This precludes us accepting non-integer arguments to our addition primitive, and ensures that the result of an addition is also an integer.

(`intro`) Typing function abstraction.
Our premise demands that the expression e has type τ when our context is extended with a variable x of type σ. We can then conclude that the function with an argument x of type σ and body e has the function type mapping σ to τ. This ensures that function abstraction respects the types of its argument and its output.

(`app`) Typing function application.
If the expression e has a function type mapping σ to τ, and expression u has type σ, we can conclude that the application of u to e has the type τ. Essentially, the application of a function to an argument must respect the function's input and output types.

Each of these rules should hopefully feel intuitive, even if the standard notation was unfamiliar. Perhaps the most subtle part of these typing rules lies in the `intro` rule. The fact that we must extend our context with a binding for x *before* checking the type of a function body e is vital. Consider an expression for the successor function:

$$\lambda x : \texttt{Int}.\ 1 + x$$

We would expect this to have the type $\texttt{Int} \to \texttt{Int}$ — a function accepting an integer input and returning an integer output. The second premise of the `intro` rule would demand that $(1 + x)$ has type `Int`. Figure 6.3 demonstrates that this holds true if we remember to extend the context with a binding for x. If we did not extend the context with our new variable, x, we would not be able to apply the `var` rule appearing at the top right of Figure 6.3.

$$\dfrac{\dfrac{\dfrac{}{\{x : \texttt{Int}\} \vdash 1 : \texttt{Int}}\,(\text{int}) \quad \dfrac{x : \texttt{Int} \in \{x : \texttt{Int}\}}{\{x : \texttt{Int}\} \vdash x : \texttt{Int}}\,(\text{var})}{\{x : \texttt{Int}\} \vdash 1 + x : \texttt{Int}}\,(\text{plus})}{\emptyset \vdash (\lambda x : \texttt{Int}.\ 1 + x) : \texttt{Int} \to \texttt{Int}}\,(\text{intro})$$

Figure 6.3: Typing checking of the expression $\lambda x : \texttt{Int}.\ 1 + x$

Concluding this introductory look at the notations used for functional language syntax and typing judgements, Section 6.2 continues by formalising parts of TinyIdris and introducing some of the nuances present in a dependently typed software language.

6.2 A FORMALISATION OF THE TINYIDRIS LANGUAGE

This section aims to give a formalisation of the TinyIdris software programming language [28]. We leave the discussion of new features and topics specific to hardware description until Section 6.3 in order to maintain a clear split between the new research in this thesis (`toatie`) and the existing project that was reused as a foundation (TinyIdris).

We first describe the full syntax of TinyIdris's surface language ($\texttt{TT}_{\text{imp}}$), continue by looking at the smaller core language to which this elaborates (`TT`), and explore its typing rules.

6.2.1 *A grammar for* $\texttt{TT}_{imp}$

Since TinyIdris is a small (but complete) language, we can feasibly present a full grammar for its surface language, called $\texttt{TT}_{\text{imp}}$. The structures we can encode in $\texttt{TT}_{\text{imp}}$ should be familiar after Chapter 5, since it is a strict subset of our `toatie` language. Figure 6.4 shows an extended BNF description of $\texttt{TT}_{\text{imp}}$ where x and y refer to names/identifiers in a program.

$e,\ u ::= v\ \{\rightarrow w\}$	Full (type) expressions
$v,\ w ::=$	Simple expressions
$\quad$ `Type`	$\quad$ Type of types
$\quad \mid \_$	$\quad$ Implicits
$\quad \mid x$	$\quad$ Names
$\quad \mid b$	$\quad$ Binders
$\quad \mid v\ w$	$\quad$ Applications
$\quad \mid (\ e\ )$	$\quad$ Nested expression
$b ::=$	Binders
$\quad (x\ \{,\ y\}\ :\ e) \rightarrow u$	$\quad$ Π-binders
$\quad \mid \lambda\ a \Rightarrow e$	$\quad$ λ-binders
$\quad \mid$ `pat` $a \Rightarrow e$	$\quad$ Pattern variables
$a ::= x\ [: v]\ \{, a\}$	List of bound variables
$decl ::=$	Declarations
$\quad x : e$	$\quad$ Type declaration
$\quad \mid$ `data` $x : e$ `where` $\{y : u\}$	$\quad$ Data declaration
$\quad \mid \{e = u\}$	$\quad$ Pattern matching function

Figure 6.4: Approximate grammar for TinyIdris

Note that we use curly braces ($\{\ldots\}$) to indicate that the enclosed pattern can appear zero or more times, and square braces ($[\ldots]$) for a pattern which can appear zero or one time. Some future syntax figures also need to denote braces which should appear verbatim instead — we will explicitly quote these literal braces to avoid ambiguity.

Compared to our introductory example, there are many more constructs here and some of variable length. This is largely for two reasons:

1. We are now dissecting a realistic, *surface-level* language. These usually support extra syntactic niceties such as grouping several arguments that share a single type. These are small "syntactic sugar" constructs — enhancing the legibility or expression, but they compile down to the same few features in the corresponding *core* language.
2. We are now considering a more feature-complete language than the simply typed lambda calculus. We now consider extras such as named, global functions defined by pattern matching clauses, GADTs, and dependent function types (Π-binders).

Section 6.2.2 discusses point (1.) by introducing TT, the core language for TinyIdris, and some of the elaboration process which translates $\texttt{TT}_{\text{imp}}$ programs to their corresponding TT representations.

Let's take a moment to compare $\texttt{TT}_{\text{imp}}$'s features to the introductory $\lambda^{\rightarrow}$ language. First of all, as demonstrated extensively during Chapter 5, we have support for user-defined GADTs and defining functions by *pattern matching* on argument constructors over several independent clauses. GADTs are introduced with the `data` construct, and top-level functions are composed of a type declaration followed by their list of pattern matching clauses. $\texttt{TT}_{\text{imp}}$ does not concern itself with the meaning of patterns directly (preferring to leave such analysis for the TT representation).

Dependent types are supported by the Π-binder construct. We no longer have separate constructs for terms and types, since we will allow the two to mingle in a single language. Instead, there is the `Type` construct — the type of types. This particular implementation is actually a known weakness of both TinyIdris and Idris 2 (at the time of writing), since this notion of `Type` allows contradictions such as Girard's paradox within our type system [100]. We would instead prefer to implement an infinite hierarchy of `Types`, claiming this requires engineering effort only and does not fundamentally affect the ideas behind the rest of the implementation.

Finally, $\texttt{TT}_{\text{imp}}$ allows for implicits (written as `_`). These are essentially markers to indicate to the system that a term has been left blank; if the type checker can unambiguously identify the implicit term's value it will be substituted automatically behind the scenes. The system's ability to solve a hole left by an implicit is quite basic here but is substantially extended in both `toatie` and the full Idris 2 implementation. Even automating small, simple implicits is attractive since we often need to supply the type

checker with redundant information — for example, passing explicit type arguments to polymorphic functions or trivially inferable arguments to GADT data constructors.

Now considering the wider compilation pipeline, a TinyIdris program source file is first parsed into a $\texttt{TT}_{\text{imp}}$ representation. The pipeline's next goal is then to elaborate this into a smaller core representation that is easier for us to work with: TT.

6.2.2 *The core language,* TT

TinyIdris' core language, TT, is a refinement of $\texttt{TT}_{\text{imp}}$ which is completely explicit. We demonstrate its syntax in Figure 6.5. We use an overline ($\overline{e}$) as a synonym for indicating zero or many occourances ($\{e\}$). The main features are, in short:

1. No implicit terms. These are initially replaced with metavariables and will be resolved during unification.
2. We represent our pattern matching as case trees — a tree of case statements, each scrutinising a single expression's constructor form [101]. This is both for better software evaluation of our pattern matching and for ease of the implementation of coverage checking (a feature omitted from TinyIdris).
3. All binders are grouped together. This is a small structural choice. Since we quite often treat all binders in the same way when manipulating TT expressions, it is convenient to abstract over them.

Just like our introductory $\lambda^{\rightarrow}$ example, a type checking process over TT expressions can help catch ill-formed programs. In the presence of dependent types, the type checking process can require *evaluation* of expressions since we might need to decide whether or not two expressions are equivalent. The rules followed by the type checker are still rather minimal — captured entirely in Figure 6.6. Although terms and types occupy a single, shared language, we will call (possibly) term-level expressions e/u and type-level expressions S/T by convention. We also provide prose translations of these judgements below:

`ctxt`: We perform our type checking relative to a context, Γ. The first three rules ($\texttt{ctxt}_{\varnothing}$, $\texttt{ctxt}_{\lambda}$, $\texttt{ctxt}_{\Pi}$) describe how valid contexts can be constructed. Both λ and Π-binders can be added to the context as long as their binding type really is a *type*.

`var`: There are two simple rules which are complementary to these context construction judgements; $\texttt{var}_{\lambda}$ and $\texttt{var}_{\Pi}$ for resolving the types of variables bound in the context.

$e, u, S, T ::=$	Terms
x	Variables
$\mid b\ e$	Binders
$\mid e\ u$	Application
$\mid$ `Type`	Type of types
$\mid \_$	Erased term
$\mid ?m$	Metavariable

$b ::=$	Binders
$\lambda x : S\ .$	Lambdas
$\mid \Pi x : S\ \rightarrow$	Functions
$\mid$ `pat` $x : S\ .$	Pattern variables
$\mid$ `pty` $x : S\ .$	Pattern types

$c ::=$	Case trees
`case` $x : S$ `of` $\overline{a}$	Case
$\mid x$	Singleton term
$\mid$ `Unmatched`	Missing valid case
$\mid$ `Impossible`	Impossible case

$a ::=$	Case alternatives
$C\ \overline{x} \Rightarrow c$	Constructor case
$\mid \_$	Default case

$g ::=$	Global definitions
$\lambda \overline{x}.\ c : S$	Pattern matching definition
$\mid$ `Hole` $: S$	Placeholder for a hole term
$\mid D\ : S$	Type constructor
$\mid C\ : S$	Data constructor

Figure 6.5: The syntax for the `TT` language

`conv`: For the non-trivial rules, we introduce a means to *convert* between "equivalent" types. In full, the `conv` judgement states that if a term e has type S, and that S and T are convertible, x also has type T. We will address the details of what makes two terms "convertible" shortly.

`app`: Our `app` judgement looks much like that of the previous $\lambda^{\rightarrow}$ example with one important addition — the return type might *depend* on the applied argument's term. In prose, it says that if e has a function type $\Pi x :\ S \rightarrow T$, and u has type S, we can conclude that the application $e\ u$ has the type $T[u/x]$ (read as T with u substituted for x). This is the most central judgement for introducing dependent types.

`intro`: The remaining two judgements ($\texttt{intro}_{\Pi}$ and $\texttt{intro}_{\lambda}$) both let us introduce new binder abstractions. If S is a type and T is a type (when the context is extended with $\Pi x : S$), we can say that $\Pi x : S \rightarrow T$ is also a type. Likewise, if the function type $\Pi x : S \rightarrow T$ is a valid type, and e has type T (when the context is extended with $\lambda x : S$) then we can say that the lambda abstraction $\lambda x : S.\ e$ has the type $\Pi x : S \rightarrow T$.

$$\frac{}{\Gamma \vdash \underline{\text{valid}}}\ (\text{ctxt}_\emptyset) \qquad \frac{\Gamma \vdash S : \texttt{Type}}{\Gamma; \lambda x : S \vdash \underline{\text{valid}}}\ (\text{ctxt}_\lambda) \qquad \frac{\Gamma \vdash S : \texttt{Type}}{\Gamma; \Pi x : S \vdash \underline{\text{valid}}}\ (\text{ctxt}_\Pi)$$

$$\frac{(\lambda x : S) \in \Gamma}{\Gamma \vdash x : S}\ (\text{var}_\lambda) \qquad \frac{(\Pi x : S) \in \Gamma}{\Gamma \vdash x : S}\ (\text{var}_\Pi)$$

$$\frac{\Gamma \vdash e : S \qquad \Gamma \vdash T : \texttt{Type} \qquad \Gamma \vdash S \simeq T}{\Gamma \vdash e : T}\ (\text{conv})$$

$$\frac{\Gamma \vdash e : (\Pi x : S \to T) \qquad \Gamma \vdash u : S}{\Gamma \vdash e\ u : T[u/x]}\ (\text{app})$$

$$\frac{\Gamma \vdash S : \mathit{Type} \qquad \Gamma; \Pi x : S \vdash T : \texttt{Type}}{\Gamma \vdash (\Pi x : S \to T) : \texttt{Type}}\ (\text{intro}_\Pi)$$

$$\frac{\Gamma \vdash (\Pi x : S \to T) : \texttt{Type} \qquad \Gamma; \lambda x : S \vdash e : T}{\Gamma \vdash (\lambda x : S.\ e) : \Pi x : S \to T}\ (\text{intro}_\lambda)$$

Figure 6.6: Typing judgements for TinyIdris

Although these typing rules are the heart of TinyIdris, their main logic is presented in approximately 90 lines of code (albeit this is only facilitated by plumbing and supporting features over a few thousand lines of code). Beyond the type system, we will now only lightly touch on the surrounding features including the unification of terms and the compilation of pattern matching clauses. While important, these are already well studied in the literature and our hardware description context does not conflict with the existing methods.

The motivation for compiling pattern matching clauses (used extensively in Chapter 5) to case trees is twofold: improved evaluation performance and ease of analysis (e.g. coverage checking). We illustrate the translation between pattern matching and case trees by a standard `zipWith` example for lists in Listing 6.1.

A naive pattern matching approach may consider each clause in turn, selecting a clause only when every one of the patterns on the LHS matches. For the worst-case behaviour of the `zipWith` example, this might require inspecting both list arguments three times. Some of this work is clearly duplicated since we begin the inspection process anew for each clause, discarding the information we've already learned about the arguments while matching previous clauses. TinyIdris instead compiles these pattern matching clauses to a case tree for better performance [101]. This process combines all clauses into a single tree which inspects each argument at most *once* — we now share information learned about the arguments across all clauses. Listing 6.2 shows an equiv-

Listing 6.1: A `zipWith` function via pattern matching

```
zipWith : (a,b,c : Type) → (f : a → b → c) →
          List a → List b → List c
pat a, b, c, f, ys ⇒
  zipWith a b c f (Nil _) ys
    = Nil _
pat a, b, c, f, xs ⇒
  zipWith a b c f xs (Nil _)
    = Nil _
pat a, b, c, f, x, xs, y, ys ⇒
  zipWith a b c f (Cons _ x xs) (Cons _ y ys)
    = Cons _ (f x y) (zipWith _ _ _ f xs ys)
```

alent case tree structure for our `zipWith` example. Note that the case tree only inspects arguments which actually deconstruct a data type (not arguments which only provide a name to a formal parameter) and we examine each of these arguments at most once.

Listing 6.2: A case tree representation for `zipWith`

```
PMDef [[a, b, c, f, xs, ys]] ⇒
case xs : List a of
  { Nil _ ⇒ Nil c
  | Cons _ v vs ⇒ case ys : List b of
    { Nil _ ⇒ Nil c
    | Cons _ w ws ⇒ (Cons c (f v w) (zipWith a b c f vs ws))
    }
  }
```

The presence of dependent types makes the handling of pattern matching more complex. Refining the structure of one argument may affect the *type* of another. This can be handled as in [18] and has consequences for the later topics of impossible cases, coverage checking, and the handling of inaccessible patterns.

Another important feature of TinyIdris is the simple *unification* process, which attempts to reconcile the equality between two terms. This powers the dependent type checking (the `conv` rule in particular) and the completion of implicits in the source listing (e.g. the _ terms in Listing 6.1). A unification algorithm attempts to decide on the convertibility of two terms, giving one of three responses:

Yes: The two terms unify. Informally they have the same normal form (i.e. they "normalise" to the same value). This implies that we need to be able to normalise expressions during type checking. For this end, TinyIdris makes good use of normalisation-by-evaluation and reuses the same code as the REPL's evaluator.

No: The two terms definitely do *not* unify. There is an unresolvable conflict between the two terms.

Maybe: The subtle case where the two terms *might* unify but we don't yet have enough information to know for sure. This is an important and valid option. Here we generate new unification constraints, satisfying the preconditions for the current unification problem. These new constraints might be solvable later.

This topic is well studied (see [17, 18, 102]) and, although it is a vital part of the implementation of many functional languages, we have not identified any aspects of this which are specific to hardware description. As such, we will not linger on the details of TinyIdris' implementation of unification.

Now that we have an appreciation for the structure TinyIdris, the base *software* language for `toatie`, let's consider the novel topics of adapting this for practical hardware description.

6.3 THE toatie CORE LANGUAGE

The alert reader will have noticed that some of the most important concepts and annotations demonstrated throughout Chapter 5 were missing from our discussion of TinyIdris. These constructs are our introductions to the language to support type-safe hardware description. Throughout this section, we will consider these additions as part of four (colour coded) groups:

Erasure: Our approach to ensuring that non-synthesisable data types can be erased from our circuit descriptions *before* circuit run-time. We use this in particular to allow non-synthesisable terms to direct type checking for otherwise synthesisable data constructors.

Staging: We introduce multistage-programming constructs to ensure that the elaboration of a circuit can always complete in full, without knowledge of values presented at a circuit's run-time. This is vital since we must elaborate the circuit to a fixed-structure during compile time — the *structure* of the circuit cannot be influenced by its run-time arguments.

Synthesis: We provide features to support synthesis of circuit descriptions into a form interoperable with downstream EDA tools. This comprises of a few reasonable restrictions on the core language, a synthesis scheme to translate programs in TT to an internal representation of circuit structures, and a VHDL netlist generator.

Sugar: A few quality-of-life features reintroduced from Idris 2 to better support the examples presented in Chapter 5. It is the "sugar" sprinkled on top to make the experience a little more sweet.

We offer a full description of the surface language ($\texttt{TT}^{\mathcal{T}}_{\text{imp}}$) and the syntax of the core language ($\texttt{TT}^{\mathcal{T}}$) upfront and then discuss the implementation details and new typing judgements of each group in turn. As an overview, the additions to $\texttt{TT}^{\mathcal{T}}$ in Figure 6.7 are reasonably modest and of the (colour coded) new feature groups can largely be implemented independently from each other. The additions to the surface language in Figure 6.8 may appear substantial but these are mostly attributable to simple quality-of-life additions which are completely removed by elaboration to $\texttt{TT}^{\mathcal{T}}$ (such as inline case statements and natural number, list, and vector literals).

$e, u, S, T ::=$	Terms
x	Variables
$\mid b_s\ e$	Staged Binders
$\mid e\ u$	Explicit application
$\mid e\ \{u\}$	Implicit application
$\mid \texttt{Type}$	Type of types
$\mid \_$	Erased term
$\mid ?m$	Metavariable
$\mid [\![e\ :\ S]\!]$	Quote
$\mid \langle e \rangle$	Code type
$\mid \sim e$	Escape
$\mid !e$	Evaluate

$b ::=$	Binders
$\lambda x : S\ .$	Explicit λ
$\mid \lambda\{x : S\}\ .$	Implicit λ
$\mid \Pi x : S\ \rightarrow$	Explicit Π
$\mid \Pi\{x : S\}\ \rightarrow$	Implicit Π
$\mid \texttt{pat}\ x : S\ .$	Pattern variables
$\mid \texttt{pty}\ x : S\ .$	Pattern types
$\mid x \mapsto e\ :\ S\ .$	Let bindings

$c ::=$	Case trees
$\texttt{case}\ x : S\ \texttt{of}\ \overline{a}$	Case
$\mid x$	Singleton term
$\mid \texttt{Unmatched}$	Missing valid case
$\mid \texttt{Impossible}$	Impossible case

$a ::=$	Case alternatives
$C\ \overline{x} \Rightarrow c$	Constructor case
$\mid [\![x\ :\ S]\!] \Rightarrow c$	Quote case
$\mid \_$	Default case

$g ::=$	Global definitions
$\lambda\overline{x}.\ c : S$	Pattern matching definition
$\mid \texttt{Hole} : S$	Placeholder for a hole term
$\mid D_p\ : S$	Parameter type constructor
$\mid D_s\ : S$	Simple type constructor
$\mid C\ : S$	Data constructor

Figure 6.7: The syntax for the $\texttt{TT}^{\mathcal{T}}$ language with additions highlighted

$e,\ u ::= v\ \{\rightarrow w\}$	Full (type) expressions
$v,\ w ::=$	Simple expressions
$\quad$ `Type`	$\quad$ Type of types
$\quad \mid \_$	$\quad$ Implicits
$\quad \mid x$	$\quad$ Names
$\quad \mid b$	$\quad$ Binders
$\quad \mid (\ e\ )$	$\quad$ Nested expression
$\quad \mid v\ w$	$\quad$ Standard application
$\quad \mid v\ \text{`\{'}\ w\ \text{`\}'}$	$\quad$ Irrelevant application
$\quad \mid [\![e]\!]$	$\quad$ Quote
$\quad \mid \langle e \rangle$	$\quad$ Code type
$\quad \mid \sim v$	$\quad$ Quote escape
$\quad \mid\ !v$	$\quad$ Quote evaluation
$\quad \mid\ \text{`}[e\ \{, u\}]$	$\quad$ List literal
$\quad \mid\ \ [e\ \{, u\}]$	$\quad$ Vect literal
$\quad \mid$ `n`	$\quad$ Natural number literal
$\quad \mid$ `case` e `of` $\{\ u \Longrightarrow rhs\ \}$	$\quad$ Inline case
$b ::=$	Binders
$\quad (x\ \{,\ y\}\ :\ e) \rightarrow u$	$\quad$ Explicit Π-binders
$\quad \mid\ \text{`\{'}\ x\ \{,\ y\}\ :\ e\ \text{`\}'} \rightarrow u$	$\quad$ Implicit Π-binders
$\quad \mid \lambda\ a \Rightarrow e$	$\quad$ Explicit λ-binders
$\quad \mid \lambda\ \text{`\{'}\ a\ \text{`\}'} \Rightarrow e$	$\quad$ Implicit λ-binders
$\quad \mid$ `let` l `in` e	$\quad$ Let-binders
$\quad \mid$ `pat` $a \Rightarrow e$	$\quad$ Pattern variables
$a ::= x\ [: v]\ \{, a\}$	List of bound variables
$l ::= x\ [: v] = e\ \{, l\}$	List of let-bound variables
$rhs ::=$	RHS of clauses
$\quad e$	$\quad$ Valid clause RHS
$\quad \mid$ `impossible`	$\quad$ Impossible clause RHS
$decl ::=$	Declarations
$\quad x : e$	$\quad$ Type declaration
$\quad \mid$ `data` $x : e$ `where` $\{y : u\}$	$\quad$ Data declaration
$\quad \mid$ `simple` $x : e$ `where` $\{y : u\}$	$\quad$ Synthesisable data declaration
$\quad \mid \{e = rhs\}$	$\quad$ Pattern matching function

Figure 6.8: Approximate grammar for $\mathtt{TT}^{\mathcal{T}}_{\text{imp}}$ with additions highlighted

6.3.1 Sugar from Idris 2

We reintroduce some reasonably straightforward features which were removed during the pruning of Idris 2 down to TinyIdris. These features mostly offer improved ergonomics or expressivity to the programmer but do not have a substantial impact on our ability to capture circuit descriptions in particular.

The most impactful of these reintroductions is simple, non-recursive let-bindings. These allow us to bind an expression to a local variable name inside a global function. We exploit let-bindings as an explicit way to denote *sharing*. An expression let-bound to a local name should be evaluated at most once, but may appear many times in its scope — this explicit sharing is preserved in a circuit description's final netlist.

We include two complementary typing judgements in Figure 6.9 to handle the introduction and elimination of let-bindings.

$$\frac{(\texttt{let}\ x \mapsto e : S) \in \Gamma}{\Gamma \vdash x : S}\ (\text{val}_{\texttt{Let}})$$

$$\frac{\begin{array}{cc}\Gamma \vdash e : S & \Gamma;\texttt{let}\ x \mapsto e : S \vdash u : T \\ \Gamma \vdash S : \mathit{Type} & \Gamma;\texttt{let}\ x \mapsto e : S \vdash T : \texttt{Type}\end{array}}{\Gamma \vdash (\texttt{let}\ x \mapsto e : S\ .u) : T[e/x]}\ (\text{intro}_{\texttt{Let}})$$

Figure 6.9: Type judgements for `let` bindings

The $\text{val}_{\texttt{Let}}$ rule states that if the name x is let-bound to a term e of type S in our context, then the name x also has type S at its point of use. The $\text{intro}_{\texttt{Let}}$ rule is extremely similar to our previous intro_{λ} rule — we may introduce a well-typed binding if its scope (u) is also well typed in our newly extended context.

The second important feature we reintroduce is coverage checking of pattern matching functions. This ensures that *all* possible valid combinations of inputs are handled by a function's pattern matching definition. Although this may seem like a superfluous feature for a proof-of-concept, it actually plays an important role in our reasoning about proofs. A proof must be fully covering in order for us to sensibly hold any confidence in its soundness. A proof which relies on a non-covering function could otherwise type-check but is, by definition, missing logic for some scenarios. Implementing coverage checking becomes more nuanced in the presence of dependent types since refining one argument via pattern matching may give us more information about the type of another argument. This newly refined type may make certain constructors impossible — for example `VCons` is not a valid choice of constructor for type `Vect 0 a`.

The coverage checking is facilitated by two standard features:

↪ The detection of impossible clauses. These clauses contain patterns which contradict each other. The example back in Listing 3.6 provides one simple example of an impossible pattern being detected and safely dismissed. The programmer may omit any impossible clauses or can opt to write them explicitly.

↪ Checking a case tree for missing, possible clauses. This is made a reasonably simple task when we start from the case tree representation. We generate a set of all potentially missing clauses by traversing the case tree node-by-node, noting any unhandled constructors. From this set, we filter our results which are impossible clauses and false positives generated when multiple clauses overlap. If any missing clauses remain we throw a compile-time error.

We augment the behaviour of our pattern matching clauses in Sections 6.3.2 and 6.3.3 when discussing handling of inaccessible patterns.

Another, largely cosmetic, reintroduction is in-line case statements, allowing the programmer to inspect an intermediate local result via simple pattern matching. Note that we do not include the "with" views from Idris 2 for local dependent pattern matching — auxiliary global functions must be used instead. In-line case statements are conspicuously missing from our description of $\texttt{TT}^{\mathcal{T}}$. This is because the elaboration of $\texttt{TT}^{\mathcal{T}}_{\text{imp}}$ identifies case statements and lifts them out to a new global function automatically. From this point, the standard top-level pattern matching machinery is invoked.

Finally, we add special parsing and printing rules for `Nats`, `Lists`, and `Vects` literals. Although just a small feature to improve readability of source and REPL outputs, it may be interesting to see that we rely on implicits which will be solved during unification, even within the compiler itself. For example, the translations of lists and vectors below both leave fields implicit — it would be quite difficult to infer the type of list elements as early as the parsing stage!

```
 Nat:          4 ⟷ S (S (S (S Z)))

List:  `[a,b,c] ⟷ Cons {_} a (Cons {_} b (Cons {_} c (Nil {_})))

Vect:     [a,b] ⟷ VCons {_} {_} a (VCons {_} {_} b (VNil {_}))
```

6.3.2 *Irrelevance and Erasure*

We introduce the motivation for irrelevance and erasure in hardware descriptions by providing an example. Listing 6.3 gives possible definitions for two types that we should hope are synthesisable to hardware descriptions.

Listing 6.3: Possible `Bit` and `Vect` definitions: Are they synthesisable?

```
1 simple Bit : Type where
2   O : Bit
3   I : Bit
4
5 simple Vect : Nat → Type → Type where
6   VNil  : (a : Type)                                  → Vect Z     a
7   VCons : (a : Type) → (k : Nat) → a → Vect k a → Vect (S k) a
```

The `Bit` type is trivially synthesisable (as explored formally in Section 6.4). However, is this `Vect` definition likely to be synthesisable if all of its elements are also synthesisable? Intuitively it feels like we should allow such a definition. However, there are two arguments within these which we will struggle to encode in hardware: the element type `(a : Type)` and the current length `(k : Nat)`. Natural numbers are clearly not synthesisable in general since they describe an infinite set ($0 \rightarrow \infty$). Synthesis of types poses a similar issue.

The insight which allows such definitions to be synthesisable is noticing that the `a` and `k` constructor arguments only exist in order to satisfy the type checker. We do not need their values at circuit run-time. This is to say that `a` and `k` are *relevant* during type checking but *irrelevant* during the circuit's run-time. If our compiler chooses to erase all irrelevant terms from a description before compiling to a netlist, the `Bit` and `Vect` examples can be synthesised using the approach discussed in Section 6.4. While erasure is an important feature for the *performance* of software languages with dependent types, it is absolutely vital for the synthesisability of dependently typed hardware descriptions!

We implement the extensions proposed by the Implicit Calculus of Constructions (ICC*, a more verbose version of ICC) from [62] to track irrelevance of terms and to ensure the program uses irrelevant terms consistently. In $\mathtt{TT}^{\mathcal{T}}_{\text{imp}}$ and $\mathtt{TT}^{\mathcal{T}}$, we introduce *implicit* versions of application, λ binders, and Π binders to highlight irrelevance. This is denoted by curly braces (`{…}`) around the argument term or the bound name.

From [62], we can use an extraction translation to perform the erasure of irrelevant positions from the term-level of an expression. Figure 6.10 recursively defines this translation for only the constructs which directly contain irrelevance annotations.

$$
\begin{array}{lll}
\mathcal{E}[\![x]\!] & = x & \text{(variables)} \\
\mathcal{E}[\![\Pi(x : S) \to T]\!] & = \Pi(x : \mathcal{E}[\![S]\!]) \to \mathcal{E}[\![T]\!] & \text{(Explicit } \Pi\text{)} \\
\mathcal{E}[\![\Pi\{x : S\} \to T]\!] & = \forall(x : \mathcal{E}[\![S]\!]) \to \mathcal{E}[\![T]\!] & \text{(Implicit } \Pi\text{)} \\
\mathcal{E}[\![\lambda(x : S).\, e]\!] & = \lambda(x : \mathcal{E}[\![S]\!]).\, \mathcal{E}[\![e]\!] & \text{(Explicit } \lambda\text{)} \\
\mathcal{E}[\![\lambda\{x : S\}.\, e]\!] & = \mathcal{E}[\![e]\!] & \text{(Implicit } \lambda\text{)} \\
\mathcal{E}[\![e\ u]\!] & = \mathcal{E}[\![e]\!]\ \mathcal{E}[\![u]\!] & \text{(Explicit application)} \\
\mathcal{E}[\![e\ \{u\}]\!] & = \mathcal{E}[\![e]\!] & \text{(Implicit application)}
\end{array}
$$

Figure 6.10: Partial definition of the ICC* extraction translation

The rules for the explicit versions of these constructs leave the expressions unchanged. The implicit application and λ binder rules completely omit their argument and bound name, respectively. This is what provides our *erasure* — we completely remove all term-level expressions marked as irrelevant. The implicit Π binder does retain its bound name after translation, but this is permitted since it is a type-level construct.

The $\mathcal{E}$ translation scheme only applies to well-typed terms, so let us also explore the new typing judgements that ensure we have used irrelevant expressions legally. We split the existing judgements for `app`, $\texttt{intro}_\Pi$, and $\texttt{intro}_\lambda$ into their explicit and implicit forms. All of the judgements for the explicit forms are identical to the originals from Figure 6.6. The implicit rules for application and Π binders behave like their explicit counterparts except that they maintain the implicit annotations. The important rule is in judging the introduction of implicit λ binders ($\texttt{i-intro}_\lambda$). Here we add the restriction that the bound name must not appear as a free variable (discovered by the FV function) in the extraction of the binder's body.

Perhaps a more intuitive way of considering the $x \notin \mathrm{FV}(\mathcal{E}[\![e]\!])$ restriction thus: once a parameter is marked as inaccessible/implicit, it must never reappear in an explicit position. Of course, a parameter marked as implicit may continue to appear in implicit positions, assisting the type checking process. Likewise, explicit parameters may be "demoted" into implicit positions without consequence. The *only* issue is that of lifting an implicit to an explicit — if we have already stated that a term is not required at circuit run-time, it would be hypocritical to then use it at circuit run-time!

One final consequence of implementing irrelevance and erasure is its impact on our ability to perform pattern matching. Previously, we have compiled pattern matching descriptions into case trees and largely left the evaluation of the scrutinee and choice of the appropriate alternative until run-time. However, in the presence of erasure, there will be many patterns which we simply cannot analyse at run-time — erased terms

$$\frac{\Gamma \vdash e : (\Pi x : S \to T) \qquad \Gamma \vdash u : S}{\Gamma \vdash e\ u : T[u/x]}\ \text{(e-app)}$$

$$\frac{\Gamma \vdash e : (\Pi \{x : S\} \to T) \qquad \Gamma \vdash u : S}{\Gamma \vdash e\ \{u\} : T[u/x]}\ \text{(i-app)}$$

$$\frac{\Gamma \vdash S : \mathit{Type} \qquad \Gamma; \Pi x : S \vdash T : \texttt{Type}}{\Gamma \vdash (\Pi x : S \to T) : \texttt{Type}}\ (\text{e-intro}_\Pi)$$

$$\frac{\Gamma \vdash S : \mathit{Type} \qquad \Gamma; \Pi \{x : S\} \vdash T : \texttt{Type}}{\Gamma \vdash (\Pi \{x : S\} \to T) : \texttt{Type}}\ (\text{i-intro}_\Pi)$$

$$\frac{\Gamma \vdash (\Pi x : S \to T) : \texttt{Type} \qquad \Gamma; \lambda x : S \vdash e : T}{\Gamma \vdash (\lambda x : S.\ e) : \Pi x : S \to T}\ (\text{e-intro}_\lambda)$$

$$\frac{\Gamma \vdash (\Pi \{x : S\} \to T) : \texttt{Type} \qquad \Gamma; \lambda \{x : S\} \vdash e : T \qquad x \notin \text{FV}(\mathcal{E}[\![e]\!])}{\Gamma \vdash (\lambda \{x : S\}.\ e) : \Pi \{x : S\} \to T}\ (\text{i-intro}_\lambda)$$

Figure 6.11: New $\texttt{TT}^{\mathcal{T}}$ typing judgements for irrelevance

will already be discarded and are not necessarily recoverable. To complicate the issue further, there are many implementations of reasonable functions which *do* require us to pattern match on irrelevant terms in order to satisfy the type checker. See Listing 3.9 for such an example when implementing the append function for two `Vects`.

It seems we both *need* to allow pattern matching on erased arguments, and *cannot* allow such pattern matching! The solution to this conflict is to distinguish "inaccessible" patterns from typical run-time pattern matching. Inaccessible (or "dot") patterns are those which are presupposed to match — claiming that the given pattern is the only possible matching term [61]. If this is true, we do not have to match against this dot pattern again at run-time (we have statically verified that this is the case already during type checking) and so we can allow even erased terms to appear in the dot pattern. This approach also appears in the MiniAgda project [103] with a similar motive: supporting size indices in a dependently typed language for termination checking, while ensuring these sizes are erased from the generated run-time code.

Given that we already have compilation of pattern matching clauses to case trees and a coverage checker, we can implement dot patterns quite easily for covering functions:

1. Reorder each clause's patterns such that we scrutinise all run-time patterns before any dot patterns.
2. Compile to a case tree representation.
3. Perform coverage checking on the case tree.
4. Traverse the (fully covering) case tree to ensure that all dot pattern scrutinees have exactly one alternative, throwing a compile-time error if we encounter a dot pattern with multiple alternatives.

In `toatie`, all implicit patterns (enclosed by `{…}`) will be interpreted as dot patterns automatically. The programmer is then free to write pattern matching functions with irrelevant arguments without tracking mentally where and when each term is used — `toatie` ensures that previously erased terms are never going to reappear at run-time with this pattern matching implementation and the typing judgements from Figure 6.11.

Although we have chosen to implement irrelevance (and then direct our erasure) using ICC*, there are many alternative methods. One tempting choice is to have a core based Quantitative Type Theory, as in Idris 2. As well as aligning the codebase with Idris 2 more closely, this would gain us *two* features: irrelevance and linearity. We have not considered the use of linearity for circuit description here, although [104] presents some interesting insights on the topic. The Proto-Quipper-D project does make extensive use of linear types, although this is largely to enforce restrictions present in quantum computing which are not necessary for traditional digital circuits.

6.3.3 Staging

Just as erasure contributes to synthesisable circuit descriptions by elimination of non-synthesisable terms, we claim that staging constructs contribute to synthesisability by forcing a distinction between a circuit's (compile-time) elaboration and its run-time behaviour. Since one of `toatie`'s main goals is to support the description and verification of entire circuit *families* at once, it is necessary to support an elaboration process from a parameterised circuit family description to a single concrete member of that family. This is analogous to the elaboration process present in traditional HDLs, especially when VHDL's `generics` are present in the description.

Perhaps the main difference in our approach is that the elaboration process is itself entirely programmable and user-defined! A lack of expressivity in the developer control of elaboration is exactly what drives many digital designers to construct ad hoc

circuit generators in general purpose software programming languages — an option with minimal type safety, discussed and firmly discouraged back in Section 4.6. In contrast to traditional HDLs, our approach is not tied to a few predefined structures such as iterative `for generate` statements, and the designer can use the full `toatie` language to define their own elaboration directed by arbitrarily complex data structures. We have already given a flavour for this style of programmable elaboration in Section 5.2.3 where we transformed a simple dot product circuit to one with an adder tree structure, exploiting higher-order functions and dependently typed folds.

In our case, we use techniques from multistage programming to enforce a strong distinction between the compile-time elaboration stage and the circuit run-time stage. Multistage programming is well studied in the literature, including its theory and implementation for simply typed languages [65, 105] and dependently typed languages [64, 106]. Kiselyov et al. have even considered its use for hardware description [66], albeit in a simply typed setting. We apply Kiselyov's observations on the use of multistage programming for the type-safe elaboration of circuit descriptions to our dependently typed language. This approach serves a number of use cases for circuit description:

↪ As documentation to other programmers:

  - Which parameters must be supplied before a function can elaborate to a synthesisable circuit, or…
  - Which parameters are only present for type checking purposes and can be left unapplied even during synthesis, or…
  - Which parameters are only presented to the circuit at run-time.

↪ Ensuring that elaboration stage can be performed *in full* before the circuit run-time. We are essentially using the type system to reason about the dependencies between these two stages and ensuring they are causal. If elaboration depends on a value only available during the circuit's run-time, we cannot finish elaboration and would be left with a non-synthesisable description.

As an introduction to these issues, let's consider a simple example without the use of `toatie`'s staging constructs. Listing 6.4 defines a function, `rep`, that replicates a given bit n times. The width of this output is directed by the natural number encoding of an unsigned binary word argument.

From the caller's perspective, we are posed with an interesting question: Which arguments must be passed to `rep` to make it synthesisable? Surely we must provide a

Listing 6.4: A problematic example without staging constructs — Ambiguous `rep`

```
1  rep : (w : Nat) → {x, b : Nat} →
2        Unsigned w x → Bit b → Vect x (Bit b)
3  pat bn, b ⇒
4    rep Z {0} {bn} UNil b = VNil {_}
5  pat n, bn, xsn, xs, b ⇒
6    rep (S n) {_} {bn} (UCons {n} {xsn} {0} xs 0) b
7      = append {_} _ _ (rep n {_} {_} xs b)
8                       (rep n {_} {_} xs b)
9  pat n, bn, xsn, xs, b ⇒
10   rep (S n) {_} {bn} (UCons {n} {xsn} {1} xs I) b
11     = VCons {_} {_} b (append {_} _ _ (rep n {_} {_} xs b)
12                                       (rep n {_} {_} xs b))
```

value for the unsynthesisable `Nat` parameter, but what about the `Unsigned` parameter? From the function's type, it seems entirely plausible that we could satisfy the `Unsigned` parameter with a run-time value (only known *after* elaboration). After some careful analysis of the function's implementation, however, we realise that this is not the case! The value of the `Unsigned` parameter directly effects the structure of the circuit (the width of our output vector) so we reason that its value must be known during elaboration — the process responsible for flattening the structure to a known, fixed topology.

This issue for the caller could be simply resolved by allowing the author of `rep` to tag the arguments with an indication of when each value must be supplied. However, we suggest the use of multistage programming to document such intent, as well as solving a deeper challenge on the side of the function's author: Given the intended staging of each parameter, how can we be sure that our elaboration will always be successful?

Strictly typed multistage programming allows us to enforce the causality required for successul elaboration at a type level. To demonstrate this, we present two alternative versions of `rep` using staging constructs: one with a compile-time `Unsigned` argument which does type-check, and one version with a run-time `Unsigned` argument which refuses to type-check since elaboration becomes impossible.

While `repOK` type-checks and synthesises as hoped, the `repBad` version (which explicitly claims that the `Unsigned` argument can be available only at run-time) gives a compile-time error. The staging constructs are enough here to statically know that `repBad` requires pattern matching on a run-time values at elaboration time. This gives the developer more confidence that their entire circuit family should be synthesisable within our language — the burden of this analysis has moved from the developer to our type checker. We can identify quite general misuse of staged terms, not just the dubious pattern matching present in this example. Of course, it is possible to describe

Listing 6.5: Two staged `rep`s which do (left) and do not (right) type-check

```
repOK : (w : Nat) → {x, b : Nat} →
        Unsigned w x →
        ⟨Bit b⟩ → ⟨Vect x (Bit b)⟩
pat bn, b ⇒
  repOK Z {0} {bn} UNil b = ⟦ VNil {_} ⟧
pat n, bn, xsn, xs, b ⇒
  repOK (S n) {_} {bn}
        (UCons {n} {xsn} {0} xs 0) b
    = ⟦ append {_} {_} {_}
          ~(repOK n {_} {_} xs b)
          ~(repOK n {_} {_} xs b) ⟧
pat n, bn, xsn, xs, b ⇒
  repOK (S n) {_} {bn}
        (UCons {n} {xsn} {1} xs I) b
    = ⟦ VCons {_} {_} ~b (append {_} {_} {_}
          ~(repOK n {_} {_} xs b)
          ~(repOK n {_} {_} xs b)) ⟧
```

```
⟩ OK!
```

```
repBad : (w : Nat) → {x, b : Nat} →
         ⟨Unsigned w x⟩ →
         ⟨Bit b⟩ → ⟨Vect x (Bit b)⟩
pat bn, b ⇒
  repBad Z {0} {bn} ⟦ UNil ⟧ b = ⟦ VNil {_} ⟧
pat n, bn, xsn, xs, b ⇒
  repBad (S n) {_} {bn}
         ⟦ UCons {n} {xsn} {0} xs 0 ⟧ b
    = ⟦ append {_} {_} {_}
          ~(repBad n {_} {_} ⟦ xs ⟧ b)
          ~(repBad n {_} {_} ⟦ xs ⟧ b) ⟧
pat n, bn, xsn, xs, b ⇒
  repBad (S n) {_} {bn}
         ⟦ UCons {n} {xsn} {1} xs I ⟧ b
    = ⟦ VCons {_} {_} ~b (append {_} {_} {_}
          ~(repBad n {_} {_} ⟦ xs ⟧ b)
          ~(repBad n {_} {_} ⟦ xs ⟧ b)) ⟧
```

```
⟩ Case tree requires ambiguous pattern-
  match on quoted arg, {arg:3}
```

a circuit family quite similar to the `rep` function which operates entirely at run-time but we must satisfy the type checker that we have accounted entirely for the different possible structures at run-time.

Given these examples as motivation, let's look at how we implement the type checking under our explicit staging constructs. First of all, we introduce new syntax in $\mathtt{TT}^{\mathcal{T}}_{\text{imp}}$ for four staging constructs inspired by [64]:

↪ `⟦ ... ⟧` A quote deferring the evaluation of a given term until circuit run-time.

↪ `⟨...⟩` A `Code` type encoding a static type for a term available only at circuit run-time.

↪ `~(...)` An escape, which splices a quoted term into a lower (but still circuit run-time) level. This construct helps us flatten valid recursive definitions of circuits within our staging restrictions.

↪ `!(...)` An evaluation, forcing the reduction of a *closed* circuit run-time term. A seldom required construct, similar to an escape but specifically for splicing a quoted term into stage 0 if it contains no free variables.

Next, $\mathtt{TT}^{\mathcal{T}}$ is extended by forcing each application and binder to track the stage at which it appears — stage 0 for any elaboration-time code, and a positive stage for any circuit run-time expressions. This tagging is performed entirely automatically during the translation from $\mathtt{TT}^{\mathcal{T}}_{\text{imp}}$ to $\mathtt{TT}^{\mathcal{T}}$. The current stage is tracked as a part of our translation, being incremented whenever we enter a quote or a code type, and being decremented whenever we encounter an escape or evaluation.

With this information gleaned directly from the source listing, we can have the type checker ensure that we use all bound variable names and functions in a way which respects their staging. Figure 6.12 lists the augmented typing rules for our staging constructs. The `quote`, `code type`, `eval`, and `escape` rules determine how we can safely lift between adjacent stages. The $\texttt{var}_\lambda$, $\texttt{var}_\Pi$, and $\texttt{val}_{\texttt{let}}$ rules check any bound names at their point of use for staging consistency. The $n \leq m$ prerequisite in these rules ensures that run-time names (from a nonzero stage) can never appear in an elaboration-time expressions (in a zero stage), hence preserving the causality between elaboration and circuit run-time.

$$\frac{\Gamma \vdash_{n+1} e : S}{\Gamma \vdash_n [\![e]\!] : \langle S \rangle}\ (\text{quote}) \qquad \frac{\Gamma \vdash_n S : \texttt{Type}}{\Gamma \vdash_n \langle S \rangle : \texttt{Type}}\ (\text{code type})$$

$$\frac{\Gamma \vdash_0 e : \langle S \rangle \qquad FV(e) = \emptyset}{\Gamma \vdash_0 !e : S}\ (\text{eval}) \qquad \frac{\Gamma \vdash_n e : \langle S \rangle}{\Gamma \vdash_{n+1} \sim e : S}\ (\text{escape})$$

$$\frac{(\lambda x :_n S) \in \Gamma \qquad n \leq m}{\Gamma_m \vdash x : S}\ (\text{var}_\lambda) \qquad \frac{(\Pi x :_n S) \in \Gamma \qquad n \leq m}{\Gamma_m \vdash x : S}\ (\text{var}_\Pi)$$

$$\frac{(x \mapsto e :_n S) \in \Gamma \qquad n \leq m}{\Gamma_m \vdash x : S}\ (\text{val}_{\text{Let}})$$

Figure 6.12: Main type judgements for our staging constructs

There are a few additional rules we must add to allow proper unification between staged terms, again, as per [64]. These are shown in Figure 6.13. These rules encode the convertibility of a given term and the escape/evaluation of a quoted version of the original term ($\texttt{conv}_{\texttt{quote}}$ and $\texttt{conv}_{\texttt{escape}}$), as well as the equivalence between the quotations of two convertible terms ($\texttt{conv}_{\texttt{quote}}$).

$$\frac{\Gamma \vdash e : S}{\Gamma \vdash ![\![e]\!] \simeq e}\ (\text{conv}_{\text{quote}}) \qquad \frac{\Gamma \vdash e : S}{\Gamma \vdash \sim[\![e]\!] \simeq e}\ (\text{conv}_{\text{escape}})$$

$$\frac{\Gamma \vdash e, u : S \qquad \Gamma \vdash e \simeq u}{\Gamma \vdash [\![e]\!] \simeq [\![u]\!]}\ (\text{conv}_{\text{quote}})$$

Figure 6.13: `toatie`'s extra conversion rules for staging

Our final special handling of staged terms is that of pattern matching. Exactly like our handling of irrelevant terms in Section 6.3.2, it would be problematic to allow pattern matching of quoted terms in general. In much the same way, we also often *require* some degree of refinement on quoted arguments. We solve this by applying the same

machinery for inaccessible patterns — we may refine circuit run-time patterns if and only if they are unambiguously identified by their surrounding accessible patterns.

For the examples presented in Chapter 5, it is likely possible to implement the staging purely as a type annotation (omitting the term-level quotes and escapes). While this would improve the brevity of the circuit descriptions, `toatie` keeps the quotes and escapes explicit for clarity and allowing for future work which relies on a true multi-stage system (such as *reconfigurable* circuits).

Although we employ staging constructs to hardware description for very different reasons than we did erasure, it remains an interesting insight that both appear as sensible *optimisations* in the software world, but absolute *necessities* in the hardware world.

6.4 CIRCUIT SYNTHESIS

This section details the important step of transforming a type checked program in $\mathtt{TT}^{\mathcal{T}}$ down to a circuit netlist. We do this in three main steps:

↪ Automatically synthesising bit representations for user-defined data types.

↪ Normalising a top-level circuit description into a form trivially translated to a circuit netlist.

↪ Generating a netlist acceptable by downstream EDA tools, such as Xilinx's Vivado, for an FPGA target.

This section of `toatie` is perhaps the most idealised — we implement enough to support all of the examples in Chapter 5, but acknowledge its limitations in this section. These limitations are only present for our proof-of-concept and we address them with reference to relevant literature to demonstrate how they may be effectively handled without restructuring our core language. Although many real-world examples which do successfully synthesise are presented, the metatheory of the proposed approach is deferred for future study.

As with any HDL, we must place some reasonable restrictions on our top-level descriptions if they are to be synthesisable. We detail these restrictions in Section 6.4.1. The rest of this section details how we choose to synthesise the resulting circuit descriptions into a VHDL netlist.

6.4.1 Restrictions for synthesisability

In much the same way as VHDL's requirement that all `generic` arguments are supplied at the top-level, `toatie` requires that all elaboration-time parameters are supplied at the top-level. There are, however, some extra nuances to these restrictions — some more permissive, some more restrictive, and some purely due to the expressivity of our staging annotations. A synthesisable top-level circuit must be:

1. In a form whose *extraction* is a single code type. This code type may, itself, be a function type describing the inputs and outputs of the circuit. This restriction disallows unapplied, explicit arguments to the top-level circuit (similar to undefined generics for our circuit family) but does permit unapplied irrelevant arguments (useful only for the type checking of the circuit).
2. Recursion is allowed but only during elaboration (at stage zero or during escaping of other staged terms). Recursion at run-time describes a possibly dynamic structure and cannot be synthesised. We have already provided the staging constructs required to force bounded, recursive elaboration-time structures to be flattened before synthesis.
3. The argument types and return type of our extracted circuit should all be synthesisable. We distinguish between non-synthesisable types (`parameter` types) and synthesisable ones (`simple` types). As well as being subject to extra checks, terms of a `simple` type are not allowed to influence the dependent typing of any other term. In other words, a run-time value cannot influence the shape (or type) of another run-time value.

These three rules should result in a circuit description whose extraction is (at least at its top-level) simply-typed, first-order, and monomorphic. Section 6.4.2 details the requirements that a `simple` type must meet in order to be synthesisable.

At this point, we also reckon with one outstanding issue in our circuit descriptions. We only offer a best-effort support for removing intermediate non-synthesisable terms in our normalisation-by-evaluation system. We opt for an untyped normalisation stage and the removal of non-synthesisable terms cannot be completely guaranteed. This is a choice for our proof of concept, but CλaSH in [19] details how to normalise a circuit description (with a typed directed term rewrite system) from similar language, System FC. They also provide proofs that their approach will *always* remove non-representable terms. This property is perhaps more important for CλaSH since they do not entertain the native staging constructs that we lean on so heavily to reason about elaboration.

6.4.2 *Simple types, parameter types, and bit representations*

Inspired by Proto-Quipper-D in [58], we introduce two different kinds of data to distinguish between synthesisable terms and non-synthesisable terms. Borrowing Proto-Quipper-D's terminology, we call these `simple` and `parameter` types, respectively, and they exist in the same namespace.

As a rule of thumb, `parameter` types best describe elaboration-time terms, while `simple` types capture run-time values. To permit dependent types in Proto-Quipper-D descriptions, the interpretation of dependent typing is changed for `simple` types. Just like our application to digital circuits, they cannot allow the shape of a circuit to be directed by a run-time value. Proto-Quipper-D's novel solution to this is, instead, allowing types to only depend on the *shape* of a `simple` value. The "shape" is a property known at elaboration time. For our purposes, we have not encountered any scenarios which demand this allowance for shape-dependence, instead having a single `parameter` value direct the shape of multiple `simple` types. To simplify the implementation we instead choose to completely exclude terms of a `simple` type from influencing other types in a Π-binding.

Our proof-of-concept attempts to handle *all* synthesisable data types in a generic fashion. While there are clear motivations for special handling of certain primitives in FPGA targets (such as explicit use of their specialised resources for arithmetic and block memories), we follow the approach justified in "Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine":

> *Compilers sometimes implement the built-in types (list, tuples, numbers) in special magic" ways, and the programmer pays a performance penalty for user-defined types. We take the view that the general mechanisms used for user-defined types should be made efficient enough to use for built-in types too.*
>
> — SIMON PEYTON JONES IN [107]

For every `simple` type, we attempt to synthesise it to a fixed width bit representation by using a modest set of new recursive rules and the unification machinery already required for an implementation of a dependently typed language.

To begin, let's consider the most fundamental synthesisable type: `Bit`. This example is simple since each constructor has zero arguments. In general, we can intuit that there are two possible values a `Bit n` can occupy — either a `0` or `1`. Since there are two options, we only need a single bit in our netlist to represent the value. This process is a

matter of quizzing the unification engine as to the number of constructors which could possibly match a given type. We can then represent this constructor tag with $\lceil \log_2 N \rceil$ bits per N constructors.

Listing 6.6: A synthesisable definition of the `Bit` type

```
1    simple Bit : Nat → Type where
2    0 : Bit 0
3    I : Bit 1
```

There are, however, some surprising nuances in the synthesis of such a description. In a situation where we are certain the type is `Bit 0`, there is actually only one possible constructor — it is a constant! In this case, we can reduce the term to a unit. In a sense, we do not propagate unique constants through the netlist, but rather specialise the rest of the circuit on this constant value. This is usually an advantage of our approach, but care must be taken for the top-level interface when exporting circuits for integration with other tools — a constant optimised away will impact our bit encoding. For this reason we recommend all external interfaces use simply typed arguments, leaving all implicit optimisations internal to the `toatie` description.

A slightly more involved example requires constructors with non-zero arities. The most intuitive of which is a synthesisable vector, as defined in Listing 6.7.

Listing 6.7: A definition of a (potentially) synthesisable `Vect` type

```
1 simple Vect : Nat → Type → Type where
2   VNil  : {a : Type}                                → Vect Z     a
3   VCons : {a : Type} → {k : Nat} → a → Vect k a → Vect (S k) a
```

This encounter also prompts us to reckon with polymorphism. Given a length `n`, and assuming the type `a` is itself synthesisable, we can synthesise a `Vect n a`. We can use the same analysis of constructor tags as in our `Bit` example. However, we must also analyse the arguments to each constructor. We can describe this process quite concisely given a helper function $PC(D\ \overline{x})$ which uses the unification engine to find all of the possible data constructors (with their specialised types) which could have possibly returned the type $D\ \overline{x}$. The process is then captured by two rules: one to translate applied type constructors, and one to translate the specialised data constructors:

$$
\begin{aligned}
&\mathcal{B}_{\mathrm{D}}[\![D\ \overline{x}]\!] = && \text{Translation for ty cons}\\
&\qquad \lceil \log_2 |PC(D\ \overline{x})| \rceil + && \text{(Tag bits)}\\
&\qquad \max\{\mathcal{B}_{\mathrm{C}}[\![C_i\ \overline{y_i}]\!] \mid \forall\ C_i\ \overline{y_i} \in PC(D\ \overline{x})\} && \text{(Widest possible data con)}
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{B}_{\mathrm{C}}[\![C\ \overline{y\ :\ T}]\!] = && \text{Translation for data cons}\\
&\qquad \sum_{e\ :\ S}^{\overline{y\ :\ T}} \mathcal{B}_{\mathrm{D}}[\![S]\!] && \text{(Sum of all data con arg types)}
\end{aligned}
$$

In essence, each type synthesises down to a set of tag bits (to uniquely identify which data constructor was used) and a set of field bits (which encode each of the tagged data constructor's non-irrelevant arguments). The data constructor arguments are encoded recursively, each having their own tag and field bits. There is also some padding required to ensure that field widths of *all* data constructors are equal.

While this definition is quite similar to CλaSH, we differentiate the approach by letting our definitions contain primitive recursion, which is unrolled by *PC* and the unification engine. To be able to satisfy our requirement for primitive recursion, every `simple` type must conform to the following restrictions:

1. Every non-irrelevant argument for every data constructor is also synthesisable. (Irrelevant arguments *can* be non-synthesisable.) This ensures that, in our polymorphic `Vect` definition, the element type (`a`) must represent a synthesisable type.
2. A recursively defined type is permissible if there is primitive recursion on (at least) one of the type constructor arguments:
   - ↪ There must be a terminating case on this recursive position.
   - ↪ Every other constructor must have their recursion be *structurally decreasing* in this position.
3. The type constructor must be applied enough to make 1) and 2) identifiable solely through unification of the top-level problem — i.e. we must have compile-time knowledge of any polymorphic type arguments and indices which direct recursion before attempting synthesis.

To demonstrate this process more concretely, let's perform this analysis by hand for a type of `Vect 2 (Bit b)`, where b is unknown. Figure 6.14 shows a graph for this analysis where all recursive definitions are shown with pink boxes/purple arrows and all terminal definitions are shown with blue boxes/green arrows. A vector of two bits is, and clearly should be, represented by two bits only. Every constructor of the `Vect` type is uniquely identified by its length and incurs zero tag bits. Each element is `Bit b` and incurs one tag bit. Notice that all irrelevant arguments are ignored for bit representation purposes, since they should not appear in the synthesised netlist.

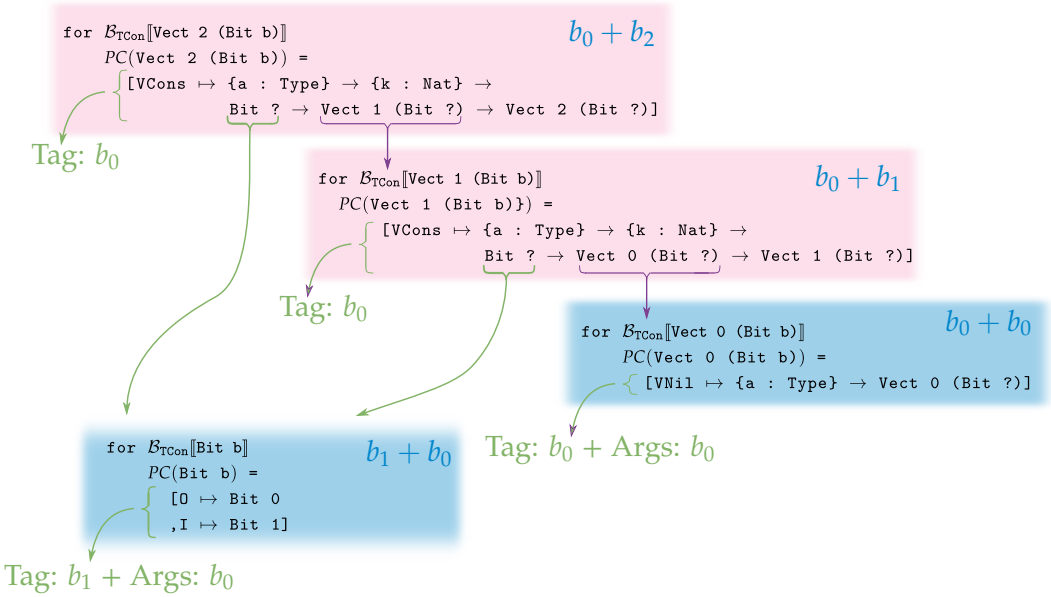

Figure 6.14: Steps for bit representation of `Vect 2 (Bit b)`

We can directly synthesise types featuring sums of products or primitive recursion using our general transform rules. For example, we can entirely natively synthesise the binary trees, `Maybe` types for encoding data with valid flags, and vectors with multiple different `Cons` options (e.g. `RCons`, `GCons`, and `BCons` for colour vectors) as defined in Listing 6.8.

While our approach to automatically deriving bit representations has similarities to Proto-Quipper-D [59] (namely generation in the presence of irrelevant arguments), there are three clear additions. These are, in part, encouraged by the control structures possible in digital circuits, rather than the Quantum circuit descriptions of [59]. These additions are:

↪ We separate the "tag" bits from the "field" bits. With these explicit tag bits, we can synthesise `toatie`'s choice constructs (such as case expressions) directly with user-defined data types.

Listing 6.8: Examples of synthesisable data types

```
simple BTree : Type → Nat → Type where
  Leaf : {a : Type} → BTree a 0
  Node : {a : Type} → {n : Nat} →
         a → BTree a n → BTree a n → BTree a (S n)

simple Maybe : Type → Type where
  Nothing : {a : Type}      → Maybe a
  Just    : {a : Type} → a → Maybe a

simple RGBVect : Nat → Type → Type where
  RGBNil : {a : Type} → RGBVect Z a
  RCons  : {a : Type} → {k : Nat} →
           a → RGBVect k a → RGBVect (S k) a
  GCons  : {a : Type} → {k : Nat} →
           a → RGBVect k a → RGBVect (S k) a
  BCons  : {a : Type} → {k : Nat} →
           a → RGBVect k a → RGBVect (S k) a
```

↪ We loosen their restrictions on synthesisable types. We can synthesise recursive types with ambiguous constructors (where more than one constructor is possible), facilitated by our tag bits. This is demonstrated in the `RGBVect` example in Listing 6.8.

↪ We synthesise directly from the standard GADT syntax instead of a reversed, function style.

Although more restrictive than theoretically necessary, our choice to support only primitive recursion in `simple` types was proven sufficient to capture all of the examples in Chapter 5.

6.4.3 *Normalisation*

We introduce one final intermediate representation before we attempt to generate a netlist, called $\texttt{CExp}^{\mathcal{T}}$. This normalised form can be thought of as a lily pad on which we hop between the $\texttt{TT}^{\mathcal{T}}$ and netlist representations. Here, our entire goal is to dramatically simplify a $\texttt{TT}^{\mathcal{T}}$ program into a more restricted form which is directly amenable to circuit generation.

We aim to transform any $\texttt{TT}^{\mathcal{T}}$ program (which satisfied the restrictions from Section 6.4.1) into a normalised form with the syntax shown in Figure 6.15.

τ, σ ::=	Types
$\quad D_{\mathtt{s}}\ \overline{\tau}$	$\quad$ Fully applied simple type
a ::=	Argument expressions
$\quad x$	$\quad$ Local variable
$\quad \mid C_{\mathtt{s}}\ \overline{a}$	$\quad$ Fully applied simple data constructor
e ::=	Subexpressions
$\quad x$	$\quad$ Local variable
$\quad \mid C_s\ \overline{a}$	$\quad$ Fully applied simple data constructor
$\quad \mid \mathtt{case}\ x\ \mathtt{of}\ \overline{alt}\ [\mathtt{default}\ a]$	$\quad$ Case with optional default
$\quad \mid \pi_i^C x$	$\quad$ Projection
$alt ::= C_{\mathtt{s}}\ \overline{x} \rightarrow a$	Alternatives
$g ::= \lambda\ \overline{x : \tau}\ .\ \mathtt{let}\ \overline{y : \sigma \mapsto e}\ \mathtt{in}\ z$	Top-level circuit

Figure 6.15: The syntax for $\mathtt{CExp}^{\mathcal{T}}$ in normal form

This represents a version of a top-level circuit with several restrictions. Our circuit's internals are just a list of let-bound subexpressions and the circuit's output is a reference to one of these let-bindings. Each let-bound subexpression and circuit input is given a name and tagged with its type. All types at this point ought to be a fully applied `simple` type since `parameter` types are not synthesisable and we aim to have eliminated any higher-order constructs. The type annotations exist only so we can reason about the bit representation of a term during netlist generation.

Each subexpression can be only one of four, easily synthesisable, forms:

↪ A local variable, referencing one of the previous let-bound names.

↪ A `simple` data constructor fully applied with all of its arguments. Note that each argument must be a local variable or a nested constructor application.

↪ A case statement scrutinising a local variable. A default branch is optional.

↪ A *projection* returning the i^{th} argument from a fully applied data constructor, C. The target of the projection is specified as a local variable.

Each of these four constructs have a clear translation to a circuit netlist, as described in Section 6.4.4. A more mature version of `toatie` may wish to extend $\mathtt{CExp}^{\mathcal{T}}$ with primitives which map to specialised FPGA resources (such as DSP blocks and Block RAMs), as well as a registering construct for synchronous signals. Indeed, CλaSH's

normal form does make these additions, as defined in Section 4.2.2 of [19]. Beyond this, the independently designed $\texttt{CExp}^{\mathcal{T}}$ normal form is nearly a subset of CλaSH's. We omit function applications in favour of exhaustive inlining, and omit primitive operations in favour of handling everything via our support for user-defined data types. The only addition in $\texttt{CExp}^{\mathcal{T}}$ is that we choose to allow nested constructor applications — waiting until netlist generation to flatten them out into a single bit vector.

Although the *surface* languages of `toatie` and CλaSH are really quite distinct, they do both settle on a extremely similar normal forms just before netlist generation. This is, perhaps, a good indication that both have captured the fundamental essence of functional hardware description.

Now that we have defined the target of our normalisation scheme, let's explore the translations from $\texttt{TT}^{\mathcal{T}}$ down to our $\texttt{CExp}^{\mathcal{T}}$ normal form. When possible, we follow the lead of the Idris 2 implementation (`CExp`) but there are major differences between the goals for Idris 2's software code generation and our netlist generation.

The broad process to convert $\texttt{TT}^{\mathcal{T}}$ to $\texttt{CExp}^{\mathcal{T}}$ is:

1. Translate to a new representation with:
   - ↪ Saturated applications and constructors, via η-abstraction for partial applications in $\texttt{TT}^{\mathcal{T}}$.
   - ↪ Inline case statements instead of top-level pattern matching. This allows us more opportunity to reorder expressions during the rest of the process.
   - ↪ All irrelevant terms erased. We are moving towards a synthesisable description; these must be removed.
   - ↪ We (temporarily) erase all types, except in function definitions.
2. Evaluate & inline the description as far as possible. Here, the circuit gets specialised by any parameters passed to the circuit family. Any reducible expressions inside the quoted circuit run-time are also simplified. Every function call is inlined. We expect termination here for total descriptions since the (non-irrelevant) arguments for the elaboration stage should be fully applied and recursion on run-time values is forbidden.
3. Lift any complex subexpressions out to their own let-bound name, as in Figure 6.15. Any unused let-bindings are erased. Each subexpression is either a local variable, saturated `simple` data constructor, a case expression scrutinising a local variable, or a projection.

4. We annotate each let-binding with its type. Since all internal typing information was erased in step 1, we attempt to regenerate this information via an annotation pass. While this discard/regenerate process is overly restrictive, it is strong enough to synthesise all examples in Chapter 5.

Since the goal of the normalisation process is to end up with a single top-level expression as defined in Figure 6.15, we need to ensure that all other constructs are eliminated. Noting that we erase all irrelevant terms (with the extraction translation from Section 6.3.2) and discard all typing annotations, we identify the main remaining constructs for elimination by comparing $\texttt{CExp}^{\mathcal{T}}$ and $\texttt{TT}^{\mathcal{T}}$:

1. Staging constructs
2. λ-abstraction and applications
3. Terms using `parameter` types

The four staging constructs prove easy to dismiss. They are useful during type checking to ensure that our use of variables is causal between stages, and to ensure our top-level circuit has all elaboration-time arguments fully applied. However, beyond these checks, we can simply erase the staging annotations with full confidence the elaboration can complete in full — the type checker has already told us so! None of our normalisation process is directed by the staging annotations.

The elimination of λ-abstractions and applications is also straightforward using a normalisation-by-evaluation approach. Since no recursion is directed by run-time free variables and all applications are saturated by this point, the normalisation scheme can easily inline all functions and specialise their bodies as fully as possible as it does so. This leaves no λ-abstractions and partially applied functions.

Finally, can we ensure that all terms with `parameter` types are eliminated? Our starting point is our top-level circuit's extracted code type, which is a function between only `simple` types. Within this expression, we might encounter intermediate values with `parameter` types. These might then influence branching constructs via case expressions. CλaSH provides a set of normalisation rules specific to their non-representable types (or `parameter` types) which can provably eliminate them from the interior of an otherwise synthesisable description. This includes substituting let-bound `parameter` types into the scope of the let-binding, and a classic "case of case" optimisation (also present in Idris 2) to eliminate structures which scrutinise `parameter`

types. Our implementation cannot provide these same guarantees because our proof-of-concept normalisation process is largely untyped for simplicity. CλaSH uses its typing information to direct which normalisation rules should be applied but `toatie` does not retain that information throughout the translation. Some of these intermediate `parameter` types might be removed through our normalisation-by-evaluation process, but this is not guaranteed. A more mature implementation of `toatie` should include a rewritten normalisation process which carries our typing information throughout (without needing to change $\texttt{TT}^{\mathcal{T}}$ or $\texttt{CExp}^{\mathcal{T}}$). We believe that this is not precluded by any of the other differences between `toatie` and CλaSH.

The erasure of type annotations during `toatie`'s normalisation stage is unfortunate for one other reason too. The Glasgow Haskell Compiler (GHC) offers good evidence for the benefits of maintaining types throughout a compilation pipeline [108]. Although it burdens the developers with extra work, it serves as an excellent sanity check for any optimisations implemented. Regardless, the simplicity of the current normalisation process in `toatie` gives confidence towards an important property — if normalisation *does* complete, we can largely trust the netlist to be correct.

6.4.4 Netlist generation

Generating a netlist interoperable with vendor EDA tools from $\texttt{CExp}^{\mathcal{T}}$ is a trivial translation. Following the lead from [19], we present our relatively simple synthesis rules visually in Figure 6.16. Our $\texttt{CExp}^{\mathcal{T}}$ starting point is a single entity whose inputs and output are defined by its type. The body of the circuit contains named signals translated from each of the let-bound subexpressions. The circuit output is driven by one of these intermediate signals. Following this, we have very few constructs left to handle:

- ↪ Local variable references are simply a wire, connected to the variable's signal — achieved purely in routing.
- ↪ Constructor applications are concatenations of other signals into a single named signal. The tag for this application is driven by a constant.
- ↪ Case expressions synthesise down to a set of alternatives (which can be safely evaluated simultaneously since they are *pure* functions), followed by a multiplexer to select the correct output.
- ↪ Projection is simply a splice, addressing only part of another named signal.

While the graphical representation of the scheme should help nurture an intuition, we also present a more formal version. Our implementation translates to VHDL, pro-

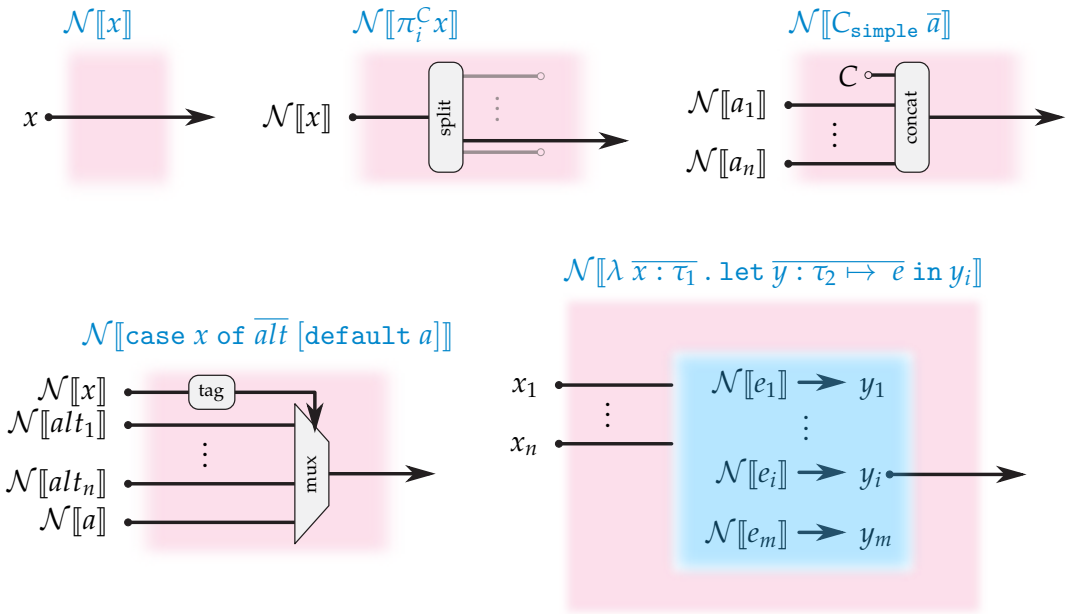

Figure 6.16: Graphical translation from $\texttt{CExp}^{\mathcal{T}}$ to a circuit structure

viding integration with most downstream EDA tools. Since we currently only make use of simple splices, concatenations, and multiplexers, it should be a simple exercise to implement back-ends for Verilog, among others. To concisely describe the VHDL implementation, we first introduce a set of helper functions to reason about the bit representations of a type.

$\texttt{type}(e)$: Get the type of a term

$\texttt{tagI}(\tau)$: Get the indices of the tag bits for a given type

$\texttt{tagB}(\tau, C)$: Get the tag bits for a given constructor

$\texttt{fieldI}(\tau, C, i)$: Get the indices for a given field/argument of a data type

$\texttt{padB}(\tau, C)$: Get the number of padding bits for a given constructor (padding out the representation of this particular data constructor up to the worst-case size for its type)

An entity declaration is generated with ports for all top-level inputs and a single output, called `res`. The architecture is equipped with signals defined for all let-bound names. From this point, we can generate the architecture's body with the rules presented in Figure 6.17. The angle bracket notation is used to indicate indexing of a bus and an ampersand represents bus concatenation:

$$
\begin{array}{lll}
\mathcal{N}[\![\lambda\ \overline{x : \tau}\ .\ \texttt{let}\ \overline{y : \sigma \mapsto e}\ \texttt{in}\ z]\!] & & \text{Top-level circuit} \\
\quad = \overline{y\ \leftarrow\ \mathcal{N}[\![e]\!]} & & \text{(Let-bound signals)} \\
\quad\quad \texttt{res}\ \leftarrow\ z & & \text{(Output routing)} \\
\\
\mathcal{N}[\![x]\!] & & \text{Local variable reference} \\
\quad = x & & \\
\\
\mathcal{N}[\![\pi_i^C\ x]\!] & & \text{Projection of field} \\
\quad = \mathcal{N}[\![x]\!]\ \langle\ \texttt{fieldI}(\texttt{type}(x),\ C,\ i)\ \rangle & & \text{(Indexing of } x\text{)} \\
\\
\mathcal{N}[\![C_{\mathtt{s}}\ \overline{a}]\!] & & \text{Constructor application} \\
\quad = \texttt{tagB}(\texttt{type}(C_{\mathtt{s}}\ \overline{a}),\ C) & & \text{(Constructor tag bits...)} \\
\quad\quad \&\ \overline{\mathcal{N}[\![a]\!]} & & \text{(with field bits...)} \\
\quad\quad \&\ \texttt{padB}(\texttt{type}(C_{\mathtt{s}}\ \overline{a}),\ C) & & \text{(and padding bits)} \\
\\
\mathcal{N}[\![\texttt{case}\ x\ \texttt{of}\ \overline{C_{\mathtt{s}}\ \overline{y} \rightarrow e}\ [\texttt{default}\ a]]\!] & & \text{Case expression} \\
\quad = \texttt{with}\ \mathcal{N}[\![x]\!]\ \langle\ \texttt{tagI}(\texttt{type}(x))\ \rangle\ \texttt{select} & & \text{(Scrutinise tag)} \\
\quad\quad \leftarrow \overline{\mathcal{N}[\![e]\!]\ \texttt{when}\ \texttt{tagB}(\texttt{type}(y),\ C)} & & \text{(Alternatives)} \\
\quad\quad\quad [\mathcal{N}[\![a]\!]\ \texttt{when others}]; & & \text{(Default)}
\end{array}
$$

Figure 6.17: Translation scheme from normalised $\texttt{CExp}^{\mathcal{T}}$ to a VHDL architecture body

The only rule which substantially leans on VHDL-specific constructs is the multiplexing of case expressions, using VHDL's `with/select` assignment. This construct offers a convenient way to express explicit choices for multiplexing as well as handling a default case. This maps very neatly to our core language since case expressions can have a non-exhaustive list of alternatives, followed by a default branch. For an implementation which outputs a netlist more directly (for example, an EDIF file) the burden of handling default cases would likely lie on `toatie`'s netlist generator, rather than on the downstream EDA tools such as Vivado.

The only other nuance to mention is that our scheme in Figure 6.17 allows *nested* constructor applications. Each argument to a constructor application can itself be a constructor application. This helps minimise the number of internal signals we need to define, especially when constructing larger structures.

6.4.5 *Synthesis examples*

To illustrate the synthesis process from start to finish, this section steps through a series of simple examples. We will make a point of discussing *why* the source description meets the synthesis requirements from Section 6.4.1, and the "*what*" behind its normalised $\texttt{CExp}^{\mathcal{T}}$ representation and final VHDL output.

The top-level source code for a single-bit adder is shown in Listing 6.9, noting that `addU` is already provided by our standard library — the definition is only shown here for reference.

Listing 6.9: A synthesisable single-bit adder for `Unsigned`

```
-- Top-level adder circuit
myadd : {x,y : Nat} →
        ⟨Unsigned 1 x → Unsigned 1 y → Unsigned 2 (plus x y)⟩
pat x, y ⇒
  myadd {x} {y} =
    ⟦ λxs ⇒ λys ⇒ ~(addU 1 {x} {y} {0} ⟦ xs ⟧ ⟦ ys ⟧ ⟦ 0 ⟧) ⟧

-- Unsigned adder family (from Data.Unsigned)
addU : (w : Nat) → {x,y,c : Nat} →
       ⟨Unsigned w x⟩ → ⟨Unsigned w y⟩ → ⟨Bit c⟩ →
       ⟨Unsigned (S w) (plus c (plus x y))⟩
pat c, cin ⇒
  addU 0 {0} {0} {c} ⟦ UNil ⟧ ⟦ UNil ⟧ cin
    = ⟦ UCons {_} {0} {c} UNil ~cin ⟧
pat w, c, xsn, xn, xbs, xb, ysn, yn, ybs, yb, cin ⇒
  addU (S w) {_} {_} {c} ⟦ UCons {w} {xsn} {xn} xbs xb ⟧
                         ⟦ UCons {w} {ysn} {yn} ybs yb ⟧ cin
    = ⟦ case (addBit {_} {_} {_} ~cin xb yb) of
          pat a, b, prf, cin', lsb
          ⇒ (MkBitPair {a} {b} {_} {prf} cin' lsb) ⟹
               let rec = ~(addU _ {_} {_} {_} ⟦ xbs ⟧ ⟦ ybs ⟧ ⟦ cin' ⟧)
                   ans = UCons {_} {_} {_} rec lsb
               in eqInd2 {_} {_} {_}
                         {prfAddU c xn yn a b xsn ysn prf}
                         {λh ⇒ Unsigned (S (S w)) h} ans
      ⟧
```

The top-level `myadd` circuit needs to meet a few restrictions in order to be synthesisable. These were described in the abstract form back in Section 6.4.1. Using `myadd` as a concrete example, we ensure that:

↪ The *extraction* of `myadd` is a single code type. This is true since the `x` and `y` arguments are marked as irrelevant and will be erased during extraction. The remaining type is a single code type — albeit a function type, `Unsigned 1 _ → Unsigned 1 _ → Unsigned 2 (plus _ _)`.

↪ The argument types and return type of the extraction are all synthesisable. This is true since `Unsigned` is declared as a `simple` type constructor, and the `Unsigned 1 _` / `Unsigned 2 _` types are both synthesisable (even with the second argument erased).

↪ There is no recursion in the circuit's run-time. While `addU` *is* defined recursively, this recursive call appears directly in an escaped quote (line 21) and will be evaluated during circuit elaboration.

We also take no risks with the erasure of intermediate non-synthesisable terms. The only non-synthesisable terms present after erasure exist in stage zero and should be reduced by the elaboration's normalisation-by-evaluation.

This source representation is then parsed directly into its $\texttt{TT}^{\mathcal{T}}_{\text{imp}}$ representation before being type checked and expanded into a $\texttt{TT}^{\mathcal{T}}$ program. Since we are focusing on the circuit synthesis process here, we rejoin our `myadd` example in the subsequent representation — normalised $\texttt{CExp}^{\mathcal{T}}$. A listing of this form is shown in Listing 6.10. This representation captures the circuit after erasure, full inlining, and elaboration. We should only expect to encounter the simplified constructs from Figure 6.15: a top-level definition with a list of let-bindings, each of which are either local variable references, constructor applications, one-level case expressions, or projections.

We can read the $\texttt{CExp}^{\mathcal{T}}$ representation for `myadd` as four main stages, only one of which is responsible for the combinatorial logic we might expect to see. Step one (lines 4–7) explicitly projects the two single `Bits` out of our `Unsigned` inputs. These lines prepare new local names for each `Bit` to be used as the scrutinee of subsequent case expressions. Note that the full type of both of these are `Bit [_]`. We do not know if the bit represents a `0` or a `1` — the synthesised bit representation will be wide enough to encode both constructors.

Step two (lines 10–24) performs the combinatorial logic required by our adder. These three case expressions scrutinise the `Bits` gathered in the previous step. Since each case inspects a single bit, we can easily imagine synthesising these to `MUX2` structures. The output used going forward is `cv:14`, defined as the pair of our carry output and sum. With the circuit's carry input tied to zero, we would expect the carry output to be governed by $x \,\&\, y$, and the sum by $x \oplus y$. The form we see in Listing 6.10 is generated by a case tree instead of using standard logic gate primitives such as &, |, +, and $\oplus$. We can demonstrate the equivalence by constructing truth tables for both encodings, and we also demonstrate an automatic reinterpretation later (Figures 6.18 and 6.19) using the open source synthesis tool Yosys.

Listing 6.10: Normalised $\texttt{CExp}^{\mathcal{T}}$ representation for `myadd`

```
1  myadd ([xs, ys]) : =
2
3    -- Project bits out from unsigned inputs
4    let {cv:3} : (Bit [_]) =
5         prj^UCons_1 (local xs) in
6    let {cv:5} : (Bit [_]) =
7         prj^UCons_1 (local ys) in
8
9    -- Perform full-adder logic
10   let {cr:12} : (BitPair [_]) =
11        case (local {cv:5}) of {
12          O [] ⇒ MkBitPair[ (O []) (I []) ])
13          I [] ⇒ MkBitPair[ (I []) (O []) ])
14        } in
15   let {cr:13} : (BitPair [_]) =
16        case (local {cv:5}) of {
17          O [] ⇒ MkBitPair[ (O []) (O []) ]
18          I [] ⇒ MkBitPair[ (O []) (I []) ]
19        } in
20   let {cr:14} : (BitPair [_]) =
21         case (local {cv:3}) of {
22          O [] ⇒ local {cr:13}
23          I [] ⇒ local {cr:12}
24        } in
25
26   -- Project bits out from pairs
27   let {cv:7} : (Bit [_]) =
28        prj^MkBitPair_1 (local {cr:14}) in
29   let {cv:6} : (Bit [_]) =
30        prj^MkBitPair_0 (local {cr:14}) in
31
32   -- Construct unsigned output
33   let {cr:15} : (Unsigned 1 (plus [_] (double 0))) =
34        UCons[ (UNil []) (local {cv:6}) ] in
35   let {rec:9} : (Unsigned 1 (plus [_] (double 0))) =
36        (local {cr:15}) in
37   let {ans:11} : (Unsigned 2 (plus [_] (double (plus [_] (double 0))))) =
38        UCons[ (local {rec:9}) (local {cv:7}) ] in
39
40   -- Return output
41   local {ans:11}
```

The final two stages (lines 27–30 and 33–38) encode the simple routing required to construct the final output. First, we project out each bit from the `BitPair [_]` in `cv:14`. Next, we recombine them into the output `Unsigned 2 [_]` representation. We will later see that, since both of these types only have one possible constructor, the constructor tags are empty. This means that this pair of unpacking and packing steps actually compiles down to routing only. A circuit operating over data types with ambiguous constructors might inspect or introduce new constructor tags here, resulting in some extra logic.

Now we consider the representation of `myadd` in the final step of our compiler pipeline: the VHDL output. This is a simple translation structurally, but demonstrates the important topic of bit representation for our GADTs. Every signal has an explicit wordlength and known layout; we know the length of the tag bits for each type and the position of each argument for each constructor.

Listing 6.11 shows the full VHDL output for the `myadd` example. Since the translation between $\texttt{CExp}^{\mathcal{T}}$ and VHDL is essentially a $1:1$ mapping, we maintain the four main stages from Listing 6.10. With reference back to Listing 6.10, it is clear to see simple translation at work:

Local variable	$\rightarrow$	Routing
Projection	$\rightarrow$	Indexing
Case expression	$\rightarrow$	`With/select` assignment
Constructor application	$\rightarrow$	Concatenation

All four of these rules are used in the construction of Listing 6.11, albeit with an empty tag for both constructor applications.

Although VHDL generation is the final step in `toatie`'s compilation pipeline, we can continue the broader FPGA/ASIC development flow and pass our design into a synthesis tool. We generate schematics for the `myadd` circuit by passing Listing 6.11 through the open source GHDL and Yosys synthesis tools. Vendor EDA tools should provide equivalent results. Figures 6.18 and 6.19 shows two versions of our schematic: one direct visualisation of our VHDL, and one after technology mapping.

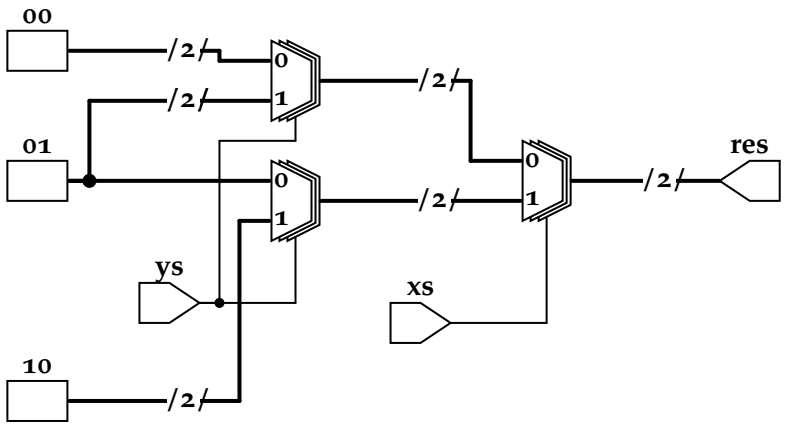

Figure 6.18: Schematic for `myadd` directly from VHDL output

Listing 6.11: VHDL representation for `myadd`

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity myadd is
  port (
    xs : in std_logic_vector (0 downto 0);
    ys : in std_logic_vector (0 downto 0);
    res : out std_logic_vector (1 downto 0)
  );
end myadd;

architecture behaviour of myadd is
  signal \cv_3\ : std_logic_vector (0 downto 0);
  signal \cv_5\ : std_logic_vector (0 downto 0);
  signal \cr_12\ : std_logic_vector (1 downto 0);
  signal \cr_13\ : std_logic_vector (1 downto 0);
  signal \cr_14\ : std_logic_vector (1 downto 0);
  signal \cv_7\ : std_logic_vector (0 downto 0);
  signal \cv_6\ : std_logic_vector (0 downto 0);
  signal \cr_15\ : std_logic_vector (0 downto 0);
  signal \rec_9\ : std_logic_vector (0 downto 0);
  signal \ans_11\ : std_logic_vector (1 downto 0);

begin
  -- Project bits out from vector inputs
  \cv_3\ <= xs(0 downto 0);
  \cv_5\ <= ys(0 downto 0);

  -- Perform full-adder logic
  with \cv_5\(0 downto 0) select \cr_12\ <=
    "0" & "1" when "0",
    "1" & "0" when "1",
    "00" when others;

  with \cv_5\(0 downto 0) select \cr_13\ <=
    "0" & "0" when "0",
    "0" & "1" when "1",
    "00" when others;

  with \cv_3\(0 downto 0) select \cr_14\ <=
    \cr_13\ when "0",
    \cr_12\ when "1",
    "00" when others;

  -- Project bits out from pairs
  \cv_7\ <= \cr_14\(0 downto 0);
  \cv_6\ <= \cr_14\(1 downto 1);

  -- Construct vector outputs
  \cr_15\ <= "" & \cv_6\;
  \rec_9\ <= \cr_15\;
  \ans_11\ <= "" & \rec_9\ & \cv_7\;

  -- Assign output
  res <= \ans_11\;

end behaviour;
```

The version after Yosys's technology mapping demonstrates that our case tree version is directly equivalent to the expected & and $\oplus$ logic gates. Later place and route algorithms are then specialised to the architecture of the target device — accounting for the primitive LUT dimensions, routing configurations, etc.

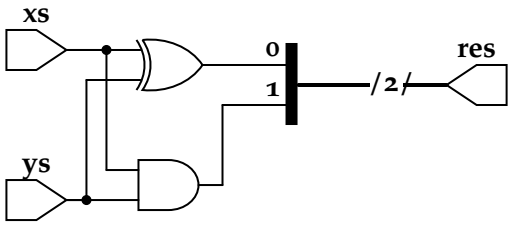

Figure 6.19: Schematic for `myadd` after synthesis with GHDL and Yosys, with mapping to logic gates

This concludes our in-depth look at the synthesis process for a single adder example. The following subsections give a very brief look at some other small examples, highlighting further common use cases.

Routing — Mirroring a binary tree

Listing 6.12 shows a program which mirrors/reflects binary tree structures. The mirroring process *should* be possible entirely through the circuit's routing. This example provides us with a sanity check to ensure that we do not infer gruesome overheads in our synthesis process — it should be implemented "for free" in a larger circuit's routing. This also acts as a reminder that we can think of evaluation of stage zero code as a user-defined elaboration stage.

For clarity, `mirror` encodes a circuit family for mirroring a binary tree of any size. Our top-level circuit is `mymirror`, which specialises `mirror` for trees with a depth of three and a node type of `Bit`. Figure 6.20 visualises the binary tree structure and how the result of the mirroring process should appear.

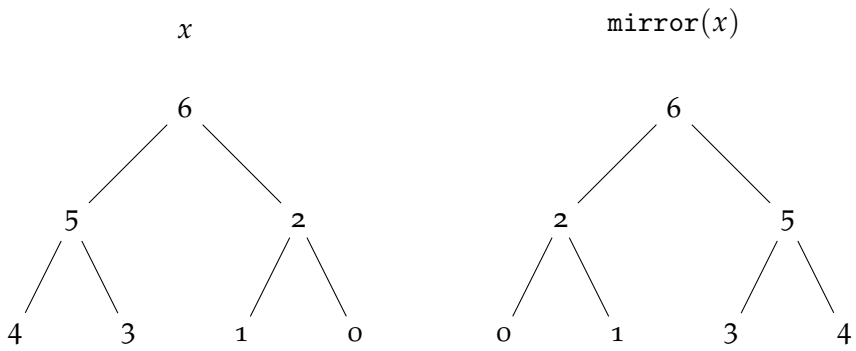

Figure 6.20: Binary tree mirroring example for depth of three

Listing 6.12: A mirroring function for binary trees

```
import Data.Nat

simple Bit : Type where
  O : Bit
  I : Bit

simple BTree : Nat → Type → Type where
  Leaf : {a : Type} → BTree Z a
  Node : {a : Type} → {n : Nat} → a → BTree n a → BTree n a →
         BTree (S n) a

mirror : {a : Type} → (n : Nat) → ⟨BTree n a⟩ → ⟨BTree n a⟩
pat a ⇒
  mirror {a} Z ⟦ Leaf {_} ⟧ = ⟦ Leaf {_} ⟧
pat a, n, x, tl, tr ⇒
  mirror {a} (S n) ⟦ Node {_} {n} x tl tr ⟧
    = ⟦ Node {_} {_} x ~(mirror {_} _ ⟦ tr ⟧ ) ~(mirror {_} _ ⟦ tl ⟧ ) ⟧

mymirror : ⟨BTree 3 Bit → BTree 3 Bit⟩
mymirror = ⟦ λt ⇒ ~(mirror {Bit} 3 ⟦ t ⟧ ) ⟧
```

As before, there is no ambiguity between constructors for any `BTree` types with a known depth. This means no bits are used to store the tag. A depth three binary tree for `Bits` should occupy 7 bits (one for each node). When deriving the bit representation, we traverse data constructor arguments in a left → right order. For `BTree` data, the ordering is equivalent a preorder traversal. For example, the tree x in Figure 6.20 would be flattened to the sequence $6, 5, 4, 3, 2, 1, 0$.

Figure 6.21 shows the final schematic for our `mymirror` circuit after `toatie`'s compilation and visualisation with Yosys. We confirm that this simplifies down to routing only; all of its structure is completely determined during our elaboration.

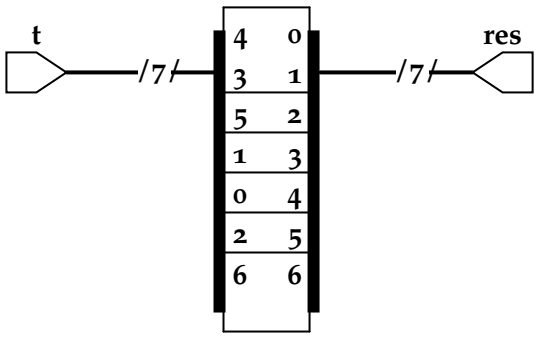

Figure 6.21: Final schematic for `mymirror`

Structured data — keeping us honest with `Maybe`

Now we present an example demonstrating how we can actually inspect the structure of our data types during circuit run-time. This relies on automatic separation of a type's constructor tag from the representation of its fields. We show this with a circuit operating over a `Maybe` type. This data type is constructed either as an invalid sample (`Nothing`) or a valid sample with its payload (`Just x`). Using `Maybe` keeps us honest in our descriptions with potentially invalid signals, especially for polymorphic functions. Here we cannot rely on invalid data since we have no access to its payload — only the fact that the sample *is* invalid! We preclude the designer's temptation to use bogus inputs when working around edge-cases.

The source example presented in Listing 6.13 shows such a circuit which negates an input bit only when it is valid. Any valid input produces a valid negated output, and any invalid input should always remain invalid. This is implemented with the polymorphic `map` function for `Maybe`, which enjoys the type-safety described earlier.

Listing 6.13: An `Maybe` example for principled used of valid-gated data

```
import Data.Nat

simple Bit : Type where
  O : Bit
  I : Bit

simple Maybe : Type → Type where
  Nothing : {a : Type}      → Maybe a
  Just    : {a : Type} → a → Maybe a

map : {a, b : Type} → (f : ⟨a → b⟩) → ⟨Maybe a⟩ → ⟨Maybe b⟩
pat a, b, f, ma ⇒
  map {a} {b} f ma =
    ⟦ case ~ma of
        pat x ⇒ Just {a} x ⟹ Just    {_} (~f x)
        Nothing {a}        ⟹ Nothing {_}
    ⟧

not : Bit → Bit
not O = I
not I = O

maybeNot : ⟨Maybe Bit → Maybe Bit⟩
maybeNot = ⟦ λx ⇒ ~(map {Bit} {Bit} ⟦ λy ⇒ not y ⟧ ⟦ x ⟧ ) ⟧
```

Before considering the output schematic, we consider the bit representation of `Maybe Bit`. This synthesises down into a two bit structure: the most significant bit represents the constructor tag (`0` for `Nothing` or `1` for `Just`), and the least significant bit represents the encapsulated data `Bit`.

Figure 6.22 shows the final netlist, after synthesis with Yosys. The input's valid bit is passed directly to the output's valid bit, ensuring that we preserve the validity flag of our data. The data itself is only negated when it *is* valid — ensured by the $valid \;\&\; \overline{data}$ gate. We note that the higher-order `map` function is completely specialised and results in an implementation we would have hand-coded in a more traditional HDL.

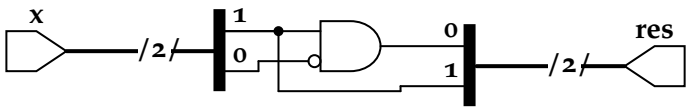

Figure 6.22: Final schematic for `maybeNot`

Larger designs — A DFT example

Finally, we highlight an example with slightly more complexity. We offer a top-level wrapper for the radix-2 DIT implementation of a DFT, as detailed in Section 5.4.4. Beyond the greater complexity of this example after synthesis, the source also contains some interesting complexities considered back in Section 5.4.4. These topics included heavy use of proofs to demonstrate equivalence with a direct DFT implementation, and a heterogeneous collection of signals with different wordlengths (facilitated by our `HWords` type). The top-level wrapper for the previous description is given in Listing 6.14.

Listing 6.14: Top-level description for a two-sample, 8-bit unsigned, radix-2 Cooley-Tukey FFT

```
1  import Examples.FFT.TwiddlesN2
2  import Examples.FFT
3
4  dft_2 : {xs : Vect 2 ZZ} →
5          ⟨HWords 2 [8,8] xs →
6           HWords 2 [9,9] (dit {2} {oscilate} twOscil2 (PDouble 1 POne) xs)⟩
7  pat xs ⇒
8    dft_2 {xs} =
9      ⟦ λbs ⇒ ~(circDIT {2} {oscilate}
10                       twOscil2 (PDouble 1 POne) [8,8] {xs} ⟦ bs ⟧ )
11     ⟧
```

The final netlist is shown in Figure 6.23, demonstrating that all of our source-level constructs reasoning about the radix-2 DIT structure do indeed synthesise down to equivalent netlist constructs. This offers a more complex circuit than our other examples presented in this chapter and is only limited by readability, not by the compiler itself.

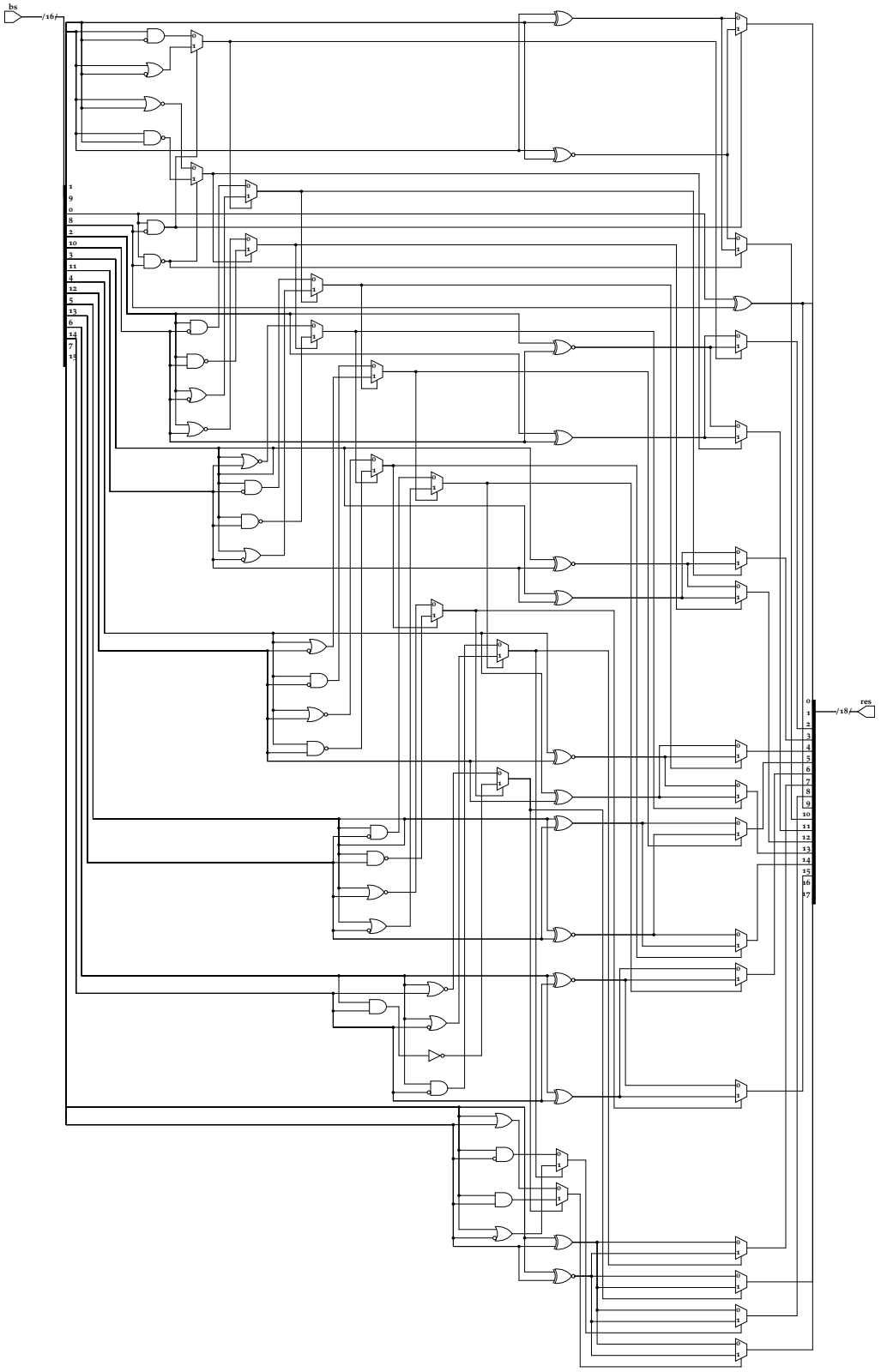

Figure 6.23: Schematic for `dft_2`

6.5 FURTHER WORK

The current implementation of `toatie`'s compiler and synthesis scheme has been substantiated against the plethora of examples from throughout Section 6.4.5 and Chapter 5. However, this section suggests four avenues for further work, towards both strengthening the presented theory and maturing its implementation.

6.5.1 A fully typed synthesis scheme

We acknowledge that our prototype circuit normaliser would be more powerful if type information was retained throughout the entire process. As it stands, types and irrelevant terms are erased, the description is evaluated as far as possible, and an attempt is made to reconstruct just enough type information to identify each term's bit representation. This is purely for convenience of implementation. Chapter 4 of [19] demonstrates how the elimination of intermediate non-synthesisable terms is much more powerful in the presence of typing annotations. In particular, a distinction can be made between terms that ought to be synthesisable and terms which are not. Different sets of rules can be applied, including inlining and specialisation of non-synthesisable terms in order to safely eliminate them from our netlist. For `toatie`, it would also be possible to synthesise a wider range of data types if type information is retained. Currently, it must be able to reconstruct enough of each term's type in order to determine a bit encoding, which will fail when presented with enough ambiguity. Consistent typing can also offer a sanity check for any future optimisations included in the compiler.

6.5.2 Formalisation of synthesisability requirements

Chapters 5 and 6 offer a range of circuit families which `toatie` can synthesise, which is an encouraging result. Future research could help formalise this system and verify our suggested restrictions for synthesisability. A rigorous, metatheoretic study could eventually prove or dismiss important properties of the system presented in this thesis, such as completeness of certain compiler passes. This is a pressing matter since bugs in the circuit synthesis process will undermine the confidence given by theorems proved in the core language. Since `toatie` is itself implemented in a dependently typed language, there is even scope for encoding proofs about the compiler's implementation in its source. For example, one could attempt to verify that the synthesis of netlists preserves the semantics of the original description, or that the synthesis process can always complete when given a valid source.

6.5.3 *Rebase on Idris 2*

It is expected that an engineering effort to rebase `toatie`'s concepts on top of a rich, dependently typed language such as Idris 2 would be extremely worthwhile. A port to Idris 2 in particular is appealing since `toatie`'s original foundation was itself a very stripped down version of Idris 2. With enough squinting, the two have a shared internal structure. This rebase ought to provide a few benefits, nearly for free:

↪ We gain access to the existing ecosystem of libraries for Idris. These can be reused directly in stage zero, and a subset of them might be applicable to higher stages.

↪ We get access to all the usability features of Idris. This importantly includes its language server, offering text-editor support for case splitting, type-directed auto complete, and interactive theorem proving. The latter substantially lessens the burden of hand-crafting proof terms.

↪ Since Idris 2's type theory is based on quantitative type theory, it offers an implementation of erasure for free *and* the option of exploring linear types. Although linear types are an important aspect of [58], we have not yet explored their use for digital circuit descriptions.

6.5.4 *Netlist optimisations for FPGA architectures*

Finally, it might prove interesting to perform a deeper analysis of how the netlist structures generated by `toatie` interact with the optimisations present in FPGA vendor tools, such as Xilinx's Vivado software. Pattern matching is currently synthesised via a (potentially quite deep) case tree. For constructors with only two options (such as `Bit`), this can infer a deep chain of MUX2 structures. It might be the case that EDA tools can infer better circuits if the compiler foregoes the case tree structure and more directly translates an entire pattern matching clause to a shallower, wider structure. There are similar choices to be made regarding the derived bit representations for user-defined data. Ideally `toatie` would be able to choose from a set of different encodings for each data type, exploiting trade-offs between the existing encoding and the likes of larger, one-hot encodings for tags. Lastly, for best use of the available hardware, a set of primitive constructs could be added to `toatie` to encode the fundamental resources available on modern FPGAs. These include hardened blocks such as DSP48E blocks, BlockRAMs, and UltraRAMs.

6.6 SUMMARY

This chapter has discussed three broad topics, as they relate to hardware description. We have presented a formalisation of the TinyIdris software language, on which `toatie` is based. We then presented `toatie`'s core language as a superset of TinyIdris, with all modifications justified by one of four feature groups:

Erasure to remove non-synthesisable terms from otherwise synthesisable data types.

Staging to provide a type-safe, user-programmable circuit elaboration system.

Synthesis to derive netlists from a high-level circuit description.

Syntactic sugar simply to improve the ergonomics of the language.

While the topics of erasure and staging were present in discussions of software languages, we noted that these are usually employed as a secondary matter of optimisation. When encoding circuits as plain functions in the hardware description world, both of these features quickly became completely necessary for synthesisability.

The third section detailed this synthesis process. We automatically derived bit representations for any synthesisable, `simple` types, maintaining a separation between a type's tag and its fields. This streamlined circuit descriptions by allowing the designer to use the same `toatie` choice constructs for both elaboration software *and* synthesisable circuit behaviour — we could match on the inputs tag bits and choose the correct alternative at circuit run-time. While our current normalisation process has its limitations, there is existing literature explaining precisely how these challenges (for us, the elimination of intermediate non-synthesisable terms) could be readily addressed. From this normalised $\texttt{CExp}^{\mathcal{T}}$ form, circuit generation is trivial. We show that any program in $\texttt{CExp}^{\mathcal{T}}$ form could be implemented with just four simple circuit constructs: referencing signals, splitting signals, concatenating signals, and multiplexing `case` alternatives.

These observations accumulate to provide, to the best of the author's knowledge, the first HDL and compiler with dependent types, where *synthesisable* combinatorial circuits can be represented as plain functions and enjoy full functional verification in a correct-by-construction fashion.

7 CONCLUSION

7.1 THESIS REVIEW

This thesis set out to explore a new HDL design, encouraging the description and full functional verification of correct-by-construction circuit families under a single roof. More specifically, the aim was to explore one uncharted point on this design space:

↪ Encoding circuits as plain *functions*.

↪ Ascribing precise *meaning* to synthesisable data via its type.

Our discussion of existing HDLs in Chapter 2 and our in-depth look at our own novel DSP circuit family in Chapter 4 has strongly motivated these two language design choices. The new parallel filtering architecture presented in Chapter 4 is also substantial in its own right, producing frontend RFSoC filters without the need for any precious, hardened DSP48E2 resources — and occasionally with a *total* footprint smaller than just the traditional CLB *overhead*. More often, the new architecture offers a valuable trade-off for half-band filters: the traditional percentage usage of DSP48E2s is translated to $\times 0.8$ of the percentage for the generic CLB resources, leaving the specialised resources for rapid prototyping beyond the frontend filtering stage.

A functional HDL that encodes circuits as plain functions simplifies the implementation of circuits since designers can use the same set of native language features (including choice constructs like pattern matching) to describe both elaboration-time behaviour *and* circuit run-time behaviour. Ascribing precise meaning to synthesisable data, achieved with dependent types, allows our type checker to completely verify the functional behaviour of an entire circuit family at once. This is in contrast to more commonly used model checking techniques, practically limited to analysing one concrete circuit at a time. Tracking the meaning in the synthesisable data types themselves encourages a correct-by-construction strategy, where we get a theorem about our circuit's functional behaviour for "free", rather than treating verification as a wholly separate concern after the fact. Here, we can enjoy a productive, type-driven methodology for describing circuits with confidence. These benefits have been explored practically through the numerous examples in Chapter 5.

An important insight in Chapter 5 is that, although `toatie` provides an environment capable of full functional verification, a designer is still free to decide how heavily they want to lean on the type system. Section 5.2 does so minimally, but still enjoys the benefits of explicitly staged descriptions and simple type-level programming to control wordlengths with more finesse than is common. Section 5.3 explores an interesting mid-point: using the type system to track non-functional aspects of a circuit family. The running example is to encode a binary number's precise range in its type, offering optimal wordlength pruning strategies even deep within a larger DSP circuit. Section 5.4 demonstrates the extreme, revisiting many of the previous circuits while demanding full functional verification in a correct-by-construction fashion. Perhaps the most surprising finding there is how well combinatorial DSP circuits map to this style of programming. Once the fundamental arithmetic blocks are in place, the implementations of the dot product, and even a radix-2 DIT FFT, really do give their correctness proofs for *free*.

The choice to represent circuits as plain, dependently typed functions does, however, place a high burden on the language and compiler design. In terms of language design, it was found that two features used as optimisations in software programming (erasure and staging) become completely necessary for synthesisable hardware descriptions. At the compiler-level, a new synthesis scheme was needed to translate suitable plain functions and user-defined GADTs to equivalent netlists. Chapter 6 began to address these challenges with the following insights:

↪ Erasure and irrelevance are a requirement for circuit description, instead of their standard role in software programming as an optimisation. We will often need to allow non-synthesisable terms to direct the type checking for otherwise synthesisable data, especially for correct-by-construction techniques. We need a mechanism for ensuring that these non-synthesisable terms are present during type checking but are guaranteed to be erased from the final netlist.

↪ Staging constructs also become a requirement for circuit description, instead of their standard role as an optimisation. A parameterised circuit has two clear and even physically distinct phases: compile-time elaboration of the netlist and the circuit's run-time on an FPGA. The staging constructs help us control the flow of information between these two stages in a type safe way. They are used to ensure causality, where information generated in the elaboration stage can be passed forward to the circuit's run-time but we forbid information being passed from the circuit's run-time backwards to the elaborator.

↪ A new strategy for automatically synthesising bit representations for user-defined algebraic data types, including definitions with bounded recursion. The imple-

mentation of this becomes quite simple for a dependently typed language since we can directly reuse the type checker's unification process to determine all possible data constructors for a given (partially-known) type. The synthesisability of an entire circuit is then guided by a set of reasonable restrictions from Section 6.4.

↪ A prototype normalisation stage, reducing a synthesisable circuit in `toatie`'s core language down to a form which can be trivially converted to a netlist. This process includes eliminating constructs without a simple netlist semantics, such as staging annotations and λ-abstractions/applications. When successful, we return a version of the circuit with only four main constructs, all of which are trivial to encode in VHDL. These are local variable references (a wire), constructor applications (a concatenation of wires), case expressions (a set of alternative wire bundles and a multiplexer), and projection (a slice).

7.2 FURTHER WORK

There are five main themes exposed by this thesis which merit further research. While detailed in Sections 5.5 and 6.5, these five avenues are, in summary:

↪ Encoding *synchronous* circuits in `toatie`. There is strong precedence for this in the literature, and is expected to require engineering effort only. The more academic consideration here is addressing how the correct-by-construction verification techniques presented in Chapter 5 could translate to a synchronous context.

↪ Extending our circuit synthesis scheme to retain types throughout all steps. This should help broaden `toatie`'s support for synthesis of user-defined data types and elimination of non-synthesisable intermediates.

↪ Further formalisation of our synthesis process and the restrictions we suggest to encourage synthesisability. The end goal is to provide a guarantee that: 1) the semantics of any generated netlist match that of the source program, and 2) netlist generation will *always* succeed for a valid source.

↪ Rebasing the implementation of `toatie` on a rich surface language such as Idris 2 with interactive development tooling. Gaining access to the ecosystem of a more mature language would substantially improve productivity.

↪ Further investigating the interplay between `toatie`-generated netlists and downstream EDA tooling. It is possible that refinements could be made to generated netlist structures to better appeal to particular FPGA architecture resources and their vendor tooling, including alternative bit representations and synthesis of pattern matching.

7.3 CONCLUDING REMARKS

At the end of this journey, we have illustrated the returns of using dependently typed programming to describe and verify combinatorial digital circuits. Indeed, implementation and verification need not be entirely independent affairs, and one can often be used to learn more about the other. More concretely, this research has explored a set of language and compiler features which unite to actually synthesise real-world combinatorial circuits from plain, dependently typed functions.

Clearly, the "disturbing" conclusion in [1] — that only 16% of FPGA projects reach production without bugs — demonstrates a real issue with the traditional approaches to circuit verification. The dependently typed environment offered by `toatie` allows for a fundamentally different approach to verification, making it possible to catch functional errors anywhere in *an entire circuit family, at compile-time*. This digital design methodology feels particularly impactful to high-assurance contexts.

The work presented in this thesis has discovered one relatively unexplored point in the historied landscape of functional HDLs. Beyond emboldening digital designers to better craft *correct* circuits, this thesis hopes to kindle an interest in *staged, dependently typed languages* elsewhere in the wider, blooming functional hardware community.

BIBLIOGRAPHY

[1] H. Foster, Siemens EDA. *2022 Wilson Research Group functional verification study*. 2022. URL: https://blogs.sw.siemens.com/verificationhorizons/2022/10/10/prologue-the-2022-wilson-research-group-functional-verification-study/.

[2] Xilinx, Inc. *XC2000 Logic Cell Array Families*. 1985. URL: https://labmaster.com/surplus/parts/html/941557-pl/xc2000.pdf.

[3] Xilinx, Inc. *XC4000, XC4000A, XC4000H Logic Cell Array Families*. 1991. URL: https://media.digikey.com/pdf/Data%20Sheets/Xilinx%20PDFs/XC4000,A,H.pdf.

[4] Xilinx, Inc. *DS003-1 — Virtex 2.5 V Field Programmable Gate Arrays (v4.0)*. 2013. URL: https://docs.xilinx.com/v/u/en-US/ds003.

[5] Xilinx, Inc. *Xilinx Virtex-II Series FPGAs*. 2003. URL: https://www.xilinx.com/publications/matrix/virtex_bw.pdf.

[6] Xilinx, Inc. *DS112 — Virtex-4 Family Overview (v3.1)*. 2010. URL: https://docs.xilinx.com/v/u/en-US/ds112.

[7] Xilinx, Inc. *Xilinx Virtex-5 FPGAs Product Table*. 2015. URL: https://docs.xilinx.com/v/u/en-US/virtex5-product-table.

[8] Xilinx, Inc. *DS150 — Virtex-6 Family Overview (v2.5)*. 2015. URL: https://docs.xilinx.com/v/u/en-US/ds150.

[9] Xilinx, Inc. *7 Series Product Selection Guide*. 2014. URL: https://www.xilinx.com/content/dam/xilinx/support/documents/selection-guides/7-series-product-selection-guide.pdf.

[10] Xilinx, Inc. *UltraScale+ FPGAs Product Tables and Product Selection Guide*. 2015. URL: https://www.xilinx.com/content/dam/xilinx/support/documents/selection-guides/ultrascale-plus-fpga-product-selection-guide.pdf.

[11] Xilinx, Inc. *DS 926 — Zynq UltraScale+ RFSoC Data Sheet: DC and AC Switching Characteristics*. 2021. URL: https://www.xilinx.com/support/documentation/data_sheets/ds926-zynq-ultrascale-plus-rfsoc.pdf.

[12] D. Allan et al. *Software Defined Radio with Zynq Ultrascale+ RFSoC*. English. Ed. by L. Crockett, D. Northcote, and R. Stewart. 1st. Jan. 2023. ISBN: 9781739588601.

[13] Xilinx, Inc. *UG574 — UltraScale Architecture Configurable Logic Block (v1.5)*. 2017. URL: https://docs.xilinx.com/v/u/en-US/ug574-ultrascale-clb.

[14] S. M. Trimberger. "Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology". In: *Proceedings of the IEEE* 103.3 (2015), pp. 318–331. DOI: 10.1109/JPROC.2015.2392104.

[15] J. Hughes. "Why Functional Programming Matters". In: *The Computer Journal* 32.2 (Jan. 1989), pp. 98–107. ISSN: 0010-4620. DOI: 10.1093/comjnl/32.2.98. eprint: https://academic.oup.com/comjnl/article-pdf/32/2/98/1445644/320098.pdf. URL: https://doi.org/10.1093/comjnl/32.2.98.

[16] C. McBride. "Epigram: Practical programming with dependent types". In: *Advanced Functional Programming: 5th International School, AFP 2004, Tartu, Estonia, August 14–21, 2004, Revised Lectures*. Springer. 2005, pp. 130–170.

[17] U. Norell. "Towards a practical programming language based on dependent type theory". PhD thesis. Göteborg, Sweden, 2007. ISBN: 978-91-7291-996-9.

[18] E. Brady. "Idris, a general-purpose dependently typed programming language: Design and implementation". In: *Journal of Functional Programming* 23.5 (2013), pp. 552–593. DOI: 10.1017/S095679681300018X.

[19] C. Baaij. "Digital circuit in CλaSH: functional specifications and type-directed synthesis". PhD thesis. Netherlands: University of Twente, Jan. 2015. ISBN: 978-90-365-3803-9. DOI: 10.3990/1.9789036538039.

[20] J. P. P. Flor, W. Swierstra, and Y. Sijsling. "ΠI-Ware: Hardware Description and Verification in Agda". In: *21st International Conference on Types for Proofs and Programs (TYPES 2015)*. Ed. by T. Uustalu. Vol. 69. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, 9:1–9:27. ISBN: 978-3-95977-030-9. DOI: 10.4230/LIPIcs.TYPES.2015.9. URL: http://drops.dagstuhl.de/opus/volltexte/2018/8479.

[21] T. Sheard. "Types and hardware description languages". In: *Hardware Design and Functional Languages, A satellite event of ETAPS. Volume 5161*. European Joint Conferences on Theory and Practice of Software, 2007.

[22] E. Brady, J. McKinna, and K. Hammond. "Constructing Correct Circuits: Verification of Functional Aspects of Hardware Specifications with Dependent Types". English. In: *Trends in Functional Programming*. Vol. 8. Eighth Symposium on Trends in Functional Programming, which was held in New York City on April 2–4, 2007. United Kingdom: Intellect Books, 2008, pp. 159–176. ISBN: 9781841501963.

[23] C. Ramsay, L. H. Crockett, and R. W. Stewart. "On Applications of Dependent Types to Parameterised Digital Signal Processing Circuits". In: *2021 IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*. 2021, pp. 787–791. DOI: 10.1109/MWSCAS47672.2021.9531730.

[24] C. Ramsay. *Source code for: "On Applications of Dependent Types to Parameterised Digital Signal Processing Circuits"*. 2020. DOI: 10.15129/db040cb9-9e48-4823-8616-cbc0ace1b6cd. URL: https://doi.org/10.15129/db040cb9-9e48-4823-8616-cbc0ace1b6cd.

[25] C. Ramsay, L. H. Crockett, and R. W. Stewart. "Low-cost, High-speed Parallel FIR Filters for RFSoC Front-Ends Enabled by CλaSH". In: *2021 55th Asilomar Conference on Signals, Systems, and Computers*. 2021, pp. 925–932. DOI: 10.1109/IEEECONF53345.2021.9723107.

[26] C. Ramsay. *Source Code for Conifer — A playground for parallel and multiplierless FIR filters with Clash*. 2021. DOI: 10.15129/a2c118f2-48a8-40d2-8896-89b9da71a4be. URL: https://github.com/cramsay/conifer.

[27] C. Ramsay. *Source for toatie— a Hardware Description Language With Dependent Types*. 2022. DOI: 10.15129/fd83f191-2dc1-4839-adbb-684bac5ecd0c. URL: http://github.com/cramsay/toatie.

[28] E. Brady. *Source Code for TinyIdris — SPLV 2020*. 2020. URL: https://github.com/edwinb/SPLV20.

[29] L. Crockett et al. *Exploring Zynq MPSoC: With PYNQ and Machine Learning Applications*. English. Apr. 2019. ISBN: 0992978750.

[30] J. Goldsmith et al. "Control and Visualisation of a Software Defined Radio System on the Xilinx RFSoC Platform Using the PYNQ Framework". In: *IEEE Access* 8 (2020), pp. 129012–129031. DOI: 10.1109/ACCESS.2020.3008954.

[31] A. Edelman. "The Mathematics of the Pentium Division Bug". In: *SIAM Review* 39.1 (1997), pp. 54–67. DOI: 10.1137/S0036144595293959. URL: https://doi.org/10.1137/S0036144595293959.

[32] E. M. Clarke, M. Khaira, and X. Zhao. "Word Level Model Checking—Avoiding the Pentium FDIV Error". In: *Proceedings of the 33rd Annual Design Automation Conference*. DAC '96. Las Vegas, Nevada, USA: Association for Computing Machinery, 1996, pp. 645–648. ISBN: 0897917790. DOI: 10.1145/240518.240640. URL: https://doi.org/10.1145/240518.240640.

[33] J. Yuan, C. Pixley, and A. Aziz. Boston, MA: Springer US, 2006, pp. 25–35. ISBN: 978-0-387-30784-8. DOI: 10.1007/0-387-30784-2_2. URL: https://doi.org/10.1007/0-387-30784-2_2.

[34] M. Sheeran. "MuFP, a Language for VLSI Design". In: *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*. LFP '84. Austin, Texas, USA: Association for Computing Machinery, 1984, pp. 104–112. ISBN: 0897911423. DOI: 10.1145/800055.802026. URL: https://doi.org/10.1145/800055.802026.

[35] Design Automation Standards Committee. "IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language". In: *IEEE STD 1800-2009* (2009), pp. 1–1285. DOI: 10.1109/IEEESTD.2009.5354441.

[36] Design Automation Standards Committee. "IEEE Standard VHDL Language Reference Manual". In: *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)* (2009), pp. 1–640. DOI: 10.1109/IEEESTD.2009.4772740.

[37] Altera, Inc. *Intel® Quartus® Prime Software: Features*. 2022. URL: https://www.intel.com/content/www/us/en/products/details/fpga/development-tools/quartus-prime/article.html.

[38] P. J. Ashenden. *The Designer's Guide to VHDL*. eng. 3rd ed. Morgan Kaufmann series in systems on silicon. 0. Morgan Kaufmann Publishers Inc, 2010. ISBN: 0120887851.

[39] J. Decaluwe. *Verilog's Major Flaw*. Sigasi. 2010. URL: http://insights.sigasi.com/opinion/jan/verilogs-major-flaw.html.

[40] G. Chen. "A Short Historical Survey of Functional Hardware Languages". In: *ISRN Electronics* 2012 (Mar. 2012). DOI: 10.5402/2012/271836.

[41] M. Sheeran. *μFP: An Algebraic VLSI Design Language*. Technical monograph. Oxford University Computing Laboratory, Programming Research Group, 1983. URL: https://books.google.co.uk/books?id=gMtQAAAAIAAJ.

[42] "Bluespec System Verilog: Efficient, Correct RTL from High Level Specifications". In: *Proceedings of the Second ACM/IEEE International Conference on Formal Methods and Models for Co-Design*. MEMOCODE '04. USA: IEEE Computer Society, 2004, pp. 69–70. ISBN: 0780385098. DOI: 10.1109/MEMCOD.2004.1459818. URL: https://doi.org/10.1109/MEMCOD.2004.1459818.

[43] J. Bachrach et al. "Chisel: Constructing Hardware in a Scala Embedded Language". In: *Proceedings of the 49th Annual Design Automation Conference*. DAC '12. San Francisco, California: Association for Computing Machinery, 2012, pp. 1216–1225. ISBN: 9781450311991. DOI: 10.1145/2228360.2228584. URL: https://doi.org/10.1145/2228360.2228584.

[44] A. Gill et al. "Introducing Kansas Lava". In: *Implementation and Application of Functional Languages*. Ed. by M. T. Morazán and S.-B. Scholz. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 18–35. ISBN: 978-3-642-16478-1.

[45] P. Bjesse et al. "Lava: Hardware Design in Haskell". In: *SIGPLAN Not.* 34.1 (Sept. 1998), pp. 174–184. ISSN: 0362-1340. DOI: 10.1145/291251.289440. URL: https://doi.org/10.1145/291251.289440.

[46] K. Claessen and M. Sheeran. "A Tutorial on Lava: A Hardware Description and Verification System". In: Göteborg, Sweden: Chalmers University, 2000.

[47] G. Hutton. *Programming in Haskell*. Cambridge Univ Press, 2007. ISBN: 0521692695.

[48] S. Singh. "Designing reconfigurable systems in Lava". In: *17th International Conference on VLSI Design. Proceedings.* 2004, pp. 299–306. DOI: 10.1109/ICVD.2004.1260941.

[49] S. Singh. "Source code for: Xilinx-Lava Version 5". In: 2014. URL: https://github.com/satnam6502/lava.

[50] C. Baaij et al. "CλaSH : Structural Descriptions of Synchronous Hardware Using Haskell". In: *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*. Sept. 2010, pp. 714–721. DOI: 10.1109/DSD.2010.21.

[51] QBayLogic B.V. *Hackage Documentation for `clash-prelude`*. 2022. URL: https://hackage.haskell.org/package/clash-prelude.

[52] G. Érdi. *Retrocomputing in Clash: Haskell for FPGA Hardware Design*. Leanpub, Sept. 2021. URL: https://unsafeperform.io/retroclash/.

[53] G. Érdi. *Source Code for "Brainfuck on FPGA"*. 2021. URL: https://github.com/gergoerdi/clash-brainfuck.

[54] S. Lindley and C. McBride. "Hasochism: The Pleasure and Pain of Dependently Typed Haskell Programming". In: *SIGPLAN Not.* 48.12 (Sept. 2013), pp. 81–92. ISSN: 0362-1340. DOI: 10.1145/2578854.2503786. URL: https://doi.org/10.1145/2578854.2503786.

[55] K. Claessen and J. Hughes. "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs". In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*. ICFP '00. New York, NY, USA: Association for Computing Machinery, 2000, pp. 268–279. ISBN: 1581132026. DOI: 10.1145/351240.351266. URL: https://doi.org/10.1145/351240.351266.

[56] J. P. Pizani Flor and W. Swierstra. "Verified Timing Transformations in Synchronous Circuits with $\lambda\pi$-Ware". In: *Interactive Theorem Proving*. Ed. by J. Avigad and A. Mahboubi. Cham: Springer International Publishing, 2018, pp. 504–522. ISBN: 978-3-319-94821-8.

[57] E. Brady. *Type-driven development with Idris*. English. Manning Publications Co., Mar. 2017. ISBN: 978-1617293023.

[58] P. Fu, K. Kishida, and P. Selinger. "Linear Dependent Type Theory for Quantum Programming Languages". In: *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS '20. Saarbrücken, Germany, 2020, pp. 440–453. ISBN: 9781450371049. DOI: 10.1145/3373718.3394765. URL: https://doi.org/10.1145/3373718.3394765.

[59] P. Fu et al. "A Tutorial Introduction to Quantum Circuit Programming in Dependently Typed Proto-Quipper". In: *Reversible Computation*. Ed. by I. Lanese and M. Rawski. Cham: Springer International Publishing, 2020, pp. 153–168. ISBN: 978-3-030-52482-1.

[60] G. Peano. *Arithmetices principia: nova methodo*. Nineteenth Century Collections Online (NCCO): Science, Technology, and Medicine: 1780-1925. Fratres Bocca, 1889. URL: https://books.google.co.uk/books?id=z80GAAAAYAAJ.

[61] E. Brady, C. Mcbride, and J. Mckinna. "Inductive families need not store their indices". In: *Types for Proofs and Programs, Torino, 2003, volume 3085 of LNCS*. Springer-Verlag, 2004, pp. 115–129.

[62] B. Barras and B. Bernardo. "The Implicit Calculus of Constructions as a Programming Language with Dependent Types". In: *Foundations of Software Science and Computational Structures*. Ed. by R. Amadio. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 365–379. ISBN: 978-3-540-78499-9.

[63] H. Goguen, C. McBride, and J. McKinna. "Eliminating Dependent Pattern Matching". English. In: *Algebra, Meaning, and Computation*. Lecture Notes in Computer Science. Springer, 2006, pp. 521–540. ISBN: 9783540354628. DOI: 10.1007/11780274_27.

[64] E. Brady and K. Hammond. "A Verified Staged Interpreter is a Verified Compiler". In: *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*. GPCE '06. Portland, Oregon, USA: Association for Computing Machinery, 2006, pp. 111–120. ISBN: 1595932372. DOI: 10.1145/1173706.1173724. URL: https://doi.org/10.1145/1173706.1173724.

[65] W. Taha. "A Gentle Introduction to Multi-stage Programming". In: *Domain-Specific Program Generation: International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003. Revised Papers*. Ed. by C. Lengauer et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 30–50. ISBN: 978-3-540-25935-0. DOI: 10.1007/978-3-540-25935-0_3. URL: https://doi.org/10.1007/978-3-540-25935-0_3.

[66] O. Kiselyov, K. Swadi, and W. Taha. "A methodology for generating verified combinatorial circuits". In: *EMSOFT 2004 - Fourth ACM International Conference on Embedded Software*. Jan. 2004, pp. 249–258. DOI: 10.1145/1017753.1017794.

[67] W. A. Howard. "The Formulae-as-Types Notion of Construction". In: *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Ed. by H. Curry et al. Academic Press, 1980.

[68] B. C. Pierce et al. *Logical Foundations*. Ed. by B. C. Pierce. Vol. 1. Software Foundations. 2021. URL: http://softwarefoundations.cis.upenn.edu.

[69] D. B. Daniel P Friedman David Thrane Christiansen. *The Little Typer*. English. MIT Press, Sept. 2018. ISBN: 9780262536431.

[70] Z. Mou and P. Duhamel. "Fast FIR filtering: Algorithms and implementations". In: *Signal Processing* 13.4 (1987), pp. 377–384. ISSN: 0165-1684. DOI: https://doi.org/10.1016/0165-1684(87)90019-3. URL: https://www.sciencedirect.com/science/article/pii/0165168487900193.

[71] Y. Voronenko and M. Püschel. "Multiplierless Multiple Constant Multiplication". In: *ACM Trans. Algorithms* 3.2 (May 2007), 11–es. ISSN: 1549-6325. DOI: 10.1145/1240233.1240234. URL: https://doi.org/10.1145/1240233.1240234.

[72] Xilinx, Inc. *UG897 — Vivado Design Suite User Guide: Model-Based DSP Design Using System Generator (v2018,3)*. 2018. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug897-vivado-sysgen-user.pdf.

[73] P. Day et al. "A broadband superconducting detector suitable for use in large arrays". In: *Nature* 425 (Oct. 2003), pp. 817–21. DOI: 10.1038/nature02037.

[74] J. Pfau et al. "Reconfigurable FPGA-Based Channelization Using Polyphase Filter Banks for Quantum Computing Systems". In: *Applied Reconfigurable Computing. Architectures, Tools, and Applications*. Ed. by N. Voros et al. Cham: Springer International Publishing, 2018, pp. 615–626. ISBN: 978-3-319-78890-6.

[75] 3rd Generation Partnership Project (3GPP). *5G; NR; Base Station (BS) radio transmission and reception, TS 38.104 version 15.2.0 Release 15*. 2018. URL: https://www.etsi.org/deliver/etsi_ts/138100_138199/138104/15.02.00_60/%20ts_138104v150200p.pdf.

[76] Xilinx, Inc. *PG 269 — Zynq UltraScale+ RFSoC RF Data Converter v2.3*. 2020. URL: https://www.xilinx.com/support/documentation/ip_documentation/%20usp_rf_data_converter/v2_3/pg269-rf-data-converter.pdf.

[77] R. Lyons. *Understanding Digital Signal Processing*. Prentice Hall, 2011. ISBN: 9780137027415. URL: https://books.google.co.uk/books?id=arVImAEACAAJ.

[78] f. j. harris. *Multirate signal processing for communication systems*. CRC Press, 2022.

[79] Xilinx, Inc. *PG149 — FIR Compiler v7.2 LogiCORE IP Product Guide*. 2015. URL: https://www.xilinx.com/support/documentation/ip_documentation%20/fir_compiler/v7_2/pg149-fir-compiler.pdf.

[80] A. G. Dempster and M. D. Macleod. "Use of Minimum-Adder Multiplier Blocks in FIR Digital Filters". In: *IEEE Transactions in Circuits and Systems-II: Analog and Digital Signal Processing* 42.9 (1995), pp. 569–577.

[81] K. N. Macpherson and R. W. Stewart. "Low FPGA area multiplier blocks for full parallel FIR filters". In: *Proceedings. 2004 IEEE International Conference on Field- Programmable Technology (IEEE Cat. No.04EX921)*. Dec. 2004, pp. 247–254. DOI: 10.1109/FPT.2004.1393275.

[82] D. Parker and K. Parhi. "Area-efficient parallel FIR digital filter implementations". In: *Proceedings of International Conference on Application Specific Systems, Architectures and Processors: ASAP '96*. 1996, pp. 93–111. DOI: 10.1109/ASAP.1996.542805.

[83] Y.-C. Tsao and K. Choi. "Area-Efficient Parallel FIR Digital Filter Structures for Symmetric Convolutions Based on Fast FIR Algorithm". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 20.2 (2012), pp. 366–371. DOI: 10.1109/TVLSI.2010.2095892.

[84] Y.-C. Tsao and K. Choi. "Area-Efficient VLSI Implementation for Parallel Linear-Phase FIR Digital Filters of Odd Length Based on Fast FIR Algorithm". In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 59.6 (2012), pp. 371–375. DOI: 10.1109/TCSII.2012.2195062.

[85] A. Mayilavelane and B. Berscheid. "A Fast FIR filtering technique for multirate filters". In: *Integration* 52 (2016), pp. 62–70. ISSN: 0167-9260. DOI: https://doi.org/10.1016/j.vlsi.2015.07.011. URL: https://www.sciencedirect.com/science/article/pii/S0167926015000905.

[86] A. Kumar, S. Yadav, and N. Purohit. "Exploiting Coefficient Symmetry in Conventional Polyphase FIR Filters". In: *IEEE Access* 7 (2019), pp. 162883–162897. DOI: 10.1109/ACCESS.2019.2951706.

[87] Xilinx, Inc. *UG905 — Vivado Design Suite User Guide : Hierarchical Design (v2019.1)*. 2019. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug905-vivado-hierarchical-design.pdf.

[88] G. Smit, J. Kuper, and C. Baaij. "A mathematical approach towards hardware design". Undefined. In: *Dagstuhl Seminar on Dynamically Reconfigurable Architectures*. Ed. by P. Athanas et al. Dagstuhl Seminar Proceedings. eemcs-eprint-19169. Germany: Internationales Begegnungs- und Forschungszentrum für Informatik, Dec. 2010, p. 11. DOI: 10.4230/OASIcs.WCET.2010.136.

[89] M. Pickering, A. Löh, and N. Wu. "Staged Sums of Products". In: *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell*. Haskell 2020. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 122–135. ISBN: 9781450380508. DOI: 10.1145/3406088.3409021. URL: https://doi.org/10.1145/3406088.3409021.

[90] N. Xie et al. "Staging with Class: A Specification for Typed Template Haskell". In: *Proc. ACM Program. Lang.* 6.POPL (Jan. 2022). DOI: 10.1145/3498723. URL: https://doi.org/10.1145/3498723.

[91] K. Claessen and J. Hughes. "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs". In: *SIGPLAN Not.* 46.4 (May 2011), pp. 53–64. ISSN: 0362-1340. DOI: 10.1145/1988042.1988046. URL: https://doi.org/10.1145/1988042.1988046.

[92] C. Wolf. *Yosys Open SYnthesis Suite*. https://yosyshq.net/yosys/.

[93] A. Gill et al. "Introducing Kansas Lava". In: *Proceedings of the Symposium on Implementation and Application of Functional Languages*. Vol. 6041. LNCS. Springer-Verlag, Sept. 2009.

[94] P. Bjesse. "Automatic Verification of Combinational and Pipelined FFT Circuits". In: *Computer Aided Verification*. Ed. by N. Halbwachs and D. Peled. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 380–393. ISBN: 978-3-540-48683-1.

[95] E. Brady. "Resource-Dependent Algebraic Effects". In: *Trends in Functional Programming*. Ed. by J. Hage and J. McCarthy. Cham: Springer International Publishing, 2015, pp. 18–33. ISBN: 978-3-319-14675-1.

[96] E. Hogenauer. "An economical class of digital filters for decimation and interpolation". In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 29.2 (1981), pp. 155–162.

[97] J. Seamons. *CIC filter register pruning utility*. 2014. URL: http://www.jks.com/cic/cic.html.

[98] ARM. *AMBA® AXI™ and ACE™ Protocol Specification: AXI3™, AXI4™, and AXI4-Lite™, ACE and ACE-Lite™*. 2013. URL: https://documentation-service.arm.com/static/5f915b62f86e16515cdc3b1c.

[99] Heriot-Watt University. *Webpage for SPLV 20: Scottish Summer School on Programming Languages and Verification*. 2020. URL: http://www.macs.hw.ac.uk/splv/splv20/.

[100] T. Coquand. "An Analysis of Girard's Paradox". In: *Proceedings of the First Annual IEEE Symposium on Logic in Computer Science (LICS 1986)*. Cambridge, MA, USA: IEEE Computer Society Press, June 1986, pp. 227–236.

[101] S. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, Jan. 1987. URL: https://www.microsoft.com/en-us/research/publication/the-implementation-of-functional-programming-languages/.

[102] D. Miller. "Unification under a mixed prefix". In: *Journal of Symbolic Computation* 14.4 (1992), pp. 321–358. DOI: 10.1016/0747-7171(92)90011-R.

[103] A. Abel. "MiniAgda: Integrating Sized and Dependent Types". In: *Electronic Proceedings in Theoretical Computer Science*. Vol. 43. Dec. 2010, pp. 14–28. DOI: 10.4204/EPTCS.43.2.

[104] J. de Muijnck-Hughes. *Talk: Wiring Circuits is as easy as 0-1-Omega, or is it...* 2022. URL: https://jfdm.github.io/post/2022-05-31-Linear-Wirings.html.

[105] O. Kiselyov. "The Design and Implementation of BER MetaOCaml". In: *Functional and Logic Programming*. Ed. by M. Codish and E. Sumii. Cham: Springer International Publishing, 2014, pp. 86–102. ISBN: 978-3-319-07151-0.

[106] A. Kawata and A. Igarashi. "A Dependently Typed Multi-stage Calculus". In: *Programming Languages and Systems*. Ed. by A. W. Lin. Cham: Springer International Publishing, 2019, pp. 53–72. ISBN: 978-3-030-34175-6.

[107] S. L. P. Jones. "Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine - Version 2.5". In: *Journal of Functional Programming* 2 (1992), pp. 127–202.

[108] P. Hudak et al. "A History of Haskell: Being Lazy with Class". In: *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. HOPL III. San Diego, California: Association for Computing Machinery, 2007, pp. 12–1–12–55. ISBN: 9781595937667. DOI: 10.1145/1238844.1238856. URL: https://doi.org/10.1145/1238844.1238856.