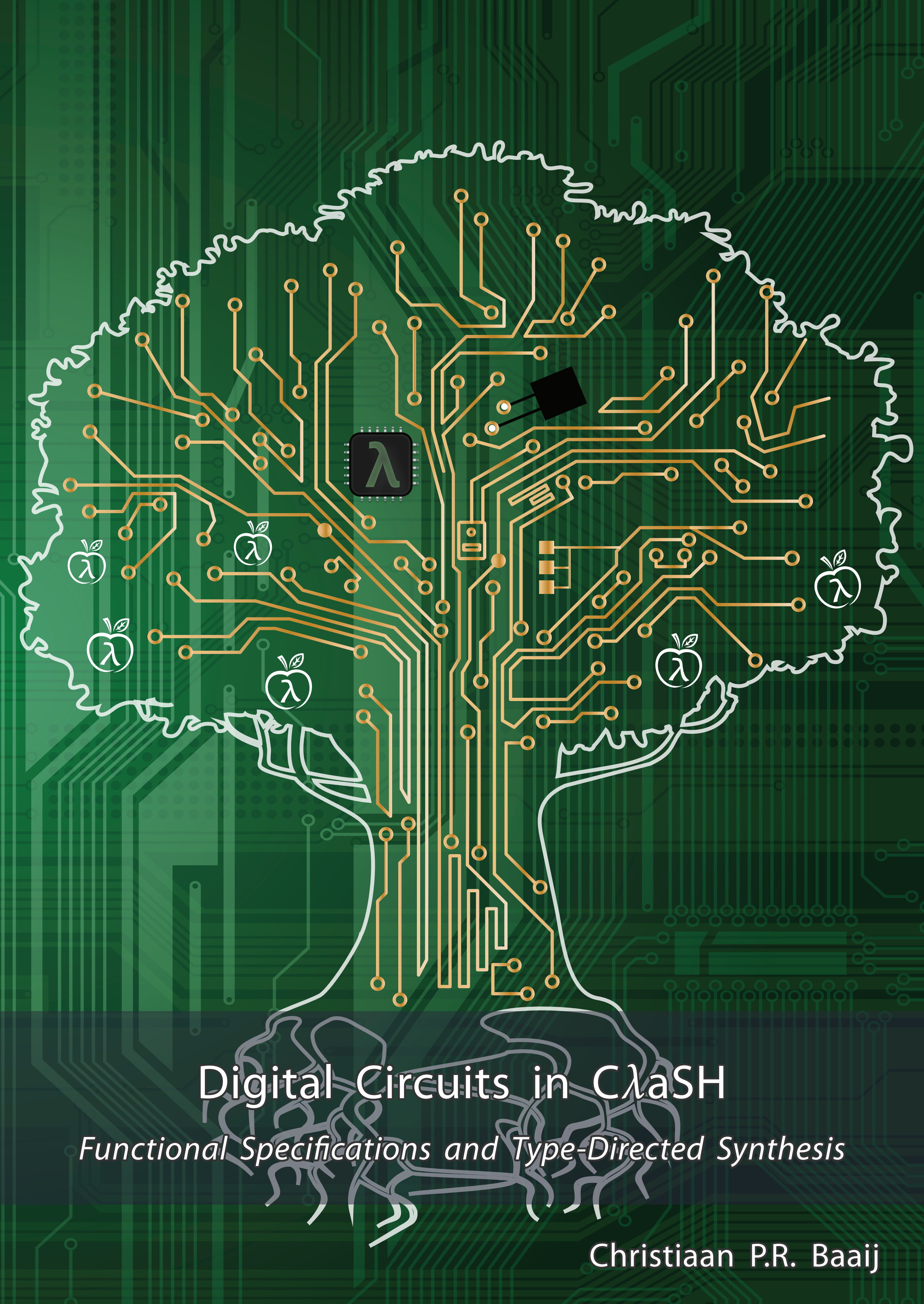


Digital Circuits in CλaSH

Functional Specifications and Type-Directed Synthesis

Christiaan P.R. Baaij

UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering, Mathematics and Computer Science, Computer Architecture for Embedded Systems (CAES) group

This research is conducted within the Service-oriented Operating Systems (S(o)OS) project (Grant Agreement No. 248465) supported under the FP7-ICT-2009.8.1 program of the European Commission.

This research is conducted within the Programming Large Scale Heterogeneous Infrastructure (Polca) project (Grant Agreement No. 610686) supported under the FP7-ICT-2013.3.4 program of the European Commission.

CTIT Ph.D. thesis Series No. 14-335
Centre for Telematics and Information Technology
University of Twente, P.O. Box 217, NL–7500 AE Enschede

This thesis was typeset using LaTeX 2_ε, Ti*k*Z, and Sublime Text. This thesis was printed by Gildeprint Drukkerijen, The Netherlands.

ISBN	978-90-365-3803-9
ISSN	1381-3617 (CTIT Ph.D. thesis Series No. 14-335)
DOI	10.3990/1.9789036538039

Digital Circuits in CλaSH

Functional Specifications and Type-Directed Synthesis

Proefschrift

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof. dr. H. Brinksma,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op vrijdag 23 januari 2015 om 14:45 uur

door

Christiaan Pieter Rudolf Baaij

geboren op 1 februari 1985
te Leiderdorp

Dit proefschrift is goedgekeurd door:

Prof. dr. ir.	G.J.M. Smit	(promotor)
Dr. ir.	J. Kuper	(assistent promotor)

ISBN 978-90-365-3803-9

Abstract

Over the last three decades, the number of transistors used in microchips has increased by three orders of magnitude, from millions to billions. The productivity of the designers, however, lags behind. Designing a chip that uses ever more transistors is complex, but doable, and is achieved by massive replication of functionality. Managing to implement complex algorithms, while keeping non-functional properties, such as area and gate propagation latency, within desired bounds, and thoroughly verifying the design against its specification, are the main difficulties in circuit design.

It is difficult to measure design productivity *quantitatively*; transistors per hour would not be a good measure, as high transistor counts can be achieved by replication. As a motivation for our work we make a *qualitative* analysis of the tools available to circuit designers. Furthermore, we show how these tools manage the complexity, and hence improve productivity. Here we see that progress has been slow, and that the same techniques have been used for over 20 years. Industry standard languages, such as VHDL and (System)Verilog, do provide means for abstractions, but they are distributed over separate language constructs and have ad hoc limitations. What is desired is a single abstraction mechanism that can capture most, if not all, common design patterns. Once we can abstract our common patterns, we can reason about them with rigour. Rigorous analysis enables us to develop correct-by-construction transformations that capture trade-offs in the non-functional properties. These correct-by-construction transformations give us a straightforward path to reaching the desired bounds on non-functional properties, while significantly reducing the verification burden.

We claim that functional languages can be used to raise the abstraction level in circuit design. Especially higher-order functional languages, where functions are first-class and can be manipulated by other functions, offer a single abstraction mechanism that can capture many design patterns. An additional property of functional languages that make them a good candidate for circuit design is purity, which means that functions have no side-effects. When functions are pure, we can reason about their composition and decomposition locally, thus enabling us to reason formally about transformations on these functions. Without side-effects, synthesis can derive highly parallel circuits from a functional description because it only has to respect the direct data dependencies.

In existing work, the functional language *Haskell* has been used as a host for *embedded* hardware description languages. An embedded language is actually a set of data types and expressions described within the host language. These data types and expressions then act like the keywords of the embedded language. Functions in the host language are subsequently used to model functions in the embedded language. Although many features of the host language can be used to model equivalent behaviour in the embedded language, this is not true for all features. One of the most important features of the host language that cannot directly be used in the embedded language, are features that model choice, such as pattern matching.

This thesis explores the idea of using the functional language Haskell *directly* as a hardware specification language, and move beyond the limitations of embedded languages. Additionally, where applicable, we can use normal functions from existing Haskell libraries to model the behaviour of our circuits.

There are multiple ways to interpret a function as a circuit description. This thesis makes the choice of interpreting a function definition as a *structural* composition of components. This means that every function application is interpreted as the component instantiation of the respective sub-circuit. Combinational circuits are then described as functions manipulating algebraic data types. Synchronous sequential circuits are described as functions manipulating infinite streams of values. In order to reduce the cognitive burden, and to guarantee synthesisable results, streams cannot be manipulated directly by the designer. Instead, our system offers a limited set of combinators that can safely manipulate streams, including combinators that map combinational functions over streams. Additionally, the system offers streams that are explicitly synchronised to a particular clock and thus enable the design of multi-clock circuits. Proper synchronisation between clock domains is checked by the type system.

This thesis describes the inner workings of our CλaSH *compiler*, which translates the aforementioned circuit descriptions written in Haskell to low-level descriptions in VHDL. Because the compiler uses Haskell directly as a specification language, synthesis of the description is based on (classic) static analysis. The challenge then becomes the reduction of the higher-level abstractions in the descriptions to a form where synthesis is feasible. This thesis describes a term rewrite system (with bound variables) to achieve this reduction. We prove that this term rewrite system *always* reduces a polymorphic, higher-order circuit description to a synthesisable variant. The only restriction is that the root of the function hierarchy is not polymorphic nor higher-order. There are, however, no restrictions on the use of polymorphism and higher-order functionality in the rest of the function hierarchy.

Even when descriptions use high-level abstractions, the CλaSH compiler can synthesize efficient circuits. Case studies show that circuits designed in Haskell, and synthesized with the CλaSH compiler, are on par with hand-written VHDL, in both area and gate propagation delay. Even in the presence of contemporary Haskell idioms and abstractions to write imperative code (for a control-oriented circuit), does the CλaSH compiler create results with decent non-functional properties. To emphasize that our approach enables correct-by-construction descriptions, we demonstrate abstractions that allow us to automatically compose components that use back-pressure as their synchronisation method. Additionally, we show how cycle delays can be encoded in the type-signatures of components, allowing us to catch any synchronisation error at compile-time.

This thesis thus shows the merits of using a modern functional language for circuit design. The advanced type system and higher-order functions allow us to design circuits that have the desired property of being correct-by-construction. Finally, our synthesis approach enables us to derive efficient circuits from descriptions that use high-level abstractions.

Samenvatting

Gedurende de laatste drie decennia is het aantal transistors in een processor met drie ordegroottes toegenomen, van miljoenen naar miljarden. De productiviteit van de ontwerpers loopt hier echter op achter. Het ontwerpen van een processor met telkens meer transistors is complex, maar doenlijk, en wordt bereikt door het veelvuldig kopiëren van functionaliteit. Het implementeren van complexe algoritmes, en het daarbij in toom houden van niet-functionele aspecten, zoals oppervlakte en propagatievertraging, en het zorgvuldig verifiëren van het uiteindelijke ontwerp, zijn de voornaamste moeilijkheden in het ontwerpen van digitale circuits.

Het is moeilijk om productiviteit van ontwerpers *kwantitatief* te bepalen; transistors per uur is geen goede maat, omdat hoge transistoraantallen kunnen worden bereikt door replicatie van functionaliteit. Als motivatie voor ons werk maken we een *kwalitatieve* analyse van de software die beschikbaar is voor ontwerpers van digitale circuits. Hierbij laten we zien hoe deze software helpt bij het beheersen van de complexiteit en dus de productiviteit verhoogt. We zien dan een geringe voortgang, waarbij dezelfde technieken al meer dan 20 jaar worden gebruikt. Talen die de standaard zijn in de industrie, zoals VHDL en (System)Verilog, verschaffen wel abstractiemogelijkheden, maar deze zijn verspreid over verschillende delen van de taal en hebben ad hoc beperkingen. Het is wenselijk om één abstractiemechanisme te hebben waarmee we veel, dan niet alle, ontwerppatronen kunnen uitdrukken. Wanneer we onze ontwerppatronen kunnen abstraheren, kunnen we er ook grondig over redeneren. Grondige analyses staan ons toe om inherent correcte transformaties te ontwerpen die afwegingen van niet-functionele eigenschappen uitdrukken. Omdat deze transformaties inherent correct zijn, is het mogelijk om tot een ontwerp te komen met de gewenste niet-functionele eigenschappen, zonder dat we extra verificatiestappen hoeven te ondernemen.

Wij beweren dat functionele talen zeer geschikt zijn om het abstractieniveau, van het ontwerpen van digitale circuits, naar een hoger niveau te tillen. Zeker hogereorde functies, waar functies andere functies kunnen bewerken, zijn geschikt als enkel abstractiemechanisme voor vele ontwerppatronen. Een andere eigenschap van functionele talen die ze geschikt maakt voor het ontwerpen van digitale circuits is dat functies vrij zijn van neveneffecten. Omdat functies geen neveneffecten hebben kunnen we op lokaal niveau redeneren over de compositie en decompositie van functies, en zodanig ook formeel redeneren over transformaties van deze functies. Vrij van neveneffecten, kan het syntheseproces zeer parallelle circuits afleiden van zo’n functionele beschrijving, omdat er alleen rekening gehouden hoeft te worden met directe afhankelijkheden.

In bestaand werk is er gekeken naar het gebruik van de functionele taal *Haskell* als kadertaal voor *ingebedde* hardwarebeschrijvingstalen. Zo'n ingebedde taal is eigenlijk een verzameling van datatypes en functies beschreven in de kadertaal, waar deze functies en datatypes dienen als trefwoorden van de ingebedde taal. Alhoewel vele aspecten van de kadertaal gebruikt kunnen worden om equivalente aspecten in de ingebedde taal uit te drukken, geldt dat niet zo voor alle aspecten van de kadertaal. Eén van de belangrijkste aspecten van de kadertaal die niet in de ingebedde taal gebruikt kan worden, zijn de aspecten die keuze uit kunnen drukken, zoals patroonherkenning.

Dit proefschrift verkent het idee om de functionele taal Haskell *direct* als hardwaresbeschrijvingstaal te gebruiken, zodat we niet meer onderhevig hoeven te zijn aan de beperkingen van ingebedde talen. Daarbij is het dan ook mogelijk, waar dat van toepassing is, om direct functies uit de standaardbibliotheken te gebruiken voor het beschrijven van digitale circuits.

Er zijn meerdere manieren om een functie als digitaal circuit te interpreteren. In dit proefschrift kiezen wij ervoor om functies te interpreteren als een *structurele* compositie van componenten. Dit betekent dat elke toegepaste functie wordt geïnterpreteerd als een nieuwe instantie van het overeenkomstige circuit. Combinatorische circuits worden beschreven als functies die algebraïsche datatypes bewerken. Synchroon sequentiële circuits worden beschreven als functies die oneindig lange reeksen van waarden bewerken. Om de cognitieve last te verlichten, en om synthetiseerbare resultaten te garanderen, kunnen zulke oneindige reeksen van waarden niet direct bewerkt kunnen worden de ontwerper. In plaats daarvan biedt het systeem een beperkte set van functies die de ontwerper toe staan de reeks op een bepaalde manier te bewerken, zoals een functie die elementsgewijs een combinatorische functie toepast op de reeks van waarden. Daarbij zijn er reeksen die expliciet zijn gekoppeld aan een specifieke klok, welk het mogelijk maakt om circuits te ontwerpen met meerdere klokken. Correcte overgangen tussen de klokdomeinen worden gecontroleerd door het typesysteem.

Dit proefschrift beschrijft de interne werking van de CλaSH *compiler*, welk eerdergenoemde circuitbeschrijvingen in Haskell omzet naar laag-niveau beschrijvingen in VHDL. Omdat de compiler Haskell direct als specificatietaal gebruikt, is synthese gebaseerd op (klassieke) statische analyse. De uitdaging zit dan in het reduceren van de hoog-niveau abstractiemechanismen die zich bevinden in de beschrijvingen naar een vorm waar synthese doenlijk is. Dit proefschrift beschrijft een termherschrijfsysteem (met gebonden variabelen) om deze reductie te bereiken. We bewijzen dat dit termherschrijfsysteem *altijd* polymorfe hogere-orde beschrijvingen van circuits reduceert naar een synthetiseerbare variant. De enige beperking is dat de functie bovenaan in de functiehiërarchie niet polymorf noch van hogere-orde is. Er zijn echter geen beperkingen in de rest van die functiehiërarchie wat betreft het gebruik van polymorfisme en hogere-orde functionaliteit.

Zelfs wanneer de beschrijvingen abstracties van een hoog niveau bevatten is de CλaSH compiler in staat hiervan efficiënte circuits te synthetiseren. Casestudies laten zien dat circuits die zijn ontworpen in Haskell, en gesynthetiseerd zijn met CλaSH, gelijkwaardig zijn aan circuits direct ontworpen in VHDL, zowel in grootte als in propagatievertraging. Ook wanneer eigentijdse Haskell idiomen worden gebruikt om imperatieve code (voor een controlegeoriënteerd circuit) te schrijven is de CλaSH compiler in staat om resultaten te genereren met degelijke niet-functionele aspecten. Om te benadrukken dat onze aanpak de gelegenheid geeft om inherent correcte beschrijvingen te ontwerpen, demonstreren wij abstracties die het mogelijk maken om circuits met elkaar te verbinden die tegendruk gebruiken als synchronisatiemethode. Ook laten we zien hoe klokslagvertragingen aan de typesignaturen van componenten kunnen worden toegevoegd, zodat we incorrecte synchronisatie tussen componenten al kunnen afvangen op het moment van ontwerpen.

Dit proefschrift laat dus zien waarom een moderne functionele taal zeer geschikt is voor het ontwerpen van digitale circuits. Het geavanceerde typesysteem en de hogere-orde functies maken het mogelijk om ontwerpen te maken die inherent correct zijn. Tenslotte zorgt onze syntheseaanpak ervoor dat we efficiënte circuits kunnen afleiden van beschrijvingen welke abstracties van een hoog niveau bevatten.

Dankwoord

November 2008, ik was op zoek naar een masteropdracht, januari 2015, ik ga promoveren. Zes jaar lang gewerkt aan hetzelfde onderwerp, waarvan het laatste jaar voornamelijk aan dit boekje. Ondertussen werken er al meerdere mensen, zelfs van buiten de vakgroep, met de software die er is geschreven, iets waar ik zeer tevreden over ben. Ook al geloof je in je eigen verhaal, geeft het toch een grote voldoening wanneer ook andere mensen jouw werk nuttig en interessant vinden.

Gedurende deze reis van zes jaar zijn er vele mensen die mij hebben geholpen met mijn werk, en nog belangrijker, ze hebben er voor gezorgd dat ik het altijd naar mijn zin heb gehad. Daarvoor wil ik hun graag bedanken.

Jan, voor de introductie tot de beste manier van programmeren, maar ook onze plezierige en uitgebreide discussies tijdens de reizen door heel Europa. Bij de eerste projectvergaderingen van SoOS had ik echt het gevoel alsof we daar niks hadden gedaan, maar daar wist jij dan altijd wel weer een positieve draai aan te geven. Nu weet ik inmiddels dat niet alles in twee dagen geregeld kan worden. Gerard, voor het zorgen voor een plek waar ik de kans kreeg om onderzoek te doen wat ik leuk vind, en, wat toch zeker heeft bijgedragen dat ik wilde gaan promoveren, dat je een groep hebt gecreëerd waar ik me als masterstudent volwaardig lid van de groep voelde.

Koen, een goed klankboord voor al jouw continue wiskunde problemen was ik nooit, maar het is wel altijd gezellig met jou op de kamer. Of je nu zelfs een gevatte opmerking maakt, of onbedoeld een opmerking maakt waar iemand anders een gevat weerwoord op heeft, zorg je altijd voor veel humor op de groep. Arjan en Philip, voor het helpen bij het oplossen van problemen van een zekere functionele aard. Gerald, voor de eerste verkenning van tijdsannotaties op de functionele beschrijvingen. Rinse, Peter, Ruud, Jaco, Erwin, hoewel de compiler natuurlijk altijd wel werkte op *mijn* computer met *mijn* voorbeelden, ben ik toch blij met de vele testcode en bugreports die door jullie zijn geleverd. Jochem, voor de interessante discussies over bitcoin en andere politieke en financiële wereldzaken. Marlous, Thelma, en Nicole, voor het regelen van hotels, vliegreizen, en nog zo vele andere zaken. Marloes voor een gezellige afsluiting van de dag wanneer we samen naar huis fietsen. Karel en Tom, voor de mooie gesprekken tijdens pauzes, borrels, en onder het gamen, en natuurlijk onze gedeelde waardering voor films met een hoog TSH[1] gehalte.

[1]Deze zal je niet terugvinden in de acronymenlijst.

Tenslotte, mijn geliefde Alexandra, voor het geduldig aanhoren als ik je terloops vertel dat ik de volgende dag voor een week weg ben voor conferentie, voor het vriendelijk herinneren dat de buren ook mijn geram op de toetsenbordplank kunnen horen, en het me bijstaan in vele achtereenvolgende weekenden toen ik doorwerkte aan dit boekje.

Christiaan
Enschede, december 2014

Contents

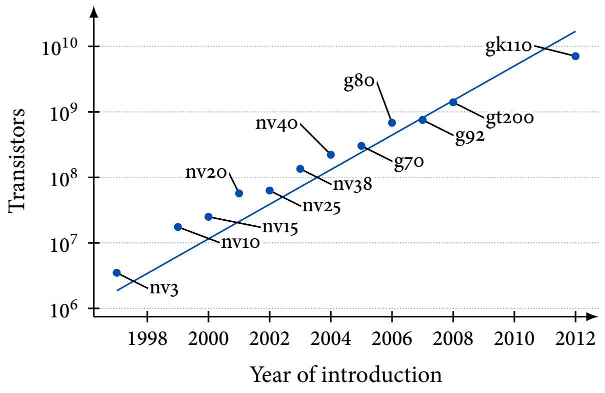
1 Introduction

In 1985[1], Intel released the 80386, a consumer-grade central processing unit *(CPU)* that had around 275.000 transistors. The Intel 80486, released 4 years later, was the first x86 CPU that crossed the 1 million transistor boundary. The largest available chip today, in terms of transistor count, is NVIDIA's GK110 GPU rounding out at about 7 billion transistors. Nearly three decades of technology scaling have thus increased the transistor count by three orders of magnitude: from millions to billions.

While transistor budgets grew by three orders of magnitude over three decades, it is much harder to determine whether the productivity of chip designer grew equally fast over the years. Figure 1.1 sets out the R&D budget of NVIDIA against the transistor count of their GPUs. We choose NVIDIA as their R&D is spent on a small product line, where the main product line is most likely taking up the largest part of their budget. If we would consider transistors per dollar spent as a measure for productivity, then NVIDIA's productivity is spectacular: while its R&D budget grows linearly, the number of transistors used in their GPU grows (almost) exponentially.

Such spectacular productivity growth is of course unlikely; it would have been widespread knowledge within the community if it would be true. Using the number of transistors as a measure for productivity is not a particularly good measure, these high transistor counts are achieved because GPUs are highly regular. GPUs fill their transistor budgets through replication: they consist out of hundreds, if not thousands, of identical cores. The same story holds for modern CPUs, for both mobile and desktop systems: they have multiple cores, sometimes in the double digits, and megabytes of cache memory. As replication is straightforward, the real complexity of these designs lies with their individual computational units and the composition of these units. When we would measure productivity in terms of transistors used for these individual units, the results are indeed not as spectacular.

[1]Chosen as a reference as it corresponds to the author's date of birth.



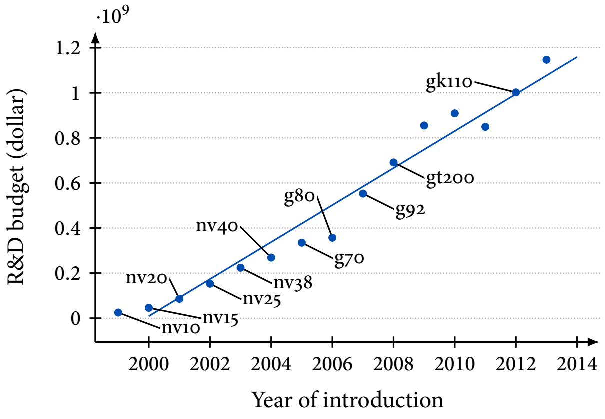


Figure 1.1 – NVIDIA: GPU transistors vs. R&D budget[2]

We can derive from the above that, measuring productivity *quantitatively* is not straightforward; actually, we are not aware of any measure in circuit design that can give a good indication for productivity. We can still, however, try to *qualitatively* determine how the tools and methodologies have improved productivity over the years, and find out where there is room for even further improvement. We will focus on the tools that help shape the design, and serve as the main implementation tools for digital circuits: hardware description languages *(HDLs)*.

[2]Transistor counts are copied from `http://en.wikipedia.org/wiki/Transistor_count#GPUs`. R&D budget are as reported on the annual 10-K reports (`http://investor.nvidia.com/sec.cfm`)

1.1 Hardware Description Languages

The two most commonly used HDLs, VHDL and Verilog, were introduced when industry shifted circuit design towards very-large-scale integration *(VLSI)*. At that time, these HDLs were used for the documentation and simulation of circuits that were *already* designed in a different format, for example with schematic capture tools. It is the advent of logic synthesis (and automated place & route) that really pushed VHDL and Verilog to the forefront of digital circuit design. Logic synthesis resulted in an incredible productivity boost compared to schematic capture tools and the manual layout process that were common practise until that time.

These logic synthesis tools work on register-transfer level *(RTL)* descriptions of a circuit. RTL describes a circuit in terms of the composition of the signals between registers, and the logical operations performed on those signals. In order to raise the abstraction level even further, and hence improve the productivity of circuit designer, the next step was to *just* describe the behaviour of the circuit, and *derive* an efficient structural description [43]. The two well-known approaches to facilitating better behavioural descriptions are:

» Extending and improving existing HDLs with features from modern programming languages, such as the object-oriented features of SystemVerilog (an extension, now successor, to Verilog).

» High-level synthesis (HLS) [13, 43] (or behavioural synthesis) of high level (programming) languages such as C or Java.

The purpose of high-level synthesis *(HLS)* is to transform a behavioural, often sequential, description of a circuit to an RTL description. HLS is not restricted to regular programming languages, it applies equally to the behavioural feature set of existing (and extended) HDLs. The code in listing 1.2 gives an RTL description of a finite impulse response *(FIR)* filter in VHDL. It is a fully parallel implementation. There is also one (purposefully included) performance issue: all multiplied values are added in a long chain, instead of using a tree of adders, leading to a longer combinational path than necessary.

The code in listing 1.1 gives a *behavioural* description of a FIR filter in C. The purpose of a HLS tool is to convert this behavioural description to an RTL description. It does not need to be a fully parallel implementation like the code in listing 1.2 though, it is also possible to map the description to a sequential implementation, one which contains only a single multiplier and a single adder. The process for determining whether the implementation should be fully parallel, fully sequential, or something in between, can either be done:

» Manually: the HLS tool provides mechanisms to, e.g., unroll and pipeline loops.

» Automatically: the HLS searches for an implementation that best fits the given size and latency restrictions.

```
void  fir_filter  (int16 *inp,  int16  coeffs [NUM_TAPS], int16 *outp) {
   static  int16  regs [NUM_TAPS];
  int32          temp = 0;
  int            i;

  for  (i  = NUM_TAPS-1; i>=0; i--) {
    if  (i  == 0)
      regs [i]  = *inp;
    else
      regs [i]  =  regs [i-1];
  }

  for  (i  = NUM_TAPS-1; i>=0; i--) {
    temp +=  coeffs [i]  *  regs [i];
  }

  *outp  = temp>>16;
}
```

Listing 1.1 – FIR Filter: Behavioural C description

For example, HLS tools can take the associativity of addition into account when summing the multiplied values and subsequently generate a tree of adder circuits automatically.

The uptake of higher-level languages for circuit design and verification in industry, be it a regular programming language or an extended HDL, is high. Use of SystemVerilog for verification and testing is considered common practise, especially in the ASIC design industry. Due to limited support from the synthesis tools, the higher level features of these HDLs are not used for the actual implementation description of a circuit. Uptake of HLS tools, such as C-to-Gates tools, is, however, much lower.

Early HLS tools, those introduced during the 1990's, showed a low adaptation for multiple reasons [41]: The quality of the generated hardware was much worse than hand-crafted designs, giving no incentive for RTL designers to switch. Also, these HLS tools focussed on the synthesis of behavioural descriptions in HDLs, instead of regular programming languages: the learning curve for these languages prohibited the adoption by algorithm designers. The (late) 2000's saw the (commercial) introduction of HLS tools that use the programming language C as the input specification language. Such tools include Catapult-C [8] and AutoPilot [77]. This significantly lowered the bar for algorithm designers and normal programmers to use these tools.

```
package types is
  type array_of_signed_16 is array (natural range <>)
                              of signed(15 downto 0);
  type array_of_signed_32 is array (natural range <>)
                              of signed(31 downto 0);
end;

entity fir is
  generic (NUM_TAPS : natural);
  port ( clk     : in  std_logic ;
         rstn    : in  std_logic ;
         inp     : in  signed(15 downto 0);
         coeffs  : in  array_of_signed_16 (NUM_TAPS−1 downto 0);
         outp    : out signed(15 downto 0));
end;

architecture rtl of fir is
  signal reg, reg_next : array_of_signed_16 (NUM_TAPS−1 downto 0);
  signal temp          : array_of_signed_32 (NUM_TAPS−1 downto 0);
begin
  -- register
  process (clk, rstn)
  begin
    if rstn = '0' then
      reg <= (others => (to_signed(0,16)));
    elsif rising_edge (clk) then
      reg <= reg_next;
    end if;
  end process;

  -- combinational logic
  reg_next <= inp & reg(NUM_TAPS−1 downto 1);

  mul_add_coeffs : for i in (NUM_TAPS−1) downto 0 generate
  begin
    mul_initial : if i = (NUM_TAPS−1) generate
      temp(i) <= reg(i) * coeffs (i);
    end generate;

    mul_add_rest : if i /= (NUM_TAPS−1) generate
      temp(i) <= temp(i+1) + (reg(i) * coeffs (i));
    end generate;
  end generate;

  outp <= temp(0)(32 downto 16);
end;
```

Listing 1.2 – FIR Filter: RTL VHDL description

Advances in compiler technology, and a focus on the digital signal processing *(DSP)* parts (instead of the control parts) within circuit designs, has resulted in a much higher quality of the hardware that is generated by contemporary HLS tools [12, 41]. That does not mean that arbitrary C programs can be converted to highly performing circuits: they almost always have to be altered so that the HLS tools can infer more parallelism. Also, although HLS tools are very good at extracting instruction- and loop-level parallelism from C programs, extracting task-level parallelism still requires manual annotation [12].

The problems that HLS tools face stems from the sequential, imperative, nature of the languages that are used for specification, and the parallel, immutable, nature of digital circuits. Even the most commonly used HDLs are based on languages that were created for sequential CPUs: VHDL is based on Ada, and Verilog on C. It thus makes sense to explore languages that are not created with a sequential platform in mind, and are hopefully better aligned with the parallel nature of digital circuits.

1.2 Functional Hardware Description Languages

The third, lesser travelled and lesser known, road to raising the abstraction level of circuit design is to use a programming paradigm that falls outside of the scope of *imperative* languages. The most studied, non-imperative, paradigm in the context of circuit design is *functional* programming. The tenets of functional programming are simply *function abstraction*, the creation of functions, and *function application*. Two other features often associated with functional languages are *purity* and *immutability*, where the two are actually closely related.

Purity is used to indicate that a function always returns the same result for an associated input; that is, the result is not influenced by side-effects, nor does a function produce any side-effects. As mutation is a side-effect, variables in pure functional languages are immutable. A variable in a functional language is thus akin to a variable in mathematics: a constant, yet unknown, value.

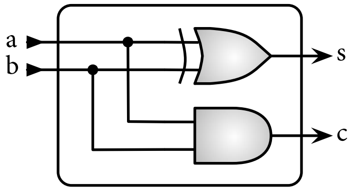
The *combinational logic* in a digital circuit is a logic function, in the mathematical sense, from its inputs to its output. The *pure* functions as those found in functional languages embody this function concept of mathematics. *Pure* functions are thus a perfect model for the *combinational logic* in digital circuits. The code in listing 1.3, describing a half adder circuit, serves as a small example to demonstrate the correspondence between functional descriptions and digital circuits.

Just like the mathematical function concept they embody, functions in functional languages are timeless: there is no notion of time that influences their behaviour. Circuits on the other hand have *propagation delays*: it takes time for a level change to propagate through a circuit. The retention behaviour of memory elements in *sequential logic* crucially depends on these propagation delays. So, although listing 1.4 is a good structural description of the combinational logic of an SR latch, the semantics of the description does not say anything about the propagation delays and hence the retention behaviour of the SR latch.

Structural description

```
halfAdder a b = (s,c)
  where
    s = xor a b
    c = and a b
```

Circuit

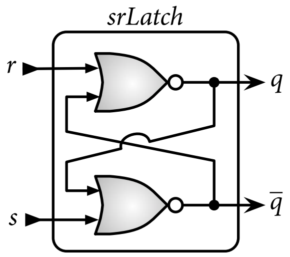


Listing 1.3 – Half adder

Structural description

```
srLatch r s = (q,nq)
  where
    q  = nor r nq
    nq = nor q s
```

Circuit



Listing 1.4 – SR Latch

Perhaps initially it seems that *pure* functions are thus a rather poor fit to *model* sequential logic. In the next subsection we will, however, show how sequential logic can still be captured intuitively in a functional language.

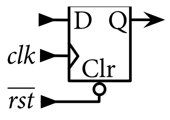
1.2.1 Sequential logic

Sequential logic in digital circuits can be divided into *synchronous* and *asynchronous* logic. In synchronous logic, all memory elements update their state in response to a clock signal. In asynchronous logic, memory elements can update their state at any time in response to a changing input signal. Although we can describe asynchronous sequential circuits in a functional language [2], in this thesis we

Behavioural description

```
dflipflop :: a    -- Initial (or reset) value
          → [a]   -- Input signal
          → [a]   -- Output: input signal where all samples are delayed
                  -- by 1 cycle
dflipflop i s = i : s -- place inital value in front of the incoming samples
```

Derived circuit



Listing 1.5 – D flip-flop

restrict ourselves to synchronous sequential logic.

The clock signal in synchronous logic is an oscillating signal that is distributed to all the memory elements such that they all observe its level change simultaneously. A crucial aspect of synchronous logic is that the interval of the clock signal must be long enough so that the input signals of the memory elements can reach a stable value. The time it takes for a signal to become stable is determined by the largest propagation delay between any two memory elements with no other memory element in between. The (combinational) logic between memory elements must hence be completely acyclic. Synchronous design allows a designer to abstract from propagation delays, and reason about state changes as if they happen instantaneously and synchronised.

Now that we can abstract away from propagation delays in synchronous sequential logic, it becomes more straightforward to model this sequential logic in a pure functional language. Where combinational logic can be modelled by functions that work on elementary values (booleans, integers, etc.), synchronous sequential logic can be modelled by functions that work on *streams* of elementary values. The elements in the stream correspond to the stable values for the consecutive clock ticks.

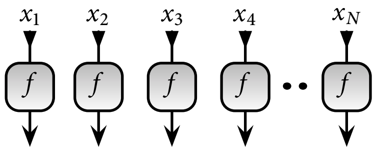
Memory elements can now be modelled as functions that add elements to the head a stream (see listing 1.5): given an stream of values *s*, adding a value *i* to the head results in a new stream, *s'*, in which every value in *s* is *delayed* by one clock cycle. Values calculated at time *t* are now available at time *t+1*. Directly working with streams can be confusing, and can lead to anti-causal descriptions (by dropping values from the stream); it is thus safer to only expose a set of primitives for stream manipulation. This aspect will be elaborated further in chapter 3.

Until now we have only discussed how to *model* sequential logic in a functional language. That doesn't mean, however, that all functional language based approaches

Haskell code

```
map f []      = []
map f (x:xs)  = f x : map f xs
```

Structural view



Listing 1.6 – *map*: parallel composition of a unary function

to hardware design need explicit descriptions of sequential logic. In chapter 2 we will see approaches where functions are a purely behavioural description, and the synthesis tool will infer, or generate, sequential logic where appropriate.

1.2.2 Higher level abstractions

While the semantic match between functional languages and digital circuits is a great technical feature, it does not directly offer the higher-level abstractions needed by hardware engineers to be productive. Where other high-level HDLs get their new design abstractions from the object-oriented programming paradigm, such as classes and interfaces in SystemVerilog, *functional* HDLs gain their high level of abstraction from their straightforward manipulation of *functions*. These so-called higher-order functional languages have functions that can receive *functions* as their arguments, or return *functions* as a result.

Higher-order functions allow many forms of design abstraction. One example is, of course, parametrising parts of the functionality of a circuit description. More generally, it is possible to capture certain design and recursion patterns as a function; where the latter are called *recursors*. One such *recursor* is the *map* function, shown in listing 1.6, which takes two arguments, a function f and a list *xs*, and applies the f to all elements in *xs*. When we take a structural view of the *map* function (bottom part of listing 1.6), we see that application of *map* to a concrete function f translates to a parallel composition of the circuit f. Aside from parallel composition, higher-order functions can capture many more connection and composition patterns commonly found in digital circuits. Further benefits of higher-order functions and *recursors* will be discussed in greater detail in chapter 3.

Another abstraction found in functional languages is polymorphism, where a function is not tied to a fixed type for every argument, but can work on arguments of any type. Combined with strong static typing and extensive and principled type *inference*, designers can write functions that are:

- » Reusable and parametric: due to polymorphism.
- » Correct: due to strong, static, typing.
- » Concise: due to the absence of type annotations, as types are inferred.

1.2.3 Challenges in synthesising functional HDLs to circuits

We have seen that the semantics of pure functional languages match the semantics of combinational logic when we have functions which process elementary objects, and of sequential logic when we have functions which process streams. Given that there is such a semantic match, synthesis from descriptions made in a functional language to a low-level format, such as a netlist, should thus be straightforward. While this is true for simple functions, synthesis of functions that use higher-level abstraction mechanisms is more difficult. We highlight the synthesis difficulties using the *map* function of listing 1.6 as an example:

- » The *map* function is polymorphic, so we cannot trivially determine how many wires are needed to connect all the components.
- » The *map* function is higher-order, its first argument is a function. We cannot encode functions as bits that flow through wires.
- » The *map* function is recursive, which is problematic when you view function definitions as structural descriptions of a component. Under such an approach, recursive function applications will be synthesized to self-instantiation of a component. This in turn leads to, unrealisable, infinite structures.

The exact synthesis of functional languages as proposed in this thesis, and further elaboration of the challenges and their solutions, will be described in chapter 4.

Aside from the theoretical challenges of synthesising higher-order and recursive descriptions, there is also the practical burden of implementing the actual simulation and synthesis tools. Especially in the academic setting this has resulted in incomplete toolsets. One popular approach to alleviate the implementation burden is to create an *embedded* domain specific language *(DSL)* for circuit design, which is the approach taken by, for example, the Lava HDL [7]. An embedded DSL is, as the name suggests, not a stand-alone language, but actually a library defined within a general purpose language. An embedded language has the syntax of the host language, where the data types and functions of the DSL library act as a new set of keywords.

Synthesis for these embedded languages works in a non-standard way, where the standard way would be performing a static analysis of the source code. The library functions and data types in an embedded language are actually small, composable, circuit generators. Simply executing the top-level function of the design within the host language will generate the complete circuit. One technical difficulty is that these circuit generators will, in the presence of feedback loops, generate infinite trees, which have to be folded back into a graph structure [24]. One deficit of the embedded language approach is that not all of the (desirable) features of the host-language can be used for circuit description. Most importantly, the choice-constructs (such as case-statements) of the host language cannot be used to describe choice-constructs in the eventual circuit; we will elaborate why in chapter 2. A designer will have to use one of the choice-functions offered by the embedded DSL library; which are often inferior in terms of expressibility compared to those offered by the host language.

1.3 Research questions

The main goal of this thesis is to *further improve the productivity of circuit designers*. As shown in the previous sections, there are multiple avenues we could explore in order to achieve higher productivity. In this thesis we chose to further explore the domain of functional hardware description languages, due to the semantic match between functional languages and digital circuits, and the high-level abstraction mechanisms available in functional languages. Being more productive is, however, not just achieved by being able to abstract functionality, we also need:

» To be able to express common idioms in circuit design straightforwardly.

» Decrease the amount of time spent on the verification of circuit designs.

» Reason confidently about non-functional properties, such as chip area and gate propagation delays.

This thesis therefore seeks answers to the following questions:

» How can functional languages be used to express both combinational *and* sequential circuits idiomatically?

» How can we support correct-by-construction design methodologies using a functional language?

» How can we use the high-level abstractions without losing performance, and have a straightforward cost model?

1.4 Approach and contributions of the thesis

In a previous section we described the use of *embedding* in order to create a new HDL, but then also highlighted that the embedded approach has its own problems. Instead of either embedding a HDL in a functional language, or creating a completely new language from scratch, this thesis explores the idea of using an existing functional language *directly* for the purpose of circuit description.

This thesis makes the choice of using the functional language *Haskell* for circuit design. We choose Haskell because of the many abstractions offered by its expressive type-system, polymorphism, higher-order functions, and pattern-matching constructs. Haskell's extensive type-derivation and near lack of syntax and keywords additionally leads to readable and concise circuit descriptions. Although there are other functional languages which have very similar properties, we specifically choose Haskell because:

- It is a *pure* functional language, meaning that it has *pure* functions, which, as mentioned earlier, map very well to combinational logic.
- It has a non-strict semantics, meaning that arguments to a function are only evaluated when their value is needed; the advantages of which are described in chapter 3.

Also, instead of creating a complete toolset from scratch, we adapt an existing Haskell compiler. We start with the existing Glasgow Haskell compiler *(GHC)* [64] and its associated libraries and tools. We extend the set of libraries with a library that has circuit-specific data types and functions, such as: arbitrary-width integers, registers, etc. Since our circuits are just Haskell programs, simulation is done in GHC by either:

- Applying a circuit description to its inputs within the GHC Haskell interpreter, or, if extra simulation speed is desired,
- Compiling the circuit description, together with its inputs, into an (optimized) executable, and execute the compiled program.

Aside from having designed a library for circuit design, we have also created a synthesis tool that converts the Haskell descriptions to low-level, synthesisable, VHDL. Also for this synthesis tool we can reuse large parts of GHC, which exposes its internals as a library. Our efforts mainly focussed on the synthesis of GHCs intermediate language, which is much smaller than Haskell. We used the GHC library functions for parsing and type checking.

One advantage of embedded DSLs not explicitly discussed earlier is that the evaluation mechanism of the host-language eliminates all high-level abstractions, such as higher-order functions. This means that the embedded DSL implementer does not have to deal with the synthesis of these abstractions. By choosing a standard synthesis approach based on static analysis for this thesis, *we* do, however, have to deal with the synthesis of these abstraction mechanisms explicitly.

Contributions

For the synthesis of these higher-level abstraction mechanisms, we chose an approach which is classic in the compilation of functional languages: compilation-by-transformation. In compilation-by-transformation, source-to-source transformations are applied exhaustively until the description has such a shape that a mapping to the target architecture is straightforward. Existing approaches are designed with instruction-set machines in mind: directly mapping their output to digital circuits would lead to highly inefficient circuits. We will elaborate on these inefficiencies in chapter 4. This thesis explores a term rewrite system *(TRS)*, a specific form of compilation-by-transformation, that removes abstraction mechanisms from a description that have no direct mapping to a digital circuit, but without introducing any inefficiencies.

This thesis is a continuation of the work done in [4] and [38], which resulted in the original prototype for the synthesis tool and circuit library: "CAES language for synchronous hardware *(CλaSH)*". We want to note that, from now on, we will refer to the triple: Haskell, our library for circuit design, and our synthesis tool, as the CλaSH language. This thesis improves upon [4] and [38] by providing a better approach for the *composition* of sequential circuit specifications, which we will discuss in chapter 3. Additionally, the rewrite system described in chapter 4 can correctly synthesise a larger class of specifications than the system described in [38], and also comes with a correctness proof.

1.5 Structure of the thesis

The next chapter starts with an overview of a select number of hardware description languages, focussing mostly on industrially used languages such as VHDL and Verilog, and on *functional* HDLs. The chapter will highlight the merits and disadvantages of the individual languages, the details of their synthesis (and problems therein), and compare them to the CλaSH language.

The subsequent chapter, chapter 3, describes the CλaSH language in greater detail. It highlights how the abstraction mechanisms in functional languages are highly beneficial in the creation of high-level, parametric, circuit designs. One important aspect discussed in length is how CλaSH deals with the concept of *state*. Additionally, we make our case for basing CλaSH on a *non-strict* language, as opposed to a *strict* language.

In chapter 4 we delve into the aspects of the synthesis from CλaSH to netlist-level VHDL. We discuss both the general setup of the CλaSH compiler, and in greater depth the term rewrite system (TRS) that removes abstractions such as higher-order functionality. The chapter highlights the importance of *types* in synthesis, and how they guide the synthesis process. Correctness of the transformations, completeness of the system (that all abstractions with no counterpart in a digital circuit are removed), and termination of the CλaSH compiler, are important aspects, and are discussed in this chapter.

Usability and effectiveness of the CλaSH language and compiler are demonstrated in chapter 5 using several mid-size circuit designs. These designs cover both data and control oriented aspects found in digital circuits.

Finally, this thesis concludes with chapter 6, where we discuss and summarise what we have achieved by building the CλaSH language and compiler. Specifically, we will address the advantages and disadvantages of using a general-purpose functional programming language Haskell as a starting point for a HDL. The chapter ends with recommendations for further research.

2

Hardware Description Languages

Abstract – *In order to increase productivity, hardware description languages must have the ability to abstract common idioms and patterns. Over the years, conventional hardware description languages have acquired more methods for abstraction, but these new aspects are sometimes non-trivial to use or are limited in scope as to what they are able to abstract. New languages have more powerful abstraction mechanisms, but as a result, their synthesis to RTL has become more complex, and is in certain situations limited. These limitations in synthesis also limits the expressivity of the designer. We compare the abstraction capabilities of existing hardware description languages, and their respective limitations, and elaborate where CλaSH either makes improvements or makes a different trade-off.*

2.1 Introduction

There are many description languages for hardware, both analogue and digital, and their introduction and revision dates span several decades. In the context of this thesis we will, however, focus on languages for synchronous, digital, circuit design; or at least those languages of which their synthesis tools produce a synchronous digital circuit. We narrow the overview of HDLs and their comparison with the CλaSH language even further to those languages that are currently accepted in industry (such as Verilog), and existing *functional* HDLs. The comparison with the industrially accepted languages is there to warrant the research into new HDLs in general, where the comparison with functional HDLs is there to demonstrate

Parts of this chapter have been published in [CB:7] and [CB:13].

that CλaSH captures a new and relevant point in the design space in the field of functional HDLs in particular.

For the languages such as VHDL and Verilog we describe the design abstraction available, and which parts of these languages are synthesisable. As CλaSH distinguishes itself as a new point in the design space of functional HDLs, we will describe these functional languages in more detail. Also their synthesis is discussed in more detail, as this aspect usually plays an important role (and not an afterthought as it was for VHDL) in the features available in these languages.

2.2 Standard hardware description languages

With *standard* languages we mean HDLs that are commonly used in industry, taught in courses on digital design, and have support in tools from multiple vendors. These languages are: VHDL, Verilog, and by extension SystemVerilog.

2.2.1 VHDL

VHDL has several abstractions available that allow for parametric and generative circuit design: *generics* (c.f. listing 2.1) and *configurations* on the parametric side, and *generate statements* (c.f. listing 2.2) on the generative side. This section only gives a short overview of these language features to demonstrate the means of abstraction in VHDL. Completely elaborating these features falls outside the scope of this thesis, and we refer the reader to works such as [3] for further details.

Parametrisation

In VHDL, design entities can be parametrised by certain *constant* values using *generics*. As of VHDL-2008 [34], the generics have been extended to: type, function, and package generics. Type generics basically added a form of polymorphism to the VHDL language, where function generics add higher-order functionality. An example of a polymorphic, higher-order, entity is shown in listing 2.1. There are several caveats to these new generics:

» Support for VHDL-2008, especially for the new generics, is either non-existent or fairly limited in synthesis tools[1].

» Functions only support the sequential subset of VHDL, not the concurrent one. There is hence no means to parametrise a component in concurrent logic using generics, a designer must use configurations for this.

» Explicitly mapping every type generic is tedious and error-prone, especially when compared to type-inference which is prevalent in functional languages.

[1] At the time of this writing, the only synthesis tool that we have found to fully support type and function generics is: Synopsys Synplify(Pro/Premier), version I-2013.09-1

```
entity incrementer is
  generic (type data_type;
           function increment (x: data_type) return data_type);
  port (inp  : in data_type;
        outp : out data_type;
        inc  : in std_logic );
end;
architecture rtl of incrementer is
begin
  outp <= increment(inp) when inc = '1';
end;
```

Listing 2.1 – Type and Function Generics

Aside from generics, there are also *configurations* as a means for parametrisation. Using configurations, declared component interfaces can be instantiated to different design architectures. This can be performed globally using a configuration *declaration*, or locally, using a configuration *specification* in the declarative part of e.g. a *block* declaration. Where configuration *declarations* can be used to configure any instantiated component in the design hierarchy, configuration *specifications* can only be used to configure components in the same scope as the configuration specification.

A disadvantage of configurations and component declaration is that this configurability, unlike generics, is not visible at the interface of a design, its entity declaration. You cannot pass a configuration from one component to the other; whereas generics can be passed from one component to the other. This makes configurations highly non-modular, they are *only* useful in the context of a complete design hierarchy.

The verbosity of generics and configurations (and perhaps VHDL in general) makes these features under-used. Having two feature-incomplete, instead of just one feature-complete, constructs for parametric design is also a disadvantage. For example, it would be preferable to have *component* generics (and a deprecation of configuration specifications) in a future version of VHDL, so that parametrisation is captured by a single concept: *generics*. Additionally, there is a disparity as to where these parametrisation features can be used: where entities can have function generics, functions themselves cannot have any kind of generics.

Higher-order functional HDLs, such as CλaSH, enable parametrisation by having functions as both arguments and result. As functions are the only abstraction mechanism, there is no feature disparity either. Additionally, type-inference ensures that we have polymorphism without explicitly propagating type annotations through our design – while still maintaining type safety.

Iterative generation: for ... generate

```
gen_label : for index in static_range generate
begin
  ...
end generate;
```

Conditional generation: if ... generate

```
gen_label : if boolean_expression generate
begin
  ...
end generate;
```

Listing 2.2 – Iterative and Conditional generation in VHDL

Generate statements

VHDL has *generate* statements that facilitate the iterative and conditional compile-time generation of other *concurrent* statements (ref. listing 2.2); where concurrent statements include things like: signal assignment and component instantiation, but also other *generate* statements. The range, for iterative generation, and the boolean expression, for conditional generation, must be *static*: completely reducible at compile- / elaboration-time. Enforcing a static range or expression is achieved by restricting the construction of the range expression or boolean expression: variable, port, and signal name references are not allowed.

The sequential parts of VHDL, *functions*, *procedures*, and *processes*, also contain *for*-loops and *if*-statements. Synthesis tools *often* elaborate these statements exhaustively, completely un-rolling *for*-loops and removing unchosen branches in *if*-statements. As such, the *for*-loops and *if*-statements could be seen as the *generative* part of VHDL for *sequential* statements; where the earlier discussed *generate* structures are there for the *concurrent* part of VHDL. Unlike the range expressions and boolean expressions in *generate* statements, static reducibility in the for-loops and if-statements is, of course, not enforced as part of the semantics of VHDL.

2.2.2 Verilog

This subsection, and the next on SystemVerilog, only give a short overview of the abstraction mechanisms available in these languages. For a complete elaboration of the details of these language features, we refer the reader to works such as [63].

Verilog [33] has abstractions for parametric and generative designs that are similar in nature to VHDL. Where VHDL has *generics*, Verilog has *parameters*. Like VHDL prior to the 2008 incarnation, parameters can only parametrise constants in the design, *not* functionality or types. However, unlike VHDL, Verilog allows *parameters* in *all* design entities: *modules*, *functions*, and *tasks*. Although it should be noted that *tasks* and *functions* in Verilog can only exist within a *module*, and are *not* top-level design entities; *functions* in VHDL *are* top-level design entities. Verilog also has *configurations*; however, where VHDL allows *configuration* specifications within an architecture, Verilog only supports configurations as a top-level construct.

Generative constructs, in the form of *generate* blocks, support both conditional and iterative generation. Aside from boolean conditions, Verilog also supports *case*-statements as conditional generation blocks.

Being related to the C programming language, Verilog also has compile-time *macros* through a pre-processor. Using `‘define` and `‘ifdef...‘else...‘endif`, code can be conditionally synthesised; and could hence be classified as a (conditional) generative construct of Verilog. An advantage of *macros* over *generate* blocks is that *macros* can be used outside of a module definition, e.g. to conditionally generate a module interface. An advantage of *generate* blocks is that they enable two different instances of the same module to be configured individually.

2.2.3 SystemVerilog

SystemVerilog is a proper extension to Verilog, and since 2009 the two languages are merged into the IEEE standard 1800-2009; there is now only SystemVerilog. SystemVerilog extends Verilog parameters with *type* parameters, hence supporting polymorphic designs. Support for these *type* parameters is present in both FPGA and ASIC tooling. Unlike VHDL-2008, there are no *function* or *task* parameters. This does not mean that functionality cannot be abstracted: SystemVerilog introduces a new design element called an *interface*.

An *interface* can bundle, aside from ports and wires, functionality in the form of *functions*, *tasks*, and *procedural blocks*. Unlike *modules*, *interfaces* can be made into ports; for both *modules* and *interfaces* themselves. These interface ports can also be *generic*, meaning that the choice for a concrete interface is deferred to when a module (or higher-level interface) is instantiated. Listing 2.3 showcases all of the above points. The interface *map_i* has a *generic* interface port, *f*. Nota bene, the interface *f* should have a task or function called *run*, which is called on line 6. The *map_i* interface is hence *parametrised* over the *run* task, or function, offered by the interface *f*. Finally, on line 17, a concrete instance of the *addOne_i* interface is created, which is subsequently passed to a concrete instance of the *map_i* interface on line 22.

Although the presented SystemVerilog code is certainly not idiomatic, ASIC synthesis tools are able to generate a netlist for Listing 2.3. The presented technique does not facilitate the abstraction over all the types of behaviour in SystemVerilog. Tasks and functions only allow a subset of SystemVerilog within their bodies: for example, tasks cannot have *procedural blocks* such as *always_comb*. Further investigation is the higher-order possibilities of SystemVerilog are hence warranted.

```
interface map_i #( parameter N=32, parameter type ELEMTYPE=logic )
                 ( interface f );
 task automatic run ( input  ELEMTYPE arg [N−1:0]
                    , output ELEMTYPE res [N−1:0]);
   for ( int mapIter=0; mapIter < N; mapIter+=1)
     f.run(arg[mapIter], res [mapIter]) ;
 endtask
endinterface

interface addOne_i;
   task automatic run (input integer a, output integer b);
     b = a + 1;
 endtask
endinterface

module top (input integer in [7:0], output integer out [7:0]);
 addOne_i addOne(); // create instance of 'addOne_i' interface
 // create intance of 'map_i' interface where:
 //   * parameter N is set to the size of 'in'
 //   * parameter ELEMTYPE is set to ' integer '
 //   * the inferface port is instantiated with 'addOne'
 map_i #(.N($size(in)), .ELEMTYPE(integer)) map(addOne);
 always_comb
   map.run(in,out);
endmodule
```

Listing 2.3 – Higher-Order SystemVerilog

2.2.4 BlueSpec SystemVerilog

BlueSpec SystemVerilog *(BSV)* [49] is a hardware description language with a syntax similar to SystemVerilog [35]. It is a high-level language that features *guarded atomic transactions* to model complex concurrent circuits. A transaction only starts when the assertion of its corresponding *guard* holds. The *atomicity* aspect says that individual transactions can be reasoned about as if they exist in isolation, even though multiple transactions are actually run concurrently. There are both *implicit* and *explicit* guards, the *explicit* guards are the ones added by a designer, where the *implicit* guards are added by the compiler, for aspects such as access to a memory.

BSV has both polymorphic typing and higher-order functions. Unlike for example type generics in VHDL-2008, BSV does not require explicit type assignments, these assignments are inferred. As opposed to Haskell, almost all declarations[2], whether

[2] Method definitions implementing a specified *interface* do not need any type annotations.

```
fun mult(x, y, acc) =
      if (x=0 | y=0) then acc
         else mult(x<<1, y>>1, if y.bit0 then acc+x else acc)
```

Listing 2.4 – Shift-Add Multiplier in SAFL [47]

they are variables, functions, or any other construct, do have to be annotated with a type; in Haskell even declarations can have their type inferred. Whether this is a restriction incurred by either the syntax or the underlying type-inference algorithm is unclear.

Synthesis

The synthesis from a BSV description to RTL-level Verilog is performed in two stages, which corresponds to the static and dynamic semantics of the language:

» A description is partially evaluated according to the static semantics, this includes the elimination / propagation of higher-order functions.

» The resulting description after partial evaluation is actually a set of rewrite rules. The second synthesis transformation instantiates all these rules in parallel, and adds scheduling logic in case there are conflicting preconditions [31].

2.3 Functional Languages

This section describes the features of existing functional hardware description languages. It provides a more detailed account of the synthesis of these languages, as it influences their expressivity in certain cases, and because synthesis is an important aspect of this thesis.

2.3.1 Conventional Languages

SAFL

SAFL [47] presents itself as a *Statically Allocated Parallel Functional Language*. Although the name alludes to SAFL being a general purpose functional language, the only existing compiler [59] produces solely RTL-level Verilog. The *Statically Allocated* aspect of SAFL refers to its *unique* feature that the size of *the text of the program* fully determines the size of the circuit. This very aspect is achieved by instantiating SAFL functions as a circuit *at most once*. Multiple function calls, including recursive calls, hence do *not* lead to multiple instantiations of the same component, a single instance will be accessed through multiplexers and arbiters. Primitive functions and operators are, however, duplicated.

Calls to f *are serialised*

```
fun f x = ...
fun main(x,y) = g(f(x),f(y))
```

Duplication of f *leads to parallel execution*

```
fun f  x = ...
fun f' x = ...
fun main(x,y) = g(f(x),f'(y))
```

Listing 2.5 – Serialised calls vs. Parallel execution through duplication [47]

The SAFL example in listing 2.4, copied from [47], shows the definition of a shift-add multiplier; it highlights the effect of being *statically allocatable*. The recursive call of *mult* will not introduce a static expansion of the logic of *mult*, but will instead lead to a (delayed) feedback loop (including the necessary control and arbitration logic).

Static allocation causes function calls to be serialised, even when they are independent. To increase the level of parallelism, a function can be duplicated, and the independent calls can refer to a unique duplicate. An example of this is shown in listing 2.5. The consequence of this duplication is of course an increase in size (by the size of *f*). Similar transformations can be (mechanically) applied to the shift-add multiplier of listing 2.4 to double the amount of work per clock cycle, at the cost of increasing the size of the circuit (although the size of the arbitration logic would stay the same).

The SAFL language has several restrictions, some of which are due to being statically allocatable. SAFL uses recursion to model feedback, but this recursion is limited to *tail*-recursion only. Having only tail-recursion means that no additional memory facilities are needed to store intermediate results. Higher-order functions are also not supported for similar reasons, higher-order functions introduce the risk of needing an infinite store. It is possible to restrict the use of higher-order functions which would not introduce these storage implication, but they are not implemented; see [47] for more details. SAFL is also restricted in the available data types, it only has integer-values (of a specifiable bit-width) and labelled product types (also known as records).

Verity

Verity [22] is a functional hardware description language which, like SAFL, describes circuits *behaviourally*. It features (synthesis) support for higher-order functions, recursion (using a fixed-point combinator called *fix*), and mutable references. The synthesis scheme behind Verity is described in a series of papers called *Geometry of Synthesis (GOS)* [19–21, 23]. Verity has an underlying *affine* type system;

Fixed-point combinator

A fixed-point combinator is a higher-order function *fix* that satisfies the equation:

```
fix f = f (fix f)
```

It is so named because, by setting $x = \mathit{fix}\ f$, it represents the solution to the fixed point equation:

```
x = f x
```

As a simple demonstration, we first present the recursive definition of the factorial function:

```
fact n = if n == 0 then 1 else n * fact (n − 1)
```

and then using a fixed-point combinator:

```
fact = fix fact'
fact' f n = if n == 0 then 1 else n * f (n − 1)
```

Nested - Allowed

$\lambda f\ g\ x.\ f\ (g\ x)$

Nested - Disallowed

$\lambda f\ x.\ f\ (f\ x)$

Parallel - Allowed

$\lambda f\ g\ x\ y.\ f\ x\ ||\ g\ y$

Parallel - Disallowed

$\lambda f\ x.\ f\ x\ ||\ f\ x$

Sequential - Allowed

$\lambda f\ x.\ f\ x\ ;\ f\ x$

Listing 2.6 – Affine typing - allowed identifier use

in an affine type systems values may not be duplicated. In Verity this means that identifiers can be used *at most once* in a parallel and nested context, where such restrictions do not apply to a sequential context. See listing 2.6 for examples that highlight these restrictions. As a result, just like for SAFL, the size of the circuit can be determined by the *size of the program text.*

The affine typing also facilitates separate compilation: synthesis does not require whole-program transformation. Because affine typing ensures that variables are used at most once, including those with a function value, free variables and higher-order function arguments simply give rise to extra input and output ports for the generated component. That is, the program:

```
import <print>
λf x. print (f x)
```

lib1.ia

```
import <print>
 let f = λx. print x
in export f
```

lib2.ia

```
import <print>
 let g = λx. print x
in export g
```

main.ia

```
import "lib1"
import "lib2"

# Will error with:
# lib1.ia: ‘print‘ already used in ‘lib2.ia‘
f 4; g 7
```

Listing 2.7 – Invalid linking

Gives rise to a component with the *obvious* input port for *x*, and an output port for the result (in this case just a control signal, a *ready* flag), but also:

- An output port corresponding to the input of *f*, and an input port corresponding to the output of *f*.
- An output port corresponding to the input of *print*, and an input port corresponding to the output of *print*.

During *link* time all the components are properly connected. The linking process does, however, not resolve any potential conflicts, such as the conflict shown in listing 2.7. In this case the functions *f* and *g* both use the *print* function, and the linker cannot connect the single *print* component to the *f* and *g* components; even though *main* uses *f* and *g* in a sequential fashion.

Although the *affine* typing rules seem overly cumbersome, especially the nested application restriction ($\lambda f\, x.\, f\,(f\, x)$), the developer-facing type system is actually more lenient. The Verity compiler uses a process called *serialisation* [21] that transforms,

- The non-affine expression:

$$\textbf{let } f = \lambda g\ x.\ g\ (g\ x)\ \textbf{in } f\ (\lambda y.\ y + 1)\ 0$$

- To the affine expression:

$$\textbf{let } f = \lambda g\ h\ x.\ g\ (h\ x)\ \textbf{in } f\ (\lambda y.\ y + 1)\ (\lambda y.\ y + 1)\ 0$$

At the moment, recursion in Verity is only possible using the *fixed-point* combinator, *fix*. Recursion using *fix* is always *unfolded in time*. So, the circuit derived from the description given in listing 2.8, contains all the logic needed for one instantiation of the body of *fix*. Just like in SAFL, control logic is added so that the circuit exhibits the behaviour of the recursive description.

Bound variables, free variables, and closed expressions.

A *free* variable denotes a place in an expression where substitution may take place. A *bound* variable is a variable that was previously free, but has been *bound* to a specific value. In programming language terms, a free variable is a variable reference that refers to neither a local variable, nor a function argument. So in the expression:

```
λx. y x
```

the x in the application *(y x)* is a bound variable because it refers to the argument x, and y is free. A closed expression is an expression with *no* free variables.

```
let fib = fix λf.λx.
  if x < 1$32
     then 0$32 # Integer value '0' represented by 32 bits
     else if x < 2
             then 1
             else f(x-1) + f(x-2)
```

Listing 2.8 – Fibbonaci function in Verity

Unfolding in space, although described in [23], is not implemented in the current incarnation of the Verity compiler. Transforming an *unfolding in time* to an *unfolding in space* would thus have to be performed manually. The restrictions that apply to the *fix* make this less than ideal:

- The expression f in, *fix f*, must be closed (f may not refer to variables outside of *fix*).
- Parallel composition is *not* allowed within *fix*.

The latter restriction means that the *unfolding in time* must be *completely* converted to *unfolding in space*, as a half-way point would have parallel composition within fix, which is not allowed.

The data types supported by Verity are: integer (with a specifiable bit-width), tuples, and arrays. Because Verity is higher-order, any algebraic data type (sum-types and product-types) could be encoded using a Church-encoding [5]. When using these encoded data types within *fix*, they must either be abstracted over, or be redefined, because *fix* only accepts closed expressions.

```
data Signal a = Signal (Stream a) (D a)
data Stream a = a :~ Stream a

newtype D a = D AST
data AST = Var    String
         | Entity String [AST]
         | Lit    Integer
```

Listing 2.9 – Simplified dual embedding [27]

```
and2   :: Signal Bool → Signal Bool → Signal Bool
or2    :: Signal Bool → Signal Bool → Signal Bool
pureS  :: Rep a ⇒ a → Signal a
(.==.) :: (Rep a,Eq a) ⇒ Signal a → Signal a → Signal Bool
mux    :: Rep a ⇒ Signal Bool → (Signal a, Signal a) → Signal a
```

Listing 2.10 – Excerpt of Lava Functions and Combinators (Simplified) [25]

2.3.2 Embedded Languages

An embedded domain specific language (DSL) is, as the name suggests, not a standalone language, but actually a library defined within a general purpose language. An embedded language has the syntax of the host language, where the data types and functions of the DSL library act as a new set of keywords.

Lava

Lava [7, 26] is a DSL for structural hardware description embedded in Haskell. All values that eventually end up in the circuit are of an abstract *Signal* type; where abstract means that the data-constructors for this data type are not available to the circuit designer. This *Signal* type can be thought of as a stream of temporally spaced values, where each individual element corresponds to a single clock cycle. In Kansas Lava [27] such a *Signal* type is internally even represented by both a *Stream* data type, and tree data type representing the structure of the circuit (ref. listing 2.9) – a subject we will return to shortly. Because these *Signal* types are abstract, a designer cannot manipulate these values directly, instead, (an extensive set of) combinators and functions offered by the Lava library must be used. The signatures for such functions are shown in listing 2.10.

Synthesis of Lava does not follow the traditional path of static analysis, instead, the Lava library functions are so-called *smart* constructors for a *tree* data type. Executing a function that translates such a data type to e.g. VHDL will then simply calculate the entire structure. Polymorphism, (finite) recursion, and higher-order functions are simply handled by the execution mechanisms of Haskell, and are hence not the problem of the Lava implementers.

The observant reader will notice that we said that the Lava functions create a *tree* data-structure, and not, as one might expect for a structural HDL, a *graph* data-structure. When descriptions have feedback loops, the Lava/Haskell description, will indeed describe an *infinite* tree – the synthesis function must hence first convert this infinite tree to a graph before VHDL can be generated. Techniques such as observable sharing [24] must be used to tag the nodes in the *infinite* tree so that cycles can be detected.

Haskell has a rich set of choice-constructs, including features such as guards and pattern matching. In the style of embedding chosen by Lava, these choice-constructs can, however, not be used to model choice-structures in the circuit. This is a direct consequence of using Haskell's evaluation mechanism to construct the circuit graph: choice-constructs can be used to *guide the construction* of the circuit graph, but it is not possible to *observe* all the alternatives. The problem is that functions must operate on values of type *Signal a* in order to build the data-structure that will represent the circuit. The *Signal* type is, however, *not* transparent in terms of pattern-matching. That is, given a value *s*, of type *Signal Bool*, the expression **case** *s* **of** {*True* → ... ; *False* → ...} , is invalid. That is, even though *True* and *False* are indeed the constructors of *Bool*, they are not the constructors of *Signal Bool*.

The earlier versions of Lava [7] did not support custom data-structures; there was only support for Bit-lists and integers. The most recent incarnation, Kansas Lava [26, 27], does support custom data types. A data type must have an *instance* for the *Rep* type class if it is to be used as value-type for a *Signal*. There are meta-programming facilities, using Template Haskell [28, 60], that automate the instance generation if the custom data type has either of the following properties:

» It has an instance for the *Integral* type class, that is, it can be represented as an integral value.

» It has an instance for the *BitRep* type class, which specifies how a data type can be converted to and from a list of *Bits*.

There are, however, no meta-programming facilities that automate the instance generation for the above two type classes.

One might argue that custom data types are of little use, as the elimination form for data types, *pattern-matching*, is one of the Haskell choice-constructs which cannot be synthesised within the *embedded* approach. Using so-called *choice combinators*, we can, however, partially emulate pattern-matching. Listing 2.11 shows a set of these *smart* constructors to emulate Haskell's **case**-expressions. Comparing an actual **case**-expression with the *cASE* choice combinator in listing 2.12, we can observe that the *emulation* closely matches real case-expressions.

Pattern matching in Haskell facilitates two distinct concepts:

```
cASE :: (Rep a, Eq a) ⇒ Signal a → [(Maybe a, Signal b)] → Signal b
cASE _ []                = error "empty case"
cASE _ [(_,e)]           = e
cASE _ ((Nothing,e):_)   = e
cASE r ((Just a,e):ps)   = mux (r .==. pureS a) ((cASE r ps), e)

(==>) :: (Eq a, Rep a) ⇒ a → Signal b → (Maybe a, Signal b)
a ==> s = (Just a,s)

oTHERWISE :: (Eq a, Rep a) ⇒ Signal b → (Maybe a, Signal b)
oTHERWISE s = (Nothing, s)
```

Listing 2.11 – Choice Combinators (Adaptation of [78])

***case**-Expression*

```
case opc of
  Add → x + y
  Mul → x * y
  _   → 0
```

cASE choice combinator (ref. listing 2.11)

```
cASE opc
  [ Add ==> x + y
  , Mul ==> x * y
  , oTHERWISE 0
  ]
```

Listing 2.12 – **case**-Expression vs. Choice Combinator (ref. listing 2.11)

» Scrutinising a value according to the shape of its constructor, and *selecting* the appropriate alternative.

» Binding the values of the fields of a data type constructor to variables, called *projection*.

The *choice combinator* shown in listing 2.11 and listing 2.12 only emulates the first feature, scrutinising the shape of a constructor and selecting the appropriate alternative. Another DSL embedded in Haskell, HHDL [76], advocates the use of *first class patterns* in the style of [55] to emulate *both* features of pattern matching: selection and projection. Where *ordinary* pattern matching binds values in the pattern itself, the *first class patterns* approach employs lambda terms to bind values. The general shape of an alternative (pattern + expression) in the *first class patterns* approach is:

__case__-Expression

```
case mbA of
  Just x   → x
  Nothing → 0
```

First Class Patterns (ref. HHDL [76])

```
match mbA
  [ pJust pvar --> λ(x :. Nil) → x
  , pNothing    --> λNil → enabledS 0
  ]
```

Listing 2.13 – **case**-Expression vs. First class patterns (ref. HHDL [76] and [55])

$$pCon\ pat_1\ ..\ pat_n\ \text{-->}\ \lambda(x_1\ :.\ ..\ :.\ x_m\ :.\ Nil)\ \rightarrow\ altExpr$$

Where --> is a function that combines the pattern on the left-hand side and the expression on the right-hand side. The *pCon* pattern *function* emulates the pattern / constructor of the original data type, e.g., *pJust* emulates the *Just* pattern / constructor of the *Maybe* data type. The *pCon* pattern function takes *n* patterns as arguments, matching the arity of the pattern / constructor of the original data type. These patterns are of the form:

» *pCon* pat_1 .. pat_n: Another constructor pattern.

» *pvar*: Variable pattern, indicating that the value should be bound to a name.

» *pconst c*: Constant pattern, match again *c*.

» *pwild*: Wildcard pattern.

Using *pvar*, the pattern function(s) build a heterogeneous list of values, which are bound to names in the lambda term that is on the right-hand side of the --> combinator. The order of the bound values matches the syntactic order of the *pvar* functions in the pattern, and nested patterns result in nested heterogeneous lists.

Comparing *ordinary* pattern matching with *first class patterns* in listing 2.13, we conclude that the expressiveness of both forms is equal. The implementation for the functions presented in listing 2.13 can be found in appendix A. There are, however, at least two caveats to the *first class patterns* approach:

» They are (far) more verbose than *ordinary* patterns.

» There is a syntactic (and hence cognitive) distance between the name binding the value (the lambda-term) and the bound value itself (the *pvar* expression).

One additional aspect that should be taken into a account is that the bound values are tagged to indicate that they are *enabled*. Bound values are of type *Signal*

```
mATCH mbA
  [ pJust  pVar --> λ(x :.  Nil) →
                      let k =  registerEnabled  0 (x + (enabledS k))
                      in  k
  , pNothing    --> λNil → 0
  ]
```

Listing 2.14 – First Class Patterns and Enabled Values

(Enabled a), and *not* simply of type *Signal a*. This is important because all the alternatives will be executing in parallel in the eventual circuit, where some of the alternatives might contain memory. The circuit in listing 2.14 contains such an alternative: it accumulates the values that are inside the *Just* constructors; it uses the *registerEnabled* function, which only stores a new value when the enable tag of its input is asserted. If the bound value would not have this enable tag, the register would accumulate bogus values during cycles when *mbA* is *Nothing*, an obviously unwanted behaviour.

It should be noted that this approach, or perhaps limitation, of using explicit *enable* values is only needed when first-class patterns are completely implemented as *smart constructors*. When the *mATCH* function would be a primitive of the language, it is most likely possible to add the enable signals automatically in the netlist generation procedure.

ForSyDe

The ForSyDe [58] system uses Haskell to specify abstract system models. A designer can model systems using heterogeneous models of computation, which include: continuous time, synchronous, and untimed models of computation. Using so-called domain interfaces a designer can simulate electronic systems which have both analogue and digital parts. ForSyDe has several backends, including simulation and automated synthesis, although automated synthesis is restricted to the synchronous model of computation.

ForSyDe splits design entities into three hierarchical levels; from bottom to top they are (ref. listing 2.15):

Function: Defines, partially, the internal behaviour of a process.

Process: The main design unit in ForSyDe. Processes consist of values and functions which determine its state and behaviour. Processes can only be created using well-defined *process constructors*. An example of such a process constructor is *mealySY*, which takes two functions, one that calculates the next state and one that calculates the current output, and the value for the initial state. This process then behaves like a standard *Mealy* machine [44].

```
1  -- A function which adds one to its input
2  addOnef :: ProcFun (Int32 → Int32)
3  addOnef = $(newProcFun [d|addOnef :: Int32 → Int32
4                            addOnef n = n + 1        |])
5
6  -- Process which uses addOnef
7  plus1Proc :: Signal Int32 → Signal Int32
8  plus1Proc = mapSY "plus1Proc" addOnef
9
10 -- System definition associated with the process
11 plus1SysDef :: SysDef (Signal Int32 → Signal Int32)
12 plus1SysDef = newSysDef plus1Proc "plus1" ["inSignal"] ["outSignal"]
```

Listing 2.15 – Hierarchical System Definition [1]

Aside from *process constructors*, processes can also be defined by:

» The composition of other processes.
» The instantiation of a system.

System: A simple wrapper around a process, defining its name, the names of its inputs, and the name of its output.

ForSyDe uses Template Haskell [28, 60], a meta-programming facility for Haskell, to convert the lowest level in the hierarchy, functions, to netlists. Template Haskell provides compile-time observation of the AST of a Haskell declaration, and compile-time generation and insertion of new AST nodes. Looking at lines 3–4 of listing 2.15, the [*d*| ... |] constructs provides the compile-time observation of the enclosed expression. The *newProcFun* then converts the AST of the function to an AST of a declaration representing a netlist node; the $(...) construct subsequently inserts this generated AST in the complete AST of the Haskell / ForSyDe program. ForSyDe uses observable sharing techniques, as those discussed in the section on Lava, to convert (compositions of) processes to a complete netlist.

Using Template Haskell, ForSyDe could observe any Haskell function and convert it to a netlist. However, this part of the synthesis flow must now rely on standard static analysis techniques, and can no longer benefit from Haskell's evaluation mechanism to simply calculate the netlist graph – which is what happens in Lava. The result is that only a limited subset of Haskell can, currently, be converted to a netlist, the ForSyDe compiler has the following restrictions:

» No structured expressions on the RHS of a function (no **where**-clauses, and no **let**-bindings)
» Only a very limited set of functions may be used.
» No higher-order functions.

» No pattern-matching in a function definition
» Patterns in **case**-statements are limited to literals, variables, and wildcards; constructor-patterns are not supported.
» No type-class methods.
» No support for *guards*.

ForSyDe does support synthesis of custom data types, but without pattern-matching on constructors, their usefulness is reduced.

2.4 Conclusions

2.4.1 Standard Languages

The standard hardware description languages all allow type and function parametrisation in some shape or form. A recurring theme in all of these languages is that the type parameters have to be specifically annotated. In Haskell, and by extension CλaSH, types are automatically inferred. This type inference relieves the designer from the cognitive burden of:

» Determining the types of all values, whether they be for input or output ports, or wires.
» Determining which types must be parametrised to get the most general hardware design.

There is an additional syntactic burden in the standard languages of having to explicitly instantiate all the parameters. In CλaSH, this instantiation happens automatically, thereby improving the readability of hardware designs.

Also, although functionality can be parametrised in the standard languages, functions are not first-class. As a result, parametrising functionality is not really encouraged by the language:

» There is no single, straightforward, parametrisation mechanism. In VHDL, depending on the type of logic you want to parametrise, you should either use *configurations*, or *function generics*. Their distinct disadvantages have been described in an earlier section.
» In SystemVerilog, modules and interfaces can *only* be parametrised in terms of interfaces, not modules. Even though modules are considered the default design entities, exemplified by the fact that the top-level design entity must be a module.

In CλaSH, where functions are first-class, and also the only design entities, these restrictions do not apply. Designers also do not have to learn and use a new syntax to parametrise functionality: function arguments, on both the abstraction and application side, are indistinguishable from other arguments. This first-class approach lowers the barrier to parametrise functionality, and thereby encourages the creation of generic, reusable, hardware descriptions.

2.4.2 Functional Languages

Conventional

SAFL is a first-order functional language, meaning that it does not have first-class functions; actually, it has no means whatsoever to parametrise functionality. CλaSH, being a higher-order functional language, thus has a clear advantage over SAFL in this aspect. In SAFL, both concurrent and recursive function calls are performed sequentially through automatically inferred arbitration logic. This feature enables the straightforward creation of *space-efficient* circuits. Higher degrees of parallelism, and thus time-efficiency, are achieved through duplication: creation of a copy of a function in the case of concurrent access, and manual unrolling in the case of recursive calls. In CλaSH, concurrent calls lead to automatic duplication of the called function, and recursive functions are exhaustively unrolled; where the latter has the potential for non-termination. CλaSH thus enables a *time-efficient* circuit by default. Making a circuit sequential has to be done manually in CλaSH, including the creation of (potentially complex) arbitration logic. For regular structures, such as those found in algorithms for DSP applications, the parallel-by-default approach of CλaSH is most likely preferable. For irregular circuits, such as CPUs, the sequential-by-default approach of SAFL might be preferable.

The first-order limitation imposed by the static allocability aspect in SAFL is overcome through the use of an affine type system in Verity. Restricting identifier use in parallel and nested settings to exactly once, space usage is still fully determined by *the size of the text of the program*. Where the affine restriction is strictly imposed on free variables, bound variables are implicitly duplicated through a process called serialisation. Where SAFL thus only supports explicit duplication of functions to achieve higher levels of parallelism, Verity supports implicit duplication through the use of higher-order functions. However, only local, let-bound, functions can be duplicated; *external* functions, being free variables, cannot be passed to a higher-order function that intends to duplicate the functionality. The exact effects of these limitations on designer productivity require further study. Being similar to SAFL, the advantages that SAFL has over CλaSH also are also advantages that Verity has over CλaSH. Unlike SAFL, Verity and CλaSH are on par in terms of parametrising functionality as both have first-class functions. A disadvantage of Verity, compared to both SAFL and CλaSH, is that there can be no *partial*, space versus time, trade-off in a circuit description: a function must be either be completely elaborated over time, or completely elaborated in space, there is no half-way point. The reason for this is that the parallel composition is not allowed within the fixpoint combinator.

In terms of data types, CλaSH is superior to both SAFL and Verity. CλaSH supports many user-defined data types, including sum, product, and inductive data types; SAFL and Verity are limited to primitive types (integer, boolean, etc.) and product types (records and arrays). Additionally, CλaSH is the only language of the three that has polymorphism.

Embedded

CλaSH has a distinct advantage over both Lava and ForSyDe in terms of user-defined data types, being the only language that supports the full range of choice constructs; especially pattern matching. In Lava, pattern matching can be emulated through smart constructors and first-class patterns, but there are many aspects that make them more cumbersome than traditional pattern matching.

Section 2.3.2 lists a number of features of the Haskell language that cannot be synthesised by the ForSyDe embedded compiler; all these features *can* be synthesised with the CλaSH compiler. Something that can be achieved with ForSyDe through explicit system *wrapping* and system *instantiation*, but not with Lava, is the creation of a hierarchical netlist. That is, the synthesis process in Lava generates one big VHDL file that contains a single entity. CλaSH creates hierarchical netlists by default: it creates multiple VHDL files, each containing a design entity corresponding to a (first-order) function. Hierarchical netlists facilitate post-synthesis analysis of non-functional properties, such as chip area and propagation delay, at the individual function level. Something that is far less complex than analysing the complete system.

3
CAES Language for Synchronous Hardware

Abstract – *The CλaSH compiler sees Haskell programs as digital circuit descriptions, in particular, it sees them as structural circuit descriptions. By taking a structural view, the implicit parallelism in the descriptions is synthesised to (sub-)circuits actually running in parallel. In such a structural view, choice constructs (e.g. case-expressions) are mapped to multiplexers, and all the alternatives of the choice construct will be operating (speculatively) in parallel; only one of the outputs of the alternatives will, however, be routed to the output of multiplexer at any one time. This approach only works for languages that have pure functions, such as Haskell, because only then is the observable behaviour of the circuit the same as the observable behaviour of the function from which it is synthesised. Only a semantic subset of Haskell programs have a circuit equivalent under a structural view: unbounded recursion would correspond to a circuit with infinite structure, which is not realisable. The subset of Haskell that can be synthesised to an equivalently behaving circuit under the structural view is called CλaSH, the language.*

3.1 Introduction

In chapter 1 we already gave a short introduction to the correspondence between pure functional languages and digital circuits. In this chapter we will rehash some of that information, but also completely elaborate why the semantics of *Haskell* specifically makes it a good choice for the specification of digital circuits. Unlike the example code listings in chapter 1 and chapter 2, we will also elaborate the syntax (and semantics) of the code in (more) detail.

Large parts of this chapter have been published in [CB:7] and [CB:9].

The chapter starts with the specification of combinational logic in Haskell, followed by simulation of these circuits, in essence highlighting the semantic match of the descriptions and actual combinational circuit logic. The section after that will discuss the higher-level features of Haskell, such as polymorphism and higher-order functions, and show their particular usefulness for circuit descriptions. The chapter ends with the descriptions of sequential logic.

Throughout this thesis, and certainly within this chapter, we view all Haskell description as a *structural* specification of the digital circuit. Certain descriptions, when viewed structurally, actually describe *infinite* structures. The compiler for the descriptions, called CλaSH, cannot synthesise those *infinite* descriptions to a low-level netlist format. We will henceforth refer to Haskell under the structural view as: CλaSH, the language. Due to the aforementioned restrictions regarding infinite structures, CλaSH is a (semantic) subset of Haskell.

3.1.1 A structural view

We already highlighted that we take a *structural* view on the specifications of our digital circuits. Of course, a designer creates such a specification with the intent that the circuit exhibits a certain *behaviour*. So what exactly is this *structural* view? In all fairness, a structural view, is not the view of the designer, but of the compiler or synthesis tool. A compiler with a structural view just sees the circuit description as an arrangement of the individual parts and the relations between those parts. During translation from specification to actual circuit, a structural synthesis tool will preserve as much of the arrangement and relations as possible.

A circuit designer will, aside from reasoning about the behaviour of a circuit, also want to reason about the non-functional properties of the circuit. Such properties include for example the size of the circuit, or the maximum propagation delay between latches or registers. The structural view taken by the compiler permits a very straightforward mental model for these non-functional properties. There are no more, nor fewer (modulo optimisations), components than the designer specified, which makes reasoning about area straightforward. The number of components between two latches is also fixed, making reasoning about propagation delays also straightforward.

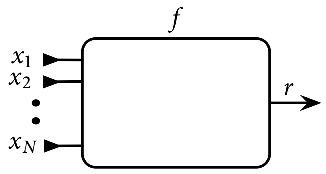
The structural view is a double-edged sword: it enables straightforward reasoning about non-functional properties, but also burdens the designer with these non-functional properties. It is up to the designer to, e.g., add pipelining to decrease maximum propagation delays, or add control circuitry to reuse components over multiple time-slots. A compiler with a structural view does not, in general, create more components than directly specified in the description.

In the next couple of sections we will describe how the language features of CλaSH correspond to digital circuits given a structural view.

Function abstraction

```
f x₁ x₂ .. x_N = r
```

Component definition



Listing 3.1 – Correspondence between function abstractions and component definitions

3.2 Combinational logic

3.2.1 Function abstraction and application

Two basic elements of a functional language are function abstraction and function application. With the structural view, a function corresponds to a *component* in a digital circuit, where:

» The function arguments become the input ports (or pins).

» The result of the function is connected to the output port.

A visual correspondence between functions and components is depicted in listing 3.1. There we see a function f with N arguments (x_1 to x_N) and a result r. Below that, we see that this corresponds to a component f, with N input ports (x_1 to x_N), and an output port which is connected to the wire r carrying the result of the component.

Function *application* is subsequently seen as component *instantiation*, where:

» The applied arguments are connected to the input ports of the instantiated component.

» The output port is connected to a the wire that carries the result.

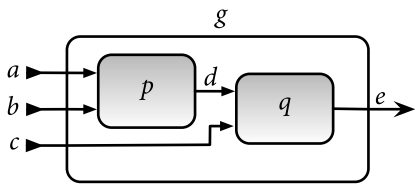
A more visual correspondence between function application and component instantiation is given in listing 3.2. There we see a function g, which, in its **where**-clause, applies two functions p and q, binding their results to the variables d and e. In the corresponding component g we see the two instantiated components p and q, where their respective output ports are connected to the wires labelled d and e.

Although drawn as single lines in the previous two figures, the connections between the components and ports can consist out of multiple wires. The number of wires needed for these connections are of course determined by the values that flow through them. In section 3.2.2, where we discuss *types*, we will elaborate how these numbers are exactly derived.

Function application

```
g a b c = e
  where
    d = p a b
    e = q d c
```

Component instantiation



Listing 3.2 – Correspondence between function application and component instantiation

In the introduction we said that certain specifications describe infinite structures, these specifications are ones that contain recursive functions. Under the structural view, the recursive/self-application of a function corresponds to the unbounded self-instantiation of the synthesised component, hence resulting in an infinite structure. Although reasoning about infinite structures is not necessarily difficult, these infinite structures cannot be realised as actual circuits. By unfolding the definition of the functions at their place of recursive application, a compiler can remove one layer of self-instantiation at a time. Under the structural view, only those recursive functions for which the exact number of recursive applications can be fully determined at compile-time are thus realisable as a circuit.

Where we have to restrict the creation and use of recursive functions for these functions to be realisable as a circuit under the structural view, the structural view poses no such limitations on value recursion. Actually, value recursion in a description often corresponds to a feedback loop in the realised circuit, as depicted in listing 3.3 (which was already shown in chapter 1).

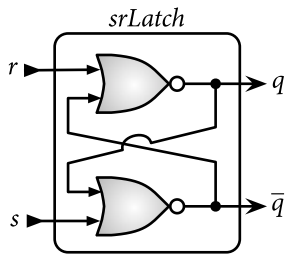
3.2.2 Types

As already hinted to earlier in this chapter, it are the *types* of the variables and functions that inform us how many wires are needed to connect the components in our circuit. Compilers generally have a notion of *primitive* types, types which the designer can use, but cannot define him- or herself. For CλaSH, there is a similar notion in determining the number of bits to encode values of certain types, and hence the number of wires needed to connect components. There is a set of types for which the compiler has hardcoded knowledge on how many bits are needed to represent its values, were refer to this set of types as *hardcoded* types. The other types will simply be referred to as *user-defined* types.

Value recursion

```
srLatch r s = (q,nq)
  where
    q  = nor r nq
    nq = nor q s
```

Feedback loop



Listing 3.3 – Correspondence between value recursion and feedback loops

Hardcoded types

The following types in CλaSH have hardcoded bit-widths:

***Signed n*, *Unsiged n*:** These two types are internally defined as:

```
newtype Signed   (n :: Nat) = S Integer
newtype Unsigned (n :: Nat) = U Integer
```

And are used to represent signed and unsigned integers respectively. Internally they are represented by infinite-precision integers, but the functions operating on these values treat them as n-bit integers, where n refers to their type-parameter which is a type-level *Nat*ural number[1]. The number of bits needed to encode these types is hence this equal to the number n.

***Vec n a*:** Where this type is defined as:

```
data Vec :: Nat → * → * where
  Nil  :: Vec 0 a
  (:>) :: a → Vec n a → Vec (n + 1) a
```

This type specifies a list (or *Vec*tor) of elements, where the length is statically encoded in the type as its first parameter. The second type parameter denotes its element type[2]. Given that the bit-width can be calculated for the element type, the total bit-width of the entire vector is simply its length times the bit-width of the element.

[1] Haskell has a strict phase separation between compile-time and run-time, meaning that type-level numbers are not the same as term-level values. So even though a type-level number and the term-level value can share the same *symbol* for their representation, i.e. the symbol 2, they are completely distinct.

[2] The kind (type of types) $\star$ represents all simple types, like *Int* and *Char* → *Bool*.

User-defined types

A designer may define a completely new type by using the **data** keyword. The following *algebraic* types have a fully determinable bit-width:

Product types: A product type has a single constructor which has multiple fields, such as *tuples*, and *record* types. An example of a product type would be:

```
data Enabled a = Enabled Bool a
```

Which has one constructor, *Enabled*, and two *fields*: one of type *Bool* and one with the type of the type parameter *a*. The bit-width of a product type is simply the sum of the bit-width of the types of its fields.

Simple sum types (enumerations): Simple sum types, or enumerations, have multiple constructors, but where the constructors have no fields. A well known enumeration is the *Bool*ean type:

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

The bit-width for these types is: $\lceil \log_2 c \rceil$, where c is the number of constructors.

Sum types: Sum types (sometimes called sum-of-product types to differentiate from simple sum types) have multiple constructors, where the constructors can have multiple fields. A well know sum type is the *Either* type:

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

Which has two constructors, and each constructor has one field. The bit-width of a sum type is calculated by:

1. For every constructor: sum the bit-width of their respective fields.
2. Calculate the maximum of these sums.
3. Add $\lceil \log_2 c \rceil$, where c is the number of constructors.

Recursively defined types can, in general, not be given a fixed bit-width: it usually not possible to determine how many times a recursive type can be unfolded. That is also the main reason why the *Vec*tor type has a hard-coded bit-width calculation. However, the very fact that there is a hard-coded calculation indicates that it is not impossible to determine the bit-width for certain recursive types. For *Vec*tors, the size can be fully determined due to the fact that it has a natural number as a type-parameter which restricts both which and how many of its constructors must be used to create a valid value. Generalising such an analysis to other recursive data types is considered future work.

3.2.3 Choice

In Haskell, choice can be achieved by a large set of syntactic elements, consisting of: **case** expressions, **if**–**then**–**else** expressions, multiple-clause function definitions, and guards. The most general of these are the (pattern-matching) **case** expressions: all others forms of choice can be translated to **case** expressions. We hence use the circuit interpretation for **case** expressions as the circuit interpretation for all *choice* constructs.

In general, a **case** expression is of the form:

```
case  subject  of
  C_1 x_1 .. x_M  →  alternative_1
  ..
  C_N y_1 .. y_K  →  alternative_N
```

Where an *alternative* is chosen based on the constructor shape ($C_1..C_N$) of the *subject*. Aside from choosing an alternative, the **case** expressions also binds the fields corresponding to the constructor to variables ($x_1..x_M$), which can be used in the selected alternative. These two aspects of **case**-expressions, selection and projection, have separate circuit interpretations.

We start with the interpretation for selection. The structural view on the selection aspect gives us a *multiplexer*, where:

» The wires representing the constructor part of the *subject* are connected to the selection port.

» The result of the alternatives are connected to the input ports.

For the synthesis of expressions to circuits we assume that all the functions that we will be working with are side-effect free, which is an aspect that can be statically asserted in Haskell. When the expressions in the alternatives are side-effect free, the parallel calculations happening in the eventual circuit will not interfere with each other. The observed behaviour of the complete circuit, of what in software terms would be called "speculative" parallel execution, is hence faithful to the semantics of the **case**-expression[3].

Two, semantically equal but syntactically distinct, code examples of a counter circuit, and the circuit interpretation that corresponds to both specifications, are given in listing 3.4. The function counts up or down depending on the *direction* variable, and has a *wrap* variable that determines the wrap-around point of the counter.

The other aspect of **case** expressions, aside from *selection*, is binding the values of constructor fields to variables. This aspect is called *projection*. Given that a value of a data type with fields travels over N wires, projection is simply continuing with M of these wires that represent the field. Listing 3.5 demonstrates this *projection* aspect of **case** expressions and the corresponding circuit interpretation for a simple product type. In this figure we mark the bit-width (in the form of bit-indices) of the individual wires to get a better understanding of the circuit interpretation for projection.

[3]Assuming no alternative evaluates to bottom, which also would not have a circuit representation.

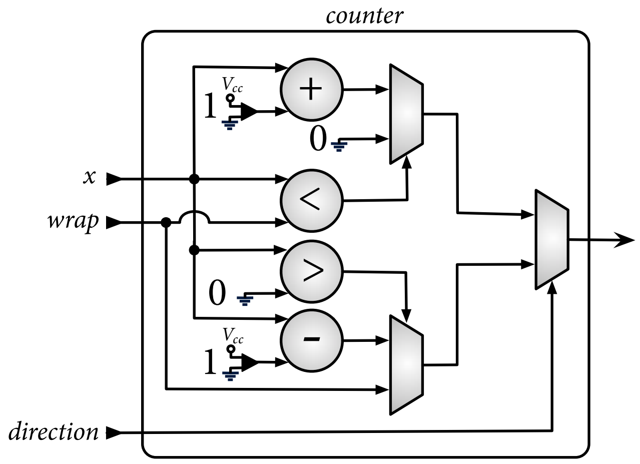
*Description using **case** and **if**–**then**–**else** expressions*

```
data Direction = Up | Down

counter wrap direction x = case direction of
  Up   → if x < wrap then
              x + 1     else
              0
  Down → if x > 0     then
              x − 1     else
              wrap
```

Description using pattern matching and guards

```
data Direction = Up | Down

counter wrap Up   x | x < wrap  = x + 1
                    | otherwise = 0

counter wrap Down x | x > 0     = x − 1
                    | otherwise = wrap
```

Circuit interpretation



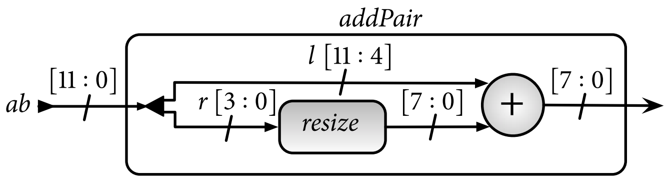
Listing 3.4 – Counter circuit

In listing 3.6 we see the projection for a sum type, and the corresponding circuit interpretation. Here we see that the wires representing *b* and *y* use overlapping bit-indices ([7 : 0] and [7 : 7] out of the original [8 : 0]). This means that, when the *sf* value is constructed by *SumS8*, the sub-circuit representing the alternative for the *SumB* branch is not really operating on a *Bool*ean value. There is an analogous situation in the other alternative when *sf* value is constructed by *SumB*. However,

Haskell code

```
data Pair = Pair (Signed 8) (Signed 4)

addPair ab = case ab of
  Pair l r → l + resize r
```

Circuit interpretation



Listing 3.5 – Projection of a product type

even though the alternatives will be working on erroneous values in the above situations, because the expressions belonging to the alternatives are side-effect free, their corresponding sub-circuits will not affect the observable behaviour of the complete circuit. That is, the multiplexer will never output the value belonging to the erroneous alternative, and there is no information flow between the alternatives.

A final side-note that we want to make regarding both past and future circuit diagrams in this thesis: we will (often) draw individual ports and wires for the individual fields in a *product* type. Drawing separate wires, instead of splitting and combining wires, improves readability and does not affect the semantics of the diagram.

3.3 Higher level abstractions

3.3.1 Polymorphism

Polymorphism allows a function to handle values of different data types in a uniform way. Haskell supports *parametric* polymorphism, meaning that functions can be written to work on values with an arbitrary type. As an example of a parametric polymorphic function, consider the type of the *fst* function, which returns the first element of a tuple:

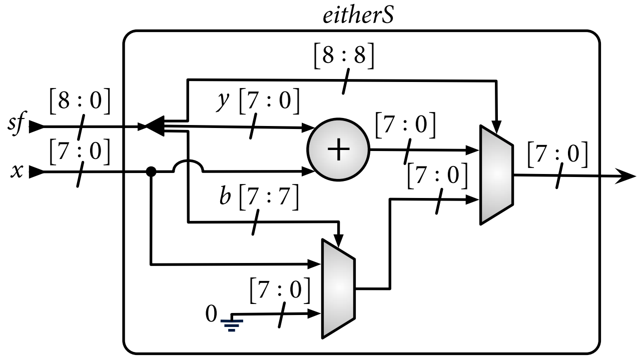
```
fst :: (a,b) → a
fst (x,_) = x
```

The type of *fst* is parametrised in *a* and *b*. This means that *fst* works for any tuple, regardless of what elements it contained in the tuple. This kind of polymorphism is very useful in circuit design, examples include: routing signals without knowing their exact type, or, specifying vector operations that work on vectors of any length

Haskell code

```
data SumF =  SumS8 (Signed 8)
          |  SumB Bool

eitherS  sf  x = case  sf  of
   SumS8 y → x + y
   SumB  b → if  b then x  else  0
```

Circuit interpretation



Listing 3.6 – Projection of a sum type

and element type. Polymorphism also plays an important role in most higher-order functions, as will be shown in the next subsection.

Another type of polymorphism supported in Haskell is *ad-hoc* polymorphism. In ad-hoc polymorphism, a function with a single interface can have different functionality depending on the type it is instantiated for. A primary example are the numeric operators ((+), (*), etc.), that have a single interface (e.g.: (*) :: *Num a* ⇒ *a* → *a* → *a*), but operate differently whether applied to e.g. integers or complex numbers.

In Haskell, ad-hoc polymorphism is achieved through *type classes*, where a class definition provides the general interface of a function, and class *instances* define the functionality for the specific types. For example, certain numeric operators are gathered in the *Num* class (listing 3.7); so if we want to have those operators work on values of our type we must create an *instance* of *Num* for that type.

By prefixing a type signature of a function with class constraints (e.g.: *Num a* ⇒), type parameters are constrained to types that have an instance for that class. For example, the arguments of the *subtract* function must have a type which has an instance for the *Num* type class because the *subtract* function subtracts them using the (-) operator (listing 3.8).

Type class definition (partial)

```
class Num a where
  (+) :: a → a → a
  (*) :: a → a → a
  (−) :: a → a → a
  ...
```

Instance for Integer (partial)

```
instance Num Integer where
  (+) = plusInteger
  (−) = minusInteger
  (*) = timesInteger
  ...
```

Instance for Complex (partial)

```
data Complex a = !a :+ !a

instance ( RealFloat a) ⇒ Num (Complex a) where
  (x:+y) + (x':+y') = (x+x') :+ (y+y')
  (x:+y) − (x':+y') = (x−x') :+ (y−y')
  (x:+y) * (x':+y') = (x*x'−y*y') :+ (x*y'+y*x')
  ...
```

Listing 3.7 – The *Num* type class and instances for *Complex* and *Integer*

```
subtract :: Num a ⇒ a → a → a
subtract x y = y − x
```

Listing 3.8 – Function using a type-class method

CλaSH supports both parametric polymorphism and ad-hoc polymorphism. A circuit designer can specify his own type classes and corresponding instances. There is, however, one constraint on parametric polymorphism: the function at the root of the function hierarchy[4] cannot have polymorphic arguments, nor a polymorphic result. This constraint is sufficient to infer and propagate the *concrete* types in the rest of function hierarchy. As elaborated in section 3.2.2, we need to know the concrete types of the arguments and results to determine the bit-width for the values flowing through the circuit.

[4] In Haskell the function at the root of the function hierarchy is called *main*, in CλaSH it is *topEntity*.

```
map :: (a → b) → [a] → [b]
map _ []       = []
map f (x:xs)   = f x : map f xs
```

Listing 3.9 – The, higher order, *map* function

```
vmap :: (a → b) → Vec n a → Vec n b
vmap _ Nil       = Nil
vmap f (x:>xs)   = f x :> vmap f xs
```

Listing 3.10 – The, higher order, *vmap* function

3.3.2 Higher-order functions

In Haskell, functions are *first class*, meaning that a function can have other functions both as arguments, and as its result. Functions that have other functions as arguments are called *higher-order* functions. An ubiquitous higher-order function in Haskell, and CλaSH, is the *map* function: the code for the map function is give in listing 3.9.

This *map* function applies a function f to all values in the list xs. For conversion to a circuit there are two problems with the *map* function:

» It works with the *recursive* list type, a type for which we cannot determine the number of bits needed to encode a list value. Another problem is that the list can be infinite.

» It is a *recursive* function, which, under the structural view, potentially describes infinite structure.

The equivalent function for *Vectors*, the *vmap* function (shown in listing 3.10), does, however, not suffer from these problems:

» The *Vector* type has a hard-coded bit-width calculation, so we can determine the number of bits needed to encode a *Vector* value. Because the length of the *Vector* is encoded as a natural number in its type, it is also impossible to create an infinitely long *Vector*. It can be *very* long, but never, by definition, infinitely long: infinity is not a member of the natural numbers.

» The function can be completely unfolded, even when the function is not applied to a concrete vector, there are hence no infinite structures. The reason that we can completely unfold the definition is that the type of the *Vector* allows us to infer which constructor must be used. When the vector is of length zero, it must be created with *Nil*, while non-zero length vectors must be created with :>.

So when we have:

```
topEntity  :: Vec 4 Bool → Vec 4 Bool
topEntity  xs4 = vmap not xs4
```

The compiler can *correctly* unfold *vmap* once to:

```
topEntity  :: Vec 4 Bool → Vec 4 Bool
topEntity  xs4 =
  let x3  = case xs4 of (x :> _)  → x
      -- xs3 :: Vec 3 Bool
      xs3 = case xs4 of (_ :> xs) → xs
  in  not x3 :> vmap not xs3
```

We repeat this process until we arrive at:

```

topEntity  :: Vec 4 Bool → Vec 4 Bool
topEntity  xs4 =
  let x3  = case xs4 of (x :> _)  → x
      -- xs3 :: Vec 3 Bool
      xs3 = case xs4 of (_ :> xs) → xs
  in  not x3 :> let
      x2  = case xs3 of (x :> _)  → x
      -- xs2 :: Vec 2 Bool
      xs2 = case xs3 of (_ :> xs) → xs
  in  not x2 :> let
      x1  = case xs2 of (x :> _)  → x
      -- xs1 :: Vec 1 Bool
      xs1 = case xs2 of (_ :> xs) → xs
  in  not x1 :> let
      x0  = case xs1 of (x :> _)  → x
      -- xs0 :: Vec 0 Bool
      xs0 = case xs1 of (_ :> xs) → xs
  in  not x0 :> Nil
```

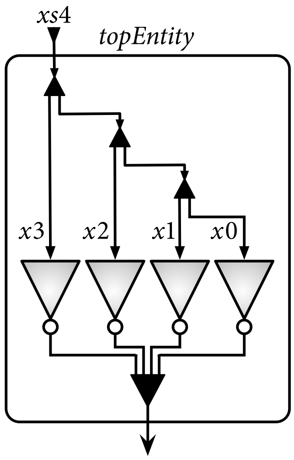
The completely unfolded definition corresponds to four not-gates in parallel, one for each element in the vector. We can see the original circuit description, and the circuit representation of the completely unfolded definition in listing 3.11.

Due to the aforementioned properties of the vector data type, all structurally recursive functions over vectors can be completely unfolded at compile-time. Higher-order functions over vectors can hence be used to capture many (parallel) composition patterns that are common in digital circuits. In listings 3.12 to 3.14 we see three such vector functions (*vmap*, *vzipWith*, and *vfoldl*), and their structural interpretations. These general structures of the corresponding circuits only emerge when the element types of the vectors have a straightforward bit-encoding.

Haskell code

```
topEntity  :: Vec 4 Bool → Vec 4 Bool
topEntity xs4 = vmap not xs4
```

Circuit interpretation

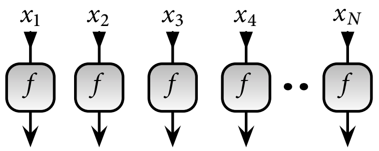


Listing 3.11 – Four not-gates in parallel

Haskell code

```
vmap f xs
```

Circuit interpretation



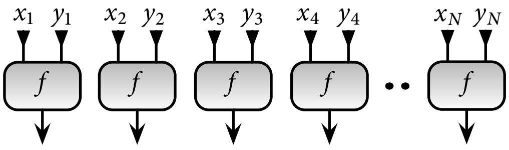
Listing 3.12 – *vmap*: parallel composition of a unary function

That is, taking a look at listing 3.15, the circuit interpretation of *vzipWith* (\$) (*vmap* (+) *xs*) *ys* is not the first, erroneous, interpretation where we see both the *vmap* and *vzipWith* stucture. The element types of the vector resulting from *vmap* (+) *xs* are functions, which, under the structural view, have no (straightforward) bit-encoding. The proper circuit interpretation is thus the second circuit, where the *vmap*-related structure is completely eliminated, and only the *vzipWith* structure remains.

Haskell code

```
vzipWith f xs ys
```

Circuit interpretation

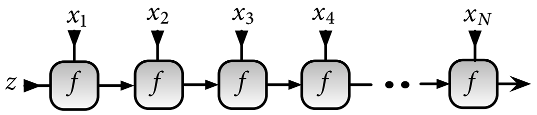


Listing 3.13 – *vzipWith*: parallel composition of a binary function

Haskell code

```
vfoldl f z xs
```

Circuit interpretation



Listing 3.14 – *vfoldl*: left-associative reduction

This leads us to place a constraint on the use of higher-order functionality similar to the constraint placed on the use of polymorphism. That is, the function at the top of the function hierarchy can have neither:

» Arguments that are, or contain, values of a function type, nor,

» A result that is, or contains, a value of a function type.

Just as for polymorphism, higher-order functionality can be used in all other parts of the function hierarchy.

An example of a common circuit where polymorphism and higher-order functions lead to a very concise and natural description is: a crossbar. The code for this example can be seen in listing 3.16. The *crossbar* function selects those values from *inputs* that are indicated by the indexes in the vector *selects*. The interplay of polymorphism and higher-order functions results in a description that is both concise, and, without any form of annotation, highly parametric:

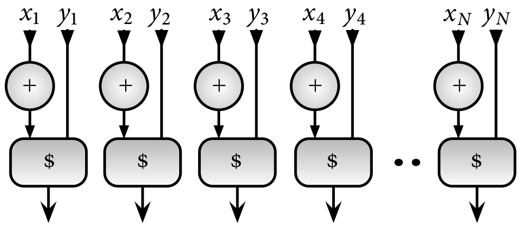
» It is polymorphic in the number of inputs, which is defined by the vector length of *inputs*.

» It is polymorphic in the number of outputs, which is defined by the vector length of *selects*.

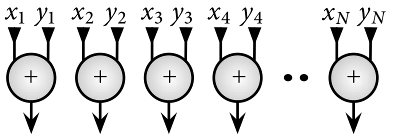
Haskell code

```
vzipWith ($) (vmap (+) xs) ys
```

Erroneous circuit interpretation



Proper circuit interpretation



Listing 3.15 – Erroneous and proper circuit interpretation

» It is polymorphic in the values of the data, which defined by the element type of *inputs*.

The *crossbar* function is just one of many examples where polymorphism and higher-order functions lead to a concise design that is naturally parametric; we will see more examples in chapter 5.

3.4 Sequential logic

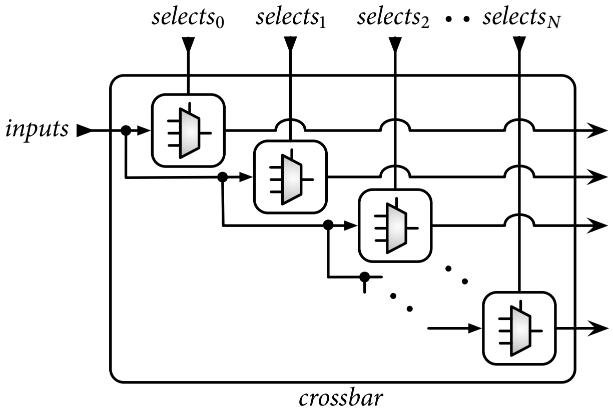
Until now, we have only witnessed how we can describe *combinational* circuit in Haskell. As already highlighted in chapter 1, the behaviour of a function in Haskell is not influenced by any notion of time. Hence, under the structural view, functions in Haskell map most faithfully to combinational circuits. Does this mean we cannot describe sequential logic under this structural view?

When we take a look at a language that has a *behavioural* synthesis approach, such as Verity [23], we see that the resulting circuits have sequential elements. As already described earlier, the structural interpretation of general recursive functions are circuits with infinite structure, the behavioural interpretation taken by e.g. Verity does, however, not suffer from this problem. Verity's GoS [23] approach introduces sequential elements in the circuit, resulting in a circuit that calculates the result of the recursive function over multiple cycles. The behavioural synthesis approach thus translated the time-independent behaviour of the function, to a circuit whose behaviour is time-dependent. The created circuit gets extra ports to indicate when

Haskell code

```
crossbar inputs selects = vmap (inputs !!) selects
```

Circuit interpretation



Listing 3.16 – Crossbar

the data-ports contain a valid value: only at moments these ports are asserted does the circuit exhibit the same external behaviour as the time-independent behaviour of the original function.

In a way, specifications subject to the behavioural synthesis approach do not actually *describe* sequential logic, but the specification gets *mapped* to sequential logic. We also already saw in chapter 1 that the circuit interpretation of certain specifications under the structural view give rise to sequential circuits, such as the description of an SR-latch (listing 3.17). However, the semantics of the resulting sequential circuit do not match the time-oblivious semantics of the functional specification. What we desire is a construct in Haskell where the semantics of the functional specification matches the semantics of resulting sequential circuit.

3.4.1 Synchronous sequential circuits

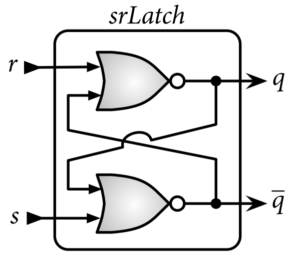
Quoting chapter 1:

> Sequential logic in digital circuits can be divided into *synchronous* and *asynchronous* logic. In synchronous logic, all memory elements update their state in response to a clock signal. In asynchronous logic, memory element can update their state at any time in response to a changing input signal. The clock signal in synchronous logic is an oscillating signal that is distributed to all the memory elements such that they all observe its level change simultaneously. A crucial aspect

Haskell code

```
srLatch :: Bit → Bit → (Bit , Bit)
srLatch r s = (q,nq)
  where
    q  = nor r nq
    nq = nor q s
```

Circuit interpretation



Listing 3.17 – SR Latch

of synchronous logic is that the interval of the clock signal must be long enough so that the input signals of the memory elements can reach a stable value. The time it takes for a signal to become stable is determined by the largest propagation delay between any two memory elements with no other memory element in between. The (combinational) logic between memory elements must hence be completely acyclic. Synchronous design allows a designer to abstract from propagation delays, and reason about state changes as if they happen instantaneously and synchronised.

In a synchronous sequential circuits we are hence not interested in the exact times and levels of a signal, but only in the sequence of stable values of a signal. We can model that abstraction of a signal as a *stream*, where the element in the stream correspond to the stable values for the consecutive clock ticks:

```
data Signal a = a :- Signal a
```

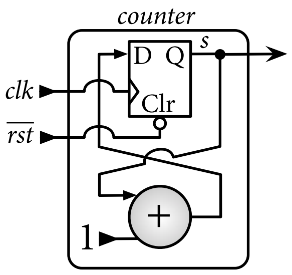
Functions defined over *Signals* can now be used to specify synchronous *sequential* circuits.

We can now use value recursion to specify feedback loops where, unlike the feedback loop in the SR-latch specification (listing 3.17), we have a one-to-one correspondence between the functional semantics of the specification, and the synchronous sequential semantics of the derived circuit. For example, given:

Haskell code

```
counter = s
  where
    s = register 0 (s + 1)
```

Sequential circuit interpretation



Listing 3.18 – Counter circuit

» A function, *register*, whose result is a *Signal* where the initial sample is *register*'s first argument, followed by the samples of *register*'s second argument. The behaviour of which corresponds to a d-flipflop with an asynchronous reset.

```
register :: a →Signal a →Signal a
register i s = i :− s
```

» And a *Num* instance for *Signal*.

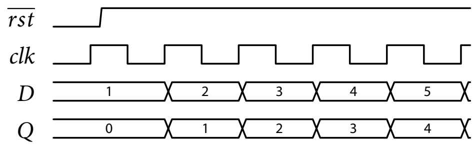
```
instance Num a ⇒Num (Signal a) where
  (a :− as) + (b :− bs) = a + b :− as + bs
  fromInteger i = i :− fromInteger i
  ...
```

We can create a circuit that counts the number of clock cycles since the last asynchronous reset using the specification shown in listing 3.18. The value recursion on *s*, under the structural view, gives rise to the feedback loop between the register and the adder. In listing 3.19 we can see that when we elaborate the specification of *counter*, in accordance with the functional semantics, we get a stream of values corresponding to the number of elapsed clock cycles. The second part of listing 3.19 shows the timing diagram of the structurally derived circuit, where we can see that there is truly a one-to-one mapping between the behaviour of the specification and the behaviour of the synthesised circuit.

Elaboration of the counter *function*

```
-- definition of 's'
register 0 (s + 1)
-- definition of 'register'
0 :- s + 1
-- definition of 's'
0 :- register 0 (s + 1) + 1
-- definition of 'register'
0 :- (0 :- s + 1) + 1
-- definition of '+' and 'fromInteger'
0 :- 0 + 1 :- s + 1 + 1
-- definition of '+'
0 :- 1 :- s + 1 + 1
-- definition of 's'
0 :- 1 :- register 0 (s + 1) + 1 + 1
-- definition of 'register'
0 :- 1 :- (0 :- (s + 1)) + 1 + 1
-- definition of '+' and 'fromInteger'
0 :- 1 :- 0 + 1 + 1 :- s + 1 + 1 + 1
-- definition of '+'
0 :- 1 :- 2 :- s + 1 + 1 + 1
```

Timing diagram of the counter *circuit*



Listing 3.19 – Elaboration of the *counter* specification (listing 3.18) and the timing diagram of the structurally derived circuit

3.4.2 A safe interface for *Signal*

We have seen that adding an element to the head of the *Signal* stream corresponds to a memory element in the derived circuit:

```
register :: a → Signal a → Signal a
register i s = i :- s
```

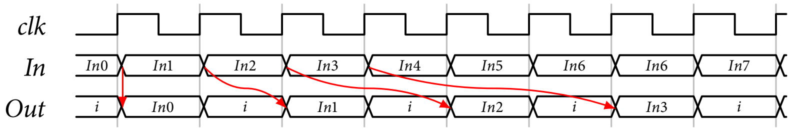
Many forms of *Signal* manipulations do, however, not have a corresponding circuit interpretation. For example, the following specification:

```
delayInc :: a → Signal a → Signal a
delayInc i (s :- ss) = i :- s :- delayInc i ss
```

```
class Functor f where
  fmap :: (a → b) → f a → f b

((⟨$⟩)) :: Functor f ⇒ (a → b) → f a → f b
f ⟨$⟩ a = fmap f a

class Functor f ⇒ Applicative f where
  pure  :: a → f a
  ((⟨∗⟩)) :: f (a → b) → f a → f b

instance Functor Signal where
  fmap f (s :− ss) = f s :− fmap f ss

instance Applicative Signal where
  pure a                   = let s = a :− s in s
  (f :− fs) ⟨∗⟩ ~(a :− as) = f a :− fs ⟨∗⟩ as
```

Listing 3.20 – *Applicative Functor* [42] interface for *Signal*

Would result in a circuit that has the following waveform:



Where we can clearly see that subsequent input values are delayed by an increasing number of clock cycles on the output. Such a circuit would need an infinite amount of memory and is clearly not realisable.

To protect the circuit designer from himself, it is preferable to hide the constructor, (:−), and instead offer an interface to the *Signal* data type, and accompanying primitive(s), through which he can only specify realisable sequential circuits. We have already seen the most important primitive, the *register* function, which is the only primitive that can introduce memory elements in a circuit. *Signal*s can subsequently only be manipulated through an *Applicative Functor* [42] interface, which is shown in listing 3.20.

Using the operators of the *Applicative* and *Functor* type class we can apply a function that only works on the individual elements of a *Signal*, a function representing a *combinational circuit*, to operate on all elements in the *Signal* sequentially. We can elaborate the use of these operators more clearly using the example shown in listing 3.21.

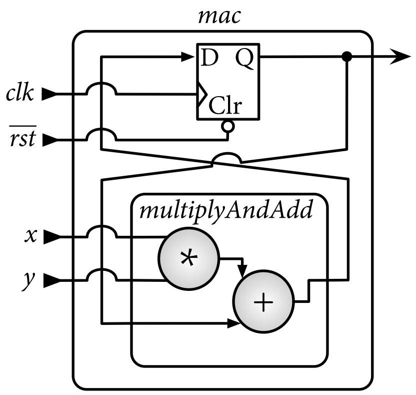
Combinational multiply-and-add specification

```
multiplyAndAdd :: Signed 7 → Signed 7 → Signed 7 → Signed 7
multiplyAndAdd x y acc = x * y + acc
```

Sequential multiply-and-accumulate (MAC) specification

```
mac :: Signal (Signed 7) → Signal (Signed 7) → Signal (Signed 7)
mac x y = acc
  where
    acc  = register 0 acc'
    acc' = multiplyAndAdd ⟨$⟩ x ⟨*⟩ y ⟨*⟩ acc
```

Derived MAC circuit



Listing 3.21 – Multiply-and-Accumulate circuit

At the top of listing 3.21 we see the combinational *multiplyAndAdd* function that multiplies the two number *x* and *y*, and adds a third number, *acc*, to the result of this multiplication. In the middle part of listing 3.21 we create the sequential multiply-and-accumulate *(MAC)* function, where we apply the (combinational) *multiplyAndAdd* function, which expects arguments of type *Signed 7*, to variables of a *Signal* type, *Signal (Signed 7)*, using the ⟨$⟩ and ⟨*⟩ operators. The derived circuit is depicted at the bottom of listing 3.21.

3.4.3 Abstractions over *Signal*

All sequential circuits will ultimately be composed out of:

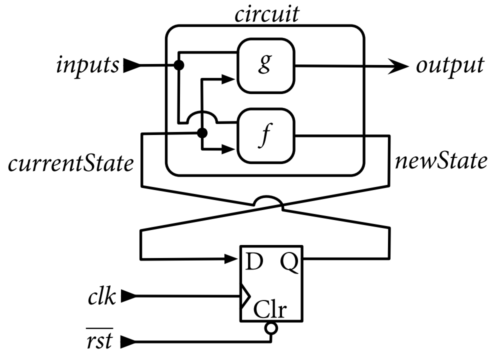
» The *register* function, and,

» The *pure*, ⟨$⟩, and ⟨*⟩ functions that lift combinational values and functions to the sequential *Signal* domain.

Haskell code

```
circuit :: State → Input → (State ,Output)
circuit currentState inputs = (newState, output)
  where
    newState = f currentState inputs
    output   = g currentState inputs
```

Sequential circuit interpretation



Listing 3.22 – Mealy machine

Although working directly with these functions and operators suffices for small circuits, it is preferable to take a more principled approach when designing larger circuits. One such approach, originally put forward as the main technique for designing sequential circuits in CλaSH prior to the *Signal* type ([38] and [CB:7]), is to design every sequential circuit by solely specifying the combinational circuit part of a Mealy machine [44]. A circuit specification would follow the template given in listing 3.22.

Where both the new state of the circuit, and the output of the circuit, are a function of the current state and the inputs. The multiply-and-accumulate (MAC) circuit is described as follows in the Mealy machine style:

```
macT :: Signed 7 → (Signed 7, Signed 7) → (Signed 7, Signed 7)
macT acc (x,y) = (acc', acc)
  where
    acc' = x * y + acc
```

All that is missing to describe the complete behaviour of the circuit is its initial state, the contents of the memory element on the first clock cycle. What is needed is a function that takes as arguments the standardised Mealy machine description, the initial content of the memory, and cast it into a function that works on *Signals*. This *lifting* function is defined in listing 3.23. The complete specification of the

```
(⟨^⟩) :: (s → i → (s,o)) → s → (Signal i → Signal o)
mealyT ⟨^⟩ initS =
  λinp → let r    = register initS (fst ⟨$⟩ outp)
             outp = mealyT ⟨$⟩ r ⟨*⟩ inp
         in snd ⟨$⟩ outp
```

Listing 3.23 – Creating Mealy machines

```
(⟨^⟩) :: (s → i → (s,o)) → s → (Signal i → Signal o)
mealyT ⟨^⟩ initS =
  λinp → let r         = register initS r'
             (r', outp) = unbundle (mealyT ⟨$⟩ r ⟨*⟩ inp)
         in outp
```

Listing 3.25 – Creating Mealy machines

MAC circuit, with its cycle-by-cycle behaviour determined by *macT*, and an initial accumulation value of zero, is then given by:

```
mac = macT ⟨^⟩ 0
```

One set of functions that we need to introduce before we go on to the next topic, are the *bundle* and *unbundle* functions, as they will be used in chapter 5. The observant reader will notice that in the definition of ⟨^⟩ that we used (*fst* ⟨$⟩*outp*), and (*snd* ⟨$⟩*outp*), to get the individual elements out of the tuple *Signal (s,o)* returned by *mealyT*. That is, we can not write: (*r'*, *outp*) = *mealyT* ⟨$⟩ *r* ⟨*⟩ *inp*, because the expression: *mealyT* ⟨$⟩ *s* ⟨*⟩ *inp*, has type *Signal (s,o)*, and not *(Signal s, Signal o)*. The synchronisity assumption behind the *Signal* type does, however, ensure that the two types are *isomorphic*. We capture this isomorphism with the *Bundle* type class shown in listing 3.24, where we see both the class definition and two instance definitions.

The *Bundle* type class defines the associated type [11], *Unbundled a*, which captures the *other side of the isomorphism* with *Signal a*, and the *bundle* and *unbundle* functions that capture the transformations between these types. In the *instance* declarations we see how *Unbundled a* captures the other side of the isomorphism, where *Unbundled (a,b) = (Signal a, Signal b)* which is isomorphic to *Signal (a,b)*, and *Unbundled (a,b,c) = (Signal a, Signal b, Signal c)* which is isomorphic to *Signal (a,b,c)*. Using the functions from the *Bundle* class we can now write the (⟨^⟩) combinator as shown in listing 3.25.

Bundle type class

```
class Bundle a where
  type Unbundled a
  bundle   :: Unbundled a → Signal a
  unbundle :: Signal a → Unbundled a
```

Bundle instance for 2-tuple

```
instance Bundle (a,b) where
  type Unbundled (a,b) = (Signal a, Signal b)
  bundle   (a,b) = (,) ⟨$⟩ a ⟨*⟩ b
  unbundle ab    = (fst ⟨$⟩ ab,snd ⟨$⟩ ab)
```

Bundle instance for 3-tuple

```
instance Bundle (a,b,c) where
  type Unbundled (a,b,c) = (Signal a, Signal b, Signal c)
  bundle   (a,b,c) = (,,) ⟨$⟩ a ⟨*⟩ b ⟨*⟩ c
  unbundle abc     = ((λ(x,_,_) → x) ⟨$⟩ abc
                     ,(λ(_,x,_) → x) ⟨$⟩ abc
                     ,(λ(_,_,x) → x) ⟨$⟩ abc)
```

Listing 3.24 – Bundle type class and instances

3.4.4 Multiple clock domains

The *Signal* constructs gives rise to synchronous sequential circuits that are synchronised to a single, unnamed, clock. We can generalise the *Signal* construct by naming the clock the circuit is synchronised to, and consequently give rise to compositions of circuits that are synchronised to different clocks. We implement these *Signal*s with a named clock as a *newtype* wrapper around the *Signal* data type, using a *phantom* type to represent the clock to which the circuit is synchronised:

```
newtype CSignal (clk :: Clock) a = CSignal (Signal a)
```

The *kind* of the type parameter *clk* is *Clock*, which is a promoted data type [75] given by:

```
data Clock = Clk Symbol Nat
```

Where *Symbol* is a *type*-level *String*, and *Nat* is a *type*-level natural number. The *Symbol* is the actual *name* of clock, and the *Nat* represents the period of the clock, where this period is dimensionless. The periods are dimensionless as we are only interested in relative differences between clocks. That is, when we compare, *Clk* "A" 725, with, *Clk* "B" 225, the only statement that we want to make is that "A" has a $\frac{29}{9}$ longer period than "B". The reason to include both the name and the period of the clock will be made clear at the end of this section.

```
data Clock = Clk Symbol Nat

data SNat (n :: Nat)      where SNat :: KnownNat n      ⇒ SNat n
data SSym (s :: Symbol)   where SSym :: KnownSymbol s ⇒ SSym s

data SClock (clk :: Clock) where
  SClock :: SSym name → SNat rate → SClock (Clk name rate)
```

Listing 3.26 – *SClock* singleton type

For similar safety reasons as those for hiding the (:−) constructor for the *Signal* type, so too will the *CSignal* constructor be hidden from the circuit designer. As for the *Signal* type, we expose one primitive, *cregister*, which can introduce memory elements in circuit specifications working on *CSignals*:

```
cregister :: SClock clk → a → CSignal clk a → CSignal clk a
cregister _ i (CSignal s) = CSignal (i :− s)
```

Which has nearly the same signature as the *register* function, except that it gets an extra *singleton* [15] argument that represents the clock to which the register is synchronised. The singleton type, *SClock clk*, is used as the value representation, or witness, of the *clk* type which encodes the clock to which the circuit is synchronised to. We define *SClock* in listing 3.26, where the constructor, *SClock*, uses the singleton types *SSym* and *SNat* as arguments for the name and period of the clock.

We need these singleton types because Haskell has a clear phase-distinction between values / terms and types, meaning, amongst others, that types cannot be indexed by terms. In a dependently typed language, such as Idris [9], we could have defined *cregister* as:

```
cregister : (clk : Clock) → a → CSignal clk a → CSignal clk a
cregister _ i (CSignal s) = CSignal (i :− s)
```

Where *clk* would be a normal value, instead of a type.

Composition of multiple clock domains

By encoding the clock in *CSignal*'s type we ensure, in a type-safe manner, that circuits synchronised to one clock do not access signals belonging to a circuit synchronised to another clock by accident. That is, applying a function *f*, with the type:

```
f :: CSignal (Clk "A" 725) Bool → CSignal (Clk "A" 725) Bool
```

To an argument *a*, with the type:

```
a :: CSignal (Clk "B" 225) Bool
```

Gives rise to a type error. These so-called clock domain crossings should only happen under clear intent of the circuit designer. That is, the designer will have to explicitly compose *f* with function a *sync* that has the type:

```
sync :: CSignal (Clk "B" 225) Bool → CSignal (Clk "A" 775) Bool
```

Which can synchronise the signals in clock domain "B" to clock domain "A".

Signals that originate from a circuit synchronised to a clock "A", and connected to a circuit synchronised to a clock "B", are considered *asynchronous* inputs for the circuit synchronised to clock "B". Asynchronous inputs can induce meta-stability in bi-stable circuits, such as d-flipflops, which are used as memory elements in synchronous circuits. Memory elements in a meta-stable state output a value that is neither logically 0 nor logically 1, but somewhere in-between (or in some cases oscillating between the two, in a rate different from the clock rate). Such an error value can quickly lead to a complete (logical) failure of the circuit. Synchronisers are circuit elements that, as the name suggests, synchronise asynchronous inputs to the clock to which the rest of circuit is synchronised to. Although they do not completely prevent the memory elements in a circuit from ever going into a meta-stable state, they reduce the chance at meta-stability to a statistically acceptable level (e.g. once every ten years).

There are many possible designs for these synchronisers, each with their own suitability for a specific situation. The fact that meta-stability leads to failure, and that there are many possible synchroniser designs, led us the design where the clocks are encoded at the type-level in *CSignal*, thereby always forcing the designer to make an (informed) choice in synchroniser design when crossing clock domains.

Synchronisation primitive

Although the synchronisation circuits can be described in terms of the functions and primitives described earlier in this chapter, we need one extra primitive that does the actual conversion from, *CSignal ClkA a*, to, *CSignal ClkB a*. This primitive, *unsafeSynchroniser*, whose exact behaviour is described in appendix B, will basically be represented by a simple *wire* in the synthesised circuit. The name of the primitive is prefixed with the word *unsafe* because, although it converts a signal from one clock domain to the other, it does nothing to reduce the chance for a memory element to exhibit meta-stable behaviour. The type signature of *unsafeSynchroniser* is:

```
unsafeSynchroniser :: SClock clk1
                   → SClock clk2
                   → CSignal clk1 a
                   → CSignal clk2 a
```

Where we can see that the function has two singleton arguments, representing the clock corresponding to the incoming signal, and the clock representing the outgoing signal, respectively. The exact behaviour of the synchronisation primitive conforms to a specific scenario:

» All the clocks in the system are derived from a single master clock, that is, they are synchronised.

» There is a shared reset signal for all clock domains, which is de-asserted just before the simultaneous rising edges of all clock signals. All circuits thus *start* at the same time.

Working with *derived* clocks in a multi-clock system is very common (in FPGAs), so the behaviour of the synchronisation primitive covers many use-cases. This does, however, mean that, if we are in a situation where the circuits have completely independent clocks, the simulation behaviour will not correspond to every possible observed behaviour of the synthesised multi-clock circuit. It will, in that situation, only correspond to the *one* (rare) case that the clocks are accidentally in-sync.

So, in case the eventual circuit does work with derived clocks, and the circuit designer wants to exploit this feature in a multi-clock design, the designer will be able to observe the desired behaviour in simulation. When the eventual circuit works with independent clocks, the designer must make sure that the correct functioning of the circuit is not dependent on the *in-sync* behaviour exhibited by the synchronisation primitive.

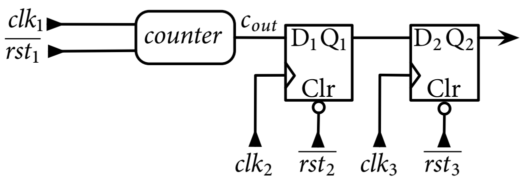
In listing 3.27, we can see: the code for a multi-clock circuit, the derived circuit, and the timing diagram corresponding to the derived circuit. The circuit has three clocks: one with a period of 3, *clk1*, one with a period of 2, *clk2*, and one with a period of 4, *clk3*. The circuit consists of a counter circuit, that starts counting from zero, and is synchronised to *clk1*. The counter is connected to a register, *R1*, synchronised to *clk2*, which is subsequently connected to a register, *R2*, synchronised to *clk3*. The top of the timing diagram shows the resets and three clocks, and the outputs of the individual sub-circuits. The clocks and the resets correspond to the scenario sketched earlier. There are shaded vertical lines indicating when the three clocks are synchronised: red when all three clocks are synchronised, and grey when only two of the three clocks are synchronised.

The bottom of the timing diagram shows two *hypothetical* signals that describe inputs of the two registers *R1* and *R2*. By *hypothetical* we mean: what the inputs of the registers *R1* and *R2* would have looked like if their inputs were synchronised to the same clock as the registers themselves. These signals, D_1^{in} and D_2^{in}, are derived by time-shifting Q_1 and Q_2 one cycle of their respective clocks into the past.

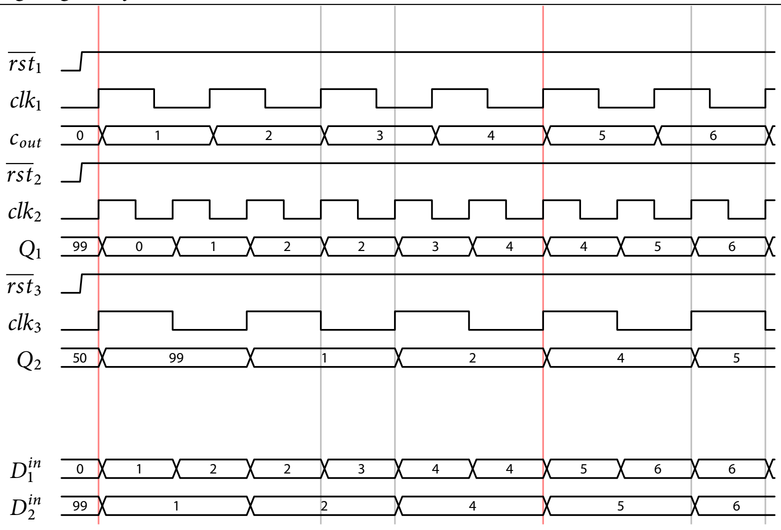
Design with three clock domains

```
clk₁  :: SClock (Clk "clk₁" 3)
clk₁  = SClock SSym SNat

clk₂  :: SClock (Clk "clk₂" 2)
clk₂  = SClock SSym SNat

clk₃  :: SClock (Clk "clk₃" 4)
clk₃  = SClock SSym SNat

counter clk = let s = cregister clk 0 (s + 1) in s

multiClock = cregister clk₃ 50
           $ unsafeSynchroniser clk₂ clk₃
           $ cregister clk₂ 99
           $ unsafeSynchroniser clk₁ clk₂
           $ c_out
  where
    c_out = counter clk₁
```

Derived circuit



Timing diagram of the derived circuit



Listing 3.27 – Multi-clock design: specification, circuit, and timing diagram

The synchronisation primitive, *unsafeSynchroniser*, is responsible for the creation of the streams corresponding to the timing diagrams of these *hypothetical* signals. So:

» Given the stream of values corresponding to the timing diagram of c_{out}: `[0,1,2,3,4, ..)`.

» The *unsafeSynchroniser* primitive creates the *oversampled* signal that corresponds to the timing diagram of D_1^{in}: `[0,1,2,2,3,4,4, ..)`.

Similarly:

» Given the stream of values corresponding to the timing diagram of Q_1: `[99 ,0,1,2,2,3,4,4, ..)`.

» The *unsafeSynchroniser* primitive creates the *compressed* signal corresponding to the timing diagram of D_2^{in}: `[99,1,2,4, ..)`.

Final remarks

Earlier we said we would explain why both the name and period of the clock are encoded in the *Clk* type: the reason that we do is to have a simple and safe synchronisation primitive. We encode the name of the clock so that we can distinguish two independent clocks that have the same period. Two independent clocks, even if they have the same period, must always be synchronised for the meta-stability reasons stated earlier. We should hence not model such a system as having a single clock. Choosing to encode the characteristics that set independent clocks with the same period apart, such as phase-difference and drift-rates, would complicate both the *CSignal* type, and the behaviour of the *unsafeSynchroniser* primitive. Only encoding the name, but not the period, of the clock is potentially unsafe: it would lead to a design where the periods would be specified as part of *unsafeSynchroniser*. This could give rise to situations where the *same* clock is erroneously attributed with *different* periods at distinct locations of *unsafeSynchroniser*.

3.5 Conclusions and future work

This chapter showed how we can model / specify both combinational and synchronous sequential circuits in CλaSH. Under the structural view, we can reason straightforwardly how a specification is synthesised to a circuit. In general, every function application is synthesised to an instantiation of the corresponding component. As a result, synthesised circuits are maximally parallel given their specification. A downside of the structural view is that sequential execution has to be manually specified: a designer will need to specify his own control logic to execute an algorithm sequentially. In SAFL [47] for example, which takes a behavioural synthesis approach, functions are only instantiated once, and function application is translated to control logic to facilitate concurrent access resulting from multiple function applications. In SAFL we would thus get a circuit that is (almost[5]) maximally sequential, and we have to duplicate functions to get a more parallel circuit. In Verity [22] there is also no need to specify one's own control logic: recursion using its *fix*-point construct is mapped to sequential circuitry. Parallel composition, however, has to be manually specified.

In CλaSH, we model feedback loops by *value recursion*. Haskell's non-strict evaluation mechanism makes both the specification and simulation of these circuit specifications far more natural than in a strict setting. In a strict setting, in order to simulate our specifications, we would have to make sure that all computations involved in a feedback loop are properly delayed by creating a *thunk*. One strict operation somewhere in the function hierarchy would force the complete computation of the feedback loop. As we cannot know if a function will be used in a feedback loop, we can only anticipate by creating thunks everywhere, basically mimicking non-strict evaluation.

In an earlier version of CλaSH, [CB:7], we did not use the *Signal* type to model the values manipulated by synchronous sequential circuits. Instead, we modelled synchronous sequential circuits by solely describing the combinational part of a Mealy machine (in the same way as we have seen them in section 3.4.3). As a result, a composition of sequential circuits also had to be modelled in the form of a Mealy machine. Additionally, the arguments and result representing the state of the subcomponents must be aggregated in the argument and result of the bigger Mealy machine. That is, given two Mealy machine descriptions, f and g, the function h that composes them will look, in general form, like:

```
h ((fS,gS),hS) inp = (((fS',gS'),hS'),outp)
  where
    (fS',  ...) = f fS  ...
    (gS',  ...) = g gS  ...
    hS'         = ...
    outp        = ...
```

Where *fS*, *gS*, *fS'*, and *gS'*, correspond to the state of f and g, and, *hS* and *hS'* correspond to the state of h. The downside of this old approach is that the aggregation is very repetitive, pollutes the local namespace of a function, and is prone to errors that break the Mealy machine abstraction. By the latter we mean that sub-states of the Mealy machines must be aggregated unchanged, and the current and updated states have to appear in the same order across the argument and the result. That is, the following definition of h would be erroneous:

```
h ((fS,gS),hS) inp = (((gS',fS'),hS'),outp)
  where
    ...
```

[5]Primitive operations in SAFL are implicitly duplicated, and will hence operate in parallel.

Where the updated state of *g*, *gS'*, is aggregated as the updated state of *f*, and visa-versa. Depending on the types of *fS'* and *gS'*, such a switch might be caught as a type-error, or it might, in a more unfortunate situation, not be caught as a type error.

In [CB:9] we describe how we encapsulate our Mealy machines description in an automata arrow [53], and use the arrow notation [52] for the composition of sequential circuits. The encapsulation in an automata arrow abstracts the process of subdivision and aggregations of substates, thus solving the problems mentioned earlier regarding manual subdivision and aggregation. Using arrows does, however, have two downsides:

» Arrows can only have one input: multiple inputs must hence be aggregated in a tuple. Functions / Arrows representing a sequential circuit can hence not be specified in a curried form. Consequently, if one wants to partially apply an arrow, he / she will need to wrap the arrow in a lambda abstraction first.

» The arrow notation has its own local scope, meaning variables bound within the arrow notation can only be referenced within the arrow-notation block, and not from the **where** clause.

By using the *Signal* type as presented in this chapter we suffer from neither of the above problems:

» There is no special syntax for the composition of functions representing sequential circuits, and consequently no separate scope. Composition can be done within regular **where**-clauses and **let** -expressions.

» We are not forced to aggregate the inputs of a function representing a sequential circuit into a tuple, meaning that a sequential circuit can be modelled as a curried function. Consequently, partial application of a function representing a sequential circuit is just as straightforward as partially applying any other curried function.

3.5.1 Future work

Working with values of the *Signal* type precludes the use of straightforward pattern matching in the specifications of sequential circuits. In section 3.4.3 we already saw that a *Signal* of a product type is isomorphic to the product of *Signal* types, a fact exploited by the *Bundle* type-class. Pattern matching on *Signals* of product types is hence possible by manually inserting *unbundle* at the place we want to pattern match. A possible extension, beyond what is currently supported by either Haskell or the language extensions of GHC, is to insert *bundle* and *unbundle* automatically, knowing that the isomorphism in the *Bundle* type class is guaranteed by the synchronisity assumption underlying *Signal*. We could subsequently use pattern matching on product types in specifications of sequential circuits, without having to worry where to add *bundle* and *unbundle*.

Pattern matching on *Signals* of sum types is even more difficult. First of all, there is no isomorphism between a *Signal* of a sum-type, and a sum(-of-products) of a *Signal*, meaning that we cannot define a *Bundle* instance for sum types. What is, however, possible is to emulate patterns and pattern matching using the techniques for Lava, shown earlier in chapter 2. The disadvantages of those techniques have also already been described in chapter 2. Those techniques also do not offer a way for functions to be defined in terms of multiple clauses. Future work lies in defining a synchronous semantics for pattern matching on sum-types from which an implementation, perhaps using source-to-source transformations to a specification using the techniques in chapter 2, can follow.

4
Type-Directed Synthesis

Abstract – *A straightforward synthesis from functional languages to digital circuits transforms variables to wires. The types of these variables determine the bit-width of the wires. Assigning a bit-width to polymorphic and function-type variables within this direct synthesis scheme is impossible. Using a term rewrite system, polymorphic and function-type binders can be completely eliminated from a circuit description, given only minor and reasonable restrictions on the input.*

4.1 Introduction

In the previous chapter we discussed the use of Haskell for specifications of synchronous digital circuits and the structural interpretation of these descriptions. This chapter will discuss the CλaSH compiler, whose purpose is to produce a *netlist* from the high-level Haskell description. Industry standard tools can then be used for further processing, such a programming an FPGA or creating an ASIC.

This chapter is split up roughly into two parts, the first part describes the compiler pipeline and certain engineering details. The second part provides details about the *synthesis* process. The translation from a (functional) description to a netlist is called *synthesis* in hardware literature (where it would be called *compilation* in software literature). The set of rules and transformations that together describe the conversion procedure from *description* to *netlist* is called a *synthesis scheme*.

The CλaSH compiler uses a term rewrite system (TRS), with bound variables, to drive the synthesis process. The objective of the TRS is to produce a *normal form* of the description which can be trivially converted to a netlist. This chapter provides proofs that the transformations of the TRS preserve the types and semantics of

Parts of this chapter have been published in [CB:13] expressions. Additionally we prove that, given reasonable restrictions, the TRS will always find the normal form.

The next subsection gives both a definition for netlists, and an introduction to synthesis schemes by describing a specific instance for a small functional language. The definition and introduction are both informal, but hopefully instil an intuition for the process of transforming a functional description to an actual circuit.

4.1.1 Netlists & Synthesis

A netlist is a textual description of a digital circuit [17]. It lists the components that a circuit contains, and the connections between these components. The connection points of a component are called ports, or pins. The ports are annotated with the bit-width of the values that flow through them. A netlist can either be hierarchical or flattened. In a hierarchical netlist, sub-netlists are abstracted to appear as single components, of which multiple instances can be created. By instantiating all of these instances, a flattened netlist can be created.

A synthesis scheme defines the procedure that transforms a (functional) description to a netlist. Synthesis schemes defined for different languages, which nonetheless have common aspects, will be called a synthesis scheme *family*. The CλaSH compiler uses a synthesis scheme, which we will call $\mathcal{T}_{C\lambda}$, that is an instance of the larger synthesis scheme family that will be referred to as $\mathcal{T}$. The following aspects are shared by all instances of $\mathcal{T}$:

» It is completely syntax-directed.

» It creates a hierarchical netlist.

» Function *definitions* are translated to components, where the arguments of the function become the input ports, and the result is connected to the output port.

» Function *application* is translated to an instance of the component that represents the corresponding function. The applied arguments are connected to the input ports of the component instance.

To demonstrate $\mathcal{T}$, a simple functional language, $\mathcal{L}$, is introduced in figure 4.1. $\mathcal{L}$ is a pure, simply-typed, first-order functional language. A program in $\mathcal{L}$ consists of a set of function definitions, which always includes the *main* function. Expressions in $\mathcal{L}$ can be: variable references, primitives, or function application. Figure 4.3 shows a small example program defined in the presented functional language.

The synthesis scheme for $\mathcal{L}$, called $\mathcal{T}_{\mathcal{L}}$, is defined by two transformations: $[\![\ ]\!]_p$ and $[\![\ ]\!]_e$, in which $[\![\ ]\!]_p$ is initially applied to the *main* function to create the hierarchical netlist. A graphical definition of the $[\![\ ]\!]_p$ and $[\![\ ]\!]_e$ transformations is depicted in figure 4.2. Again, the purpose of this subsection is to give an intuition for the synthesis process, not to give a formal account of $\mathcal{T}_{\mathcal{L}}$. The transformation $[\![\ ]\!]_p$ creates a component definition for a function f, where input ports correspond to

$p ::= f\ \overline{x} = e;\ p$	Function definitions
$\quad \mid\ main\ \overline{x} = e$	Main function
$e ::= x$	Argument reference
$\quad \mid\ \otimes\ \overline{e}$	Primitive
$\quad \mid\ f\ \overline{e}$	Function application

Figure 4.1 – $\mathcal{L}$: a simple functional language

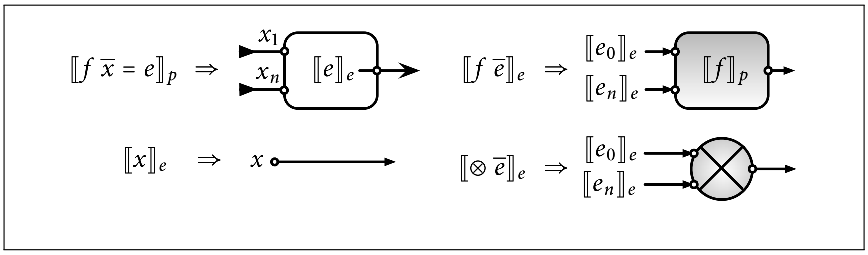

$[\![f\ \overline{x} = e]\!]_p \Rightarrow$
x_1
x_n
$[\![e]\!]_e$
$[\![f\ \overline{e}]\!]_e \Rightarrow$
$[\![e_0]\!]_e$
$[\![e_n]\!]_e$
$[\![f]\!]_p$
$[\![x]\!]_e \Rightarrow x$
$[\![\otimes\ \overline{e}]\!]_e \Rightarrow$
$[\![e_0]\!]_e$
$[\![e_n]\!]_e$

Figure 4.2 – $\mathcal{T}_{\mathcal{L}}$: A synthesis scheme for $\mathcal{L}$

```
double x = x * x
main x y = (double x) + (double y)
```

Figure 4.3 – Example program in $\mathcal{L}$

the argument of f. Additionally, $[\![\ ]\!]_p$ creates an output port for the result of the expression e, which is connected to the outcome of the $[\![\ ]\!]_e$ transformation applied to e.

Figure 4.2 shows that $[\![\ ]\!]_e$ transforms a reference to an argument x, to a connection with the input port x. Function application of a function f is transformed to an instantiation of the component f. The $[\![\ ]\!]_p$ transformation will be called for the definition of f in case there is no existing component definition. Arguments to f are subsequently transformed by $[\![\ ]\!]_e$, and the outcome of these transformations are connected to the input ports of the component f. The process for the transformations of primitives is analogous to that of functions, except that $[\![\ ]\!]_p$ will not be called for the definitions.

Applying the synthesis scheme $\mathcal{T}_{\mathcal{L}}$ to the example program given in figure 4.3 results in the (graphical representation of the) netlists depicted in figure 4.4. The netlist representation of *main* shows that synthesis schemes belonging to $\mathcal{T}$ ex-

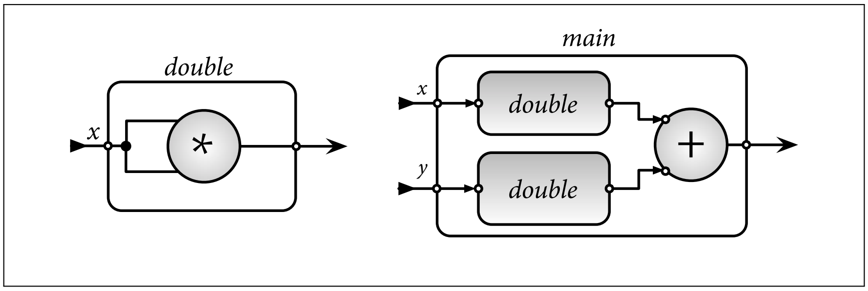


Figure 4.4 – Netlist of the example program in figure 4.3, created by $\mathcal{T}_{\mathcal{L}}$

ploit the *implicit parallelism* present in (pure) functional languages: as there are no dependencies between the operands of the addition, they are instantiated side-by-side. During the actual operation of the circuit, electricity flows through all parts simultaneously, and the instances of *double* will actually be operating in parallel.

4.2 Compiler pipeline

Now that we have seen an informal introduction to the synthesis of functional languages, we will move on to some of the engineering details of the CλaSH compiler. The CλaSH compiler is not a complete rewrite of a Haskell compiler targeted at doing circuit synthesis. It actually reuses a large part of GHC. The CλaSH compiler is roughly split into three parts:

Front-end: The front-end of the compiler exists of parsing, type-checking, and desugaring the Haskell code. For this part of the compiler, CλaSH uses the GHC API [65]. The GHC API exposes the internals of the GHC compiler as a library for other projects to use. The output of this stage is not an abstract syntax tree *(AST)* of the Haskell code, but an AST of the internal core language of GHC: System FC [62, 68, 69, 75].

Normalisation: The description in System FC still contain constructs which are non-trivial to translate to a netlist. The normalisation process uses a term rewrite system (TRS), as mentioned earlier in the chapter, to transform the System FC description into a shape where these untranslatable constructs no longer exists. The inner workings of this TRS are discussed in great detail in the second part of this chapter. The result of this stage of the compiler is thus a System FC description in a *normal form* suitable for a translation to a netlist.

Netlist generation: When the System FC description has been transformed into its suitable normal form, it is converted to a generic netlist data type. This generic netlist data type is currently pretty printed to VHDL, although generation of Verilog is not precluded.

Lambda calculus

The lambda calculus is a formal system for expressing computation in terms of functions. Its grammar for expressions is:

$$
\begin{array}{lcll}
e, u & ::= & & \text{Expressions} \\
 & | & x & \text{Variable reference} \\
 & | & \lambda x.e & \text{Abstraction} \\
 & | & e\ u & \text{Application}
\end{array}
$$

Where the expression $\lambda x.e$ is an anonymous function with an argument named x, and body e. A computational step is described by beta reduction:

$$(\lambda x.e)\ u \Longrightarrow e[u/x]$$

Which states that, when there is an application of an abstraction, all occurrences of x in e are substituted by the expression u.
The lambda calculus can be expanded to express more properties of an expression. For example, we can annotate function arguments with a type, $\lambda x : Bool.e$, and subsequently check if expressions are correct according to their types. Continuing this line of reasoning we can start abstracting over the types of function arguments, $\Lambda a.\lambda x{:}a.e$, where the function is now *polymorphic* in its argument.

A large part of the compiler, and certainly this thesis, centres around System FC, so we will first introduce System FC before moving on with discussing the rest of the compiler pipeline.

4.2.1 System FC

The syntactically rich Haskell language is desugared to a much smaller language, called System FC [62], by the GHC API [65]. The actual AST corresponds most to a combination of the calculi System $F_C^{\uparrow}$ [75] and System FC_2 [68], extended with recursive let-expressions. We will, however, present our work in the context of the latest incarnation of System FC, as described by Weirich et al. [69]. This latest version generalises the earlier work on System FC. So even though the implementation of the CλaSH compiler works with an AST corresponding to an older version of System FC, the proofs and theory as presented in this thesis carry over to the implementation.

System FC [69] is a Church-style polymorphic lambda-calculus with first-class type quality proofs which are called *coercions*. It supports generalized algebraic data types *(GADTs)* [74], and type-level functions (encoded in the Haskell source language in the form of type families [11, 16]). Novel in the work of Weirich et al. [69], compared to earlier versions of System FC, is the unified type and kind

Typing judgements

A typing *judgement* is a formal statement about the type of an expression. The most basic form is:

$$\Gamma \vdash e \ : \ \tau$$

which expresses: in the context Γ, the expression e, has the type τ. These judgements are used in type derivation rules. For example, the following inference rule:

$$\frac{\Gamma, x : \sigma \vdash e \ : \ \tau}{\Gamma \vdash \lambda x : \sigma . e \ : \ \sigma \to \tau}$$

states that: if we can derive that e has the type τ in the context Γ, extended with the variable x of type σ. Then we can derive that e abstracted over x, $\lambda x : \sigma . e$, has the function type $\sigma \to \tau$. Sometimes we add a tag to a judgement, by subscripting the turnstile ($\vdash$), to indicate that the judgements refers to a particular set of type derivation rules. For example, the judgement:

$$\Gamma \vdash_{\text{co}} \gamma \ : \ \tau \sim \sigma$$

states that the coercion γ witnesses the equality between types τ and σ. The $_{\text{co}}$ subscript in $\vdash_{\text{co}}$ indicates that this is a judgement in the typing derivation rules for coercions.

(the type of types) hierarchy. We will present a slightly adapted version, where the expression language is extended with: recursive let-expressions, primitive operations, and projections. The grammar for this slight variant of System FC is given in figure 4.5; it uses, like the rest of this chapter, the notation described in figure 4.7. For space reasons, the grammar for coercions is put in a separate figure, figure 4.6.

Coercions

Coercions are the distinguishing feature of System FC, although they play only a small role in this thesis. Coercions witness the *non-syntactical* equality of two types, that is, the judgement:

$$\Gamma \vdash_{\text{co}} \gamma \quad : \quad \tau_1 \sim \tau_2$$

checks that the coercion γ proves that τ_1 and τ_2 are equal. Using casts, $e \triangleright \gamma$, we can *safely* coerce the types of expressions:

$$\frac{\Gamma \vdash_{\text{tm}} e \ : \ \tau_1 \qquad \Gamma \vdash_{\text{co}} \gamma \ : \ \tau_1 \sim \tau_2}{\Gamma \vdash_{\text{tm}} e \triangleright \gamma \ : \ \tau_2}$$

So given that the expression e has type τ_1, and the coercion γ is a witness of the equality $\tau_1 \sim \tau_2$, the type of $e \triangleright \gamma$, is τ_2.

H	$::=$		Type constants
	$\mid$	$(\rightarrow)$	Arrow
	$\mid$	$\star$	Type/Kind
	$\mid$	T	Type constructor
	$\mid$	K	Promoted data constructor
w	$::=$		Type-level names
	$\mid$	a	Type variables
	$\mid$	F	Type functions
	$\mid$	H	Type constants
σ, τ, κ	$::=$		Types and Kinds
	$\mid$	w	Names
	$\mid$	$\forall a : \kappa . \tau$	Polymorphic types
	$\mid$	$\forall c : \phi . \tau$	Coercion abstr. type
	$\mid$	$\tau_1\ \tau_2$	Type/kind application
	$\mid$	$\tau \rhd \gamma$	Casting
	$\mid$	$\tau\ \gamma$	Coercion application
ϕ	$::=$	$\sigma \sim \tau$	Propositions (coercion kinds)
e, u	$::=$		Expressions
	$\mid$	x	Variables
	$\mid$	$\lambda x : \tau . e$	Abstraction
	$\mid$	$e_1\ e_2$	Application
	$\mid$	$\Lambda a : \kappa . e$	Type/kind abstraction
	$\mid$	$e\ \tau$	Type/kind application
	$\mid$	$\lambda c : \phi . e$	Coercion abstraction
	$\mid$	$e\ \gamma$	Coercion application
	$\mid$	$e \rhd \gamma$	Casting
	$\mid$	K	Data constructors
	$\mid$	$\mathbf{case}\ e\ \mathbf{of}\ \overline{p \rightarrow u}$	Case decomposition
	$\mid$	$\mathbf{let}\ \overline{x : \sigma = e}\ \mathbf{in}\ u$	Recursion let-expression
	$\mid$	$\otimes$	Primitive operation
	$\mid$	$\pi_i^k\ e$	Constructor field projection
p	$::=$		Patterns
	$\mid$	$K\ \Delta\ \overline{x : \tau}$	Constructor pattern
	$\mid$	$\_$	Default pattern
Δ	$::=$		Telescopes
	$\mid$	$\varnothing$	Empty
	$\mid$	$\Delta, a : \kappa$	Type variable binding
	$\mid$	$\Delta, c : \phi$	Coercions variable binding

Figure 4.5 – System FC grammar

γ, η	$::=$		Coercions
	$\mid$	c	Variables
	$\mid$	$C\,\overline{\rho}$	Axiom application
	$\mid$	$\langle\tau\rangle$	Reflexivity
	$\mid$	$\mathbf{sym}\,\gamma$	Symmetry
	$\mid$	$\gamma_1 \fatsemi \gamma_2$	Transitivity
	$\mid$	$\forall_{\eta}(a_1, a_2, c).\gamma$	Type/kind abstraction congruence
	$\mid$	$\forall_{(\eta_1,\eta_2)}(c_1, c_2).\gamma$	Coercion abstraction congruence
	$\mid$	$\gamma_1\;\gamma_2$	Type/kind application congruence
	$\mid$	$\gamma(\gamma_2, \gamma_2')$	Coercion application congruence
	$\mid$	$\gamma \rhd \gamma'$	Coherence
	$\mid$	$\gamma@\gamma'$	Type/kind instantiation
	$\mid$	$\gamma@(\gamma_1, \gamma_2)$	Coercion instantiation
	$\mid$	$\mathbf{nth}^{i}\,\gamma$	n^{th} argument projection
	$\mid$	$\mathbf{kind}\,\gamma$	Kind equality extraction
ρ	$::=$	$\tau \mid \gamma$	Type or coercion

Figure 4.6 – System FC coercion grammar

$w\,\overline{\sigma}$	$\equiv w\;\sigma_1\;..\;\sigma_n$	$e\,\overline{u}$	$\equiv e\;u_1\;..\;u_n$
$\tau \to \sigma$	$\equiv (\to)\;\tau\;\sigma$	$\lambda\overline{x:\sigma}.e$	$\equiv \lambda x_1 : \sigma_1 ..\,.\lambda x_n : \sigma_n.e$
$\overline{\tau} \to \sigma$	$\equiv \tau_1 \to .. \to \tau_n \to \sigma$	$\overline{x:\sigma = e}$	$\equiv \{x_1 : \sigma_1 = e_1,\;..,\;x_n : \sigma_n = e_n\}$
$\forall\overline{a:\kappa}.\sigma$	$\equiv \forall a_1 : \kappa_1 ..\,.\forall a_n : \kappa_n.\sigma$	$\overline{p \to u}$	$\equiv \{p_1 \to u_1,\;..,\;p_n \to u_n\}$

Figure 4.7 – Notation

As stated earlier, System FC supports type level functions, through the use of coercion axioms. Copying the example from Weirich et al. [69], the following type family declaration in Haskell:

```
type family F a :: *
type instance F Bool = Int
```

declares a type function called *F*, which has a single argument. And it subsequently declares that, when this type argument is *Bool*, then the result of the type function is *Int*. This in turn gives rise to the following *axiom* in System FC:

$$axF \quad : \quad F\;Bool \sim Int$$

which states that the axiom *axF* is a witness to the equality between the type function *F* applied to *Bool*, and the data type *Int*. So, given a Haskell function that has the type:

```
f :: a → F a → Char
```

the Haskell expression, *f True* 3, is desugared to the System FC expression:

```
f Bool True (3 ▷sym axF)
```

Where the literal 3, which has type *Int*, is coerced to type, *F Bool*, using, **sym** *axF*, as the witness. The *symmetry* coercion constructor, **sym**, creates, as its name suggests, a witness for the equality symmetric to the equality witnessed by its argument:

$$\frac{\Gamma \vdash_{co} axF \;:\; F\ Bool \sim Int}{\Gamma \vdash_{co} \mathbf{sym}\ axF \;:\; Int \sim F\ Bool}$$

Having a consistent set of axioms is paramount for soundness of the type system, as we do not want to be able to build coercions that witness *bogus* equalities such as *Bool* ~ *Int*. We refer the reader to [62] for further discussion on consistency of the axiom set.

GADTs

GADT are also encoded with coercions, where pattern matching on constructors binds coercion variables, which facilitate *type assumptions* in the context of the matched pattern. For example, the following Haskell code:

```
data DT a where
  MkI :: Int  → DT Int
  MkB :: Bool → DT Bool

f :: a → DT a → a
f i (MkI j) = i + j
f a (MkB b) = a && b
```

gives rise to the following System FC code:

```
MkI : ∀ a :*. ∀ c:a~Int  → Int  → DT a
MkB : ∀ a :*. ∀ c:a~Bool → Bool → DT a

f = Λa:*.λx:a.λds:DT a.case ds of
  MkI (cv:a~Int)  (j:Int)  → ((x ▷cv) + j)  ▷sym cv
  MkB (cv:a~Bool) (b:Bool) → ((x ▷cv) && b) ▷sym cv
```

where the data type constructors, *MkI* and *MkB*, both have three arguments: a type variable (corresponding to the type parameter of the type constructor *T*), a coercion variable, and an argument for the field. This coercion argument will carry a witness that the type variable a is actually equal to either *Int* or *Bool*.

When we pattern match on the constructor *MkI*, we can safely assume that the argument x, of type a, is actually of type *Int*. We thus coerce x to be of type *Int* using a cast, so that it can be safely applied to $(+ : Int \to Int \to Int)$, together with the field j. The result is subsequently coerced to type a by, **sym** cv : $Int \sim a$.

In general, and this is checked by a well-formedness judgement on the context (appendix C), type constructors T, are of kind:

$$T : \forall\overline{a:\kappa}.\star.$$

The *parameters* of the type can only be type and kind variables, but not coercion variables. Data constructors K, in general, are of type:

$$K : \forall\overline{a:\kappa}.\forall\Delta.\overline{\sigma} \to T\ \overline{a}.$$

Where the list of type arguments, $\overline{a:\kappa}$, matches and saturates the parameters $\overline{a}$ of the type constructor T. Next follows a telescope, of the form $\forall\Delta.\tau$, which is a list of nested quantified types (see figure 4.5 for the grammar of telescopes). Every variable bound in the telescope scopes over the remainder of the telescope, and the quantified type, in the above case $\overline{\sigma} \to T\ \overline{a}$. The telescope corresponds to the *existential* arguments of a data constructor, and contains both type and coercion arguments. For example, the Haskell datatype declaration:

```
data Vec :: Nat → * → * where
  Nil  :: Vec 0 a
  Cons :: a → Vec n a → Vec (n+1) a
```

gives rise to the System FC constructors:

```
Nil  : ∀ n:Nat.∀ a:*. ∀ c:n~0 → Vec n a
Cons : ∀ n:Nat.∀ a:*. ∀ n1:Nat.∀ c:n~n1+1.a → Vec n1 a → Vec n a
```

where, for the *Cons* constructor, n:*Nat* and a:* correspond to the parameters of the type constructor, and ∀ n1:*Nat*.∀ c:n~n1+1 is the telescope, the existential arguments of the constructor.

Extensions

We extend the presentation of System FC in Weirich et al [69] by: default patterns, recursive let-expressions, primitives, and projections. Case-decompositions in System FC are only well-formed when their alternatives are exhaustive in the possible constructors of the subject. By including the default pattern, being exhaustive no

$\boxed{\Gamma \vdash_{\text{tm}} e \ : \ \tau}$ **Expression typing**

$$\text{T\_CASE}\ \frac{\Gamma \vdash_{\text{tm}} e \ : \ T\,\overline{\tau'} \qquad \overline{\Gamma \vdash_{\text{alt}} p_i \to u_i \ : \ T\,\overline{\tau'} \to \tau}}{\Gamma \vdash_{\text{tm}} \textbf{case}\ e\ \textbf{of}\ \overline{p \to u} \ : \ \tau}$$

$$\text{T\_LETREC}\ \frac{\overline{\Gamma, \overline{x : \sigma} \vdash_{\text{bind}} x_i : \sigma_i \leftarrow e_i} \qquad \Gamma, \overline{x : \sigma} \vdash_{\text{tm}} u \ : \ \tau}{\Gamma \vdash_{\text{tm}} \textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ u \ : \ \tau}$$

$$\text{T\_PRIM}\ \frac{\vdash_{\text{wf}} \Gamma \qquad \otimes : \tau \in \Gamma}{\Gamma \vdash_{tm} \otimes \ : \ \tau}$$

$$\text{T\_PROJ}\ \frac{\Gamma \vdash_{\text{tm}} e \ : \ T\,\overline{\tau'} \qquad \Gamma \vdash_{\text{alt}} K_k\ \Delta\ \overline{x : \tau'} \to x_i \ : \ T\,\overline{\tau'} \to \tau}{\Gamma \vdash_{\text{tm}} \pi_i^k\ e \ : \ \tau}$$

$\boxed{\Gamma \vdash_{\text{alt}} p \to e \ : \ \sigma \to \tau}$ **Alternative typing**

$$\text{T\_ALTDEF}\ \frac{\Gamma \vdash_{\text{tm}} e : \tau \qquad \Gamma \vdash_{\text{ty}} \tau \ : \ \star \qquad \Gamma \vdash_{\text{ty}} \sigma \ : \ \star}{\Gamma \vdash_{\text{alt}} \_ \to e \ : \ \sigma \to \tau}$$

$$\text{T\_ALTCON}\ \frac{\begin{array}{l} \Gamma \vdash_{\text{ty}} \tau \ : \ \star \\ K : \forall \overline{a : \kappa}. \forall \Delta. \overline{\sigma} \to (T\,\overline{a}) \in \Gamma \\ \Delta' = \Delta[\overline{\tau'/a}] \\ \overline{\sigma' = \sigma[\overline{\tau'/a}]} \\ \Gamma, \Delta', \overline{x : \sigma'} \vdash_{\text{tm}} u \ : \ \tau \end{array}}{\Gamma \vdash_{\text{alt}} K\ \Delta'\ \overline{x : \sigma'} \to u \ : \ T\,\overline{\tau'} \to \tau}$$

$\boxed{\Gamma \vdash_{\text{bind}} x : \sigma \leftarrow e}$ **Binding typing**

$$\text{T\_BIND}\ \frac{\Gamma \vdash_{\text{tm}} e \ : \ \sigma}{\Gamma \vdash_{\text{bind}} x : \sigma \leftarrow e}$$

FIGURE 4.8 – Extended typing rules for System FC

longer entails enumerating all the constructor patterns. The default pattern is only chosen when none of the constructor patterns can be matched.

Let-expressions introduce local recursion, and, in the context of this thesis, are mainly used to model feedback loops. We note that the let-binders are also allowed to form a non-recursive binding group.

There are certain operations that are primitive to the system, such as addition of integers. Primitives are well-formed when they are monomorphic, and only have data types (T in the grammar given by figure 4.5) as arguments and result.

The projection construct, $\pi_i^k\ e$, projects (or extracts) the i'th field of the k'th constructor of a data type T, from the expression e. This construct is not introduced by the desugaring process from Haskell to System FC. It is created by the normalisation process where it is (type-)safe to do. When the data type has only one constructor we omit the k annotation, and just write π_i.

We present the typing rules for these extensions in figure 4.8 and the operational semantics for the extensions in figures 4.9, 4.10, and 4.11. For an overview of all the typing rules and operational semantics we refer the reader to appendix C. Most of the typing rules are straightforward.

In the typing rule for primitives, T_Prim, we perform a well-formedness check to ensure that the primitive is monomorphic and only has data types (T) as arguments and result. Type checking for alternatives has now moved to two separate judgements, of the form $\Gamma \vdash_{\mathrm{alt}} p \to e \ :\ \sigma \to \tau$, one for the default pattern (T_AltDef) and one for constructor patterns (T_AltCon). We refer the reader to [69] for the workings of the typing judgement for constructor patterns. The type rule for projection, T_Proj, uses the typing judgement for constructor pattern alternatives, where the matched constructor is the k^{th} constructor of the data type, and the expression is a variable reference to the i^{th} field of the constructor.

The T_LetRec rule is the typing-rule for recursive let-expressions. In the premise it says that every binding i, $x_i : \sigma_i = e_i$, is checked in the context of the environment, Γ, and all let-bound variables, $\overline{x : \sigma}$, are checked using the bind typing rule $\vdash_{bind}$. The typing rule for bindings says that, if the expression e has the type σ, then e bound to the variable x with type σ is a valid binding.

On to the operational semantics. To support recursion of let-expressions, we use an extra context Σ_{let} that keeps track of the bindings in the recursive group. This extra context does not influence the proof of the type preservation theorem (theorem C.4.1) for the existing typing rules and operational semantics as presented in [69]. The rule S_LetRec, takes a step on the body of the let-expressions, with the bindings in scope of the extra context Σ_{let}, and the result is the let-expression with the one-step reduced body.

The progress theorem roughly states that if an expression is not a *value*, always one of the step-reduction rules applies so that an evaluator for expressions can make *progress*. Values, and their types, *value types*, are defined by the grammar:

$$\begin{array}{lll} v & ::= & \lambda x : \sigma . e \mid \Lambda a : \kappa . e \mid \lambda c : \phi . e \mid K\, \overline{\tau}\, \overline{\rho}\, \overline{e} \mid \otimes\, \overline{v} \\ \xi & ::= & \sigma_1 \to \sigma_2 \mid \forall a : \kappa . \sigma \mid \forall c : \phi . \sigma \mid T\, \overline{\sigma} \end{array}$$

Compared to [69], we have extended our values with unsaturated primitives, $\otimes\, \overline{v}$. And, as we can see by the $\overline{v}$, the arguments of a primitive are reduced to values, meaning that primitives are strict in all of their arguments. We also update the canonical forms lemma, lemma 4.2.1.

$\boxed{\Sigma_{\text{let}}; e \longrightarrow e'}$ Step reduction

$\Sigma_{\text{let}} ::= \varnothing \mid \Sigma_{\text{let}}, x \mapsto e$

$$\text{S\_CaseDef}\ \frac{\_ \to u_i \in \overline{p \to u} \qquad \text{No other matches}}{\Sigma_{\text{let}}; \textbf{case}\ K_i\ \overline{\tau}\ \overline{\rho}\ \overline{e}\ \textbf{of}\ \overline{p \to u} \longrightarrow u_i} \qquad \text{S\_Var}\ \frac{\Sigma_{\text{let}}(x) = e}{\Sigma_{\text{let}};\ x \longrightarrow e}$$

$$\text{S\_LetRec}\ \frac{\Sigma_{\text{let}}, \overline{x \mapsto e};\ u \longrightarrow u'}{\Sigma_{\text{let}}; \textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ u \longrightarrow \textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ u'}$$

$$\text{S\_LetApp}\ \frac{}{\Sigma_{\text{let}}; (\textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ u)\ e' \longrightarrow \textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ (u\ e')}$$

$$\text{S\_LetTApp}\ \frac{}{\Sigma_{\text{let}}; (\textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ u)\ \tau \longrightarrow \textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ (u\ \tau)}$$

$$\text{S\_LetCApp}\ \frac{}{\Sigma_{\text{let}}; (\textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ u)\ \gamma \longrightarrow \textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ (u\ \gamma)}$$

$$\text{S\_LetCast}\ \frac{}{\Sigma_{\text{let}}; (\textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ u) \rhd \gamma \longrightarrow \textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ (u \rhd \gamma)}$$

$$\text{S\_LetFlat}\ \frac{}{\begin{array}{l}\Sigma_{\text{let}}; \textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ (\textbf{let}\ \overline{x' : \sigma' = e'}\ \textbf{in}\ u) \longrightarrow \\ \textbf{let}\ \overline{x : \sigma = e}, \overline{x' : \sigma' = e'}\ \textbf{in}\ u\end{array}}$$

$$\text{S\_LetCase}\ \frac{}{\begin{array}{l}\Sigma_{\text{let}}; \textbf{case}\ (\textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ u)\ \textbf{of}\ \overline{p \to u'} \longrightarrow \\ \textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ (\textbf{case}\ u\ \textbf{of}\ \overline{p \to u'})\end{array}}$$

Figure 4.9 – Operational semantics of System FC, default patterns and let-expression

Lemma: 4.2.1 (Canonical forms). *Say* $\Sigma \vdash_{\text{tm}} v : \sigma$, *where* Σ *is a closed context and* v *is a* value. *Then* σ *is a value type. Furthermore,*

1. *If* $\sigma = \sigma_1 \to \sigma_2$ *then* v *is* $\lambda x : \sigma_1.e$ *or* $K\ \overline{\tau}\ \overline{\rho}\ \overline{e}$ *or* $\otimes\ \overline{v}$.
2. *If* $\sigma = \forall a : \kappa.\sigma'$ *then* v *is* $\Lambda x : \kappa.e$ *or* $K\ \overline{\tau}\ \overline{\rho}\ \overline{e}$.
3. *If* $\sigma = \forall c : \phi.\sigma'$ *then* v *is* $\lambda c : \tau_1 \sim \tau_2.e$ *or* $K\ \overline{\tau}\ \overline{\rho}\ \overline{e}$.
4. *If* $\sigma = T\ \overline{\tau}$ *then* v *is* $K\ \overline{\tau}\ \overline{\rho}\ \overline{e}$ *or* $\otimes\ \overline{v}$.

Because we can construct co-inductive data types using our recursive let-expressions, it might be the case that the body of a let-expression does not reduce to a closed *value*, even though the expression is closed. The following example:

```
IntStream : *
SCons : Int → IntStream → IntStream

let f = SCons 1
    r = f r
in  r
```

$\boxed{\Sigma_{\text{let}}; e \longrightarrow e'}$ Step reduction

$\Sigma_{\text{let}} ::= \varnothing \mid \Sigma_{\text{st}}, x \mapsto e$

$$\text{S\_PrimLet}\ \frac{}{\Sigma_{\text{let}}; \otimes\,(\textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ u) \longrightarrow \textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ (\otimes\, u)}$$

$$\text{S\_PrimCast}\ \frac{\Gamma \vdash_{\text{co}} \gamma : T \sim T}{\Sigma_{\text{let}}; \otimes\,\overline{v}\,(v' \rhd \gamma) \longrightarrow \otimes\,\overline{v}\,v'} \qquad \text{S\_Prim}\ \frac{\Sigma_{\text{let}}; \overline{e} \longrightarrow \overline{e'}}{\Sigma_{\text{st}}; \otimes\,\overline{e} \longrightarrow \otimes\,\overline{e'}}$$

$$\text{S\_PrimDelta}\ \frac{\otimes\,\overline{v} \longrightarrow_{\delta} v'}{\Sigma_{\text{st}}; \otimes\,\overline{v} \longrightarrow v'}$$

Figure 4.10 – Operational semantics of System FC, primitives

will reduce to:

```
IStream : *
SCons : Int → IStream → IStream

let f = SCons 1
    r = f r
in  Cons 1 r
```

where the body of the let-expression is indeed a value, a fully applied constructor, but it is not closed. We therefore rephrase the progress theorem as theorem 4.2.1.

Theorem: 4.2.1 (Progress). *Assume a closed*[1]*, consistent, context* Γ. *If* $\Gamma \vdash_{\text{tm}} e_1 : \tau$ *and* e_1 *is not a value* v*, a coerced value* $v \rhd \gamma$*, or a let-abstracted version of either, then there exists an* e_2 *such that* $\Sigma_{\text{let}}; e_1 \longrightarrow e_2$.

As for the details behind the step reduction rules themselves, we want to note that most are quite straightforward. The rules for let-expressions are such that when the body of a let-expressions is a value, the application with either a term, type, or coercion can be moved inwards for further reduction. In case the let-expression is the subject of a cast, then the cast can be moved inwards by S_LetCast. The S_LetFlat rule merges a nested let-expression into a single let-expression. Finally, when the let-expression is the subject of a case-decomposition, then the let-bindings are moved outwards by S_LetCase.

The step reduction rules for primitives are such that the arguments of a primitive application become a value. Also, as we can see in the rule S_Prim, primitives are strict in all their arguments. Finally, the S_PrimDelta performs a δ-reduction step on a, primitive applied to all values, to compute a new value. For example, +# 3 4 $\longrightarrow_{\delta}$ 7.

[1]A context is closed if it does not contain any expression variable bindings.

$\boxed{\Sigma_{\text{st}}; e \longrightarrow e'}$ Step reduction

$\Sigma_{\text{st}} ::= \varnothing \mid \Sigma_{\text{st}}, x \mapsto e$

$$\text{S\_Proj}\ \frac{\Sigma_{\text{let}}; e \longrightarrow e'}{\Sigma_{\text{let}}; \pi_i^k\ e \longrightarrow \pi_i^k\ e'}$$

$$\text{S\_ProjKPush}\ \frac{\begin{array}{l} K : \forall \overline{a : \kappa}.\forall \Delta.\overline{\sigma} \to (T\ \overline{a}) \in \Sigma \\ \Psi = \mathsf{extend}(\mathsf{context}(\gamma); \overline{\rho}; \Delta) \\ \overline{\tau'} = \Psi_2(\overline{a}) \\ \overline{\rho}' = \Psi_2(dom\ \Delta) \\ \text{for each } e_i \in \overline{e} \\ \quad e_i' = e_i \rhd \Psi(\sigma_i) \end{array}}{\Sigma_{\text{st}}; \pi_i^k\ (K\ \overline{\tau}\ \overline{\rho}\ \overline{e} \rhd \gamma) \longrightarrow \pi_i^k\ (K\ \overline{\tau'}\ \overline{\rho}'\ \overline{e'})}$$

$$\text{S\_ProjLet}\ \frac{}{\Sigma_{\text{let}}; \pi_i^k\ (\mathbf{let}\ \overline{x : \sigma = e}\ \mathbf{in}\ u) \longrightarrow \mathbf{let}\ \overline{x : \sigma = e}\ \mathbf{in}\ (\pi_i^k\ u)}$$

$$\text{S\_ProjMatch}\ \frac{K_k\ \Delta_k\ \overline{x : \sigma} \to x_i}{\Sigma_{\text{let}}; \pi_i^k\ (K_k\ \overline{\tau}\ \overline{\rho}\ \overline{e}) \longrightarrow e_i[\overline{\rho}/\Delta_k]}$$

Figure 4.11 – Operational semantics of System FC, projection

Finally, the step reduction rules for projections match those of normal case-decomposition. Projections are only introduced by the normalisation process where the subject of the projection is guaranteed to reduce to the correct constructor. The most *involved* rule of the step-reduction rules for projections is S_ProjKPush, which is nearly a one-to-one copy of the S_KPush rule. The purpose of the S_ProjKPush rule is to *push* the cast through the data constructor application, so that the S_ProjMatch rule can subsequently be applied. We will explain the S_ProjKPush rule using the example in listing 4.1.

At the top of listing 4.1 we see the Haskell definitions for the type family F and datatype DT. The declaration of f is not really Haskell, as the projection construct does not exist in Haskell, it is there mainly for demonstrative purposes. In the middle of listing 4.1 we see the equivalent System FC axioms, and the expression f before the S_ProjKPush rule. At the bottom of listing 4.1 we see the expression f after the S_ProjKPush rule. The S_ProjKPush is as complex as it is, because it must apply many rewrites, and create new coercions, in order to be type preserving. The coercion around the *Mk* data constructor witnesses the equality:

$$\langle DT \rangle\ (\mathbf{sym}\ axFChar) \ :\ DT\ Int \sim DT\ (F\ Char)$$

So in order to push this coercion down, we must cast the data constructor field, *Haskell*

```
type family F Int  = Bool
type family F Char = Int

data DT a where
  Mk :: F a → DT a

f :: F (F Char)
f = π₁((MkDT True :: DT Int) :: DT (F Char))
```

System FC, Before S_ProjPush

```
axFInt  : F Int ~ Bool
axFChar : F Char ~ Int
Mk      : ∀ a.F a → DT a

f = π₁((Mk Int (True ▷sym axFInt)) ▷(⟨DT⟩ (sym axFChar)))
```

System FC, After S_ProjPush

```
f = π₁(Mk (F Char) ((True ▷sym axFInt) ▷(⟨F⟩ nth¹(⟨DT⟩ (sym axFChar)))))
```

Listing 4.1 – S_ProjKPush Example

which before S_ProjKPush has type:

$$\frac{\Gamma \vdash_{\mathrm{tm}} \mathit{True} \;:\; \mathit{Bool} \qquad \Gamma \vdash_{\mathrm{co}} \mathbf{sym}\; \mathit{axFInt} \;:\; \mathit{Bool} \sim F\; \mathit{Int}}{\Gamma \vdash_{\mathrm{tm}} \mathit{True} \rhd \mathbf{sym}\; \mathit{axFInt} \;:\; F\; \mathit{Int}}$$

to be of type, $F\ (F\ \mathit{Char})$, after the application of S_ProjKPush. To do this, we create the coercion:

$$\frac{\Gamma \vdash_{\mathrm{co}} \langle F\rangle \;:\; F \sim F \qquad \Gamma \vdash_{\mathrm{co}} \mathbf{nth}^1(\langle DT\rangle\ (\mathbf{sym}\ \mathit{axFChar})) \;:\; \mathit{Int} \sim F\ (\mathit{Char})}{\Gamma \vdash_{\mathrm{co}} \langle F\rangle\ \mathbf{nth}^1(\langle DT\rangle\ (\mathbf{sym}\ \mathit{axFChar})) \;:\; F\ \mathit{Int} \sim F\ (F\ \mathit{Char})}$$

The operation that performs all transformations is called the *lifting* operation, $\Psi(\cdot)$. The lifting operation uses the *lifting context* Ψ to know which rewrites it should perform and how new coercions should be created. Elaboration of the lifting context, the lifting operation, and the functions: extend, and context, that are needed to build the lifting context, fall outside the scope of this thesis. We refer the reader to section 5 of [69] for a complete explanation of these concepts.

A complete overview of the typing rules and operational semantics can be found in appendix C, accompanied with the proofs for type preservation and progress. We now move on to discussing the remainder of the compiler pipeline.

Representable types

τ_r	::=	$T_r\ \overline{\tau_r}$	Representable data types
	$\mid$	$(F\ \overline{\tau})_r$	Representable type function result

Representable expressions

t	::=	$\lambda\overline{(x:\tau_r)}.\mathbf{let}\ \overline{y:\tau_r = r}^{+}\ \mathbf{in}\ y_j$	Top-level function
r	::=	x	Local variable reference
	$\mid$	$f\ \overline{x}$	Saturated top-level function
	$\mid$	$K\ \overline{\tau_r}\ \varnothing\ \overline{x}$	Saturated data constructor
	$\mid$	$\otimes\ \overline{x}$	Saturated primitive
	$\mid$	$\mathbf{case}\ x\ \mathbf{of}\ \overline{p \rightarrow y}$	Case decomposition
	$\mid$	$\pi_i^k\ x$	Projection

Patterns

p	::=	$\_$	Default case
	$\mid$	$K\ \varnothing\ \_$	Matches data constructor

Figure 4.12 – System FC in Normal Form

4.2.2 Normal form

As pointed out in chapter 3, we view Haskell descriptions, *operationally*, as a structural composition of (sub-)circuits. Partly as a bi-implication[2] to this structural view, the CλaSH compiler uses a member of the $\mathcal{T}$ family of synthesis schemes, which we will call $\mathcal{T}_{C\lambda}$. An important aspect of $\mathcal{T}$ is that the arguments and results of functions become the input and output ports of components. These ports are annotated with a bit-width so that it is known how many wires are needed to make connections between ports. Because System FC is a polymorphic, higher-order language, the arguments and results of functions can contain polymorphic or function-typed values. *It is generally impossible or impractical to represent such values by a fixed number of bits.* In order to run $\mathcal{T}_{C\lambda}$, all values that cannot be represented by a fixed bit-width, will have to be eliminated from the functional description.

The second stage of CλaSH's compilation chain, *normalisation*, transforms a System FC description into a normal form which is trivially convertible to a netlist. One of the most important aspects of this normal from is that it is completely monomorphic, and first-order. The grammar of the normal form is shown in figure 4.12, where all top-level functions are of the from described by the non-terminal t.

When looking at top-level functions accepted by the non-terminal t we can see how deriving a netlist from our normal-form is trivial:

[2] A structural view implies using a variant on $\mathcal{T}$ (sans the syntax-directed aspect) for synthesis, using $\mathcal{T}$ as a synthesis scheme implies having a structural view.

» The initial term-abstractions define the input ports of the component.

» The variable reference in the body of the (recursive) let-expression is the output port of the component.

» The let-binders, of which there should be at least one, describe the composition of the subcomponents, where:

  – A variable reference refers to either, one of the input ports of the component, or the output of one of the other sub-expressions.
  – A saturated top-level function leads to the instantiation of the corresponding component, where the input ports are mapped to the wires corresponding to the local variables, and the output port is mapped to the let-bound variable.
  – The other constructs, constructor application, primitive application, case-decomposition, and projection, will have hard-coded translations to netlist primitives.

The normal form comes with additional side-conditions, some syntactic, some semantic. As indicated by the overline with a $^+$ superscript, the top-level let-expression should contain at least one let-binding. The variable reference in the body of the let-expressions refers to one of the let-bound variables. The expressions are also completely in administrative normal form *(ANF)*, which for case-decompositions means that both the subject, and the expressions in the alternatives, are references to either lamba- or let-bound variables. At the moment, data types with *existential* arguments are not considered representable. There are hence no data constructors with existential arguments in our normal form, nor patterns which bind existential variables. This is indicated in the grammar with the $\varnothing$ mark. The alternatives of case-decompositions do explicitly *not* refer to pattern-bound variables. This aspect of the normal form is highlighted by marking pattern variables as *wild*, using _, in the grammar for patterns. *Wild* means that a binder is not referenced in an expression.

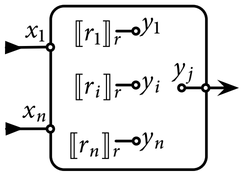
Also note that all bound variables must have *representable* type τ_t, which is either a representable data type $T\ \overline{\tau}$, or a type function which, transitively, has a representable data type $T\ \overline{\tau}$ as a result. By representable we mean that it should be trivial to encode the values of the type with a fixed number of bits, and for those reasons excludes: polymorphic types, function types, and most recursive data types.

4.2.3 From normalised System FC to a netlist

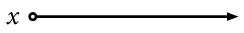
Figure 4.13 depicts the final phase of our $\mathcal{T}_{C\lambda}$ synthesis scheme, the conversion from System FC in its normal form to a netlist. As we did for $\mathcal{T}_{\mathcal{L}}$, we use graphical, schematic, representations for the transformations from System FC to a netlist. There are two transformations: $[\![\ ]\!]_t$ and $[\![\ ]\!]_r$, each referring to one of the non-terminals for expressions in our grammar of the System FC normal form (figure 4.12).

a) The transformation for top-level functions, $[\![\ ]\!]_t$, creates a new component, that:

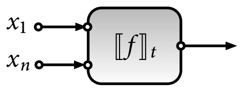
a) $[\![\lambda\overline{(x:\tau_r)}.\textbf{let}\ \overline{y:\tau_r = r}^{+}\ \textbf{in}\ y_j]\!]_t \quad \Rightarrow$



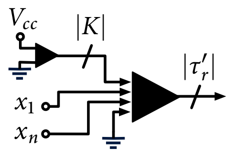
b) $[\![x]\!]_r \quad \Rightarrow$



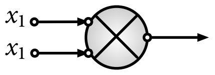
c) $[\![f\ \overline{x}]\!]_r \quad \Rightarrow$



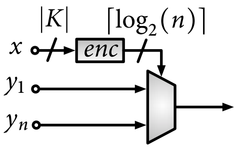
d) $[\![K\ \overline{\tau_r}\ \varnothing\ \overline{x}]\!]_r \quad \Rightarrow$



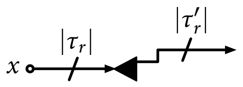
e) $[\![\otimes\ \overline{x}]\!]_r \quad \Rightarrow$



f) $[\![\textbf{case}\ x\ \textbf{of}\ \overline{p\ \rightarrow\ y}]\!]_r \quad \Rightarrow$



g) $[\![\pi_i^k\ x]\!]_r \quad \Rightarrow$



Legend:

▶	=	Combine wire bundles
◀	=	Split wire bundles
$\lvert K\rvert$	=	Bit-width of the constructor K
$\lvert\tau\rvert$	=	Bit-width for values of type τ

Figure 4.13 – From System FC in normal form to a netlist

   » Has an input port for every lambda-bound variable.
   » Instantiates all let-bound expressions side-by-side, using the $[\![\ ]\!]_r$ transformation for every expressions, and connects their results to *anchor points* corresponding to the let-bound variable. Variable references can now be synthesized to either connections to these anchor points, or to connections to the input ports.
   » Has an output port which is connected to one of the anchor points corresponding to the variable reference y_j.

b) As already hinted to earlier, variable references are transformed to connections to either anchor points or input ports by the $[\![x]\!]_r$ transformation.

c) Saturated function applications are turned into component instantiations by $[\![f\ \overline{x}]\!]_r$, where the component definition is the result of the $[\![\ ]\!]_t$ transformation applied to the expression referenced by f.

d) Saturated constructors are transformed by $[\![K\ \overline{\tau_r}\ \varnothing\ \overline{x}]\!]_r$ into:

   » A bundle of wires, of width $|K|$, encoding the constructor K. For sum types, $|K|$ is equal to $\lceil \log_2(N) \rceil$, where N is the number of constructors of the sum-type. For product types, the width of the bundle would be zero, and the constructor would hence not be encoded.
   » Every argument, in the form of a variable references, is transformed into a wire connecting to either an anchor point or input port.
   » Given that a let-binding, $y : \tau'_r = K\ \overline{\tau_r}\ \varnothing\ \overline{x}$, has type τ'_r, $|\tau'_r|$ denotes the number of bits required to encode values of type τ'. So when the wires for the constructor and for the individual fields are finally bundled together, extra wires (connected to logic 0) are added for padding to form a bundle of width $|\tau'_r|$.

e) Primitives have their custom transformation logic.

f) The $[\![\textbf{case}\ x\ \textbf{of}\ \overline{p \rightarrow y}]\!]_r$ transformation turns case-decompositions into multiplexers. In general, the variable references $\overline{y}$ become the inputs in syntactic order of the alternatives, which are subsequently connected to the referenced anchor points or input ports. The variable reference in the subject becomes the select line of the multiplexer, and is also connected to the referenced anchor point or input port. When the case-decomposition is *complete*[3] in the constructor patterns, the alternatives are reordered to match the constructor declarations of the data type. That is, given the data type definition:

```
data T = A | B
```

And the following case-decomposition in normal form:

```
case x of
  B → y1
  A → y2
```

[3]Given that a data type has N constructors, a case-decomposition is considered complete when $N-1$ constructors are matched and the N^{th} alternative is the default alternative.

The alternatives are first rearranged to:

```
case x of
  A → y2
  B → y1
```

When the case-decomposition is not exhaustive in constructor patterns, the alternatives are left in their original order, and the subject is encoded to match the order of the alternatives. That is, given the data type definition:

```
data T = A | B | C | D | E
```

And the following case-decomposition in normal form:

```
case x of
  D → y1
  B → y2
  _ → y3
```

The case-decomposition will be first transformed to:

```
case encode x of
  0 → y1
  1 → y2
  _ → y3
```

Where *encode* has the behaviour specified by the following truth table[4]:

x_2	x_1	x_0	o_1	o_0
0	0	0	1	-
0	0	1	0	1
0	1	0	1	-
0	1	1	0	0
1	0	0	1	-
1	0	1	-	-
1	1	0	-	-
1	1	1	-	-

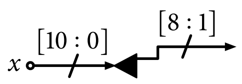
For which the corresponding logic can be derived using standard synthesis techniques [10], which in the above case would lead to: $o_1 = \neg x_0$ and $o_0 = \neg x_1$.

g) Projection is translated, using $[\![\pi_i^k\ x]\!]_r$, to splitting off the wires representing the ith field of the kth constructor from the bundle of wires connected to the anchor point or input port referenced by x. The other wires are left floating. For example, given that the variable x is of type:

[4] The '-' denotes a *don't care* value.

```
data T
  = K1 (Signed 8) Bool Bool
  | K2 Bool (Signed 8)
```

The projection, $\pi_2^2\ x$, is transformed to:



Synthesis example

Listing 4.2 demonstrates a small example where we see:

» A Haskell program.

» The corresponding System FC expression.

» The System FC expression in its normalized form.

Then at the top of listing 4.3 we see the very mechanical translation of the normalised System FC expression to a netlist, where right below that we see the same netlist with a more pleasing and more understandable layout.

The Haskell code starts with the data type declaration for *Bool*, where *False* is the first constructor, and *True* is the second. In the normalised System FC code we see that the first pattern in the case-decomposition is the *True* pattern. Then in the netlist diagram we see, in accordance to the rules of the $[\![\textbf{case}\ x\ \textbf{of}\ \overline{p \rightarrow y}]\!]_r$ transformation, that alternatives are switched with to respect the order of constructors in the data type declaration.

In the beginning of this section we discussed System FC, elaborated on the operational semantics of the extensions. We will now move on to the presentation of the normalisation phase of the CλaSH compiler, and leave the discussion why the derived netlist faithfully corresponds to the operational semantics to section 4.4.

Haskell

```
data Bool = False | True

f :: Bool → (Int8, Int8) → Int8
f x (a,b) = if x then y + 2 else y * 2
  where
    y = a * b
```

System FC

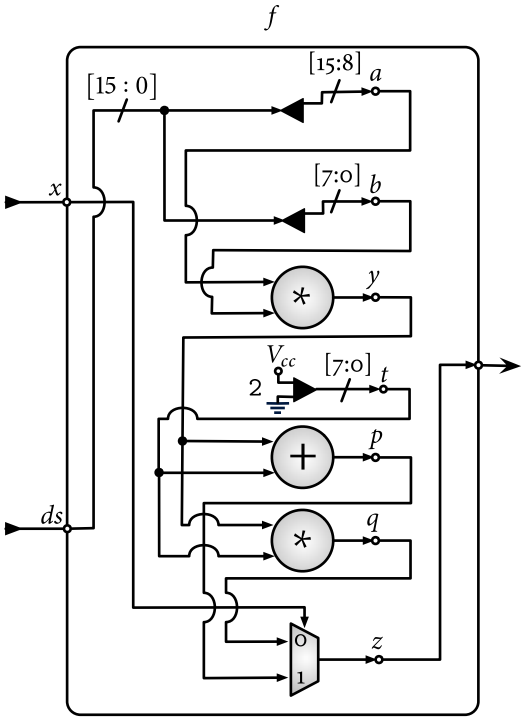
```
f = λ(x:Bool).λ(ds:Int8,Int8).case ds of
      (a:Int8,b:Int8) → let y:Int8 = a * b
                        in  case x of
                              True  → y + 2
                              False → y * 2
```

Normalized System FC

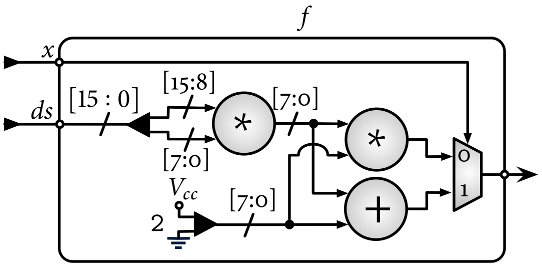
```
f = λ(x:Bool).λ(ds:(Int8, Int8)).
  let a : Int8 = π₁ds
      b : Int8 = π₂ds
      y : Int8 = a * b
      t : Int8 = 2
      p : Int8 = y + t
      q : Int8 = y * t
      z : Int8 = case x of
                   True  → p
                   False → q
  in  z
```

Listing 4.2 – Synthesis example: from Haskell to normalized System FC

Derived netlist (mechanical translation)



Derived netlist (flattened)



Listing 4.3 – Synthesis example: from normalized System FC to a netlist

4.3 Normalisation

The normalisation phase of the CλaSH compiler converts System FC expressions as they are produced by the front-end to expressions in the normal form described in section 4.2.2. The normalisation phase operates in two passes:

» The first pass eliminates, in a meaning preserving manner, all bound variables which cannot trivially be given a fixed bit-encoding.

» The second pass further simplifies the expression to facilitate the trivial translation to a netlist as described in section 4.2.3.

Given the closed global environment Γ, the set of normalized bindings Γ_n, and a binding (x, e), we apply the two passes of the normalization phase on e, to create the normalized expression e_n and the new environment Γ'. We add the binding (x, e_n) to the set of normalized bindings Γ_n to create Γ'_n. We then calculate the free variable in e_n, which are references to global binders in the environment Γ'. We then continue normalisation for each of the referenced global binders that are not in Γ'_n. We get the complete closed environment of normalized binders by taking the fixed point of the above process, starting with the closed set of global binders created by the front-end, an empty set of normalized binders, and the binding corresponding to the entry point of the program, *main*.

Definition 4.1 (Closed environment). *A closed environment is a set of binders where the difference between, the set of free variables of the bound expressions, and the binders in the environment, is the empty set.*

Definition 4.2 (Normalized environment). *A normalized environment is a closed environment where all bound expressions are in the normal form given by the grammar in figure 4.12.*

Both passes of the normalisation phase are implemented as a term rewrite system (TRS), although perhaps not a TRS in its most traditional sense, as we will see below. The rewrite rules in this chapter are presented using the format depicted in figure 4.14.

In all of these rewrite rules, the expression above the horizontal bar is the expression that has to be matched before performing the rewrite rule, and the expression below the horizontal bar is the result after applying the rewrite rule. Some rewrite

Name of the Rewrite Rule

$$\frac{\text{Matched Expression}}{\text{Resulting Expresson}} \qquad \begin{array}{l}\langle\text{Additional Preconditions}\rangle\\ \langle\text{Additional Definitions}\rangle\\ \langle\text{Updated Environment}\rangle\end{array}$$

Figure 4.14 – Format for Rewrite Rules

α-equivalence and capture-avoiding substitution

Two expressions are *α-equivalent* when their only difference is the renaming of bound variables. So the expression, $\lambda x.y\ x$, is α-equivalent to, $\lambda z.y\ z$.

A capture-avoiding substitution $e[u/x]$, is a substitution of the variable x by the expression u, in the expression e, in such a way that free variables in u do not get bound by binders in e. If a substitution does not avoid capture of free variables, then e.g., $(\lambda x.y)[x/y]$ turns the constant function $\lambda x.y$ into the identify function $\lambda x.x$. A capture-avoiding substitution could for example rename the bound variables in e that occur free in u. A capture-avoiding substitution would thus turn, $(\lambda x.y)[x/y]$, into, $\lambda z.y$, which is an α-equivalent constant function.

rules have additional preconditions, and the rewrite is only applied when these preconditions hold. Other rewrite rules have additional definitions which they use in the resulting expressions. All rewrite rules always have access to the global environment, Γ, which holds all top-level binders. There are some rewrite rules that create new top-level binders, and therefore update the global environment.

The rewrite rules have access to the following functions:

fv e	Calculates the free variables; works for types and terms.
$e[u/x]$	A capture-avoiding substitution of a variable reference x, by the expression, type, or coercion u, in the expression e.
Γ@f	The expression belonging to a global binder f in the environment Γ.
NONREP τ	Determine if the type τ is a non-representable type.
TYPEOF e	Determine the type of the expression e

Before the TRS starts, all variables are made unique, and all variable references are updated accordingly. Any new variables introduced by the rewrite rules will be unique by construction. Having hygienic expressions prevents accidental free-variable capture, and makes it easier to define meaning-preserving rewrite rules.

Definition 4.3 (Global function)**.** *Given a environment* Γ*, a global function is a variable reference to one of the binders in* Γ*.*

Definition 4.4 (Function hierarchy)**.** *Given a closed environment* Γ*, a directed graph G with variables as nodes, and a binding* (x,e)*. Let* ys *be the free variables in* e*, and* ch = [(x,y) | y ← ys] *be new edges in the graph. Let* ys' *be the subset of* ys *that are* not *nodes in G, and* es *the bindings in* Γ *corresponding to* ys'*. Let G' be the graph G, extended with nodes* ys'*, edges* ch*. Let G^+ be the graph G' extended by the nodes and edges by applying the same process on* Γ*, G', and bindings* es*. The* function hierarchy *is the graph G^* created by taking the fixed point of the above process starting with an empty graph and the binding corresponding to the entry point of the program,* main.

Aside from needing the obvious property that the normalisation phase should preserve the meaning of expressions, there are two additional properties that we wish our normalisation phase to respect as much as possible:

» It should keep the original function hierarchy intact where possible, because the netlist hierarchy will follow the function hierarchy under $\mathcal{T}_{C\lambda}$. Leaving the function hierarchy intact will make it easier to relate the non-functional properties of the netlist, such as area and propagation delay, back to the original function. If desired, it is always possible to flatten parts of the hierarchy later on, going in the other direction is non-trivial.

» Sharing should be preserved where possible. As seen in listing 4.2, synthesized circuits can also share the result of a computation by connecting the output port of one component to input ports of multiple other components. Any loss in sharing, will most likely lead to a duplication of component instantiations, and under $\mathcal{T}_{C\lambda}$ ultimately lead to a larger circuit.

4.3.1 Eliminating non-representable values

$\mathcal{T}_{C\lambda}$ can only synthesize functional descriptions if function arguments, let-bindings, and pattern-bound variables can be given a fixed bit-encoding. There are straightforward encodings for certain primitive data types, and certain algebraic data types, as we have seen in chapter 3. Data types with a fixed bit-encoding are called *representable*. Deriving a fixed bit-encoding for the following types is either not desired, or not possible:

» Function types.

» (Higher-rank) polymorphic types.

» Data types with *existential* arguments, and consequently, all GADTs.

» Recursively defined data types.

» Data types that are composed of types that are not representable.

This section shows that the TRS responsible for the first pass of the normalisation pass, the removal of non-representable values. It eliminates such values completely, given that the input adheres to the following restrictions:

» The *main* function is monomorphic.

» The arguments and the result of the *main* function are representable.

» The arguments and the result of primitives are representable.

The TRS uses a combination of inlining and specialisation, where specialisation takes on two forms:

» Specialisation of a function on one of its arguments.

» Elimination of a case-decomposition based on a known constructor.

Simply inlining all functions with non-representable arguments and results, and inlining all non-representable let-bindings, is either undesired or not possible. Inlining destroys the original function hierarchy, which we are trying to preserve, and might also lead to loss in sharing.

Defunctionalisation

Reynolds-style defunctionalisation [57] is a well-known method for generating an equivalent first-order program from a higher-order program. Reynolds' method creates data types for arguments with a function-type. Instead of applying a higher-order function to a value with a function-type, it is applied to a constructor for the newly introduced data type. Application of the functional argument is replaced by the application of a mini-interpreter. Given the following higher-order program:

```
uncurry f (a,b) = f a b
main x = (uncurry (+) x) + (uncurry () x)
```

Reynolds' method creates the following behaviourally equivalent first-order program:

```
data Function = Plus | Sub
apply Plus a b  = (+) a b
apply Sub  a b  = ()  a b

uncurry f (a,b) = apply f a b
main x = (uncurry Plus x) + (uncurry Min x)
```

Reynolds' method works on all programs, removes all functional arguments, and preserves sharing. Although commonly defined and studied in the setting of the simply typed lambda calculus, there are also variants [6, 56] of Reynolds' methods that are correct within a polymorphic type system. The disadvantage of Reynolds' method is the introduction of the mini-interpreter (which takes on the form of the *apply* function in the example).

The *apply* function will be transformed into netlist that contains an adder and subtracter whose results will be multiplexed to the output. In the above example, the *apply* function will be instantiated twice, naively leading to a circuit with three adders, two subtracters, and two multiplexers. Of course, we can specialize the two calls to *uncurry*, and subsequently *apply*, on their first argument to get a circuit that has only two adders, one subtracter, and no multiplexers. We will therefore not use (a variant of) Reynolds' method in the CλaSH compiler, but instead opt to specialise functions on their function-typed arguments immediately.

Many of the rewrite rules used by the TRS described in this chapter can also be found in optimizing compilers for functional languages, such as GHC [54]. The rewrite rules presented by Peyton Jones and Santos [54] do, however, not guarantee a first-order normal form, which the TRS presented in this chapter does (given certain restrictions on the input program).

Mitchell and Runciman [46] present a defunctionalisation method based on a TRS, which, like the TRS presented in this chapter, also uses specialisation and inlining. The presented TRS can thus be seen as an extension to the work of Mitchell and Runciman:

» It provides transformations that additionally perform monomorphisation, which includes the specialisation of: higher-rank polymorphic arguments and existential datatypes.

» It can deal with recursive let-expressions.

» It works on a typed language, and uses this type information to determine when transformations should be applied.

Rewrite rules

Many of the rewrite rules in this section are straightforward encodings of the operational semantics for System FC. Proving them correct, type- and semantics-preserving, is therefore in most cases trivial. We defer the correctness proofs of the transformations to appendix D.

The first three rewrite rules, TBeta, LetTApp, and CaseTApp, propagate type information downwards into an expression. By either removing type-variables, propagating type-information to a location for specialisation, or propagating type information to a primitive or constructor, these rewrite rules aid in the elimination of polymorphism.

TBeta

$$\frac{(\Lambda a : \kappa . e)\ \tau}{e[\tau/a]}$$

LetTApp

$$\frac{(\mathbf{let}\ \overline{x : \sigma = e}\ \mathbf{in}\ u)\ \tau}{\mathbf{let}\ \overline{x : \sigma = e}\ \mathbf{in}\ (u\ \tau)}$$

CaseTApp

$$\frac{(\mathbf{case}\ e\ \mathbf{of}\ \overline{p \to u})\ \sigma}{\mathbf{case}\ e\ \mathbf{of}\ \overline{p \to (u\ \sigma)}}$$

Now follow the coercion propagation rules, CBeta, CLetApp, CCaseApp, Push, TPush, CPush and KPush. Again, almost all of these transformations are direct implementations of the System FC operational semantics:

CBeta

$$\frac{(\lambda c : \phi . e)\ \gamma}{e[\gamma / c]}$$

LetCApp

$$\frac{(\mathbf{let}\ \overline{x : \sigma = e}\ \mathbf{in}\ u)\ \gamma}{(\mathbf{let}\ \overline{x : \sigma = e}\ \mathbf{in}\ (u\ \gamma)}$$

CaseCApp

$$\frac{(\mathbf{case}\ e\ \mathbf{of}\ \overline{p \to u})\ \gamma}{\mathbf{case}\ e\ \mathbf{of}\ \overline{p \to (u\ \gamma)}}$$

Push

$$\frac{(e\ \rhd \gamma)\ u}{(e\ (u\ \rhd \mathbf{sym}(\mathbf{nth}^1 \gamma)))\ \rhd \mathbf{nth}^2 \gamma}$$

TPush

$$\frac{(e\ \rhd \gamma)\ \tau}{(e\ (\tau\ \rhd \gamma'))\ \rhd \gamma @ (\langle \tau \rangle\ \rhd \gamma')}$$

Definitions: $\gamma' = \mathbf{sym}(\mathbf{nth}^1 \gamma)$

CPush

$$\frac{(e\ \rhd \gamma_1)\ \gamma_2}{(e\ \gamma_3)\ \rhd \gamma_1 @ (\gamma_3, \gamma_2)}$$

Definitions: $\gamma_3 = \mathbf{nth}^1 \gamma_1 \mathbin{;} \gamma_2 \mathbin{;} \mathbf{sym}(\mathbf{nth}^2 \gamma_1)$

KPush

Definitions:
$K : \forall \overline{a : \kappa}. \forall \Delta . \overline{\sigma} \to T \overline{a}$
$\Psi = \text{extend}(\text{context}(\gamma); \overline{\rho}; \Delta)$
$\overline{\tau'} = \Psi_2(\overline{a})$
$\overline{\rho'} = \Psi_2(dom\ \Delta)$
$\overline{e_i' = e_i\ \rhd \Psi(\sigma_i)}^i$

$$\frac{\mathbf{case}\ (K\ \overline{\tau}\ \overline{\rho}\ \overline{e})\ \rhd \gamma\ \mathbf{of}\ \overline{p \to u}}{\mathbf{case}\ K\ \overline{\tau'}\ \overline{\rho'}\ \overline{e'}\ \mathbf{of}\ \overline{p \to u}}$$

The next three rewrite rules, LamApp, LetApp, and CaseApp, propagate values, including non-representable ones, downwards into the expression. LamApp is preferred over β-reduction to preserve sharing. CaseApp creates a let-binding, instead of propagating the applied expression towards all alternatives, to preserve sharing.

LamApp

$$\frac{(\lambda x : \sigma . e)\ u}{\textbf{let}\ \{x : \sigma = u\}\ \textbf{in}\ e}$$

LetApp

$$\frac{(\textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ u)\ e'}{\textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ (u\ e')}$$

CaseApp

$$\frac{(\textbf{case}\ e\ \textbf{of}\ \overline{p \to u})\ e'}{\textbf{let}\ \{x : \sigma = e'\}\ \textbf{in}\ (\textbf{case}\ e\ \textbf{of}\ \overline{p \to (u\ x)})}$$

The next two rewrite rules, LetCast and CaseCast, propagate casts downward:

LetCast

$$\frac{(\textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ u)\ \triangleright \gamma}{\textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ (u\ \triangleright \gamma)}$$

CaseCast

$$\frac{(\textbf{case}\ e\ \textbf{of}\ \overline{p \to u})\ \triangleright \gamma}{\textbf{case}\ e\ \textbf{of}\ \overline{p \to (u\ \triangleright \gamma)}}$$

The next rewrite rules, BindNonRep and LiftNonRep, remove both existing let-binders, and let-binders introduced by LamApp and CaseApp, in case they bind non-representable values.

BindNonRep removes a let-binder, $x_i : \sigma_i = e_i$ (with a non-representable type σ_i), and substitutes references to x_i by the expression e_i. An extra precondition is that the binding may not be self-referencing, as removing the let-binder in such a case would lead to a new free variable. Although substitution leads to loss in sharing in the traditional sense, lifting the let-binder to the global environment, as will be done by LiftNonRep, would not result in the preservation of sharing in the case $\mathcal{T}_{C\lambda}$. The reason is that applications of the lifted function would result into multiple instantiations of the component, where the circuitry within the individual components cannot be shared. Actually, some of the sharing lost due to the substitution performed by BindNonRep might actually be recovered by a local common sub-expression elimination *(CSE)* pass.

LiftNonRep removes a let-binder, $x_i : \sigma_i = e_i$ (with a non-representable type σ_i), and substitutes references to x_i in the rest of the let-expression with an (application of a) variable reference to a *new*, *global*, binder: f. This transformation is, however, only applied when the binder is self-referencing, and cannot be removed by BindNonRep. The new global binder, f, binds the original expression e_i which is abstracted over the free local (type) variables of e_i; all references to x_i are substituted with an (application of a) variable reference to f. Again, as explained earlier, lifting the binder to the global environment will not preserve sharing, and it is one of the reasons why LiftNonRep is only applied when it is not possible to apply BindNonRep (the other reason is explained in section 4.3.3).

BindNonRep

$$\frac{\textbf{let}\ \{b_1;\ldots;b_{i-1};x_i : \sigma_i = e_i;b_{i+1};\ldots;b_n\}\ \textbf{in}\ u}{(\textbf{let}\ \{b_1;\ldots;b_{i-1};b_{i+1};\ldots;b_n\}\ \textbf{in}\ u)[e_i/x_i]} \qquad \begin{array}{l}\text{Preconditions: NONREP}(\sigma_i)\\ \bigwedge x_i \notin \overline{y}\end{array}$$

Definitions: $(\overline{a},\overline{y}) = \text{FV}(e_i)$

LiftNonRep

$$\frac{\textbf{let}\ \{b_1;\ldots;b_{i-1};x_i : \sigma_i = e_i;b_{i+1};\ldots;b_n\}\ \textbf{in}\ u}{(\textbf{let}\ \{b_1;\ldots;b_{i-1};b_{i+1};\ldots;b_n\}\ \textbf{in}\ u)[f\ \overline{z}/x_i]} \qquad \begin{array}{l}\text{Preconditions: NONREP}(\sigma_i\ )\\ \bigwedge x_i \in \overline{y}\\ \bigwedge \overline{a} = \varnothing\end{array}$$

Definitions: $(\overline{a},\overline{y : \tau'}) = \text{fv}(e_i)$; $\overline{z} = \overline{y} - \{x_i\}$

New Environment: $\Gamma \cup_\alpha \{f : \overline{\tau'} \rightarrow \sigma_i = \lambda\overline{z : \tau'}.e_i[(f\ \overline{z})/x_i]\}$

The LiftNonRep rewrite rule uses the $\cup_\alpha$ operator to indicate that the global environment is only updated with the new binder, f, if an α-equivalent expression is not already present. In case an α-equivalent expression is present in the environment, the transformed expression will refer to that existing global binder instead.

The previous rewrite rules either propagated non-representable values downwards into the expression, or lifted those values out of the expression. The next two sets of rewrite rules remove non-representable values by specialisation. The TypeSpec, CoSpec, NonRepSpec, and CastSpec, provide function argument specialisation. LetCase, CaseCase, InlineNonRep, and CaseCon, together achieve specialisation by eliminating case-decompositions of known constructors (of non-representable data types).

The TypeSpec rewrite rule matches on a type application, $f\ \tau$, of a variable reference to a global binder, f. A precondition for this specialisation is that τ cannot have any free type or coercion variables. The application is replaced by a reference to the *new* global binder f'. The new binder f' is defined in terms of the body of f specialized on the type τ. CoSpec behave like TypeSpec, but for applications of coercions to global variable references, $f\ \gamma$. NonRepSpec behaves similarly to TypeSpec for the application of a global variable on a non-representable arguments, $f\ u$. The difference is that the expression of the new binder, f', is abstracted over the free variables of the specialised argument; the transformed expression also takes these free variables into account. CastSpec is not a specialisation transformation in the traditional sense of propagating an argument expression to the function definition. Instead, it pushes the cast surrounding a reference to a global binder, f, towards the body of f and creating a new binder f' for this specialized version. The expression then references this new specialised binder f'. CastSpec is mostly present to enable firing of the other two specialisation transformations, TypeSpec and NonRepSpec.

TypeSpec, CoSpec, NonRepSpec, and CastSpec, all use the $\cup_\alpha$ operator to indicate that the global environment is only updated with a new binder if an α-equivalent specialization is not already present. In case an α-equivalent specialisation is present in the environment, the transformed expression will refer to that existing global binder instead.

TypeSpec $\quad \dfrac{(f\ \overline{e})\ \sigma}{f'\ \overline{e}}$ Preconditions: $\mathrm{fv}(\sigma) = \varnothing$ Definitions: $\overline{\tau'} = \mathrm{TYPEOF}(\overline{e})$

New Environment: $\Gamma \cup_\alpha \{f' : \overline{\tau'} \to \tau = \lambda\overline{x : \tau}.(\Gamma@f\ \overline{x})\ \sigma\}$

CoSpec $\quad \dfrac{(f\ \overline{e})\ \gamma}{f'\ \overline{e}}$ Preconditions: $\mathrm{fv}(\gamma) = \varnothing$ Definitions: $\overline{\tau'} = \mathrm{TYPEOF}(\overline{e})$

New Environment: $\Gamma \cup_\alpha \{f' : \overline{\tau'} \to \tau = \lambda\overline{x : \tau'}.(\Gamma@f\ \overline{x})\ \gamma\}$

NonRepSpec $\quad \dfrac{(f\ \overline{e})\ u}{f'\ \overline{e}\ \overline{y}}$ Preconditions: $\mathrm{NONREP}(\sigma) \wedge \overline{a} = \varnothing$ Definitions: $(\overline{a}, \overline{y : \sigma'}) = \mathrm{fv}(u)$

$\overline{\tau'} = \mathrm{TYPEOF}(\overline{e})$

$\sigma = \mathrm{TYPEOF}(u)$

New Environment: $\Gamma \cup_\alpha \{f' : \overline{\tau'} \to \overline{\sigma'} \to \tau = \lambda\overline{x : \tau'}.\lambda\overline{y : \sigma'}.(\Gamma@f\ \overline{x})\ u\}$

CastSpec	$\dfrac{(f\ \overline{e}) \rhd \gamma}{f'\ \overline{e}}$	Preconditions:	$\text{fv}(\gamma) = \varnothing$
		Definitions:	$\overline{\tau'} = \text{TYPEOF}(\overline{e})$

New Environment: $\Gamma \cup_\alpha \{f' : \overline{\tau'} \to \tau = \lambda\overline{x : \tau'}.(\Gamma@f\ \overline{x})\ \rhd \gamma\}$

The LetCase is required in specialising expressions that have a non-representable datatype. Taking the let-binders out of the case-decomposition does not affect the sharing behaviour so can be applied blindly. There is no free variable capture in the alternatives because all variables are made unique before running the TRS.

The CaseCase rewrite rule is only applied if the subject of a case-decomposition has a non-representable datatype. CaseCase is not applied blindly because the alternatives in a case-decomposition are evaluated in parallel in the eventual circuit. So the CaseCase rewrite rule generates a larger number of alternatives than present in the matched expression. A larger number of alternatives results in a larger circuit. Even though CaseCase makes the circuit larger, the intention of CaseCase is to eventually expose the constructor of the non-representable datatype to CaseCon. CaseCon eliminates the case-decomposition, and subsequently amortizes the increase in circuit size induced by CaseCase.

InlineNonRep is only applied if the subject of a case expression is of a non-representable datatype, as inlining breaks down the component hierarchy. All bound variables in the inlined expression are regenerated, and variable references updated accordingly. This preserves the assumptions made by the other rewrite rules that all variables are unique.

The CaseCon rule comes in three variants:

» A case-decomposition with a constructor application as the subject, and a matching constructor pattern.

» A case-decomposition with a constructor application as the subject, with *no* matching constructor pattern.

» A case-decomposition with one alternative, where the expression in the alternative does not reference any of the variables in the pattern.

CaseCon only creates a let-binding if the constructor in the subject exactly matches the constructor of an alternative. When the default pattern is matched, the case-decomposition is simply replaced by the expression belonging to the default alternative. The same happens when there is only one alternative and the expression in the alternative does not reference any pattern variables. Case-decompositions in System FC are exhaustive, either by enumerating all the constructors, or by including the default pattern. This means that when a constructor applications is the subject of a case-decomposition, CaseCon will always remove that case-decomposition.

LetCase

$$\frac{\textbf{case}\ (\textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ u)\ \textbf{of}\ \overline{p \to u'}}{\textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ (\textbf{case}\ u\ \textbf{of}\ \overline{p \to u'})}$$

CaseCase Preconditions: NONREP(TYPEOF(u_1))

$$\frac{\textbf{case}\ (\textbf{case}\ e\ \textbf{of}\ \{p_1 \to u_1;\ \dots;\ p_n \to u_n\})\ \textbf{of}\ \overline{p' \to u'}}{\textbf{case}\ e\ \textbf{of}\ p_1 \to \textbf{case}\ u_1\ \textbf{of}\ \overline{p' \to u'};\ \dots;\ p_n \to \textbf{case}\ u_n\ \textbf{of}\ \overline{p' \to u'}\}}$$

InlineNonRep Preconditions: NONREP(TYPEOF($f\ \overline{e}$))

$$\frac{\textbf{case}\ f\ \overline{e}\ \textbf{of}\ \overline{p \to u}}{\textbf{case}\ (\Gamma @ f\ \overline{e})\textbf{of}\ \overline{p \to u}}$$

CaseCon

$$\frac{\textbf{case}\ K_i\ \overline{\tau}\ \overline{\rho}\ \overline{e}\ \textbf{of}\ \{\dots; K_i\ \Delta\ \overline{x : \sigma} \to u_i; \dots\}}{\textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ (u_i\ [\overline{\rho}/\Delta])}$$

$$\frac{\textbf{case}\ K_i\ \overline{\tau}\ \overline{\rho}\ \overline{e}\ \textbf{of}\ \{\overline{p_{j \neq i} \to u}; \_ \to u_0\}}{u_0}$$

$$\frac{\textbf{case}\ \overline{e}\ \textbf{of}\ \{p_0 \to u_0\}}{u_0}$$ Preconditions: $\mathrm{fv}(u_0) - \mathrm{fv}(p_0 \to u_0) = \varnothing$

4.3.2 Completeness of non-representable value removal

We start with the following three definitions:

Definition 4.5 (Representable function type). *A representable function type, is a function type where the left hand side* (LHS) *of the arrow (→) is a representable type, and the right hand side* (RHS) *of the arrow is either: a representable type, or a representable function type.*

Definition 4.6 (Representable binding). *A binding* (x,e) *is a representable binding, when:*

» *All variables bound in e have a representable type.*

» *All arguments in an application in e have a representable type.*

» *The expressions e has a representable type, or a representable function type (definition 4.5).*

Definition 4.7 (Representable environment). *A representable environment is a closed environment where every binding is a representable binding.*

The first set of rewrite rules (TBeta - LiftNonRep) propagate or remove non-representable values for those syntactical elements on which the specialisation rewrite rules do not match. The second set of rewrite rules (TypeSpec - CaseCon) remove the non-representable values through specialisation. The two sets of rewrite rules ensure that the resulting environment of the normalisation phase is a *representable* environment (definition 4.7), given the restrictions in Section 4.3.1.

The restrictions on primitives are needed because those cannot be specialized on their argument, nor can their definitions be inlined. The restriction that the result type of *main* cannot be a non-representable data type, ensures that any expression calculating a non-representable data type always becomes the subject of a case-decomposition, which will be removed by the TRS.

This section will prove the following two theorems, which together ensure that, when the normalisation finishes, the resulting environment is a *representable* environment.

Theorem: 4.3.1 (Representable bindings). *When the* main *function, and all used primitives have representable (function) types then, given a binding* (x,e), (x,e) *is either a representable binding, or one of the rewrite rules applies to* e.

Theorem: 4.3.2 (Representable environment). *Given a closed environment* Γ, *an environment of representable bindings* Γ_r, *and a binding* (x,e). *Let* (e',Γ') *be the resulting tuple of exhaustively applying all the rewrite rules to* e *and* Γ, *and let* $\Gamma_r' = \Gamma_r \cup \{(x, e')\}$. *Then* (x,e') *is a representable binding; and* Γ_r' *is a representable environment or normalisation continues with* Γ', Γ_r', *and the binders corresponding to the free variables in* Γ_r'.

Notation

In order to give structure to the proofs of the above theorems we introduce a notation to represent all possible expressions. We will model *expressions* (not types or coercions) in System FC as a *set* of syntax trees. We do this, so that after every lemma, or in intermediate parts of a proof, we can show how that the set of all possible expressions *after* applying the rewrite rules exhaustively become smaller. We will ultimately end up with a set of possible expressions where our will theorems hold, thus having finished our proof.

We start with the definitions of a few basic sets. There is the set $\mathcal{V}$ which represents all *local* variables, those *not* bound in the environment, and the set $\mathcal{F}$ which represents all *global* variables, those bound in the environment. Next we have $\mathcal{P}$, the set of primitives, $\mathcal{K}$, the set of data constructors, $\mathcal{T}$, the set of types, and $\mathcal{C}$, the set of coercions. For the purpose behind our notation, to give structure to the proofs, it is not necessary to specify how these sets are generated.

Next we define our set generators, which can generate certain sets of expressions. Notice that most are parametrized in the sets from which they can draw elements.

$$
\begin{array}{ll}
\textit{lam}\ x & = \{\lambda y : \tau.e \mid y \in \mathcal{V},\ \tau \in \mathcal{T},\ e \in x\} \\
\textit{app}\ x\ y & = \{e\ u \mid e \in x,\ u \in y\} \\
\textit{tyLam}\ x & = \{\Lambda a : \kappa.e \mid a \in \mathcal{V},\ \kappa \in \mathcal{T},\ e \in x\} \\
\textit{tyApp}\ x & = \{e\ \tau \mid e \in x,\ \tau \in \mathcal{T}\} \\
\textit{coLam}\ x & = \{\lambda c : \phi.e \mid c \in \mathcal{V}, \phi \in \mathcal{C},\ e \in x\} \\
\textit{coApp}\ x & = \{e\ \gamma \mid e \in x, \gamma \in \mathcal{C}\} \\
\textit{prim} & = \mathcal{P} \\
\textit{con} & = \mathcal{K} \\
\textit{var} & = \mathcal{V} \\
\textit{fun} & = \mathcal{F} \\
\textit{let}\ x\ y & = \{\textbf{let}\ \overline{v : \tau = e}\ \textbf{in}\ u \mid \overline{v} \subseteq \mathcal{V},\ \overline{\tau} \subseteq \mathcal{T},\ \overline{e} \subseteq x,\ u \in y\} \\
\textit{case}\ x\ y & = \{\textbf{case}\ e\ \textbf{of}\ \overline{p \to u}^{+} \mid e \in x,\ \overline{p} \subseteq \{\_\} \cup \textit{pat},\ \overline{u} \subseteq \overline{y}\} \\
\textit{pat} & = \{K\ \Delta\ \overline{x : \sigma} \mid K \in \mathcal{K},\ \Delta \subseteq \textit{tele},\ \overline{x} \subseteq \mathcal{V},\ \overline{\tau} \subseteq \mathcal{T}\} \\
\textit{tele} & = \{a : \kappa \mid a \in \mathcal{V},\ \kappa \in \mathcal{T}\} \cup \{c : \phi \mid c \in \mathcal{V},\ \phi \in \mathcal{C}\} \\
\textit{cast}\ x & = \{e \rhd \gamma \mid e \in x,\ \gamma \in \mathcal{C}\}
\end{array}
$$

We define three derived set generators which we use to model nested application:

$$
\begin{array}{l}
\textit{app}^{*}\ x\ y = \{e\ \overline{u} \mid e \in (x - \textit{app}\ x\ y),\ \overline{u} \subseteq y\} \\
\textit{tyApp}^{*} x = \{e\ \overline{\tau} \mid e \in (x - \textit{tyApp}\ x\ y),\ \overline{tau} \subseteq \mathcal{T}\} \\
\textit{coApp}^{*} x = \{e\ \overline{\gamma} \mid e \in (x - \textit{coApp}\ x\ y),\ \overline{tau} \subseteq \mathcal{C}\}
\end{array}
$$

where the LHS of the nested application is not itself that specific kind of application. So $\textit{app}^{*}\ x\ y$, generates all nested *term* applications from expressions drawn from the sets x and y, but not the expressions in x that are themselves *term* applications; but it will, for example, generate a nested *term* application of a *type* application.

Note that we have not included a set generator for projections. The reason is that projections are only introduced by the second pass of the normalisation phase. Projections do not occur in the expressions generated by the front-end, nor do our rewrite rules for non-representable value elimination introduce projections.

The set of expressions we can possibly create is the fixed point of the equation: $\mathcal{S}_0 = \textit{lam}\ \mathcal{S}_0 \cup \textit{app}^{*}\ \mathcal{S}_0\ \mathcal{S}_0 \cup \textit{tyLam}\ \mathcal{S}_0 \cup \textit{tyApp}^{*}\ \mathcal{S}_0 \cup \textit{coLam}\ \mathcal{S}_0 \cup \textit{coApp}^{*}\ \mathcal{S}_0 \cup \textit{prim} \cup \textit{con} \cup \textit{var} \cup \textit{fun} \cup \textit{let}\ \mathcal{S}_0\ \mathcal{S}_0 \cup \textit{case}\ \mathcal{S}_0\ \mathcal{S}_0 \cup \textit{cast}\ \mathcal{S}_0$

For every lemma n and its corresponding proof, we will define a new set of expression, $\mathcal{S}_n$, that are possible under the restrictions of the lemma. Every subsequent set of possible expressions is a *proper subset* of the previous set, that is, $\mathcal{S}_n \supset \mathcal{S}_{n+1}$.

Proof of the theorems

For our proofs of theorem 4.3.1 and theorem 4.3.2, we will first need to define several lemmas.

Lemma: 4.3.1 (Correctly typed). *Expressions are correctly typed. Therefore:*

1. *The subject of a case-decomposition is not an abstraction.*
2. *Term-abstractions are never the subject of a type- or coercion-application.*
3. *Type-abstractions are never the subject of a term- or coercion-application.*
4. *Coercion-abstractions are never the subject of a term- or type-application.*
5. *Constructors are first applied to their type and coercion arguments, before they are applied to their term arguments.*
6. *By the well-formedness check (appendix C), primitives are monomorphic.*

Proof. The original expression is correctly typed and all of our transformations preserve correct typing.

We define our new set of possible expressions under the restrictions set by the lemma as:

$\mathcal{S}_1 = \mathit{lam}\ \mathcal{S}_1 \cup \mathit{app}^*\ (\mathcal{S}_1 - \mathit{tyLam}\ \mathcal{S}_1 - \mathit{coLam}\ \mathcal{S}_1)\ \mathcal{S}_1 \cup \mathit{tyLam}\ \mathcal{S}_1 \cup \mathit{tyApp}^*\ \mathcal{A}_1 \cup \mathit{coLam}\ \mathcal{S}_1 \cup \mathit{coApp}^*\ \mathcal{B}_1 \cup \mathit{prim} \cup \mathit{con} \cup \mathit{var} \cup \mathit{fun} \cup \mathit{let}\ \mathcal{S}_1\ \mathcal{S}_1 \cup \mathit{case}\ (\mathcal{S}_1 - \mathit{lam}\ \mathcal{S}_1 - \mathit{tyLam}\ \mathcal{S}_1 - \mathit{coLam}\ \mathcal{S}_1)\ \mathcal{S}_1 \cup \mathit{cast}\ \mathcal{S}_1$

$\mathcal{A}_1 = \mathit{app}^*\ (\mathcal{S}_1 - \mathit{prim} - \mathit{con} - \mathit{tyLam}\ \mathcal{S}_1 - \mathit{coLam}\ \mathcal{S}_1)\ \mathcal{S}_1 \cup \mathit{tyLam}\ \mathcal{S}_1 \cup \mathit{coApp}^*\ \mathcal{B}_1 \cup \mathit{con} \cup \mathit{var} \cup \mathit{fun} \cup \mathit{let}\ \mathcal{S}_1\ \mathcal{S}_1 \cup \mathit{case}\ (\mathcal{S}_1 - \mathit{lam}\ \mathcal{S}_1 - \mathit{tyLam}\ \mathcal{S}_1 - \mathit{coLam}\ \mathcal{S}_1)\ \mathcal{S}_1 \cup \mathit{cast}\ \mathcal{S}_1$

$\mathcal{B}_1 = \mathit{app}^*\ (\mathcal{S}_1 - \mathit{prim} - \mathit{con} - \mathit{tyLam}\ \mathcal{S}_1 - \mathit{coLam}\ \mathcal{S}_1)\ \mathcal{S}_1 \cup \mathit{tyApp}^*\ \mathcal{A}_1 \cup \mathit{coLam}\ \mathcal{S}_1 \cup \mathit{con} \cup \mathit{var} \cup \mathit{fun} \cup \mathit{let}\ \mathcal{S}_1\ \mathcal{S}_1 \cup \mathit{case}\ (\mathcal{S}_1 - \mathit{lam}\ \mathcal{S}_1 - \mathit{tyLam}\ \mathcal{S}_1 - \mathit{coLam}\ \mathcal{S}_1)\ \mathcal{S}_1 \cup \mathit{cast}\ \mathcal{S}_1$

For readability, we have defined two new subsets, $\mathcal{A}_1$ and $\mathcal{B}_1$, to represent the possible expressions that can form the LHS of a type application ($\mathcal{A}_1$) or the LHS of a coercion application ($\mathcal{B}_1$). □

Lemma: 4.3.2 (Only constructor, variable, function, and primitive applications). *There are only type- and coercion-applications of constructors and of (applications of) local variables. There are only term-applications of: global functions, primitives, constructors, local variables, and of the earlier mentioned type- and coercion-applications.*

Proof. By enumerating all the other possible locations for type-, coercion-, and term-applications in $\mathcal{S}_1$, and showing that they are removed by one of the transformations.

- » An application of a term-abstractions is removed by LamApp.
- » An application of a let-expressions is removed by the LetApp.
- » An application of a case-decomposition is removed by CaseApp.
- » An application of a cast is removed by Push
- » Types applied to type-abstractions are propagated by TBeta.
- » Types applied to global functions are propagated by TypeSpec.
- » Types applied to let-expressions are propagated by LetTApp.
- » Types applied to case-decompositions are propagated by CaseTApp.
- » Types applied to casts are removed by TPush.
- » Coercions applied to coercion-abstractions are propagated by CBeta.
- » Coercions applied to global functions are propagated by CoSpec.
- » Coercions applied to let-expressions are propagated by LetCApp.
- » Coercions applied to case-decompositions are propagated by CaseCApp.
- » Coercions applied to casts are removed by CPush.

Thus, our new set of possible expressions possible under the lemma, is defined by:

$\mathcal{S}_2 = lam\, \mathcal{S}_2 \,\cup\, app^*\, (prim \cup con \cup var \cup fun \cup tyApp^*\, \mathcal{A}_2 \cup coApp^*\, \mathcal{B}_2)\, \mathcal{S}_2 \,\cup\, tyLam\, \mathcal{S}_2 \,\cup\, tyApp^*\, \mathcal{A}_2 \,\cup\, coLam\, \mathcal{S}_2 \,\cup\, coApp^*\, \mathcal{B}_2 \,\cup\, prim \,\cup\, con \,\cup\, var \,\cup\, fun \,\cup\, let\, \mathcal{S}_2\, \mathcal{S}_2 \,\cup\, case\, (\mathcal{S}_2 - lam\, \mathcal{S}_2 - tyLam\, \mathcal{S}_2 - coLam\, \mathcal{S}_2)\, \mathcal{S}_2 \,\cup\, cast\, \mathcal{S}_2$

$\mathcal{A}_2 = app^*\, var\, \mathcal{S}_2 \,\cup\, coApp^*\, \mathcal{B}_2 \,\cup\, con \,\cup\, var$

$\mathcal{B}_2 = app^*\, var\, \mathcal{S}_2 \,\cup\, tyApp^*\, \mathcal{A}_2 \,\cup\, con \,\cup\, var$

Note that we still define subsets for the LHSs of type and coercion applications. □

Definition 4.8 (Representable set). *We define $\mathcal{R}_n$ as the set of expressions drawn from $\mathcal{S}_n$ whose type is representable.*

Lemma: 4.3.3 (Representable arguments). *The arguments of an application of a primitive or of a global function are representable.*

Proof. The precondition states that all arguments to primitives must have a representable type. Non-representable arguments applied to global functions are propagated by NonRepSpec.

Our new set of possible expressions is:

$\mathcal{S}_3 = \mathit{lam}\ \mathcal{S}_3 \cup \mathit{app}^*\ (\mathit{con} \cup \mathit{var} \cup \mathit{tyApp}^*\ \mathcal{A}_2 \cup \mathit{coApp}^*\ \mathcal{B}_2)\ \mathcal{S}_3 \cup \mathit{app}^*\ (\mathit{prim} \cup \mathit{fun})\ \mathcal{R}_3 \cup \mathit{tyLam}\ \mathcal{S}_3 \cup \mathit{tyApp}^*\ \mathcal{A}_2 \cup \mathit{coLam}\ \mathcal{S}_3 \cup \mathit{coApp}^*\ \mathcal{B}_2 \cup \mathit{prim} \cup \mathit{con} \cup \mathit{var} \cup \mathit{fun} \cup \mathit{let}\ \mathcal{S}_3\ \mathcal{S}_3 \cup \mathit{case}\ (\mathcal{S}_3 - \mathit{lam}\ \mathcal{S}_3 - \mathit{tyLam}\ \mathcal{S}_3 - \mathit{coLam}\ \mathcal{S}_3)\ \mathcal{S}_3 \cup \mathit{cast}\ \mathcal{S}_3$

$\mathcal{A}_3 = \mathit{app}^*\ \mathit{var}\ \mathcal{S}_3 \cup \mathit{coApp}^*\ \mathcal{B}_3 \cup \mathit{con} \cup \mathit{var}$

$\mathcal{B}_3 = \mathit{app}^*\ \mathit{var}\ \mathcal{S}_3 \cup \mathit{tyApp}^*\ \mathcal{A}_3 \cup \mathit{con} \cup \mathit{var}$

$\square$

Lemma: 4.3.4 (Representable let-bindings). *All let-bindings are representable.*

Proof. Let-bindings with a type that is not representable are either inlined by BindNonRep, or turned into a global function by LiftNonRep.

Our new set of expressions still possible under the lemma is:

$\mathcal{S}_4 = \mathit{lam}\ \mathcal{S}_4 \cup \mathit{app}^*\ (\mathit{con} \cup \mathit{var} \cup \mathit{tyApp}^*\ \mathcal{A}_2 \cup \mathit{coApp}^*\ \mathcal{B}_2)\ \mathcal{S}_4 \cup \mathit{app}^*\ (\mathit{prim} \cup \mathit{fun})\ \mathcal{R}_4 \cup \mathit{tyLam}\ \mathcal{S}_4 \cup \mathit{tyApp}^*\ \mathcal{A}_2 \cup \mathit{coLam}\ \mathcal{S}_4 \cup \mathit{coApp}^*\ \mathcal{B}_2 \cup \mathit{prim} \cup \mathit{con} \cup \mathit{var} \cup \mathit{fun} \cup \mathit{let}\ \mathcal{R}_4\ \mathcal{S}_4 \cup \mathit{case}\ (\mathcal{S}_4 - \mathit{lam}\ \mathcal{S}_4 - \mathit{tyLam}\ \mathcal{S}_4 - \mathit{coLam}\ \mathcal{S}_4)\ \mathcal{S}_4 \cup \mathit{cast}\ \mathcal{S}_4$

$\mathcal{A}_4 = \mathit{app}^*\ \mathit{var}\ \mathcal{S}_4 \cup \mathit{coApp}^*\ \mathcal{B}_4 \cup \mathit{con} \cup \mathit{var}$

$\mathcal{B}_4 = \mathit{app}^*\ \mathit{var}\ \mathcal{S}_4 \cup \mathit{tyApp}^*\ \mathcal{A}_4 \cup \mathit{con} \cup \mathit{var}$

$\square$

The next two lemmas, lemma 4.3.5 and lemma 4.3.6, form a bi-implication. The first lemma, lemma 4.3.5, shows that after all transformations have been applied exhaustively, all remaining pattern-bound variables are representable. The second lemma, lemma 4.3.6, shows that after all transformations have been applied exhaustively, all remaining lambda-bound variables are representable. As these lemmas form a bi-implication we will define our new set of possible expressions in corollary 4.3.1 which follows the two lemmas.

Lemma: 4.3.5 (Representable pattern-binders). *If there are only representable lambda-binders, then the subjects of case-decompositions are representable.*

Proof. Case-decompositions with an (application of a) constructor are removed by *CaseCon*. We thus show that, when the subject of a case-decomposition has a non-representable type, then the subject *always* becomes a constructor application. We only enumerate then cases still allowed by $\mathcal{S}_4$:

» (Applications of) global functions are inlined by InlineNonRep

- » Let-expressions will have their let-bindings extracted by CaseLet, leaving only the body.
- » Case-decompositions have their alternatives propagated to the alternatives of the encompassing case-decomposition by CaseCase.
- » Regarding casts:
  - – Casts of (applications of) global functions are propagated by CastSpec.
  - – Casts of (applications of) constructors are propagated by KPush.
  - – Casts of let-expressions are propagated by LetCast.
  - – Casts of case-decompositions are propagated by CaseCast.
- » Regarding (applications or casts of) local variables:
  - – By the precondition, it cannot be a lambda-bound variable, as it would be non-representable.
  - – It cannot be a pattern-bound variable, as that would imply that the subject of a case-decomposition has a non-representable type. By the induction hypothesis, those case-decompositions have already been removed by CaseCon.

□

Lemma: 4.3.6 (Representable lambda-bindings). *If there are only case-decompositions with representable subjects, then all lambda-binders are representable.*

Proof. The normalisation process proceeds in a top-down traversal of the function hierarchy, starting with the *main* function. We prove, by induction, that term-abstraction with a non-representable binder no longer exist:

- » We are normalising the *main* function:
  - – By the precondition on normalisation, *main* has a representable function type.
  - – The possible locations for a term-abstraction with a non-representable binder still possible in $\mathcal{S}_4$ are: as an argument of a constructor or a local variable application.
  - – However, a constructor with a non-representable argument is itself non-representable. By the precedent of the lemma, that subjects of case-decompositions are representable, so the non-representable constructor application can also only occur as an argument to a constructor or variable application.
  - – Because subjects of case-decompositions are representable, there are however no pattern-bound variables that can be the LHS of an application.
  - – When a lambda-bound variable is the LHS of an application, then the term-abstraction which introduces the binder can also only exists as an argument to an application.

  - Because *main* has a representable function type, the above mentioned application do not exist.

- » We are normalising a function other than *main*:
  - The type of the expression must be a representable (function) type. Otherwise the expression would have been inlined by InlineNonRep, or it would have entailed that the callers in the function hierarchy, and ultimately *main* would have had a non-representable (function) type.
  - The expressions has a representable (function) type; by lemma 4.3.3 the expression is specialised on all non-representable argument. Any term-abstraction that would have contributed to a non-representable function type has thus become the subject of an application. By lemma 4.3.2, those applications are removed, as witnessed by the possible expressions we can draw from $\mathcal{S}_4$.
  - Following the same reasoning as for the normalisation of *main*, term-abstractions with a non-representable type cannot occur anywhere else.

□

Corollary: 4.3.1 (No variable applications). *Applications of local variables are removed, and all bound variables are representable.*

Proof. Lemma 4.3.5 proves that, if there are only representable lambda-binders, then the subjects of case-decompositions are representable. Lemma 4.3.6 proves that, if there are only case-decompositions with representable subjects, then all lambda-binders are representable. From lemma 4.3.5 we can derive that all pattern-bound variables are representable.

Lemmas 4.3.4, 4.3.5, and 4.3.6 together show that all bound variables are representable. This entails that there are no local variables with a polymorphic or function type.

Our new set of expressions still possible under the corollary is:

$\mathcal{S}_5 = \mathit{lam}\,\mathcal{S}_5 \cup \mathit{app}^*\ (\mathit{con} \cup \mathit{tyApp}^*\ \mathcal{A}_5 \cup \mathit{coApp}^*\ \mathcal{B}_5)\ \mathcal{S}_5 \cup \mathit{app}^*\ (\mathit{prim} \cup \mathit{fun})\ \mathcal{R}_5 \cup \mathit{tyLam}\,\mathcal{S}_5 \cup \mathit{tyApp}^*\ \mathcal{A}_5 \cup \mathit{coLam}\,\mathcal{S}_5 \cup \mathit{coApp}^*\ \mathcal{B}_5 \cup \mathit{prim} \cup \mathit{con} \cup \mathit{var} \cup \mathit{fun} \cup \mathit{let}\,\mathcal{R}_5\,\mathcal{S}_5 \cup \mathit{case}\,\mathcal{R}_5\,\mathcal{S}_5 \cup \mathit{cast}\,\mathcal{S}_5$

$\mathcal{A}_5 = \mathit{coApp}^*\ \mathcal{B}_5 \cup \mathit{con}$

$\mathcal{B}_5 = \mathit{tyApp}^*\ \mathcal{A}_5 \cup \mathit{con}$

□

Definition 4.9 (Type- and coercion-applied constructor). *We define a new set generator which we use to model constructors applied to types and coercions:*

$con^{\Delta} = \{K\,\overline{\tau}\,\overline{\rho} \mid K \in \mathcal{K},\ \overline{\tau} \subseteq \mathcal{T},\ \overline{\rho} \subseteq (\mathcal{T} \cup \mathcal{C})\}$

We can now remove the explicit type and coercion applications from our possible set of expressions and redefine $\mathcal{S}_5$ as:

$\mathcal{S}_5 = lam\,\mathcal{S}_5 \cup app^{*}\,con^{\Delta}\,\mathcal{S}_5 \cup app^{*}\,(prim \cup fun)\,\mathcal{R}_5 \cup tyLam\,\mathcal{S}_5 \cup coLam\,\mathcal{S}_5 \cup prim \cup con^{\Delta} \cup var \cup fun \cup let\,\mathcal{R}_5\,\mathcal{S}_5 \cup case\,\mathcal{R}_5\,\mathcal{S}_5 \cup cast\,\mathcal{S}_5$

Lemma: 4.3.7 (No type- or coercion-abstractions). *All type-abstractions and all coercion-abstractions are removed.*

Proof. The normalisation process proceeds in a top-down traversal of the function hierarchy, starting with the *main* function. We prove, by induction, that type-abstractions and coercion-abstractions no longer exist:

- » We are normalising the *main* function:
  - By the precondition on normalisation, *main* has a representable function type.
  - The only possible location where type- and coercion-abstraction are still possible in $\mathcal{S}_5$, are as arguments in a constructor application.
  - Those constructor applications would themselves be non-representable, and can also only exists as arguments in a constructor application.
  - Because *main* has representable function type, those applications do not exist.
- » We are normalising a function other than *main*:
  - The type of the expression must be a representable (function) type. Otherwise the expression would have been inlined by InlineNonRep, or it would have entailed that the callers in the function hierarchy, and ultimately *main* would have had a non-representable (function) type.
  - The expressions has a representable (function) type; by lemma 4.3.3 the expression is specialised on all type and coercion arguments. Any type- or coercion-abstraction that would have contributed to a non-representable function type have thus become the subject of a type- or coercion-application. By lemma 4.3.2, those applications are removed, as witnessed by the possible expressions we can draw from $\mathcal{S}_5$.
  - Following the same reasoning as for the normalisation of *main*, type-abstractions and term-abstractions cannot occur anywhere else.

$\mathcal{S}_6 = lam\,\mathcal{S}_6 \cup app^{*}\,con^{\Delta}\,\mathcal{S}_6 \cup app^{*}\,(prim \cup fun)\,\mathcal{R}_6 \cup prim \cup con \cup var \cup fun \cup let\,\mathcal{R}_6\,\mathcal{S}_6 \cup case\,\mathcal{R}_6\,\mathcal{S}_6 \cup cast\,\mathcal{S}_6$

□

We now recapitulate our two theorems, theorem 4.3.1 and theorem 4.3.2, and prove them with using our lemmas. Recall definition 4.6:

Definition 4.6 (Representable binding). *A binding* (x,e) *is a representable binding, when:*

» *All variables bound in e have a representable type.*

» *All arguments in an application in e have a representable type.*

» *The expressions e has a representable type, or a representable function type (definition 4.5).*

Theorem: 4.3.1 (Representable bindings). *When the* main *function, and all used primitives have representable (function) types then, given a binding* (x,e)*,* (x,e) *is either a representable binding, or one of the rewrite rules applies to* e*.*

Proof. Combining lemmas 4.3.1 - 4.3.7 and corollary 4.3.1 we arrive at a set of possible expression forms, $\mathcal{S}_6$, when all rewrite rules have been exhaustively applied. In $\mathcal{S}_6$, all bound variables have a representable type (corollary 4.3.1), and the arguments of a primitive application and global function application are representable. Constructors with non-representable arguments are themselves non-representable. The normalisation process proceeds in a top-down traversal of the function hierarchy, starting with the *main* function. We prove, by induction, that constructors with non-representable arguments no longer exist:

» We are normalising the *main* function:
  - By the precondition on normalisation, *main* has a representable function type.
  - The only possible location where non-representable constructor are still possible in $\mathcal{S}_6$, are as arguments in a constructor application.
  - Those constructor applications would themselves be non-representable, and can also only exists as arguments in a constructor application.
  - Because *main* has representable function type, those applications do not exist.

» We are normalising a function other than *main*:
  - The type of the expression must be a representable (function) type. Otherwise the expression would have been inlined by InlineNonRep, or it would have entailed that the callers in the function hierarchy, and ultimately *main* would have had a non-representable (function) type.
  - Following the same reasoning as for the normalisation of *main*, constructors with non-representable arguments cannot occur anywhere else.

Our final set of possible expressions is hence:

$\mathcal{S}_7 = lam\ \mathcal{S}_7 \ \cup\ app^*\ (con^\Delta \ \cup\ prim \ \cup\ fun)\ \mathcal{R}_7 \ \cup\ prim \ \cup\ con \ \cup\ var \ \cup\ fun \ \cup\ let\ \mathcal{R}_7\ \mathcal{S}_7 \ \cup\ case\ \mathcal{R}_7\ \mathcal{S}_7 \ \cup\ cast\ \mathcal{S}_7$

□

Theorem: 4.3.2 (Representable environment). *Given a closed environment* Γ*, an environment of representable bindings* Γ_r*, and a binding* (x,e). *Let* (e',Γ') *be the resulting tuple of exhaustively applying all the rewrite rules to* e *and* Γ*, and let* $\Gamma'_r = \Gamma_r \cup \{(x, e')\}$. *Then* (x,e') *is a representable binding; and* Γ'_r *is a representable environment or normalisation continues with* Γ', Γ'_r*, and the binders corresponding to the free variables in* Γ'_r.

Proof. By theorem 4.3.1 exhaustively applying all the rewrite rules to an expression in the binding *(x,e)* creates a realisable binding *(x,e')*. So Γ'_r is also a set of realisable binders. When Γ'_r is closed, Γ'_r is a representable environment. When Γ'_r is not closed, the normalisation process will normalise the binders referenced in Γ'_r but are themselves not in Γ'_r. □

Now that we have proven that the normalisation process produces a *representable environment*, under the precondition that *main* and all used primitives have a representable function type, we will now move on to the termination aspects of the non-representable value removal pass.

4.3.3 Termination of non-representable value removal

There are several (combinations of) rewrite rules that induce non-termination of the unconstrained TRS. The CλaSH compiler must therefore:

» Apply the rewrite rules according to a specific strategy.

» Include termination measures.

When one of the termination measures is triggered, non-representable values remain present in the description. $\mathcal{T}_{C\lambda}$ will not be able to transform the description to a netlist when that happens.

It should be noted that these termination measures are only trigged on functions that contain unbounded (mutually) recursive function calls, or have a (mutually) recursive data type as a result; functions which cannot be synthesized by $\mathcal{T}_{C\lambda}$ anyway. It can hence be said that the CλaSH compiler can produce circuits for all *useful* circuit descriptions.

In the next three subsections we highlight where non-termination might occur, and how it is prevented. We will use Haskell instead of System FC, for purposes of brevity, to present expressions that can induce non-termination in the unconstrained TRS.

Non-termination of InlineNonRep

Although inlining is already restricted to subjects of case decompositions that have a non-representable type, inlining can still cause non-termination. This happens when a recursive function call becomes the subject of a case decomposition:

$$\begin{array}{l} \textbf{data}\ B\ a = B\ a \\ f = \textbf{case}\ f\ \textbf{of}\ B\ \_ \rightarrow B\ (\lambda x \rightarrow x) \end{array}$$

Where the type of f is data type with a field that has a function value, which is a non-representable data type. When we now apply the transformation rules, we get:

$$\begin{array}{l} \textbf{case}\ f\ \textbf{of}\ B\ \_ \rightarrow B\ (\lambda x \rightarrow x) \\ \quad \Rightarrow \quad \textit{InlineNonRep} \\ \textbf{case}\ (\textbf{case}\ f\ \textbf{of}\ B\ \_ \rightarrow B\ (\lambda x \rightarrow x))\ \textbf{of}\ B\ \_ \rightarrow B\ (\lambda x \rightarrow x) \\ \quad \Rightarrow \quad \textit{CaseCase} \\ \textbf{case}\ f\ \textbf{of}\ B\ \_ \rightarrow (\textbf{case}\ B\ (\lambda x \rightarrow x)\ \textbf{of}\ B\ \_ \rightarrow B\ (\lambda x \rightarrow x)) \\ \quad \Rightarrow \quad \textit{CaseCon} \\ \textbf{case}\ f\ \textbf{of}\ B\ \_ \rightarrow B\ (\lambda x \rightarrow x) \end{array}$$

Obviously this sequence of transformations will never terminate. Beside the fact that the expression is recursive, it also has a non-representable type, meaning it could have never been transformed to an actual circuit.

If we were to use f in an expression where its result is never used internally, such as:

$$\textit{main} = \lambda x \rightarrow \textbf{case}\ f\ \textbf{of}\ B\ \_ \rightarrow x$$

Then the order in which the transformations are applied will determine if normalisation terminates or not. If CaseCon is applied first we will terminate with the expression:

$$\textit{main} = \lambda x \rightarrow x$$

However, if we delayed the application of the CaseCon rule, we might end up with the following sequence of transformations:

$$\begin{array}{l} \lambda x \rightarrow \textbf{case}\ f\ \textbf{of}\ B\ \_ \rightarrow x \\ \quad \Rightarrow \quad \textit{InlineNonRep} \\ \lambda x \rightarrow \textbf{case}\ (\textbf{case}\ f\ \textbf{of}\ B\ \_ \rightarrow B\ (\lambda x \rightarrow x))\ \textbf{of}\ B\ \_ \rightarrow x \\ \quad \Rightarrow \quad \textit{CaseCase} \\ \lambda x \rightarrow \textbf{case}\ f\ \textbf{of}\ B\ \_ \rightarrow (\textbf{case}\ B\ (\lambda x \rightarrow x)\ \textbf{of}\ B\ \_ \rightarrow x) \\ \quad \Rightarrow \quad \textit{CaseCon} \\ \lambda x \rightarrow \textbf{case}\ f\ \textbf{of}\ B\ \_ \rightarrow x \end{array}$$

Resulting in a expression that will loop in the same way as the expression bound to f. To prevent such sequences from occurring we have adapted the strategy of our TRS to apply the CaseCon transformation before InlineNonRep.

If f were bound in a let-expression, and the bound variable is not used inside the expression, then the use of f will *disappear* from the function hierarchy through LiftNonRep or BindNonRep. In case f is applied to a term abstraction where the abstracted variable is not used, then f will also *disappear* from the function hierarchy through the combination of LamApp and, LiftNonRep or BindNonRep.

Additionally, although the TRS does not contain β-reduction as one of the rewrite rules, LamApp, LiftNonRep, InlineNonRep, and CaseCon together behave like β-reduction. This means that the typed version of $(\lambda x \rightarrow x\ x)\ (\lambda x \rightarrow x\ x)$:

$$\textbf{data}\ T = C\ (T \rightarrow Int)$$
$$(\lambda x \rightarrow \textbf{case}\ x\ \textbf{of}\ C\,h \rightarrow h\ x)\ (C\ (\lambda x \rightarrow \textbf{case}\ x\ \textbf{of}\ C\,h \rightarrow h\ x))$$

induces non-termination. To prevent this situation from happening, a function f can only be inlined a finite number of times, where the amount of inlinings per function can be set by the user of the CλaSH compiler.

Non-termination of specialisation

A situation that would cause non-termination of NonRepSpec, is when the non-representable argument that will be propagated has local free variables with a non-representable type.

$$\begin{aligned}
&\textbf{let}\ x{:}\,Int \rightarrow Int = \lambda x{:}\,Int \rightarrow x\ \textbf{in}\ f\ x\\
&\quad\Rightarrow \quad \textit{NonRepSpec}\\
&\textbf{let}\ x{:}\,Int \rightarrow Int = \lambda x{:}\,Int \rightarrow x\ \textbf{in}\ f'\ x\\
&\quad\Rightarrow \quad \textit{NonRepSpec}\\
&\textbf{let}\ x{:}\,Int \rightarrow Int = \lambda x{:}\,Int \rightarrow x\ \textbf{in}\ f'\ x\\
&\quad\Rightarrow \quad \textit{NonRepSpec}\\
&\textbf{let}\ x{:}\,Int \rightarrow Int = \lambda x{:}\,Int \rightarrow x\ \textbf{in}\ f'\ x
\end{aligned}$$

In this situation, the local free variable (which has a non-representable type) would also be applied to the specialized function. The NonRepSpec transformation would thus specialize indefinitely as the local free variables, who have a non-representable type, will remain applied to the specialized function. This is one of the reasons why the strategy of our TRS only applies the NonRepSpec transformation when all the other transformations have already been exhaustively applied. When NonRepSpec is applied last, all local free variables with a non-representable type are replaced by either a global variable or a term-abstraction (through BindNonRep and LiftNonRep).

Another reason for non-termination as a result of NonRepSpec is when a recursive function f has an argument that accumulates non-representable values.

```
main = f id
f :: (Int → Int) → Int → Int
f g x = f ((+1) . g) x
```

Which after the first specialisation becomes:

```
main = f'
f :: (Int → Int) → Int → Int
f g x = f ((+1) . g) x

f' x = f ((+1) . id) x
```

And after another round of specialisation becomes:

```
main = f'
f :: (Int → Int) → Int → Int
f g x = f ((+1) . g) x

f'  x = f'' x
f'' x = f ((+1) . ((+1) . id)) x
```

Which would lead to endless specialisations of f. To ensure termination, a NonRepSpec is only applied to a function a finite number of times, where the amount of specialisation per function can be set by the user of the CλaSH compiler.

In a situation similar to the above, TypeSpec can induce non-termination when specialising polymorphic recursive functions such as:

```
data Nested a = a :⟨: (Nested [a]) | Epsilon
 infixr 5 :⟨:

lengthN :: Nested a → Int
lengthN Epsilon      = 0
lengthN (_ :⟨: xs) = 1 + lengthN xs

main :: Nested Int → Int
main = lenghtN
```

Where *lengthN*, will first be specialised on *Int*, after that on *[Int]*, after that on *[[Int]]*, and so on. So to ensure termination, not only NonRepSpec is equipped with a termination measure that limits the number of specialisations. All specialisation functions, NonRepSpec, TypeSpec, CoSpec, and CastSpec, are only applied a finite number of times.

Looping of NonRepSpec and LiftNonRep

We have already elaborated why applying BindNonRep is preferable above LiftNonRep when possible. Below we show a situation why BindNonRep must be included to prevent non-termination.

The LiftNonRep transformation could induce non-termination in the presence of recursion. Again, we want to highlight that unbounded recursion is not synthesisable to a digital circuit under our synthesis scheme $\mathcal{T}_{C\lambda}$. The following example will result in non-termination due to interactions between LiftNonRep and NonRepSpec:

```
main = f not
f    = λa x → (a x) && (f a x)
```

In the first iteration, we create a version of f specialized on *not*: f':

```
main = f’
f    = λa x → (a x) && (f a x)
f’   = (λa x → a x && (f a x)) not
```

When the TRS starts working on f', it will first bind *not* to the variable a in a let-binding through LamApp.

```
main = f’
f    = λa x → (a x) && (f a x)
f’   = let a = not in (λx → a x && (f a x))
```

The a binder is turned into a global variable g by LiftNonRep.

```
main = f’
f    = λa x → (a x) && (f a x)
f’   = λx → g x && (f g x)
g    = not
```

The recursive call to f will now be specialized on g, to create a new version of f called f''.

```
main = f’
f    = λa x → (a x) && (f a x)
f’   = λx → g x && (f” x)
g    = not
f”   = (λa x → (a x) && (f a x)) g
```

This process will continue indefinitely, as LiftNonRep will create a new global variable, meaning the specializations are not α-equivalent. Specialization would therefore generate an infinite number of specializations of f.

This is the reason that LiftNonRep only works for self-referencing let-binders, and BindNonRep is used for all the others. When the set of functions has the form:

```
main = f’
f’   = let a = not in (λx → a x && (f a x))
f    = λa x → (a x) && (f a x)
```

Instead of creating a new function g, the function *not* is simply inlined by BindNonRep in f'.

```
main = f’
f’   = λx → not x && (f not x)
f    = λa x → (a x) && (f a x)
```

Specialization will in this case *not* create a new function f'', because the specialisation of f on *not* has been seen before.

$$\begin{aligned}
&main = f' \\
&f' \quad = \lambda x \rightarrow \; not \; x \; \&\& \; (f' \; x) \\
&f \quad = \lambda a \; x \rightarrow \; (a \; x) \; \&\& \; (f \; a \; x)
\end{aligned}$$

Strategy

We have seen that the unconstrained TRS can induce non-termination, and that in some cases extra termination measures have to be implemented in the CλaSH compiler. Other forms of non-termination could be induced by picking the transformations in the wrong order. We thus apply the transformations in a specific order. We classify our transformations in three groups:

Argument specialisation: These transformations specialise top-level functions on their argument, which are: TypeSpec, CoSpec, and NonRepSpec.

Inlining: These transformations inline top-level functions, which is only the InlineNonRep transformation.

Propagation: These transformations propagate informations downwards into the expression, and includes all the transformations not mentioned above.

The strategy that we employ is the following:

1. Apply the propagation transformations in an inner-most traversal.
2. Apply the inlining transformations in a bottom-up traversal, if these succeed, run step 1 again.
3. Apply the specialisations transformations in a bottom-up traversal.

Neither the correctness of the individual transformations, nor the guarantee of a normal form, are dependent on this specific ordering of transformations. The argument-specialisation rewrite rules are applied last, so that the fewest number of new functions is introduced, and the original function hierarchy is preserved as much as possible. Because specialisation transformations do not create expressions on which the other rewrite rules match, all rewrite rules have been applied exhaustively after the traversal with specialisation transformations. The CastSpec transformation is included in the propagation transformations, and not the specialisation transformations, because it enables firings of InlineNonRep. Putting CastSpec in the propagations transformations is hence required in order to guarantee the normal form.

4.3.4 Simplification

The normal form that facilitates a trivial translation to a netlist is repeated in figure 4.15. A top-level expression, t, is single let-expression with a variable reference as a body; it is possibly λ-abstracted over its arguments. The bindings of the let-expression are all in administrative normal form (ANF).

Representable types			
τ_r	$::=$	$T_r\ \overline{\tau_r}$	Representable data types
	$\mid$	$(F\ \overline{\tau})_r$	Representable type function result
Representable expressions			
t	$::=$	$\lambda\overline{(x : \tau_r)}.\mathbf{let}\ \overline{y : \tau_r = r}^{+}\ \mathbf{in}\ y_j$	Top-level function
r	$::=$	x	Local variable reference
	$\mid$	$f\ \overline{x}$	Saturated top-level function
	$\mid$	$K\ \overline{\tau_r}\ \varnothing\ \overline{x}$	Saturated data constructor
	$\mid$	$\otimes\ \overline{x}$	Saturated primitive
	$\mid$	$\mathbf{case}\ x\ \mathbf{of}\ \overline{p \to y}$	Case decomposition
	$\mid$	$\pi_i^k\ x$	Projection
Patterns			
p	$::=$	$\_$	Default case
	$\mid$	$K\ \varnothing\ \_$	Matches data constructor

Figure 4.15 – System FC in Normal Form, repeated

The grammar that we can extract from $\mathcal{S}_7$ of theorem 4.3.1 (from the proof of representable binders) is given in figure 4.16. It shares the type grammar with figure 4.15. The next four subsections describe the transformations that reduce an expression from the grammar in figure 4.16 to the desired normal form, given that the rewrite rules from the previous section have already been exhaustively applied. Each subsection describes a set of transformations, these sets are applied in the order that they are described, and every set has its own strategy.

Eta-expansion

The first step of the simplification process is η-expanding the top level expression in a top-down traversal until the sub-expression no longer has a function type. We subsequently apply an inner-most traversal with the transformations LamApp, LetApp, and CaseApp.

EtaExpand

$$\frac{e}{\lambda x : \sigma . e\ x} \qquad \begin{array}{l}\text{Preconditions: } \tau = \sigma \to \tau' \wedge e \neq \lambda y : \sigma'.e' \\ \text{Definitions: } \tau = \text{TYPEOF(e)}\end{array}$$

As a result, we will only have a top-level term-abstractions. Term-abstraction in the body of let-expression and alternatives of case-decompositions will have disappeared. The resulting grammar after this process is:

Representable expressions

r	$::= \lambda\overline{(x : \tau_r)}.e$	Term abstraction
	$\mid\ x$	Local variable reference
	$\mid\ f\ \overline{r}$	Saturated top-level function
	$\mid\ K\ \overline{\tau_r}\ \varnothing\ \overline{r}$	Saturated data constructor
	$\mid\ \otimes\ \overline{r}$	Saturated primitive
	$\mid\ \textbf{let}\ \overline{y : \tau_r = r}\ \textbf{in}\ r$	Let expression
	$\mid\ \textbf{case}\ r\ \textbf{of}\ \overline{p \to r}$	Case decomposition
	$\mid\ r \rhd \gamma$	Cast

Patterns

p	$::= \_$	Default case
	$\mid\ K\ \varnothing\ \overline{x : \tau}$	Matches data constructor

Figure 4.16 – System FC in representable form

t	$::= \lambda\overline{(x : \tau_r)}.r$	Term abstraction
r	$::= x$	Local variable reference
	$\mid\ f\ \overline{r}$	Saturated top-level function
	$\mid\ K\ \overline{\tau_r}\ \varnothing\ \overline{r}$	Saturated data constructor
	$\mid\ \otimes\ \overline{r}$	Saturated primitive
	$\mid\ \textbf{let}\ \overline{y : \tau_r = r}\ \textbf{in}\ r$	Let expression
	$\mid\ \textbf{case}\ x\ \textbf{of}\ \overline{p \to r}$	Case decomposition
	$\mid\ r \rhd \gamma$	Cast

Administrative normal form

We turn all application into administrative normal form (ANF) using the ANF transformation.

ANF

$$\frac{e\ u}{\textbf{let}\ \{x : \sigma = u\}\ \textbf{in}\ e\ x} \qquad \begin{array}{l}\text{Preconditions: } u \neq x \\ \text{Definitions: } \sigma = \text{TYPEOF(u)}\end{array}$$

Case-decompositions are also transformed into ANF. The first rewrite rule, SubjectANF, creates a binding for the subject of a case-decomposition if this subject is not already a local variable reference.

SubjectANF

$$\frac{\textbf{case } e \textbf{ of } \overline{p \to u}}{\textbf{let } \{x : \sigma = e\} \textbf{ in case } x \textbf{ of } \overline{p \to u}}$$

Preconditions: $e \not\equiv x$
Definitions: σ = TYPEOF(e)

The second rewrite rule, AltANF, creates bindings for the expressions in the alternatives of a case-decomposition. As it was the case for SubjANF, the expression u_k must not already be local variable references. Aside from creating let-bindings for the expressions, AltANF let-binds projections, π_i^k, for the pattern variables y_k that are referenced in the expression u_k. Note that these projections will never be *evaluated* when their argument, x, is not of the expected constructor, as the expression referencing their let-binders, u_k, is only evaluated when the case-decomposition evaluating u_k has already asserted that x has the expected constructor.

AltANF

$$\frac{\textbf{case } e \textbf{ of } \{...; p_k \to u_k : \sigma; ...\}}{\textbf{let } \{\overline{z : \tau_i = \pi_i^k\ x}; x_k : \tau = u_k[\overline{z/y_k}]\} \textbf{ in case } e \textbf{ of } \{...; p_k \to x_k; ...\}}$$

Preconditions: $u_k \not\equiv x$
Definitions: $(\varnothing, \overline{y_k : \tau}) = \text{fv}(u_k)$ - $\text{fv}(p_k \to u_k)$

These three transformations are applied in a single bottom-up traversal, and result in the following new grammar:

$r \ ::= x$	Local variable reference
$\mid\ f\ \overline{x}$	Saturated top-level function
$\mid\ K\ \overline{\tau_r}\ \varnothing\ \overline{x}$	Saturated data constructor
$\mid\ \otimes\ \overline{x}$	Saturated primitive
$\mid\ \textbf{let}\ \overline{y : \tau_r = r}\ \textbf{in}\ r$	Let expression
$\mid\ \textbf{case}\ x\ \textbf{of}\ \overline{p \to x}$	Case decomposition
$\mid\ \pi_i^k\ x$	Projection
$\mid\ r \rhd \gamma$	Casts
Patterns	
$p \ ::= \_$	Default case
$\mid\ K\ \varnothing\ \_$	Matches data constructor

Let-simplification

The next three transformations ensure that the body of a let-expression is a local variable reference, and none of the bound-expressions are themselves a let-expression. The first transformation, bodyVar, ensures that the (representable) body of a let-expression is always a local variable reference. We have already seen the second transformation, LetCast, in the non-representable value elimination pass. It ensures that casts are propagated towards the body of a let-expressions, and is needed for the third rewrite rule to fire. The third transformation, LetFlat, finally ensures that nested let-expressions are turned into a single let-expression.

BodyVar

$$\frac{\textbf{let}\ \overline{x : \sigma = u}\ \textbf{in}\ e}{\textbf{let}\ \{\overline{x : \sigma = u}, x_{n+1} : \tau = e\}\ \textbf{in}\ x_{n+1}}$$

Preconditions: $e \neq x$

Definitions: τ = TYPEOF(e)

LetCast

$$\frac{(\textbf{let}\ \overline{x : \sigma = u}\ \textbf{in}\ e)\ \triangleright \gamma}{\textbf{let}\ \overline{x : \sigma = u}\ \textbf{in}\ (e\ \triangleright \gamma)}$$

LetFlat

$$\frac{\textbf{let}\ \{b_1; ...; b_{i-1}; x_i : \sigma_i = \textbf{let}\ \overline{x : \sigma = u}\ \textbf{in}\ e_i; b_{i+1}; ...; b_n\}\ \textbf{in}\ u}{\textbf{let}\ \{b_1; ...; b_{i-1}; \overline{x : \sigma = u}; x_i : \sigma_i = e_i; b_{i+1}; ...; b_n\}\ \textbf{in}\ u}$$

The above three transformations are applied using a single bottom-up traversal. The final rewrite rule, TopLet, ensures that the expression following the top-level term-abstractions is always a let-expression. As we did for EtaExpand, TopLet is applied in a top-down traversal until the sub-expression no longer has a function type.

TopLet

$$\frac{e}{\textbf{let}\ \{x : \tau = e\}\ \textbf{in}\ x}$$

Preconditions: $(e \neq \textbf{let}\ \overline{y : \sigma}\ \textbf{in}\ z) \wedge (\tau \neq \sigma \rightarrow \tau')$

Definitions: τ = TYPEOF(e)

We finally arrive at the grammar given in figure 4.17.

Remaining casts

When looking at the last grammar given in figure 4.17, it almost matches the grammar in figure 4.15, with the exception of the casts. Coercions witness a non-syntactical equality between types. Consequently, both the coerced and uncoerced value

$$\begin{array}{lll}
t ::= \lambda\overline{(x : \tau_r)}.\mathbf{let}\ \overline{y : \tau_r = r}^{+}\ \mathbf{in}\ y_j & & \text{Top-level function} \\
r ::= x & & \text{Local variable reference} \\
\quad \mid\ f\ \overline{x} & & \text{Saturated top-level function} \\
\quad \mid\ K\ \overline{\tau_r}\ \varnothing\ \overline{x} & & \text{Saturated data constructor} \\
\quad \mid\ \otimes\ \overline{x} & & \text{Saturated primitive} \\
\quad \mid\ \mathbf{case}\ x\ \mathbf{of}\ \overline{p \to y} & & \text{Case decomposition} \\
\quad \mid\ \pi_i^k\ x & & \text{Projection} \\
\quad \mid\ r \triangleright \gamma & & \text{Casts} \\
p ::= \_ & & \text{Default case} \\
\quad \mid\ K\ \varnothing\ \_ & & \text{Matches data constructor}
\end{array}$$

Figure 4.17 – System FC in (almost) Normal Form

will have the same bit-encoding. Hence, during the translation to netlist, the casts are ignored.

4.4 Discussion

4.4.1 Properties of the normalisation phase

We have proven that the pass of the normalisation phase which removes non-representable values from the function hierarchy is complete. That is, given only minor restrictions, it reduces the function hierarchy that contains polymorphic and higher-order functions to a completely monomorphic and first-order function hierarchy. Also, every bound variable, and every argument to a function or primitive is representable.

We have also shown that the unconstrained TRS is prone to non-termination, and requires a specific strategy, and certain termination measures, to actually terminate in the presence of recursion. There is, however, no proof that this pass of the normalisation phase terminates. We want to sketch our case that there is a strong indication that the CλaSH compiler actually terminates in the presence of unbounded recursion:

» Local recursive functions are always lifted to global functions by LiftNonRep.

» Those transformations that have the potential to induce non-termination, InlineNonRep and the specialisation transformation, are equipped with termination measures that limit their number of applications on the same function.

» There is no transformation in the set of propagation transformations where the resulting expression of one transformation is the matching expression of another.

Additionally we want to state that in years of use of the CλaSH compiler there has been no report of non-termination[5].

The same statement holds for completeness of the simplification pass or normalisation. In our experience, the CλaSH compiler always reduces expressions to the desired normal form[6]. We also want to state that the expression that is the result of the first pass of normalisations, non-representable value elimination, would already be in a state in which it can be synthesized to a netlist. It would, however, not be as trivial as it is in the case of our normal form.

4.4.2 Correspondence operational semantics and netlists

The operational semantics of System FC (as given in appendix C) is a *call-by-name* semantics, where the arguments to a function are not evaluated before the function is called. In a circuit as derived by $\mathcal{T}_{C\lambda}$, all circuits are operating simultaneously. Under the assumption that the primitives are free from side-effects, the behaviour of the circuit follows the behaviour as derived from the operational semantics, even in the presence of non-termination. Given the constant function on integers, $const = \lambda x{:}\,Int\,.\lambda y{:}\,Int\,.x$, the program, $main = const\ 3\ (\mathbf{let}\ x{:}\,Int = x\ \mathbf{in}\ x)$, will reduce to the following normal form:

```
const = λx.λy. let  z: Int = x in z
main  = let  x: Int = x
             k: Int = 3
             z: Int = const k x
        in   z
```

which reduces to 3 according to the operational semantics, even though *const*'s second argument is non-terminating. The actual circuit will have a component corresponding to *const* where the first port is connected to a bundle of wires representing the literal 3, and the second argument is connect to a combinational feedback loop. The *const* component will, however, route only its first input to the output, and its second input is not even connected. Consequently, the circuit produces the constant value 3. Although we have only described the static case, the same holds for the dynamic case. When the input port of a component is connected to a combinational feedback loop (a non-terminating expression in System FC), but the computation is not dependent on this input, the circuit connected to the output is in no way influenced by the feedback loop.

The same reasoning applies to the parallel translation of case-decompositions and let-expressions. Even though some circuit paths are performing computations on

[5] Excluding incompleteness or non-termination due to incorrect implementation of the theory.
[6] See footnote 5.

nonsensical values, as those alternatives would not have been evaluated in the sequential case, the generated multiplexer will only select the output of the circuit path that corresponds to the evaluation of the chosen alternative. The circuits calculating nonsensical values have no influence on the behaviour of the other circuit paths, or of the multiplexer. The translations is hence functionally correct. From a energy performance perspective it might, however, be preferable to actually *turn off* the circuit paths corresponding to the non-chosen alternatives.

In the above, and in the actual CλaSH compiler, we assume that all primitives are side-effect free. In the presence of side-effects our translation would be incorrect, as side-effects would be performed eagerly and resources associated with the side-effect would have to be duplicated. To solve the problem of eagerly evaluating side-effects, we would have to add handshake synchronisation for every data-channel, by which a circuit can indicate that it wants the side-effect performed and a response by which the circuit knows that the side-effect is finished. Solving the implicit resource duplication problem (e.g.the impossible duplication of a VGA monitor) is far more difficult, and would require adding scheduling logic. Concluding, side-effecting primitives do not fit in our view of seeing function descriptions as a structural composition of components.

4.4.3 Recursive descriptions

A feature notably lacking from our normalisation phase is the normalisation and unrolling of bounded recursive functions. The actual implementation of the CλaSH compiler includes additional transformations which:

» Inline closed constructor applications.

» Evaluate closed primitive applications, using a δ-reduction.

» Specialize functions on closed constructor applications.

In combination with the CaseCon, the above set of transformations can unroll rudimentary recursive descriptions by creating successive specialised functions. This is, by far, not enough to unroll all bounded recursive functions.

At the moment, this lack of support for bounded recursive functions is amortised by marking the higher-order functions over vectors, such as *map* and *foldr*, as primitives. Just as for any other primitive, the CλaSH compiler has special translation logic for these primitives. This has, however, violated the conditions of our system as described in this thesis that primitives:

» Are monomorphic.

» Only operate on data types as arguments.

» That the arguments of primitives must be representable.

We want to note that they are only considered primitive for synthesis, but not for execution within the Haskell interpreter. Although we have only proven that our TRS reduces a description to normal form when primitives have representable arguments, the compiler actually reduces descriptions to a normal form almost identical to the one presented in this thesis, with the exception that primitives have non-representable arguments. Primitives may, however, not have non-representable results, as they do not have definitions that can be inlined. As future work we should hence revisit our completeness proofs with less conservative restrictions on primitives.

4.5 Conclusions

The CλaSH compiler uses a synthesis scheme, $\mathcal{T}_{C\lambda}$, that produces a description that has specific normal from. One aspect of this normal form is that arguments and results of expressions have types for which a fixed bit-encoding exists. For $\mathcal{T}_{C\lambda}$, non-representable values are those values for which no fixed bit-encoding can be determined.

Given only minor restrictions the TRS presented in this chapter removes all non-representable values from a function hierarchy while preserving:

» The behaviour.

» The original function hierarchy, where possible.

» Sharing, where possible.

These restrictions are that:

» The *main* function is monomorphic.

» The arguments and the result of the *main* function are representable.

» The arguments and result types of primitives are representable types.

These restrictions do, however, not limit the use of polymorphism or higher-order functionality in the rest of the description.

4.5.1 Future work

The two big features that are currently lacking from our synthesis approach are:

» Support for unrolling bounded recursion.

» Support for GADTs as arguments and result of *main*.

At the moment, GADTs are considered non-representable because of their *existential* coercion arguments. GADTs can therefore not be arguments or results to *main*. Support for GADTs at the root of the function hierarchy will also enable more unfolding of bounded recursive descriptions.

Extending support for GADTs

The problem with supporting GADTs as arguments to, and result of, *main*, is that they have *existential* arguments. These *existential* arguments cannot be projected and let-bound outside of the case-decomposition. We can hence not reach our desired *normal* form where case-decompositions are in administrative normal form (ANF).

There are, however, cases where we can imagine a transformation that eliminates a case-decomposition on a GADT while the constructor is not yet known. We present two such cases, the first one uses the example that we used earlier to explain GADTs:

```
MkI : ∀ a :*. ∀ c : a~Int  →  Int  →  T a
MkB : ∀ a :*. ∀ c : a~Bool →  Bool →  T a

f = Λa:*.λx:a.λds:T a.case ds of
  MkI (dt:a~Int)  (j : Int) → ((x ▷dt) + j) ▷sym dt
  MkB (dt:a~Bool) (b : Int) → ((x ▷dt) && b) ▷sym dt

main = f Int
```

After specialising f on *Int*, and propagating the type application, we get a f' that looks like:

```
f’ = λx: Int .λds: T Int . case ds of
  MkI (dt: Int~Int)  (j : Int) → ((x ▷dt) + j) ▷sym dt
  MkB (dt: Int~Bool) (b: Int)  → ((x ▷dt) && b) ▷sym dt
```

Now we have one alternative with a coercion variable that states a tautology, *Int ~ Int*, and another alternative with a coercion variable that states a contradiction, *Int ~ Bool*. There is no safe way to construct a coercion that witnesses a contradiction, so we can prune the respective alternative, leaving only the alternative with the tautology.

```
f’ = λx: Int .λds: T Int . case ds of
  MkI (dt: Int~Int)  (j : Int) → ((x ▷dt) + j) ▷sym dt
```

We can now almost use a combination of the CaseCon and AltANF transformation to eliminate the case-decomposition. The remaining problems are the references to the coercion variable *dt* in the expression. We can, however, safely replace this coercion variable with another coercion that witnesses *Int ~ Int*, the reflexivity coercion: ⟨*Int*⟩:

```
f’ = λx: Int .λds: T Int . case ds of
  MkI (dt: Int~Int)  (j : Int) → ((x ▷⟨Int⟩) + j) ▷sym ⟨Int⟩
```

Now using AltLet, followed by CaseCon, we get:

```
f' = λx:Int.λds:T Int.
        let j  : Int = π¹₁ds
            x' : Int = ((x ▷⟨Int⟩) + j) ▷sym ⟨Int⟩
        in  x'
```

Where there are no more references to existential arguments of the constructor *MkI*.

Unfolding vector operations The next example is one that ties into unfolding of bounded recursive descriptions. We define, the constructors for the *Vec* type, the *map* function on vectors, and a *main* function that maps (+1) over a vector of four integer:

```
Nil : ∀ n:Nat.∀ a:*. ∀ c:n~0 → Vec n a
Con : ∀ n:Nat.∀ a:*. ∀ n1:Nat.∀ c:n1+1~n → a → Vec n1 a → Vec n a

map = Λa:*.Λb:Int.Λn:Nat.λf:(a → b).λxs:Vec n a.case xs of
  Nil (dt:n~0) →
    (Nil 0 b ⟨0⟩) ▷(⟨Vec⟩ (sym dt) ⟨b⟩)
  Cons (n1:Nat) (dt:n~n1+1) (x:a) (xs:Vec n1 a) →
    (Cons (n1+1) b n1 ⟨n1+1⟩
          (f x)
          (map a b n1 f xs)
    ) ▷(⟨Vec⟩ (sym dt) ⟨b⟩)

main = map Int Int 4 (+1)
```

After one round of specialisation of *map* on its type and function arguments, and subsequently further propagation of these arguments, we get a new version, *map'*:

```
map' = λxs:Vec 4 Int.case xs of
  Nil (dt:4~0) →
    (Nil 0 Int ⟨0⟩) ▷(⟨Vec⟩ (sym dt) ⟨Int⟩)
  Cons (n1:Nat) (dt:4~n1+1) (x:Int) (xs:Vec n1 Int) →
    (Cons (n1+1) Int n1 ⟨n1+1⟩
          ((+1) x)
          (map Int Int n1 (+1) xs)
    ) ▷(⟨Vec⟩ (sym dt) ⟨Int⟩)
```

Again, we have an alternative with a contradiction (4~0) that we can prune:

```
map' = λxs:Vec 4 Int.case xs of
  Cons (n1:Nat) (dt:4~n1+1) (x:Int) (xs:Vec n1 Int) →
    (Cons (n1+1) Int n1 ⟨n1+1⟩
          ((+1) x)
          (map Int Int n1 (+1) xs)
    ) ▷(⟨Vec⟩ (sym dt) ⟨Int⟩)
```

Next comes the rather big step, which will requires a better formalisation in order to guarantee a correct transformation.

» Because +1, in the coercion dt:4~n1+1, is an *injective* type function, we can *calculate* that n1 must be 3.

» We can subsequently replace every occurrence of n1 by 3.

» The coercion variable then witnesses dt:4~3+1, which, after normalisation of the type function (+) becomes dt:4~4.

» We can then replace ever reference to the coercion variable dt by the coercion ⟨4⟩.

As a result we will get:

```
map' = λxs:Vec 4 Int.case xs of
  Cons (n1:Nat) (dt:4~4) (x:Int) (xs':Vec 3 Int) →
    (Cons (3+1) Int 3 ⟨3+1⟩
          ((+1) x)
          (map Int Int 3 (+1) xs)
    ) ▷(⟨Vec⟩ (sym ⟨4⟩) ⟨Int⟩)
```

We can now apply AltLet, followed by CaseCon, to get:

```
map' = λxs:Vec 4 Int.
          let x  = π²₁ xs
              xs = π²₂ xs
              x' = (Cons (3+1) Int 3 ⟨3+1⟩
                         ((+1) x)
                         (map Int Int 3 (+1) xs)
                   ) ▷(⟨Vec⟩ (sym ⟨4⟩) ⟨Int⟩)
          in  x'
```

After a final specialisation of *map* we end up with:

```
map’ = λxs:Vec 4  Int .
          let x    = $\pi_1^2$ xs
              xs’  = $\pi_2^2$ xs
              x’   = (Cons (3+1) Int 3 ⟨3+1⟩
                            ((+1) x)
                            (map” xs)
                      ) ▷(⟨Vec⟩ (sym ⟨4⟩) ⟨Int⟩)
          in  x’
```

This line of specialisations will continue until the second alternative will witness the contradiction, *dt*:0~*n*1+1, where no such natural number *n*1 exists.

Unrolling bounded recursion

In the discussion, section 4.4, we already indicated that the CλaSH compiler unrolls very rudimentary form of bounded recursion. A small extension to the current techniques implemented in the compiler is to not only inline, and specialise on, *closed* constructor applications, but also constructors applications that are not closed. Once our description is already in its normal form, inlining constructor applications does not result in loss of sharing, as all computations are already let-bound. Using the techniques in the previous subsection will also enable unrolling bounded recursive descriptions operating on vectors.

A problem with unfolding bounded recursive descriptions, is that it is not just simply inlining recursive function calls. We must also, in some cases, inline functions that form the subject of a case-decomposition containing the recursive function calls. Determining whether a function must be inlined will then suddenly start to depend on a lot of context, and that is difficult to capture in our term rewrite system (TRS). Simply inlining *all* function definitions is not desirable as we want to preserve our function hierarchy as much as possible. It might additionally lead to loss in sharing.

Challenges for future work are thus:

» To find local metrics to determine whether a function should be inlined for the purpose of unrolling bounded recursive descriptions.

» Define a partial inlining procedure that preserves sharing.

» Prove that the method for unrolling bounded recursive descriptions will always lead to a non-recursive normal form.

5 Advanced aspects of circuit design in CλaSH

Abstract – *Even when descriptions use high-level abstractions, the CλaSH compiler can synthesize efficient circuits. Case studies show that circuits designed in Haskell, and synthesized with the CλaSH compiler, are on par with hand-written VHDL, in both area and gate propagation delay. Even in the presence of contemporary Haskell idioms and abstractions to write imperative code (for a control-oriented circuit) does the CλaSH compiler create results with decent non-functional properties. To emphasize that our approach enables correct-by-construction descriptions, we demonstrate abstractions that allow us to automatically compose components that use back-pressure as their synchronisation method. Additionally, we show how cycle delays can be encoded in the type-signatures of components, allowing us to catch any synchronisation error at compile-time.*

5.1 Introduction

The previous two chapters introduced the ideas behind designing synchronous circuits in Haskell, and synthesising these descriptions to a netlist. Chapter 3 introduced some small examples, but mainly to demonstrate the ideas behind the techniques to design circuits. The CλaSH compiler has, however, been used for much larger designs. It has been used to create various DSP circuits: such as filter banks [73], particle filters [71][CB:11], spectral estimation techniques [36], a model of the human cochlea [66], and stencil computations [72]. Additionally, the use of higher-order functions to make correct-by-constructions trade-offs between a space/time unrolling of computations, and subsequent mapping to circuits, is

Parts of this chapter have been published in [CB:7] and [CB:9].

explored in [39, 70]. Finally, CλaSH has also been used to design a many-core processor architecture that is based on data-flow principles [48][CB:3].

There are hence many existing examples of using CλaSH for circuit design, and the use of higher-order to make design trade-offs. The purpose of this chapter is therefore not to give an overview how CλaSH can be used for circuit design in general. Instead, it will focus on *some* of the more advanced aspects of circuit design in CλaSH.

As an introduction, the next section will describe the design of a medium sized circuit, a streaming reduction circuit [18], which is still designed in terms of *ordinary* function composition. After that, we move on to describing the larger CλaSH demonstrator circuit, where we use two aspects that are *novel* in terms of synthesisable circuit design in Haskell (compared to e.g. [26]): multiple clock domains, and the use of monads to describe imperative, control-oriented, circuits. The last two sections will focus on the correct-by-construction composition of circuits, where one section will use higher-order functions to compose circuits that use back-pressure, and the other section will use type-level functions and numbers to statically enforce correct synchronisation.

5.2 Streaming reduction circuit

When solving the matrix equation $Ax = b$ for a big sparse matrix A, one often uses the conjugate gradient algorithm [61]. A core operation of the conjugate gradient algorithm is a sparse matrix-vector multiplication *(SMxV)*, which we can perform by taking the dot product for every row in the matrix. For an SMxV the number of multiplications and additions required for an elements in the result vector depends on the number of non-zeros in the respective row of the matrix. Working with floating pointer numbers, the multiplication and addition operations are often pipelined in order to achieve reasonable operating frequencies.

In a pipelined arithmetic logic unit *(ALU)*, a new operation can be scheduled every cycle, but it will take several cycles before the calculated result exits the pipeline. We demonstrate this behaviour in figure 5.1, where an addition is scheduled every cycle. For demonstrative purposes, in our notation the addition takes place immediately, and the result will then propagate through the pipeline.

As part of the dot product, we will have to sum all the numbers in a row of a matrix. Summing all elements in a row using a pipelined adder is, however, more complex than it is when using a non-pipelined adder. Assume for instance that we want to sum three values in a row using a 14-stage pipelined adder. Adding the first two values is trivial, after that we will have to wait 14 cycles before the result of this addition can be added to the third value. While we wait, however, there may be values from other rows that also need to be summed. We hence desire to schedule the pipeline so that it can reduce multiple rows simultaneously, of course, without accidentally summing values from different rows.

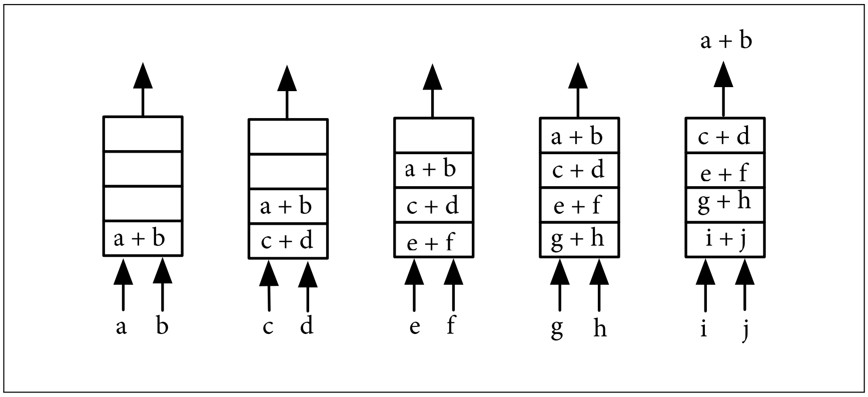


Figure 5.1 – Pipelining, 5 consecutive clock cycles

Circuits which can process variable length rows of floating point values are called *reduction* circuits. The difficulty in designing these reduction circuits can be attributed to the pipelining of the floating point operators, and that rows can have varying lengths. The advantages and drawbacks of various design approaches fall outside the scope of this thesis, for that we refer the reader to [CB:9] and [18].

The work by Gerards et al. [18] introduces a streaming reduction circuit, together with an algorithm to route appropriate inputs to the pipelined operator. The pipelined operator, with α pipeline stages, will be called $\mathcal{P}_\alpha$. In this circuit, values appear sequentially at the input, where each value is a tuple of a floating point number and a row index. Since partially reduced values coming out of the pipeline will have to reduced further, we need to store them in the partial result buffer (denoted by $\mathcal{R}$). When two partially reduced results enter the pipeline, the pipeline cannot simultaneously reduce values entering the system. We hence need to store incoming values in a FIFO buffer which we will denote by $\mathcal{I}$.

To determine whether values from the input buffer, from the end of the pipeline, and/or from the partial result buffer will be used, five rules are checked. The rules can determine which values to use, i.e. the top two values from $\mathcal{I}$ (denoted as i_1 and i_2), the output of the adder pipeline (denoted by ρ) or values from $\mathcal{R}$ (denoted by r). The five rules, in descending order of priority, are:

1. If there is a value available in $\mathcal{R}$ with the same row index as ρ, then these two values enter the pipeline together.
2. If i_1 has the same index as ρ, then i_1 and ρ enter the pipeline.
3. If there are at least two elements in $\mathcal{I}$, and i_1 and i_2 have the same index, then they enter the pipeline.
4. If there are at least two elements in $\mathcal{I}$, but i_1 and i_2 have different indexes, then i_1 enters the pipeline together with the unit element of the operation

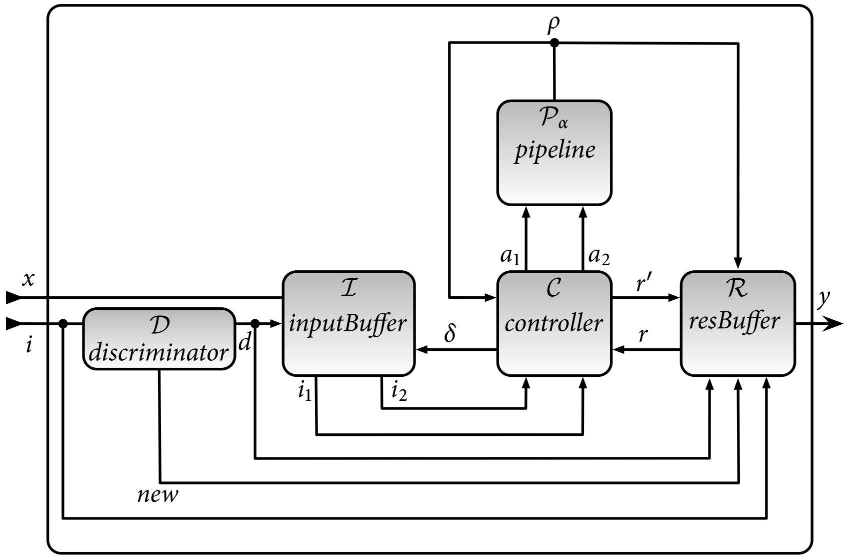


Figure 5.2 – Streaming reduction circuit

dealt with by the pipeline (thus for example, 0 in case of addition, 1 in case of multiplication).

5. In case there are less than two elements available in $\mathcal{I}$, no elements enter the pipeline.

Figure 5.2 shows the entire circuit including control signals. The controller, denoted by $\mathcal{C}$, checks which rule has to be executed. To identify rows within the reduction circuit, discriminators are used as identification. Discriminators are assigned to new rows entering the reduction circuit, and are released when a row is fully reduced and leaves the reduction circuit, after which the discriminator is reused. All elements of a row have an equal discriminator. Discriminators require less bits than row indices, as the number of rows within the reduction circuit is bounded, and thus reduces the amount of memory used by the circuit. The discriminators are assigned by the discriminator component $\mathcal{D}$.

Haskell design

We do not elaborate the complete design of the circuit in Haskell as it adds no value for the narrative, but instead highlight a few aspects. The main data type within the circuit is a *Cell*, which contains both the floating point number and the discriminator. We see the type definition, and some helper functions in listing 5.1. We use a *Maybe* data type so that we can distinguish between *unit* elements created internally (encoded by *Nothing*), and actual data (encoded by *Just*). Cells are compared on

```
newtype Cell = C (Maybe (FPData,Discr))

instance Eq Cell where
  (C (Just (_,d1))) == (C (Just (_,d2))) = d1 == d2
  _                 == _                 = False

valid :: Cell → Bool
valid (C (Just _)) = True
valid _            = True
```

Listing 5.1 – Reduction circuit, data type definitions

```
controller = unbundle . fmap controllerT . bundle

controllerT (i₁, i₂, ρ, r) =
    (a₁, a₂, δ, r')
  where
    (a₁, a₂, δ, r')
      | ρ == r     = (ρ        , r        , 0, C Nothing)
      | ρ == i₁    = (ρ        , i₁       , 1, C Nothing)
      | i₁ == i₂   = (i₁       , i₂       , 2, ρ)
      | valid i₁   = (i₁       , C Nothing , 1, ρ)
      | otherwise  = (C Nothing, C Nothing , 0, ρ)
```

Listing 5.2 – Reduction circuit, controller

their discriminator, so we define the *Cell* data type as a *newtype* wrapper[1] so that we can define a new *Eq* instance.

We can now straightforwardly implement the controller as shown in listing 5.2, where i_1 and i_2 are the two elements from the inputbuffer, ρ is the output of the pipelined operation, and r the output from the partial result buffer. The outputs a_1 and a_2 are the two arguments for the pipelined operator, δ determines how many values should be shifted out of the input FIFO buffer, and r' determines what value should be stored in the partial result buffer.

The controller is the only high-level combinational component in the reduction circuit, the other components are all sequential circuits. We model them using the Mealy machine abstraction discussed in chapter 3. One example is the *discriminator* ($\mathcal{D}$), which hands out new discriminators to the system, based on whether a new row index appears at the input. The code for this circuit is shown in listing 5.3. It

[1]We do not create a completely new data type so that we may use existing functions for *Maybe*.

```
discriminator = discriminatorT ⟨^⟩ (maxBound,maxBound)
  where
    discriminatorT (prevIndex, discr) index = ( (index, discr')
                                              , (newDiscr, discr') )
      where
        newDiscr              = index /= prevIndex
        discr' | newDiscr     = discr + 1
               | otherwise    = discr
```

Listing 5.3 – Reduction circuit, discriminator

```
rc op (x,i) = y
  where
    (new,d)       = discriminator i
    (i₁,i₂)       = inputBuffer   (x,d,δ)
    ρ             = pipeline op   (a₁,a₂)
    (r,y)         = resBuffer     (new,d,i,ρ,r')
    (a₁,a₂,δ,r')  = controller    (i₁,i₂,ρ,r)
```

Listing 5.4 – Reduction circuit

uses the ⟨^⟩ combinator to combine the transfer function and the initial state of the circuit.

We connect all the components to form the complete reduction circuit using the code shown in listing 5.4. The floating point operator *op* is passed as a parameter to the reduction circuit, making the implementation generic for all kinds of pipelined reduction operations. The *pipeline* component (or function) requires a result from the *controller*, while the *controller* requires a result from *pipeline*, i.e. the functions depend on each other's results. In figure 5.2, this is shown using the signals ρ, a_1 and a_2. These same signals are shown in listing 5.4. The result produced by the pipeline (ρ) does, however, not depend on the value produced by the controller (a_1 and a_2) within the same clock cycle. The output ρ is directly derived from the state of the pipeline, and *not* the inputs of the pipeline. Haskell's lazy evaluation will hence ensure that there are no problems with simulating this circuit. As the feedback loop contains delays, the resulting hardware will also exhibit the expected synchronous behaviour.

Table 5.1 displays the design characteristics of both the CλaSH design and a hand-optimized VHDL design where the same global design decisions are applied to both designs. Both designs are synthesized, with the Xilinx tools, for the Xilinx Virtex-4 4VLX160FF1513-10 FPGA. The figures in the table show that the results are comparable. The VHDL design is faster (and bigger) due to a higher degree of pipelining.

Table 5.1 – Design characteristics of the streaming reduction circuit

	CλaSH	VHDL
CLB Slices & LUT	4076	4734
Dffs or Latches	2467	2810
Operating Frequency (MHz)	159	171
Lines of code	211	748

5.3 CλaSH demonstrator circuit

With the intent to show that Haskell, in combination with the CλaSH compiler, can be used as a general purpose HDL, we build a demonstrator circuit for presentation at a large conference [14]. For our FPGA platform we used the Terasic/Altera DE1 Development and Education board, which contains an Altera Cyclone II EP2C20F484C7 FPGA. The circuit is able to:

» Control an audio chip via the I^2C protocol.

» Communicate with an audio chip to acquire audio samples from the line-in, and send audio samples to a speaker over the line-out.

» Read scancodes from a PS\2 keyboard, which are synthesized to an appropriate sine-wave and send to the speakers.

» Mix the audio samples from the line-in and the tones synthesised from the keyboard scancodes.

» Apply low- or high-pass filtering over the mixed audio samples using a 17-tap FIR filter.

» Analyse the spectrum of the filtered audio samples using an fast fourier transform *(FFT)*.

» Display the audio spectrum on a VGA monitor, with time on the horizontal axis, frequency on the vertical, and colours indicating the energy.

A schematic overview of the circuit is presented in figure 5.3. Groups of components run at different clock frequencies, these clock domains are highlighted by colour-accented blocks in the diagram. Most of the components run at the system clock (red) of 50 MHz. As proper timing is important for distortion-free audio, the audio-chip runs in *master* mode, meaning it is in control of all its clocks. The audio interface, mixer, and FIR filter are hence synchronised to bit sample clock (12 MHz) provided by the audio chip (blue). The FFT requires its own clock (green) as the longest combinational path in the FFT design only allowed a clock frequency of 10 MHz. It only runs at 10 MHz because the main FFT datapath is completely combinational, but it is fast enough for processing our audio samples (a sample rate of 44 KHz). So instead of pipelining, running the FFT in its own clock domain was an appropriate solution.

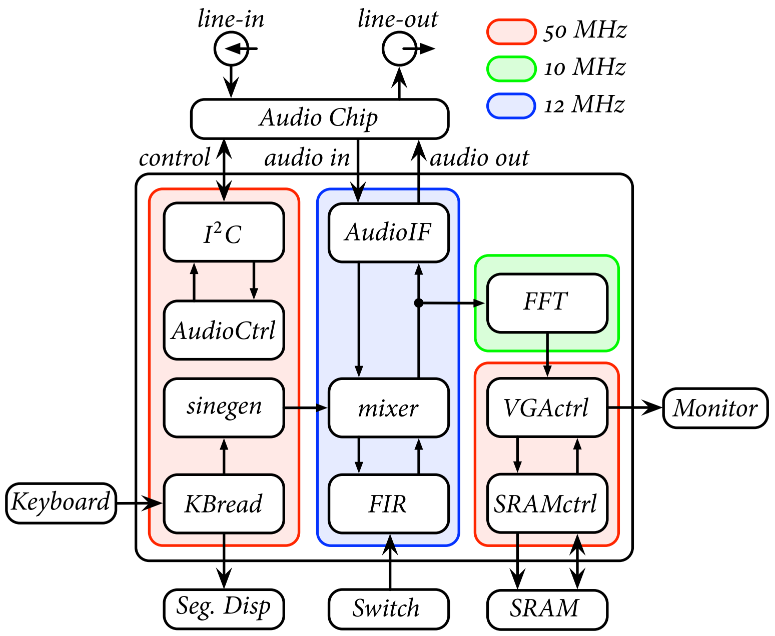


Figure 5.3 – Demonstrator circuit

We will not elaborate the design of every component, but will instead focus on the most control-oriented circuit, the I^2C bus controller. The I^2C protocol is a simple serial bus protocol that allows multiple masters and slaves on a single bus. Although there was only one master, the FPGA, and one slave, the audio chip, we nonetheless designed a complete I^2C controller. We modelled our design of the I^2C controller after an existing design [30] by Richard Herveille written in VHDL. The final design is feature compatible, modulo WISHBONE [50] compatibility, with the design by Richard Herveille.

Such features include:

» True multi-master operation, including collision detection and arbitration, that prevents data corruption when two or more masters try to control the bus.

» Clock stretching, by using the bus clock synchronisation mechanism, slave devices can slow down the transfer rate of the master, thus inserting wait-states.

» Communication of the bus status to a client of the I^2C controller, such as whether there is traffic on the bus, or arbitration is lost.

The I^2C controller is very much a control-oriented circuit, and needs to store many pieces of information. Due to its control-oriented nature, many of its operations are

Monads

Although the name and concept *monad* originate from category theory [40], we will elaborate the concept from a circuit designers point of view. A monad is a structure that expresses computation as a *sequence* of steps. In Haskell this gives rise to the *Monad* type class:

```
class Monad m where
  return :: a → m a
  (>>=)  :: m a → (a → m b) → m b
```

where the *return* function puts a value in the computational structure, and >>= (pronounced *bind*) is the sequential composition of two structures. Using the **do** notation we do not have to explicitly write our programs in terms of >>=, but the compiler will desugare it for us. The program:

```
f x = do
  y <- g x 3
  z <- h x y
  return (z + 1)
```

is desugared to:

```
f x = g x 3 >>= (λy → h x y >>= (λz → return (z + 1)))
```

State monad The *state monad* is a computational structure, where computations are executed in sequence, and a global state is *threaded* through each of these computations. Its type and *Monad* instance are:

```
type State s a = s → (a,s)

instance Monad (State s) where
  return x          = λs → (x,s)
  (State m) >>= k = λs → let (x,s') = m s in (k x) s'
```

where we see that every computation can update the state, and pass the updated state to the next computation. Using the *state monad* operations *get* and *put* we can then describe an incrementing counter as:

```
counter = do
  cnt <- get
  put (cnt + 1)
  return cnt
```

Lenses

Lenses are a recent idiom in Haskell for declaring how to focus deeply into a data structure, whether it be for viewing or updating. In this thesis we use them mainly to conveniently access fields in a record.
Given a record data type:

```
data ShiftRegister = SR { _sr :: BitVector 8, _dcnt :: Index 8 }
```

to represent the state of a shift register in terms of its data, and the number of valid elements. We will use Template Haskell [28] to create two *lenses*, *sr* and *cnt*, to focus into the respective fields of our data structure.
The strength of lenses is, for a large part, defined by the set of combinators to manipulate data in terms of lenses. For example, lets say that we want to decrement our *_dcnt* field within a state monad. Using normal record syntax we would have to write:

```
s@(SR { _dcnt = x}) <- get
put s {_dcnt = x - 1}
```

while with lenses and the −= combinator we can just write:

```
dcnt -= 1
```

which is far more self-explanatory.

not algebraic manipulations of data, but controlling the flow of data through choice structures (e.g. case-expressions). Additionally, in many situations only parts of the state of the circuit need to be updated, while others can be left as is. Sometimes, we also want to assign a certain default value to the state, and only overwrite this state in certain situations. We thus need a way to deal with both selective and *destructive* updating of the complete state of the circuit.

We use the state monad [51] to model a function with a global, updatable state, and use a concept called lenses [37] to perform selective updates. As we will see, these two concepts give an imperative feel to our implementation, which works well for control-heavy circuits. Note that we still use the Mealy machine abstraction to actually create the registers, we use the state monad to describe the combinational part. This means we have full control over where registers will be placed, and hence the cycle delays of our circuit.

In listing 5.5 we see the definition of a shift register, which is used inside the I^2C controller. The shift register is used to both write a word bit-by-bit, and read a word bit-by-bit. On the first line we define a new data type *SR* which describes the state of the shift register: the register content *_sr*, and the number of valid elements

```
 1 data ShiftRegister = SR { _sr :: BitVector 8, _dCnt :: Index 8}
 2
 3 $(makeLenses ''ShiftRegister ) -- generate lenses
 4
 5  shiftRegister :: Bool → Bool → BitVector 8 → Bit
 6                → State ShiftRegister ( BitVector 8,Bool)
 7  shiftRegister ld shift dIn coreRxd = do
 8   -- get the current state
 9   (SR {..}) <- get
10   -- shift register
11   if ld then
12     sr .= dIn
13   else when shift $
14     sr %= (<<# coreRxd)
15   -- data−counter
16   if ld then
17     dCnt .= 7
18   else when shift $
19     dCnt −= 1
20   -- When writing: bit to send, remaning bits , finished writing
21   -- When reading: N/A, word read, finished reading
22   return (msb _sr, _sr,_dCnt == 0)
```

Listing 5.5 – Shift register

in the register *_dcnt*. When the shift register is used for writing values bit-by-bit, *_dcnt* represents the number of bits that still need to be written. When the shift register is used for reading in values bit-by-bit, *_dcnt* represents the number of bits that still need to be read. On line 3 we have the compiler generate, using Template Haskell [28], two *lenses*, *sr* and *dnt*, which can peer into the fields of the *SR* data type. These lenses will allow us to succinctly update only parts of the state of the shift register. We now move on to our monadic implementation of the shift register. On line 9 we get the current state of the shift register, which brings two new variables into scope: *_sr* representing the current register content, and *_dcnt* representing the current data counter. On lines 11 to 14 we update the contents of our register. On line 12 we overwrite the register contents with our data input *dIn*, using the (.=) operator, when the load flag (*ld*) is asserted. On line 14 we modify, using the (%=) operator, the register content by shifting in the *coreRxd* input from the right. On lines 16 to 19 we update the value of our data counter. When the load new value signal (*ld*) is asserted, we reset the counter to 7 (line 17). When we are shifting (the *shift* signal is asserted), we decrement our data counter by 1 using the (−=) operator (line 19).

```
shift   .= False
ld      .= False
hostAck .= False

case _cState of
  Idle  → when go $ do
                    if startCmd then do
                         cState  .= Start
                         coreCmd .= I2C_Start
                    else if readCmd then do
                         cState  .= Read
                         coreCmd .= I2C_Read
                    else if writeCmd then do
                         cState  .= Write
                         coreCmd .= I2C_Write
                    else do
                         cState  .= Stop
                         coreCmd .= I2C_Stop
                    ld .= True
  Write → when coreAck $ do
                     if cntDone then do
                         cState  .= Ack
                         coreCmd .= I2C_Read
                     else do
                         cState  .= Write
                         coreCmd .= I2C_Write
                         shift   .= True
  Read  → when coreAck $ do
```

Listing 5.6 – I^2C controller, excerpt

Other parts of the implementation of the I^2C controller follow the same idiom:

» We first acquire the current value of the state; the individual parts of the current state are referenced by variables starting with an underscore.

» We subsequently update parts of the state using lenses and their associated combinators.

We still follow the Mealy machine idiom of describing the single cycle behaviour of our circuits, but we now use imperative(-looking) features of the Haskell language and its libraries in order to capture the imperative nature of our control-oriented circuits.

Table 5.2 – Design characteristics of the I^2C controller

	CλaSH	**VHDL**
Logic elements	144	194
Registers	66	100
Operating Frequency (MHz)	279	220
Lines of code	515	579

Table 5.3 – Design characteristics of the Cordic core

	CλaSH	**VHDL**
Logic elements	1026	1060
Registers	695	656
Operating Frequency (MHz)	207	209
Lines of code	53	175

In listing 5.6, we can see an excerpt of the code that is used to implement part of the I^2C controller. The purpose of this listing is to demonstrate the destructive update of parts of the state, in this case *ld* and *shift*. These two values are used to control the shift register we discussed earlier. By default, as implemented in line 1 and line 2, we do not want to change the content of the shift register, hence we set *ld* and *shift* to *false*. However, when we are in the *Idle* state and the *go* signal is asserted, we do want to load in a new value into the shift register. Hence we update the *ld* with the value *True* on line 19. Similarly, when we are in the *Write* state, but we have not shifted out all the bits (*cntDone* is not asserted), then we update *shift* with the value *True* (line 27). We will further discuss this imperative way of writing circuit behaviour at the end of this chapter.

Table 5.2 displays the design characteristics of both the CλaSH design and the original VHDL design [30] on which we based our implementation. The figures in the table show that the results are comparable. The reason that the VHDL code is slower is most likely due to the support for both an asynchronous and synchronous reset, where the CλaSH design can only be reset asynchronously. The higher number of registers in the VHDL is most likely due to the one-hot encoding used for the internal state machine and the internal I^2C commands, where CλaSH uses a binary encoding. This also explains the higher number of logic elements, as there are wider (due to one-hot) and more (due to the synchronous reset) multiplexers. The purpose of table 5.2 is, however, to show that the use of imperative features did not negatively impact the performance characteristics of the circuit.

As a final comparison we show the design characteristics of our CORDIC circuit in table 5.3. We use the CORDIC algorithm [67] to calculate the sinusoids for our synthesizer. We compare our design against an existing VHDL implementation [29] that has the same number of pipeline stages, and same bit-precision. Again we see that the performance of the CλaSH implementation and the VHDL implementation are comparable.

5.4 Correct-by-construction compositions

Having seen that the CλaSH compiler can synthesize circuits that have acceptable non-functional properties, we will now show how Haskell can aid us in designing circuits that are correct-by-construction. Specifically, we focus on the correct composition of components. The first subsection describes how we can use higher-order functions to compose functions that use back-pressure as their synchronisation method. The correct-by-construction aspect is that the designer is relieved of connecting the signals for back-pressure, this is done automatically. The second subsection shows how we can annotate components with their cycle delay at the type level. Here the correct-by-construction aspect is that the type system checks that compositions are correctly synchronised, i.e., enough registers have been placed on all the signal paths.

5.4.1 Back pressure

Back-pressure is a synchronisation method between two circuits where:

» Circuit B receives data from circuit A.

» Circuit B has a status line connected to circuit A indicating whether it is ready to receive new data.

» When this status line is de-asserted, circuit A should remember its output until circuit B asserts its ready status again.

When circuit A is feeding data to both circuit B and circuit C, and circuit B de-asserts its ready signal, circuit A should be able to tell circuit C that it cannot provide new data. In general, we will hence also need a forward synchronisation method, by which a circuit can assert that its output is new and/or valid.

We model circuits adhering to this synchronisation protocol using the *DataFlow* type shown in listing 5.7. It has two inputs, one corresponding to its data input, and the second input corresponding to the ready signal of the circuit it is sending data to. It also has two outputs, a data output, and its own ready signal, which will be connected to the circuit it is receiving data from. The validity signals flowing forward, and ready signals flowing backward, have the same type, meaning the granularity of synchronisation matches in both directions.

We can now define sequential composition of two circuits as the $(\ggg)$ operator, shown in listing 5.8. In both the code and the diagram we can see the mutual dependency between the function f and g, where g is expecting data from f, and f will only supply new data when g is ready. The designer must ensure that these signals do not form a combinational loop within f and g. Using the $(\ggg)$ operator, the designer can never accidentally connect the data ports to the ready ports and visa versa when composing two circuits.

Aside from sequential composition, we want other compositional forms, such a parallel composition. We see the type definitions, and their circuit representation,

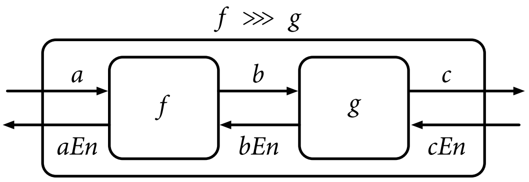
```
newtype DataFlow iEn oEn i o
  = DF
  { df :: Signal (i,iEn)      -- data input, with validity flag
       → Signal oEn           -- receiving circuit ready
       → ( Signal (o,oEn)     -- data output, with validity flag
         , Signal iEn         -- circuit is ready
         )
  }
```

Listing 5.7 – Dataflow circuits

Back-pressure, sequential composition

```
(>>>) :: DataFlow aEn bEn a b → DataFlow bEn cEn b c
      → DataFlow aEn cEn a c
(DF f) >>> (DF g) = DF (λa cEn → let (b,aEn) = f a bEn
                                     (c,bEn) = g b cEn
                                 in  (c,aEn))
```

Circuit representation



Listing 5.8 – Sequential composition of dataflow circuits

for these additional composition operators in listing 5.9. Where the parallel composition operator, (⋆⋆⋆), can actually be defined in terms of *first*, *swap*, and (>>>):

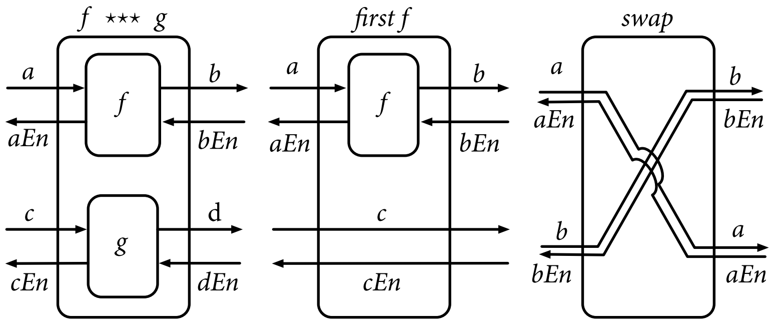
```
second f = swap >>> first f >>> swap
f ⋆⋆⋆ g  = first f >>> second g
```

Here we can also see why we parametrised the synchronisation signals in the definition of our *DataFlow* circuit type, and not fix them to be of type *Bool*. If we had fixed our synchronisation signals to *Bool*, then two circuits composed in parallel would be forced to operate in lock-step, as there would only be one ready and one valid signal to connect to both circuits.

Back-pressure, sequential composition

```
(⋆⋆⋆) :: DataFlow aEn bEn a b → DataFlow cEn dEn c d
         → DataFlow (aEn,cEn) (bEn,dEn) (a,c) (b,d)

first  :: DataFlow aEn bEn a b
         → DataFlow (aEn,cEn) (bEn,cEn) (a,c) (b,c)

swap :: DataFlow (aEn,bEn) (bEn,aEn) (a,b) (b,a)
```

Circuit representation



Listing 5.9 – Parallel composition of back-pressure circuits

Feedback

We define our combinator which adds a feedback arc to a circuit, *loop*, in listing 5.10. We see that our valid and ready flags are no longer parametrised in this circuit, but are fixed to be of type *Bool*. The reason for that is that we want to ensure that data flowing to the output, and data flowing through the feedback loop proceed in lock-step. As we will see below, asserting validity flags and ready flags in the presence of eager consumers is not trivial. By forcing the designer to create a function that operates in lock-step, we can guarantee that the feedback loop operates correctly with respect to our synchronisation protocol.

The implementation of our *loop* combinator is non-trivial, because our circuits are defined as *eager* consumers:

» When the *valid* signal is asserted on its input, then a circuit is expecting a *new* data input every clock cycle.

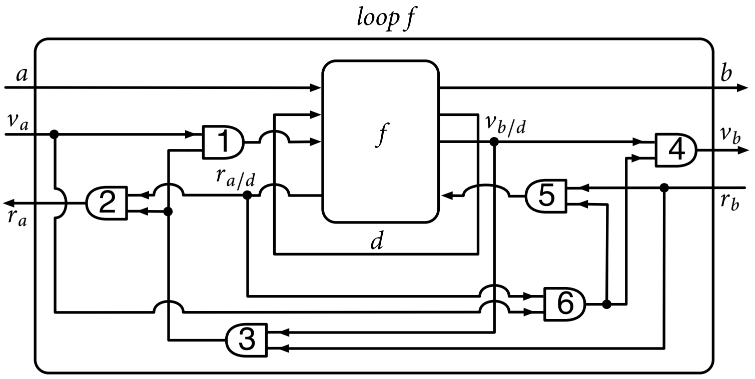
The *loop* combinator must hence ensure that:

» The input circuit and feedback circuit only produce new data when both the feedback circuit and output circuit can consume new values. That means that all data inputs should be valid, and the consuming circuits should be ready.

Feedback loop *combinator*

```
loop :: DataFlow Bool Bool (a,d) (b,d)
     → DataFlow Bool Bool a b
```

Circuit representation



Listing 5.10 – Adding feedback arcs to dataflow circuits

» The feedback circuit and the output circuit only consume new data, when both can consume new data.

We have annotated the and-gates in listing 5.10 with numbers, so we can refer to them in the explanation below.

» The circuit f should only consume new values, when both the incoming data is valid (v_a) and the fed back data is valid (v_d). But also only when the output circuit is ready (r_b), because the function f can only produce *new* values when the output circuit is ready. This logic is implemented by and-gates 1 and 3.

» The circuit on the input, providing data input a, should only produce *new* values (r_a is asserted) when:

- The circuit f is ready.
- When the fed back value is valid (v_d), because f can only consume new values when both v_a and v_d are asserted.
- The circuit on the output is ready (r_b), because f can only consume its fed back value and input value when the output circuit is also ready.

This logic is encoded by and-gates 2 and 3.

» The circuit f should only produce new data (its incoming ready flag is asserted) when:

- The circuit on the output is ready (r_b), and f itself is ready (r_d).

```
loopNI :: DataFlow Bool Bool d (b,d)
       → DataFlow Bool Bool () b
loopNI (DF f) = stub ⋙ loop f'
  where
    f'   = DF (λdin b → let (ud,v) = unbundle din
                            (_,d') = unbundle ud
                            d      = bundle (d',v)
                        in f d b)
    stub = DF (λ_ _ → (pure (True,()), pure True))
```

Listing 5.11 – Alternative feedback compbinator

- The data produced by the circuit on the input is valid (v_a). Because f can only consume the tuple of a and d simultaneously, f should not update its fed back data (d) when the other part of the tuple is not updated.

This logic is encoded by and-gates 5 and 6.

» The circuit on the output, consuming data output b, should only consume values (v_b is asserted) when:

- The output from f is valid (v_b).
- The circuit f is ready to consume its fed back result, f cannot update b and d independently.
- The data on the input is valid (v_a). When the data on the input is not valid, f cannot consume its fed back value, meaning it cannot update its output tuple.

We encode this logic with and-gates 4 and 6.

The *loop* combinator is defined in such a way that it takes a function f that consumes both an external input, and a fed back output. We can, however, imagine a situation where we have a function f that only consumes the fed back input. We can define a new combinator, *loopNI* (loop, no input), in terms of *loop* that accepts such functions. The code for this alternative combinator, *loopNI*, is shown in listing 5.11. The *stub* function will ignore its input, is always *ready*, and produces a constant stream of *unit* values. The final synthesis result will therefore no longer contain *and*-gates 1, 2, and 6. Gate 2 will be removed because its output is unused, and gates 1 and 6 will be removed because *and*ing with *true* is equal to the *and*'s second argument.

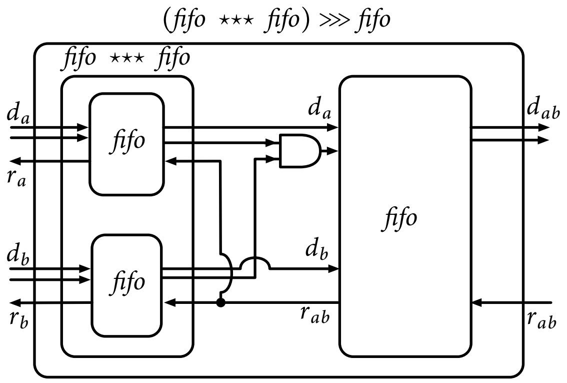
Manipulating status flags

A component that is often used in a back-pressure controlled circuit is a FIFO buffer. Consider a fifo component adhering to our *DataFlow* type:

Parallel and sequential composition

```
main = ( fifo  ⋆⋆⋆  fifo ) ⋙
         (colDF ⋙  fifo )
  where
    colDF = mapDF (uncurry (&&)) dup
    dup a = (a,a)
```

Circuit representation



Listing 5.12 – Composition of FIFO circuits

```
mapDF :: (ra  → rb)
         → (rb → ra)
         → DataFlow ra  rb  a  a
mapDF f g = DF $
  λdin rb → let (d,ra) = unbundle din
                 rb'    = fmap f  ra
                 ra'    = fmap g  rb
             in (bundle' (d,rb') ,ra')
```

Listing 5.13 – Impedence matching circuit

```
fifo  :: DataFlow Bool Bool a a
```

where the fifo will perform a shift when its input is valid, and when the output is ready. The fifo will de-assert its ready signal when it is full, and de-assert the valid signal when it is empty.

We can now compose three FIFOs, two in parallel, composed sequentially with a third, and have their synchronisation signals connected automatically. The code,

```
newtype DSignal (delay :: Nat) a = D { toSignal :: Signal a }
  deriving (Functor, Applicative ,Num)
```

Listing 5.14 – Delay annotated signals

and corresponding circuit interpretation can be seen in listing 5.12. Here we see that we need an additional function, *mapDF*. The reason for that is that we need to correctly match the status signals of two FIFOs in parallel with the third FIFO. The parallel composition is expecting two ready flags to flow backwards, and is offering two valid flags in the forward direction. The third FIFO is, however, only offering one ready flag, and expecting only one valid flag. Our final combinator, *mapDF* (listing 5.13), can thus performs arbitrary manipulation on the status flags, in order for the connections such as the above to match up.

The *mapDF* function is also useful when specifying feedback loops. For example, we can build a derivative of our *loop* combinator, one that always adds a FIFO buffer on the feedback edge:

```
loopFIFO :: DataFlow Bool Bool (a,d) (b,d)
         → DataFlow Bool Bool a b
loopFIFO f = loop (f >>> dupDF >>> second fifo >>> colDF)
  where
    dupDF = mapDF dup (uncurry (&&))
    colDF = mapDF (uncorry (&&)) dup
    dup a = (a,a)
```

where *dubDF* duplicates the valid signals flowing forward, and logically *and*'s the ready signals flowing backwards. The *colDF* operates in the opposite direction, merging the valid signals, and duplicating the ready signals.

5.4.2 Delay annotations

The dataflow method described in the previous section is one approach to make circuit self-synchronising. Such a self-synchronising approach is often used when cycle delays of computations are not fixed, i.e., when iteratively calculating a value until it has a desired property. In other cases, however, different paths through a circuit accumulate distinct delays. When these paths are merged through an operation, it is often desired that these path are synchronised, that is, that both paths have accumulated the same number of delay. In this section we describe how signals can be annotated with their accumulated delay *in their type*, and how composition of functions are only type-correct when the function arguments have the expected amount of accumulated delay.

The data type definition of our annotated signal type is given in listing 5.14. The delay annotation is a *phantom* type, just as the length annotation on our *Vec*tor

```
delay :: Vec d a
      → Delay (n − d) a
      → Delay n a
delay is (D inp) = D outp
  where
    outp = case length is of
             0 → inp
             _ → shiftRegister is inp
```

Listing 5.15 – The delay operator

type. We use natural numbers to encode our delay, meaning that we cannot encode *negative* delays, and hence anti-causal systems. Just as we did for our *Signal* type declaration in chapter 3, the constructor for *DSignal*, *D*, should not be exported. The circuit designer should only modify these annotated signals through a predefined, safe, set of functions and operators.

The first and foremost of these function is the *delay* function, whose definition is given in listing 5.15. This function adds d amount of delay to a signal. The type signature should be read as: given that we are now at time n (the output), we are seeing samples from time $n - d$ clock periods ago (the input). Internally this delay operator is just implemented as a shift register (given that the requested amount of delay is more than zero). For practical purposes we initialize this register with a set of given samples, where the number of initial samples is thus equal to the delay d.

With our most important function in place, we can now show how delay annotations enforce proper synchronisation. Our first example assumes that the type signature for the function is given:

```
-- (+) :: DSignal k Int → DSignal k Int → DSignal k Int
--
-- The following produces a type error:
f :: DSignal (n − 1) Int → DSignal n Int → DSignal n Int
f a b = a + b -- 'a' and 'b' are not synchronised
```

In our example, f expects two integer signals, where the second argument is delayed by 1 cycle compared to the first argument. The above specification would result in a type error, as the addition operator $(+)$ is expecting its arguments to have the same amount of delay. We fix our implementation by also delaying the first argument by 1 cycle:

```
f :: DSignal (n − 1) Int → DSignal n Int → DSignal n Int
f a b = delay [0] a + b -- arguments are now synchronised
```

Also, any function applying f will have to ensure that f's second argument is delayed by 1 cycle compared to its first.

```
unsafeFromSignal :: Proxy n → Signal a → DSignal n a
unsafeFromSignal _ s = D s
```

Listing 5.16 – Unsafe delay annotation

Conversion to Signal

As explained earlier, the constructor for *DSignal*, *D*, will not be exported, so that the circuit designer cannot subvert its invariants *by accident*. Of course, it is imperative that the designer can place a circuit defined in terms of *DSignal*, in a larger, less restricted, circuit defined in terms of *Signal*. The most straightforward operation is stripping the signal from its delay information:

```
toSignal :: DSignal n a → Signal a
```

which is just the *field* extractor defined as part of our newtype definition of *DSignal*.

Dropping delay information is always safe, in terms of the invariants implied by the type of *DSignal*. Adding delay information requires more careful consideration. The safest way to add a delay annotation on a normal signal is to add a delay annotation of *zero* cycles:

```
fromSignal :: Signal a → DSignal 0 a
fromSignal s = D s
```

This new function will allow a designer to coerce any *DSignal* to have zero delay by using: (*fromSignal* . *toSignal*) :: *DSignal n a* → *DSignal* 0 *a*. However, as long as we annotate functions with polymorphic delay values, the risks of subverting the invariants of *DSignal* are reduced.

Let us assume that a circuit designer wants to use our circuit defined earlier, *f*, within the function *g*, where *g* which is defined in terms of *Signal*s. Having only the *fromSignal* function available, the second argument of *g* will have to be explicitly delayed when applied to *f*:

```
g :: Signal Int → Signal Int → Signal Int
g a b = toSignal (f (fromSignal a) (delay [0] (fromSignal b)))
```

However, it might be the case that the designer knows that *g*'s second argument is actually already delayed by one clock cycle compared to *g*'s first argument. In that case, adding an extra delay to the second argument of *f* will actually cause the arguments of *f* to be *improperly* synchronised.

We hence need a third conversion function, given in listing 5.16, that allows the designer to explicitly annotate a signal with its delay. We prefix the function with the word, *unsafe*, to indicate to the designer that this function has to potential to subvert invariants of *DSignal*. Because there is a strict phase distinction between types and terms, we cannot make a function of the type *unsafeFromSignal* :: *n* → *Signal a* → *DSignal n a*, where *n* would be both a type and a term. We therefore use the *Proxy* data type, which, as the name suggests, acts like a *term*-level proxy for the type *n*. Going back to our example, we can now write *g* as:

```
g :: Signal Int → Signal Int → Signal Int
g a b = toSignal (f (fromSignal a)
                    (unsafeFromSignal (Proxy :: Proxy 1) b))
```

Unlike the *fromSignal* function, *unsafeFromSignal* is able to completely subvert the invariants of *DSignal*. That is, we can redefine *f*, erroneously, but type-correct, as:

```
f :: DSignal (n−1) Int → DSignal n Int → DSignal n Int
f a b = ((unsafeFromSignal (Proxy :: Proxy n) . toSignal) a) + b
```

The *unsafeFromSignal* function is hence a truly *unsafe* operation. Both *fromSignal* and *unsafeFromSignal* are, however, needed in order to interact with circuit specifications that are not annotated with delays.

Feedback

With delays encoded in the types, we can no longer define feedback using standard value recursion. For example, using normal *Signal*s, we can describe a MAC circuit:

```
mac x y = acc'
  where
    acc' = (x * y) + acc
    acc  = register 0 acc'
```

where value recursion is used to model the feedback loop. However, when we try to the same using *DSignal* and the *delay* function:

```
mac x y = acc'
  where
    acc' = (x * y) + acc
    acc  = delay [0] acc'
```

we get a type error[2] because *delay* is expecting *acc'* to be of type, *DSignal* (*n*−1) *a*, however, *acc'* is defined in terms of *acc*, which is of type, *DSignal n a*. Actually, the only *type*-correct feedback loops that we can specify with the operators that we have seen until now, are the unsound combinational feedback loops.

[2] Actually, in the polymorphic setting we get a function with an insoluble constraint, $n \sim n - 1$, which we cannot solve once we move to a monomorphic setting.

```
fb :: Proxy (d + 1)
   → (DSignal (n − d − 1) a → (DSignal k b, DSignal n a))
   → DSignal k b
fb _ f = let (y,D x) = f (D x) in y
```

Listing 5.17 – Feedback operator

We thus define a feedback function, listing 5.17, which additionally enforces that the feedback loop contains at least one delay element. For extra verification, we allow the designer to explicitly indicate how much delay the feedback loop must have. The type signature of the *fb* function should thus be read as:

» We want to create a feedback loop which has an internal delay of $d + 1$.

» We give a function f, that has an arbitrary delay between its input and first output, and $d + 1$ delay between its input and second output.

» We feed back the second output of the function to its input, and return its first output.

We can now correctly define our *mac* function as:

```
mac :: DSignal 0 Int → DSignal 0 Int → DSignal 0 Int
mac x y = fb Proxy $
  λacc → let acc' = (x * y) + acc
          in (acc', delay [0] acc')
```

Note that we do not explicitly have to indicate how much delay our feedback loop must have. Providing an unannotated *Proxy* constructor indicates we are content with any kind of delay. The types will, however, enforce that the feedback loop has a delay of at least 1.

5.5 Discussion

Designing circuits in Haskell

In chapter 3 we introduced several primitives for designing synchronous sequential circuits. On top of those primitive we build our Mealy machine abstractions as an initial step to specify our sequential circuits in a principled manner. In our experience, designing data-oriented circuits using *normal* function composition leads to concise and clear circuit specifications. This holds true for both combinational circuits, and sequential circuits defined using our Mealy machine abstraction. In this chapter we have also seen, in the top-level specification of the reduction circuit, that function composition works well for the high-level composition of our components. Once we move to internals of control-oriented circuits, standard function composition does, however, not always *seem* to be the right answer.

In section 5.3 we showed that we defined our I^2C controller in terms of the state monad. We did this because the imperative nature of the state monad allowed us to closely match the original, imperative, VHDL implementation [30]. However, were we to design the I^2C controller from scratch, would we have ended up with such an imperative implementation? Perhaps most of the functionality could have been implemented in a functional style. So we cannot state that control-oriented circuits can only be defined in an imperative style where function composition is unidiomatic.

If there are circuits that are truly imperative in nature, then we do know that we can specify them more idiomatically using a (state) monad. Additionally, this chapter shows that the CλaSH compiler synthesises circuits with acceptable properties from such imperative specifications. This chapter also shows the benefits of using functions from existing Haskell libraries, the state monad and lenses, to define the behaviour of circuits, and thus the advantage of using CλaSH compared to DSLs(e.g. [26] or [58]) embedded in Haskell.

Abstracting patterns

In chapter 2 we saw that existing HDL such as VHDL and Verilog offer features that can abstract certain structural, repetitive, patterns. These patterns correspond to our higher-order functions such as *map* and *foldr*. In this chapter we have seen a glimpse (in the form of the dataflow combinators) of what other kinds of design patterns we can capture with higher-order functions. Such patterns cannot be captured by the abstractions offered in VHDL or Verilog.

At the moment, circuit designers must use our dataflow combinators explicitly when structuring their circuits. Those familiar with Haskell will have noticed that our combinators look very much like the methods of the *Arrow* type class [32]. Arrows have a very convenient notation for their composition [52], sadly this notation cannot be used for our dataflow combinator. The type signatures of the combinators do not match the method signatures of the *Arrow* class. The problem is that we parametrise over the types of the synchronisation signals, which we did to enable parallel composition where both data streams progress independently.

One approach that initially *seems* like a solution is to relate the types of the *ready* flags to types of the data ports using a type function, instead of parametrising them directly. Our *BackPressure* type would then look like the code shown in listing 5.18. However, when we try to implement the very first function of the *Arrow* class we immediately run into problems:

```
instance Arrow DataFlow where
  arr :: (b → c) → BackPressure b c
  arr f = DF (λi oRdy → let (iD, iVal) = unbudle i
                             oVal       = fmap ??? iVal
                         in (bundle (fmap f o, oVal), fmap ??? oRdy))
```

```
type family En a
type instance En (a,b) = (En a, En b)
type instance En Int   = Bool
type instance En Bool  = Bool
...

newtype DataFlow i o
  = DF
  { bp :: Signal (i,En i)
       → Signal (En o)
       → ( Signal (o, En o)
         , Signal (En i)
         )
  }
```

Listing 5.18 – DataFlow type definition using type function

where the problem are the holes in our definition, indicated by ???. Our intention for *arr* is to simply pass the synchronisation flags from our input to our output. Because the flags are of different types, we need two general functions, one with the type *En i* → *En o*, and the other with type *En o* → *En i*. There is, however, no way to create such polymorphic functions; remember that in our *mapDF* function we had to specify these conversion functions explicitly for every situation. A *solution* to our problem would lie in a situation where the *Arrow* type class does not have an *arr* method [45].

Reducing post hoc verification

Using the delay-annotated signals is one of the ways to reduce the verification burden associated with circuit design. Designers can no longer, *by accident*, compose signals that are improperly synchronized. The type system will enforce that the designer is very explicit how delay and feedback are managed in the presence of delay-annotated signals. Sadly, the *unsafeFromSignal* function, needed to connect unannotated signals, can be used to subvert the invariants of our delay annotation. As it is for any unsafe function in the Haskell library (e.g *unsafeCoerce :: a -> b*), *unsafeFromSignal* must hence be used judiciously.

As future work on this issue we are considering to add the *antiDelay* function to our set of functions that can manipulate delay annotation:

```
antiDelay :: Proxy d → DSignal n a → DSignal (n − d) a
```

which brings values from *the future* into the *now*. The *antiDelay* function would allow us to implement functions with the type signature:

```
f :: DSignal (n - 1) a → DSignal n a → DSignal (n - 1) a
```

that actually use their second argument (that is, they do not solely operate combinationally on the first argument). With the combinators that we have now (without *antiDelay*) the delay of the result of a function is always *at least* equal to the largest delay of the function's arguments.

Additionally, the *antiDelay* function would, once again, allow us to implement feedback in terms of value recursion:

```
mac :: DSignal 0 Int → DSignal 0 Int → DSignal 0 Int
mac x y = acc'
  where
    acc' = (x * y) + (antiDelay Proxy acc)
    acc  = delay [0] acc'
```

instead of using an explicit combinator for feedback (*fb*).

At this moment in time we do not include *antiDelay* it in our set of combinators for two reasons:

» We are unsure of having delay-annotated signals in combination delay operators that work in opposite directions is, in practice, any safer than having unannotated signals. For example, our feedback combinator (*fb*) allowed us to enforce that feedback loops always have a delay of at least 1.

» We assume *antiDelay* does not allow us to specify anti-causual behaviour because we use natural numbers to specify delays:

  - The earliest *now*, the result of *antiDelay*, is a signal with zero delay.
  - This means that the *future*, the argument of *antiDelay*, is a signal with a positive delay, which can only be created using the *delay* function.
  - The circuit is hence realisable, where *antiDelay* is simply represented by a wire.

  We have, however, not fully verified this assumption.

6 Conclusions

The goal of this thesis is to *further improve the productivity of circuit designers.* We have discussed that there are no good indicators that allow us to measure productivity quantitatively. So instead, we decided to look qualitatively at hardware description languages. We identified four aspects of a hardware description language that improve productivity:

» The ability to abstract common (repetitive) patterns.

» To be able to express behaviour idiomatically.

» To support correct-by-construction design methods.

» To support a straightforward cost model.

In this thesis we chose to explore the functional programming paradigm for the purpose of circuit description, in particular higher-order functional languages. Because of the close semantic match between pure functions and combinational circuit logic, functional languages allow us to describe highly parallel combinational logic idiomatically. Higher-order functions are a powerful abstraction mechanism that can capture many design patterns. We therefore set out to determine how well functional languages score in the other qualitative measures to improve productivity. This gave rise to the following research questions:

» How can functional languages be used to express both combinational *and* sequential circuits idiomatically?

» How can we support correct-by-construction design methodologies using a functional language?

» How can we use the high-level abstractions without losing on performance, and have a straightforward cost model?

The answers to those questions are given on the next page.

How can functional languages be used to express both combinational *and* sequential circuits idiomatically? Pure functions operating on algebraic data types are a very good model for the combinational logic in circuits because they are semantically very close. Both concepts map to, can be mapped from, the mathematical concept of a function directly. In chapter 3 we saw that pattern matching is a very convenient language feature to express both selection (or choice) and projection. Chapter 3 also shows the very direct mapping of these two concepts in combinational circuit logic in the form of multiplexers for selection, and splitting up bundles of wires for projection. We model synchronous sequential circuits as functions operating on infinite streams of samples, where every sample corresponds to the stable value of the signal during one clock cycle. Simple synchronous circuits are specified in terms of memory primitives, and mapping functions representing combinational circuits over our streams. For more complex sequential circuits designs, we describe solely the combinational part of a Mealy machine, and use a combinator to add the memory element and feedback loop.

How can we use functional languages to support correct-by-construction design methodologies? This thesis demonstrates two features that enable correct-by-construction designs. In chapter 5 we show that higher-order functions, and advanced type annotations, give us compositions of circuits that respect certain invariants. We use higher-order functions to compose circuits with back-pressure as their synchronisation mechanism and guarantee that the synchronisations ports are always correctly connected. We use advanced type annotations to only allow compositions of circuits in such a way that all their arguments have the expected amount of accumulated cycle delay. Additionally, by encoding the clock domains in the type of signals, clock domain crossings are always explicit.

Another aspect of correct-by-construction, is that the synthesis process is correct. This thesis shows a term rewrite system (TRS) that has transformation rules that are proven to be semantic preserving. Additionally, high level abstractions without direct mapping to a circuit are guaranteed to be removed, in a meaning-preserving manner.

How can we use the high-level abstractions without losing on performance, and have straightforward cost model? By viewing a function description as a structural composition of a circuit, every function application is mapped to a component instantiation. The structural view ensures that all the implicit parallelism in the description, is mapped to a circuit with the same degree of parallelism. The size of a circuit is directly derivable from the number of applied functions. Registers are also never inferred, and are only placed where the circuit designer wants the register primitive. The circuit designer thus has direct control over gate propagation delay. From a synthesis point of view, it is thus important to preserve the original function hierarchy when removing abstractions. This is important because the circuit designer expects a very close correspondence between the function hierarchy and the component hierarchy.

6.1 Contributions

The main contributions of this thesis are:

» *A synthesisable model for combinational and sequential circuits in Haskell.* Pure functions operating on algebraic data types describe combinational circuits, while functions operating on infinite streams of values describe sequential circuits. We limit the manipulations on streams to a set of convenient primitives which include: a memory element, mapping combinational functions over streams, and isomorphisms between signals of product types and products of signal types. This set of primitives is highly expressive, but still guarantees that it only leads to realisable sequential circuit descriptions. (Chapter 3)

» *A synthesis method that preserves the function hierarchy.*
Our synthesis method uses a term rewrite system (TRS) that ensures that in the description after transformation: all bound variables, and all arguments, can be represented by a finite number of bits. Consequently, it performs both monomorphisation and defunctionalisation. It performs these operations in a typed context and preserves the original function hierarchy as much as possible. It also ensures that sharing is preserved where possible, meaning the circuit does not become larger than the designer intended. The transformations are proven to be meaning preserving, and we also prove that our TRS always finds a synthesisable normal form given only minor restrictions on the input expression. (Chapter 4)

The accompanying technical contribution is the implementation of the CλaSH compiler, and the Haskell library for circuit design. The CλaSH compiler allows us to synthesize ordinary Haskell functions to a circuit. In chapter 5 we have witnessed the merits of synthesising Haskell code directly, where we can now use the state monad [51] and lenses [37] to create imperative descriptions where such an approach is favourable.

6.2 Recommendations

In chapter 3 we introduced the *Signal* type for modelling sequential circuits. From personal communication with users of CλaSH, we know that they understand the difference between combinational and sequential circuits, but still find the *Signal* type hard to understand. Not so much the *Signal* type in itself, but the combinators of the *Applicative* type class to map combinational circuits over *Signal*s, and the fact that they cannot pattern-match on sum-types. On the short term, future work would lie in adding *idiom brackets* [42] to Haskell, so that we have a more convenient way to lift combinational circuits over *Signal*s. With idiom brackets, instead of writing:

```
multiplyAndAdd <$> x <*> y <*> acc
```

we could write:

```
(| multiplyAndAdd x y  acc  |)
```

On the longer term, future work would lie in integrating the *Signal* type in the language, and devising a synchronous semantics for this extended language, including a synchronous semantics for pattern matching.

The CλaSH compiler can, at the moment, only unfold very rudimentary recursive functions. In order to amortize this deficiency, higher-order functions operating on vectors are currently marked as primitives, for which the compiler has special translation logic. In chapter 4 we discussed that there is a tension between unrolling recursive functions, and preserving the function hierarchy. Not only do we have to inline the recursive function itself, but also functions whose result is used in the subject of the case-decompositions guarding the recursive call. Future work hence lies in finding metrics that determine whether a function should be inlined for the purpose of unrolling bounded recursive descriptions.

In chapter 5 we used type-level functions to reason about the correct delay synchronisation of signals. The feedback combinator enforced that the circuit it was adding a feedback arc to, would have at least 1 cycle of delay. It would be interesting to investigate what other kinds of correct-by-construction invariants we can encode at the type level that are useful in the context of circuit design.

In conclusion, we have seen how to effectively describe circuits in Haskell, how Haskell's high-level features improve the productivity of circuit designers, and how we can map these descriptions to efficient circuits.

A

First Class Patterns in Kansas Lava

```
module Language.KansasLava.Patterns  where

import Language.KansasLava

-- Heterogeneous  Lists
data a  :.  as  = a  :.  as
data Nil        = Nil

-- Pattern: bound variable  + parent  match  asserted
type Pattern  w a  =  Signal  a  →  Signal  Bool  →  (w, Signal  Bool)

-- Variable  pattern
pvar  ::  Rep a  ⇒  Pattern  ( Signal  (Enabled  a)  :.  Nil)  a
pvar  =  λw en  →  (packEnabled  en  w  :.  Nil ,  high)

-- Constant  pattern
pcnst  ::  (Rep a,  Eq a)  ⇒  a  →  Pattern  Nil  a
pcnst  c  =  λw _  →  ( Nil ,w .==.  pureS  c)

-- Wildcard  pattern
pwild  ::  Pattern  Nil  a
pwild  =  λw _  →  ( Nil , high)

--  Alternative :  Pattern  +  Expression
(===>)  ::  Pattern  w a  →  (w →  Signal  b)
        →  Signal  a  →  ( Signal  b,  Signal  Bool)
pat  ===> f  =  λw →  let  (ws,en)  =  pat  w high
                           r        =  f  ws
                      in  (r ,en)
```

```

-- Pattern matching
mATCH :: Rep b ⇒ Signal a → [Signal a → (Signal b, Signal Bool)]
        → Signal b
mATCH _ []     = error "empty case"
mATCH r [f]    = fst (f r)
mATCH r (f:ps) = let (e,en) = f r
                 in mux en ((mATCH r ps),e)
```

Listing A.1 – Language.KansasLava.Patterns

```
module Language.KansasLava.Patterns.Maybe where

import Language.KansasLava
import Language.KansasLava.Patterns
import Data.Default

-- Given:
-- data Maybe a = Just a | Nothing

-- The following encoding is derived:
-- Deconstructors:
deJust :: Rep a ⇒ Signal (Maybe a) → (Signal U1,Signal a)
deJust = unpack . bitwise

deNothing :: Rep a ⇒ Signal (Maybe a) → Signal U1
deNothing = fst . unpack . bitwise

-- Patterns:
pJust :: Rep a ⇒ Pattern w a → Pattern w (Maybe a)
pJust p = λw en → let (con,var)   = deJust w
                      (ws,enVar)  = p var enAll
                      enCon       = (con .==. pureS 0)
                      enPat       = and2 enCon enVar
                      enAll       = and2 enPat en
                  in  (ws,enPat)

pNothing :: Rep a ⇒ Pattern Nil (MaybeP a)
pNothing = λw _ → let con = deNothing w
                  in (Nil, con .==. pureS 1)
```

Listing A.2 – Constructors and Patterns for the *Maybe* data type

```
module Language.KansasLava where

-- Signal/Tuple conversion
unpack          :: Signal (a,b) → (Signal a, Signal b)
-- Convert by bit-representation
bitwise         :: (Rep a,Rep b,W a ~ W b) ⇒ Signal a → Signal b
-- Create a Signal from a constant
pureS           :: Rep a ⇒ a → Signal a

-- Combine data and a Boolean into an enabled Signal
packEnabled     :: Signal Bool → Signal a → Signal (Enabled a)
-- Assert that ordinary signals are always valid
enableS         :: Signal a → Signal (Enabled a)
-- Register that only updates it contents when the input is valid
registerEnabled :: Rep a ⇒ a → Signal (Enabled a) → Signal a

-- Lifted Boolean equality
(.==.) :: (Rep a,Eq a) ⇒ Signal a → Signal a → Signal Bool
-- Always 'True'
high   :: Signal Bool
-- Always 'False'
low    :: Signal Bool
-- Lifted Boolean conjunction
and2   :: Signal Bool → Signal Bool → Signal Bool
-- Multiplex two signals based on the first argument
mux    :: Rep a ⇒ Signal Bool → (Signal a, Signal a) → Signal a
```

Listing A.3 – Language.KansasLava (Simplified)

B

Synchronisation Primitive

```
unsafeSynchroniser
  :: SClock clk1 -- ^ 'Clock' of the incoming signal
  → SClock clk2 -- ^ 'Clock' of the outgoing signal
  → CSignal clk1 a
  → CSignal clk2 a
unsafeSynchroniser (SClock _ clk1) (SClock _ clk2) s = s'
  where
    t1    = fromInteger (snatToInteger clk1)
    t2    = fromInteger (snatToInteger clk2)
    s' | t1 < t2   = compress   t2 t1 s
       | t1 > t2   = oversample t1 t2 s
       | otherwise = same s

same :: CSignal clk1 a → CSignal clk2 a
same (CSignal s) = CSignal s

oversample :: Int → Int → CSignal clk1 a → CSignal clk2 a
oversample high low (CSignal (s :- ss)) = CSignal (s :- oversampleS
    (reverse (repSchedule high low)) ss)

oversampleS :: [Int] → Signal a → Signal a
oversampleS sched = oversample' sched
  where
    oversample' []     s       = oversampleS sched s
    oversample' (d:ds) (s:-ss) = prefixN d s (oversample' ds ss)

    prefixN 0 _ s = s
    prefixN n x s = x :- prefixN (n-1) x s
```

```
compress :: Int → Int → CSignal clk1 a → CSignal clk2 a
compress high low (CSignal s) = CSignal (compressS (repSchedule high
    low) s)

compressS :: [Int] → Signal a → Signal a
compressS sched = compress' sched
  where
    compress' []      s             = compressS sched s
    compress' (d:ds) ss@(s :- _) = s :- compress' ds (dropS d ss)

    dropS 0 s          = s
    dropS n (_ :- ss) = dropS (n-1) ss

repSchedule :: Int → Int → [Int]
repSchedule high low = take low $ repSchedule' low high 1
  where
    repSchedule' cnt th rep
      | cnt < th   = repSchedule' (cnt+low) th (rep + 1)
      | otherwise = rep : repSchedule' (cnt + low) (th + high) 1
```

Listing B.1 – unsafeSynchroniser

C

System FC

This appendix gives an overview of the System FC grammar, its typing rules, its operational semantics, and the proofs for type preservation and progress of the operational semantics. The presented System FC is a small extension of the System FC presented in [69]. Parts of this appendix are thus *verbatim* copies of [69]. Our extensions will be clearly marked.

C.1 Grammar

This section gives an overview of the System FC grammar. Our extensions to the language are: recursive let-expressions, primitive operations, projections, and default patterns for case-decompositions.

H	$::=$		Type constants
	$\mid$	$(\to)$	Arrow
	$\mid$	$\star$	Type/Kind
	$\mid$	T	Type constructor
	$\mid$	K	Promoted data constructor
w	$::=$		Type-level names
	$\mid$	a	Type variables
	$\mid$	F	Type functions
	$\mid$	H	Type constants
σ, τ, κ	$::=$		Types and Kinds
	$\mid$	w	Names
	$\mid$	$\forall a : \kappa.\tau$	Polymorphic types
	$\mid$	$\forall c : \phi.\tau$	Coercion abstr. type
	$\mid$	$\tau_1\ \tau_2$	Type/kind application
	$\mid$	$\tau \rhd \gamma$	Casting
	$\mid$	$\tau\ \gamma$	Coercion application
ϕ	$::=$	$\sigma \sim \tau$	Propositions (coercion kinds)

γ, η	$::=$		Coercions
	$\mid$	c	Variables
	$\mid$	$C\,\overline{\rho}$	Axiom application
	$\mid$	$\langle \tau \rangle$	Reflexivity
	$\mid$	$\mathbf{sym}\ \gamma$	Symmetry
	$\mid$	$\gamma_1 \fatsemi \gamma_2$	Transitivity
	$\mid$	$\forall_{\eta}(a_1, a_2, c).\gamma$	Type/kind abstraction congruence
	$\mid$	$\forall_{(\eta_1,\eta_2)}(c_1, c_2).\gamma$	Coercion abstraction congruence
	$\mid$	$\gamma_1\ \gamma_2$	Type/kind application congruence
	$\mid$	$\gamma(\gamma_2, \gamma_2')$	Coercion application congruence
	$\mid$	$\gamma \rhd \gamma'$	Coherence
	$\mid$	$\gamma@\gamma'$	Type/kind instantiation
	$\mid$	$\gamma@(\gamma_1, \gamma_2)$	Coercion instantiation
	$\mid$	$\mathbf{nth}^{i}\ \gamma$	n^{th} argument projection
	$\mid$	$\mathbf{kind}\ \gamma$	Kind equality extraction
ρ	$::=$	$\tau \mid \gamma$	Type or coercion
e, u	$::=$		Expressions
	$\mid$	x	Variables
	$\mid$	$\lambda x : \tau.e$	Abstraction
	$\mid$	$e_1\ e_2$	Application
	$\mid$	$\Lambda a : \kappa.e$	Type/kind abstraction
	$\mid$	$e\ \tau$	Type/kind application
	$\mid$	$\lambda c : \phi.e$	Coercion abstraction
	$\mid$	$e\ \gamma$	Coercion application
	$\mid$	$e \rhd \gamma$	Casting
	$\mid$	K	Data constructors
	$\mid$	$\mathbf{case}\ e\ \mathbf{of}\ \overline{p \to u}$	Case decomposition
	$\mid$	$\mathbf{let}\ \overline{x : \sigma = e}\ \mathbf{in}\ u$	Recursion let-expression
	$\mid$	$\otimes$	Primitive operation
	$\mid$	$\pi_i^k\ e$	Constructor field projection
p	$::=$		Patterns
	$\mid$	$K\ \Delta\ \overline{x : \tau}$	Constructor pattern
	$\mid$	$\_$	Default pattern
Δ	$::=$		Telescopes
	$\mid$	$\varnothing$	Empty
	$\mid$	$\Delta, a : \kappa$	Type variable binding
	$\mid$	$\Delta, c : \phi$	Coercions variable binding

C.2 Type system

A context Γ is a list of assumptions for term variables (x), primitives ($\otimes$), type variables/data types/data constructors (w), coercion variables (c), and coercion axioms (C).

$$\Gamma \ ::= \ \varnothing \mid \Gamma,\ x : \tau \mid \Gamma, \otimes : \tau \mid \Gamma,\ w : \kappa \mid \Gamma,\ c : \phi \mid \Gamma,\ C : \forall\Delta.\phi$$

These rules ensure that all assumptions in the context are well formed and unique (indicated by #). They additionally constrain the form of the kinds of data types, and the types of data constructors. Our addition is a well-formedness check on the types of primitives (GWF_Prim).

$\boxed{\vdash_{\text{wf}} \Gamma}$ **Context well-formedness**

$$\text{GWF\_Empty}\ \frac{}{\vdash_{\text{wf}} \varnothing} \qquad \text{GWF\_TyVar}\ \frac{\Gamma \vdash_{\text{ty}} \kappa : \star \qquad a \# \Gamma}{\vdash_{\text{wf}} \Gamma,\ a : \kappa}$$

$$\text{GWF\_TyFun}\ \frac{\Gamma \vdash_{\text{ty}} \kappa : \star \qquad F \# \Gamma}{\vdash_{\text{wf}} \Gamma,\ F : \kappa}$$

$$\text{GWF\_TyData}\ \frac{\Gamma \vdash_{\text{ty}} \forall\overline{a : \kappa}.\star : \star \qquad T \# \Gamma}{\vdash_{\text{wf}} \Gamma,\ T : \forall\overline{a : \kappa}.\star}$$

$$\text{GWF\_Var}\ \frac{\Gamma \vdash_{\text{ty}} \tau : \kappa \qquad x \# \Gamma}{\vdash_{\text{wf}} \Gamma,\ x : \tau} \qquad \text{GWF\_Prim}\ \frac{\Gamma \vdash_{\text{ty}} \overline{T} \to T : \star}{\vdash_{\text{wf}} \Gamma, \otimes : \overline{T} \to T}$$

$$\text{GWF\_Con}\ \frac{\Gamma \vdash_{\text{ty}} \forall\overline{a : \kappa}.\forall\Delta.(\overline{\sigma} \to T\ \overline{a}) : \star \qquad K \# \Gamma}{\vdash_{\text{wf}} \Gamma,\ K : \forall\overline{a : \kappa}.\forall\Delta.(\overline{\sigma} \to T\ \overline{a})}$$

$$\text{GWF\_CVar}\ \frac{\Gamma \vdash_{\text{pr}} \phi\ \mathsf{ok} \qquad c \# \Gamma}{\vdash_{\text{wf}} \Gamma, c : \phi} \qquad \text{GWF\_Ax}\ \frac{\Gamma, \Delta \vdash_{\text{pr}} \phi\ \mathsf{ok} \qquad C \# \Gamma}{\vdash_{\text{wf}} \Gamma, C : \forall\Delta.\phi}$$

The next rules ensure correct kinding of types. There are no additions to these rules.

$\boxed{\Gamma \vdash_{\text{ty}} \tau : \kappa}$ **Type/Kind formation**

$$\text{K\_Var}\ \frac{\vdash_{\text{wf}} \Gamma \qquad w : \kappa \in \Gamma}{\Gamma \vdash_{\text{ty}} w : \kappa} \qquad \text{K\_Arrow}\ \frac{\vdash_{\text{wf}} \Gamma}{\Gamma \vdash_{\text{ty}} (\to) : \star \to \star \to \star}$$

$$\text{K\_AllT}\ \frac{\Gamma,\ a : \kappa \vdash_{\text{ty}} \tau : \star \qquad \Gamma \vdash_{\text{ty}} \kappa : \star}{\Gamma \vdash_{\text{ty}} \forall a : \kappa.\tau : \star}$$

$$\text{K\_App}\ \frac{\Gamma \vdash_{\text{ty}} \tau_1 : \kappa_1 \to \kappa_2 \qquad \Gamma \vdash_{\text{ty}} \tau_2 : \kappa_1}{\Gamma \vdash_{\text{ty}} \tau_1\ \tau_2 : \kappa_2}$$

$$\text{K\_Inst}\ \frac{\Gamma \vdash_{\text{ty}} \tau_1 : \forall a : \kappa_1 \to \kappa_2 \qquad \Gamma \vdash_{\text{ty}} \tau_2 : \kappa_1}{\Gamma \vdash_{\text{ty}} \tau_1\ \tau_2 : \kappa_2[\tau_2/a]}$$

$$\textsc{K\_StateInStar}\ \frac{\vdash_{\mathrm{wf}} \Gamma}{\Gamma \vdash_{\mathrm{ty}} \star \ : \ \star} \qquad \textsc{K\_AllC}\ \frac{\Gamma,\ c : \phi \vdash_{\mathrm{ty}} \tau \ : \ \star \qquad \Gamma \vdash_{\mathrm{pr}} \phi \ \mathsf{ok}}{\Gamma \vdash_{\mathrm{ty}} \forall a : \kappa . \tau \ : \ \star}$$

$$\textsc{K\_Cast}\ \frac{\Gamma \vdash_{\mathrm{ty}} \tau \ : \ \kappa_1 \qquad \Gamma \vdash_{\mathrm{co}} \eta \ : \kappa_1 \sim \kappa_2 \qquad \Gamma \vdash_{\mathrm{ty}} \kappa_2 \ : \ \star}{\tau \rhd \eta \ : \ \kappa_2}$$

The rule ensures correct formation of propositions.

$\boxed{\Gamma \vdash_{\mathrm{pr}} \phi \ \mathsf{ok}}$ **Proposition validity**

$$\textsc{Prop\_Equality}\ \frac{\begin{array}{c}\Gamma \vdash_{\mathrm{ty}} \sigma_1 \ : \ \kappa_1 \\ \Gamma \vdash_{\mathrm{ty}} \sigma_2 \ : \ \kappa_2\end{array}}{\Gamma \vdash_{\mathrm{pr}} \sigma_1 \sim \sigma_2 \ \mathsf{ok}}$$

Next follow the rules for coercions; again, there are no additions.

$\boxed{\Gamma \vdash_{\mathrm{co}} \gamma \ : \ \phi}$ **Coercion typing**

$$\textsc{CT\_Refl}\ \frac{\Gamma \vdash_{\mathrm{ty}} \tau \ : \ \kappa}{\Gamma \vdash_{\mathrm{co}} \langle \tau \rangle \ : \ \tau \sim \tau} \qquad \textsc{CT\_Sym}\ \frac{\Gamma \vdash_{\mathrm{co}} \gamma \ : \ \tau_1 \sim \tau_2}{\Gamma \vdash_{\mathrm{co}} \mathbf{sym}\, \gamma \ : \ \tau_2 \sim \tau_1}$$

$$\textsc{CT\_Trans}\ \frac{\Gamma \vdash_{\mathrm{co}} \gamma_1 \ : \ \tau_1 \sim \tau_2 \qquad \Gamma \vdash_{\mathrm{co}} \gamma_2 \ : \ \tau_2 \sim \tau_3}{\Gamma \vdash_{\mathrm{co}} \gamma_1 \fatsemi \gamma_2 \ : \ \tau_1 \sim \tau_3}$$

$$\textsc{CT\_App}\ \frac{\begin{array}{cc}\Gamma \vdash_{\mathrm{co}} \gamma_1 \ : \ \tau_1' \sim \tau_2' & \Gamma \vdash_{\mathrm{co}} \gamma_2 \ : \ \tau_1 \sim \tau_1 \\ \Gamma \vdash_{\mathrm{ty}} \tau_1'\ \tau_1 \ : \ \kappa_1 & \Gamma \vdash_{\mathrm{ty}} \tau_2'\ \tau_2 \ : \ \kappa_2\end{array}}{\Gamma \vdash_{\mathrm{co}} \gamma_1\ \gamma_2 \ : \ \tau_1'\ \tau_1 \sim \tau_2'\ \tau_2}$$

$$\textsc{CT\_CApp}\ \frac{\begin{array}{c}\Gamma \vdash_{\mathrm{co}} \gamma_1 \ : \ \tau_1 \sim \tau_1' \\ \Gamma \vdash_{\mathrm{ty}} \tau_1\ \gamma_2 \ : \ \kappa \qquad \Gamma \vdash_{\mathrm{ty}} \tau_1'\ \gamma_2' \ : \ \kappa'\end{array}}{\Gamma \vdash_{\mathrm{co}} \gamma_1(\gamma_2, \gamma_2') : \ \tau_1\ \gamma_2 \sim \tau_1'\ \gamma_2'}$$

$$\textsc{CT\_AllT}\ \frac{\begin{array}{c}\Gamma \vdash_{\mathrm{co}} \eta \ : \kappa_1 \sim \kappa_2 \\ \Gamma, a_1 : \kappa_1, a_2 : \kappa_2, c : a_1 \sim a_2 \vdash_{\mathrm{co}} \gamma \ : \ \tau_1 \sim \tau_2 \\ \Gamma \vdash_{\mathrm{ty}} \forall a_1 : \kappa_1 . \tau_1 \ : \ \star \qquad \Gamma \vdash_{\mathrm{ty}} \forall a_2 : \kappa_2 . \tau_2 \ : \ \star\end{array}}{\Gamma \vdash_{\mathrm{co}} \forall_{\eta}(a_1, a_2, c).\gamma \ : \ (\forall a_1 : \kappa_1 . \tau_1) \sim (\forall a_2 : \kappa_2 . \tau_2)}$$

$$\textsc{CT\_AllC}\ \frac{\begin{array}{c}\begin{array}{cc}\Gamma \vdash_{\mathrm{co}} \eta_1 \ : \ \sigma_1 \sim \sigma_1' & \phi_1 = \sigma_1 \sim \sigma_2 \\ \Gamma \vdash_{\mathrm{co}} \eta_2 \ : \ \sigma_2 \sim \sigma_2' & \phi_2 = \sigma_1' \sim \sigma_2' \\ c_1 \ \#\ |\gamma| & c_2 \ \#\ |\gamma|\end{array} \\ \Gamma, c_1 : \phi_1, c_2 : \phi_2 \vdash_{\mathrm{co}} \gamma \ : \ \tau_1 \sim \tau_2 \\ \Gamma \vdash_{\mathrm{ty}} \forall a_1 : \kappa_1 : \tau_1 \ : \ \star \qquad \Gamma \vdash_{\mathrm{ty}} \forall a_2 : \kappa_2 . \tau_2 \ : \ \star\end{array}}{\Gamma \vdash_{\mathrm{co}} \forall_{(\eta_1, \eta_2)}(c_1, c_2).\gamma \ : \ (\forall c_1 : \phi_1 . \tau_1) \sim (\forall c_2 : \phi_2 . \tau_2)}$$

$$\text{CT\_Coh}\ \frac{\Gamma \vdash_{\text{co}} \gamma : \tau_1 \sim \tau_2 \qquad \Gamma \vdash_{\text{ty}} \tau_1 \rhd \gamma' : \kappa}{\Gamma \vdash_{\text{co}} \gamma \rhd \gamma' : \tau_1 \rhd \gamma' \sim \tau_2}$$

$$\text{CT\_Var}\ \frac{c : \phi \in \Gamma \qquad \vdash_{\text{wf}} \Gamma}{\Gamma \vdash_{\text{co}} c : \phi}$$

$$\text{CT\_Axiom}\ \frac{C : \forall \Delta.(\tau_1 \sim \tau_2) \in \Gamma \qquad \Gamma \vdash_{\text{tel}} \overline{\rho} \Leftarrow \Delta}{\Gamma \vdash_{\text{co}} C\,\overline{\rho} : \tau_1[\overline{\rho}/\Delta] \sim \tau_2[\overline{\rho}/\Delta]}$$

$$\text{CT\_Nth}\ \frac{\begin{array}{c}\Gamma \vdash_{\text{co}} \gamma : H\,\overline{\rho} \sim H\,\overline{\rho'} \\ \rho_i = \tau \qquad \rho'_i = \tau'\end{array}}{\Gamma \vdash_{\text{co}} \mathbf{nth}^i\ \gamma : \tau \sim \tau'}$$

$$\text{CT\_Nth1TA}\ \frac{\Gamma \vdash_{\text{co}} \gamma_1 : (\forall a_1 : \kappa_1.\tau_1) \sim (\forall a_2 : \kappa_2.\tau_2)}{\Gamma \vdash_{\text{co}} \mathbf{nth}^1\ \gamma_1 : \kappa_1 \sim \kappa_2}$$

$$\text{CT\_Inst}\ \frac{\begin{array}{c}\Gamma \vdash_{\text{co}} \gamma_1 : (\forall a_1 : \kappa_1.\tau_1) \sim (\forall a_2 : \kappa_2.\tau_2) \\ \Gamma \vdash_{\text{co}} \gamma_2 : \sigma_1 \sim \sigma_2 \\ \Gamma \vdash_{\text{ty}} \sigma_1 : \kappa_1 \qquad \Gamma \vdash_{\text{ty}} \sigma_2 : \kappa_2\end{array}}{\Gamma \vdash_{\text{co}} \gamma_1 @ \gamma_2 : \tau_1[\sigma_1/a_1] \sim \tau_2[\sigma_2/a_2]}$$

$$\text{CT\_Nth1CA}\ \frac{\Gamma \vdash_{\text{co}} \gamma : (\forall c : \kappa_1 \sim \kappa_2.\tau) \sim (\forall c' : \kappa'_1 \sim \kappa'_2.\tau')}{\Gamma \vdash_{\text{co}} \mathbf{nth}^1\ \gamma : \kappa_1 \sim \kappa'_1}$$

$$\text{CT\_Nth2CA}\ \frac{\Gamma \vdash_{\text{co}} \gamma : (\forall c : \kappa_1 \sim \kappa_2.\tau) \sim (\forall c' : \kappa'_1 \sim \kappa'_2.\tau')}{\Gamma \vdash_{\text{co}} \mathbf{nth}^2\ \gamma : \kappa_2 \sim \kappa'_2}$$

$$\text{CT\_InstC}\ \frac{\begin{array}{c}\Gamma \vdash_{\text{co}} \gamma : (\forall c_1 : \phi_1.\tau_1) \sim (\forall c_2 : \phi_2.\tau_2) \\ \Gamma \vdash_{\text{co}} \gamma : \phi_1 \qquad \Gamma \vdash_{\text{co}} \gamma_2 : \phi_2\end{array}}{\Gamma \vdash_{\text{co}} \gamma @ (\gamma_1, \gamma_2) : \tau_1[\gamma_1/c_1] \sim \tau_2[\gamma_2/c_2]}$$

$$\text{CT\_Ext}\ \frac{\Gamma \vdash_{\text{co}} \gamma : \tau_1 \sim \tau_2 \qquad \Gamma \vdash_{\text{ty}} \tau_1 : \kappa_1 \qquad \Gamma \vdash_{\text{ty}} \tau_2 : \kappa_2}{\Gamma \vdash_{\text{co}} \mathbf{kind}\,\gamma : \kappa_1 \sim \kappa_2}$$

The definitions for the erasure operation, $|\gamma|$, and the telescope argument validity judgement, $\Gamma \vdash_{\text{tel}} \overline{\rho} \Leftarrow \Delta$, can be found in [69].

The next set of rules determine whether an expressions is correctly typed. Our additions are the typing rules for let-expressions (T_LetRec), primitives (T_Prim), projections (T_Proj), and default patterns for case-decompositions (T_AltDef).

$\boxed{\Gamma \vdash_{\mathrm{tm}} e \ : \ \tau}$ **Expression typing**

$$\textsc{T\_Var}\ \dfrac{\vdash_{\mathrm{wf}} \Gamma \qquad x : \tau \in \Gamma}{\Gamma \vdash_{\mathrm{tm}} x \ : \ \tau} \qquad\qquad \textsc{T\_Abs}\ \dfrac{\Gamma,\ x : \tau_1 \vdash_{\mathrm{tm}} e \ : \ \tau_2}{\Gamma \vdash_{\mathrm{tm}} \lambda x : \tau . e \ : \ \tau_1 \to \tau_2}$$

$$\textsc{T\_App}\ \dfrac{\Gamma \vdash_{\mathrm{tm}} e \ : \ \tau_1 \to \tau_2 \qquad \Gamma \vdash_{\mathrm{tm}} u \ : \ \tau_1}{\Gamma \vdash_{\mathrm{tm}} e\, u \ : \ \tau_2}$$

$$\textsc{T\_CAbs}\ \dfrac{\Gamma,\ c : \phi \vdash_{\mathrm{tm}} e \ : \ \tau \qquad \Gamma \vdash_{\mathrm{pr}} \phi\ \mathsf{ok}}{\Gamma \vdash_{\mathrm{tm}} \lambda c : \phi . e \ : \ \forall c : \phi . \tau} \qquad\qquad \textsc{T\_CApp}\ \dfrac{\begin{array}{c}\Gamma \vdash_{\mathrm{tm}} e \ : \ \forall c : \phi . \tau \\ \Gamma \vdash_{\mathrm{co}} \gamma \ : \ \phi\end{array}}{\Gamma \vdash_{\mathrm{tm}} e\, \gamma \ : \ \tau[\gamma / c]}$$

$$\textsc{T\_TAbs}\ \dfrac{\Gamma,\ a : \kappa \vdash_{\mathrm{tm}} e \ : \ \tau}{\Gamma \vdash_{\mathrm{tm}} \Lambda a : \kappa . e \ : \ \forall a : \kappa . \tau}$$

$$\textsc{T\_TApp}\ \dfrac{\Gamma \vdash_{\mathrm{tm}} e \ : \ \forall a : \kappa . \tau \qquad \Gamma \vdash_{\mathrm{ty}} \tau' \ : \ \kappa}{\Gamma \vdash_{\mathrm{tm}} e\, \tau \ : \ \tau[\tau' / a]}$$

$$\textsc{T\_Cast}\ \dfrac{\Gamma \vdash_{\mathrm{tm}} e \ : \ \tau_1 \qquad \Gamma \vdash_{\mathrm{co}} \gamma \ : \tau_1 \sim \tau_2 \qquad \Gamma \vdash_{\mathrm{ty}} \tau_2 \ : \ \star}{e \rhd \gamma \ : \ \tau_2}$$

$$\textsc{T\_Con}\ \dfrac{\vdash_{\mathrm{wf}} \Gamma \qquad K : \tau \in \Gamma}{\Gamma \vdash_{tm} \otimes \ : \ \tau}$$

$$\textsc{T\_Case}\ \dfrac{\Gamma \vdash_{\mathrm{tm}} e \ : \ T\, \overline{\tau'} \qquad \overline{\Gamma \vdash_{\mathrm{alt}} p_i \to u_i \ : \ T\, \overline{\tau'} \to \tau}}{\Gamma \vdash_{\mathrm{tm}} \mathbf{case}\ e\ \mathbf{of}\ \overline{p \to u} \ : \ \tau}$$

$$\textsc{T\_LetRec}\ \dfrac{\overline{\Gamma, \overline{x : \sigma} \vdash_{\mathrm{bind}} x_i : \sigma_i \leftarrow e_i} \qquad \Gamma, \overline{x : \sigma} \vdash_{\mathrm{tm}} u \ : \ \tau}{\Gamma \vdash_{\mathrm{tm}} \mathbf{let}\ \overline{x : \sigma = e}\ \mathbf{in}\ u \ : \ \tau}$$

$$\textsc{T\_Prim}\ \dfrac{\vdash_{\mathrm{wf}} \Gamma \qquad \otimes : \tau \in \Gamma}{\Gamma \vdash_{tm} \otimes \ : \ \tau}$$

$$\textsc{T\_Proj}\ \dfrac{\Gamma \vdash_{\mathrm{tm}} e \ : \ T\, \overline{\tau'} \qquad \Gamma \vdash_{\mathrm{alt}} K_k\, \Delta\, \overline{x : \tau'} \to x_i \ : \ T\, \overline{\tau'} \to \tau}{\Gamma \vdash_{\mathrm{tm}} \pi_i^k\, e \ : \ \tau}$$

$\boxed{\Gamma \vdash_{\mathrm{alt}} p \to e \ : \ \sigma \to \tau}$ **Alternative typing**

$$\textsc{T\_AltDef}\ \dfrac{\Gamma \vdash_{\mathrm{tm}} e \ : \ \tau \qquad \Gamma \vdash_{\mathrm{ty}} \tau \ : \ \star \qquad \Gamma \vdash_{\mathrm{ty}} \sigma \ : \ \star}{\Gamma \vdash_{\mathrm{alt}} \_ \to e \ : \ \sigma \to \tau}$$

$$\textrm{T\_A{\scriptsize LT}C{\scriptsize ON}}\ \frac{\begin{array}{l}\Gamma \vdash_{\mathrm{ty}} \tau \ : \ \star \\ K : \forall \overline{a:\kappa}.\forall \Delta.\overline{\sigma} \to (T\,\overline{a}) \in \Gamma \\ \Delta' = \Delta[\overline{\tau'/a}] \\ \overline{\sigma' = \sigma[\overline{\tau'/a}]} \\ \Gamma, \Delta', \overline{x:\sigma'} \vdash_{\mathrm{tm}} u \ : \ \tau \end{array}}{\Gamma \vdash_{\mathrm{alt}} K\,\Delta'\,\overline{x:\sigma'} \to u \ : \ T\,\overline{\tau'} \to \tau}$$

$\boxed{\Gamma \vdash_{\mathrm{bind}} x : \sigma \leftarrow e}$ **Binding typing**

$$\textrm{T\_B{\scriptsize IND}}\ \frac{\Gamma \vdash_{\mathrm{tm}} e \ : \ \sigma}{\Gamma \vdash_{\mathrm{bind}} x : \sigma \leftarrow e}$$

C.3 Operational semantics

To support recursion of let-expressions, we use an extra context Σ_{let} that keeps track of the bindings in the recursive group. Our additions are those step reduction rules involving let-expressions, primitives, and projections.

$\boxed{\Sigma_{\mathrm{let}}; e \ \longrightarrow e'}$ Step reduction, parametrised by top-level context Γ

$\Sigma_{\mathrm{let}} ::= \varnothing \ \mid \ \Sigma_{\mathrm{let}}, x \mapsto e$

$$\textrm{S\_B{\scriptsize ETA}}\ \frac{}{\Sigma_{\mathrm{let}}; (\lambda x : \tau.e)\ e' \longrightarrow e[e'/x]} \qquad \textrm{S\_EA{\scriptsize PP}}\ \frac{\Sigma_{\mathrm{let}}; e_1 \longrightarrow e_1'}{\Sigma_{\mathrm{let}}; e_1\ e_2 \longrightarrow e_1'\ e_2}$$

$$\textrm{S\_P{\scriptsize USH}}\ \frac{\Gamma \vdash_{\mathrm{co}} \gamma : \sigma_1 \to \sigma_2 \sim \tau_1 \to \tau_2}{\Sigma_{\mathrm{let}}; (v \rhd \gamma)\ e \longrightarrow (v\ (e \rhd \mathbf{sym}(\mathbf{nth}^1 \gamma))) \rhd \mathbf{nth}^2 \gamma}$$

$$\textrm{S\_TB{\scriptsize ETA}}\ \frac{}{\Sigma_{\mathrm{let}}; (\Lambda a : \kappa.e)\ \tau \longrightarrow e[\tau/a]} \qquad \textrm{S\_TA{\scriptsize PP}}\ \frac{\Sigma_{\mathrm{let}}; e_1 \longrightarrow e_1'}{\Sigma_{\mathrm{let}}; e_1\ \sigma \longrightarrow e_1'\ \sigma}$$

$$\textrm{S\_TP{\scriptsize USH}}\ \frac{\begin{array}{l}\Gamma \vdash_{\mathrm{co}} \gamma : \forall a : \kappa_1.\sigma_1 \sim \forall a : \kappa_2.\sigma_2 \\ \gamma' = \mathbf{sym}(\mathbf{nth}^1 \gamma) \\ \tau' = \tau \ \rhd \gamma' \end{array}}{\Sigma_{\mathrm{let}}; (v \ \rhd \gamma)\ \tau \longrightarrow (v\ \tau') \rhd \gamma @(\langle \tau \rangle \rhd \gamma')}$$

$$\textrm{S\_CB{\scriptsize ETA}}\ \frac{}{\Sigma_{\mathrm{let}}; (\lambda c : \sigma_1 \sim \sigma_2.e)\ \gamma \longrightarrow e[\gamma/c]} \qquad \textrm{S\_CA{\scriptsize PP}}\ \frac{\Sigma_{\mathrm{let}}; e_1 \longrightarrow e_1'}{\Sigma_{\mathrm{let}}; e_1\ \gamma \longrightarrow e_1'\ \gamma}$$

$$\textrm{S\_CP{\scriptsize USH}}\ \frac{\begin{array}{l}\Gamma \vdash_{\mathrm{co}} \gamma : \forall c : \phi.\tau \sim \forall c' : \phi'.\tau' \\ \gamma'' = \mathbf{nth}^1 \gamma \mathbin{;} \gamma' \mathbin{;} \mathbf{sym}(\mathbf{nth}^2 \gamma) \end{array}}{\Sigma_{\mathrm{let}}; (v \rhd \gamma)\ \gamma' \longrightarrow (v\ \gamma'') \rhd \gamma @(\gamma'', \gamma')}$$

$$\textsc{S\_Comb}\ \frac{}{\Sigma_{\text{let}}; (v \rhd \gamma_1) \rhd \gamma_2 \longrightarrow v \rhd (\gamma_1 \fatsemi \gamma_2)}$$

$$\textsc{S\_Coerce}\ \frac{\Sigma_{\text{let}}; e \longrightarrow e'}{\Sigma_{\text{let}}; e \rhd \gamma \longrightarrow e' \rhd \gamma}$$

$$\textsc{S\_CaseMatch}\ \frac{K_i\ \Delta_i\ \overline{x_i : \sigma_i} \to u_i \in \overline{p \to u}}{\Sigma_{\text{let}}; \textbf{case}\ K_i\ \overline{\tau}\ \overline{\rho}\ \overline{e}\ \textbf{of}\ \overline{p \to u} \longrightarrow u_i[\overline{e/x_i}][\overline{\rho}/\Delta_i]}$$

$$\textsc{S\_CaseDef}\ \frac{\_ \to u_i \in \overline{p \to u} \qquad \text{No other matches}}{\Sigma_{\text{let}}; \textbf{case}\ K_i\ \overline{\tau}\ \overline{\rho}\ \overline{e}\ \textbf{of}\ \overline{p \to u} \longrightarrow u_i}$$

$$\textsc{S\_Case}\ \frac{\Sigma_{\text{let}}; e \longrightarrow e'}{\Sigma_{\text{let}}; \textbf{case}\ e\ \textbf{of}\ \overline{p \to u} \longrightarrow \textbf{case}\ e'\ \textbf{of}\ \overline{p \to u}}$$

$$\textsc{S\_ProjKPush}\ \frac{\begin{array}{l} K : \forall \overline{a : \kappa}.\forall \Delta.\overline{\sigma} \to (T\ \overline{a}) \in \Gamma \\ \Psi = \mathsf{extend}(\mathsf{context}(\gamma); \overline{\rho}; \Delta) \\ \overline{\tau'} = \Psi_2(\overline{a}) \\ \overline{\rho}' = \Psi_2(\mathit{dom}\ \Delta) \\ \text{for each } e_i \in \overline{e} \\ \qquad e'_i = e_i \rhd \Psi(\sigma_i) \end{array}}{\Sigma_{\text{let}}; \textbf{case}\ (K_i\ \overline{\tau}\ \overline{\rho}\ \overline{e}) \rhd \gamma\ \textbf{of}\ \overline{p \to u} \longrightarrow \textbf{case}\ K_i\ \overline{\tau'}\ \overline{\rho'}\ \overline{e'}\ \textbf{of}\ \overline{p \to u}}$$

$$\textsc{S\_Var}\ \frac{\Sigma_{\text{let}}(x) = e}{\Sigma_{\text{let}};\ x \longrightarrow e}$$

$$\textsc{S\_LetRec}\ \frac{\Sigma_{\text{let}}, \overline{x \mapsto e};\ u \longrightarrow u'}{\Sigma_{\text{let}}; \textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ u \longrightarrow \textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ u'}$$

$$\textsc{S\_LetApp}\ \frac{}{\Sigma_{\text{let}}; (\textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ u)\ e' \longrightarrow \textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ (u\ e')}$$

$$\textsc{S\_LetTApp}\ \frac{}{\Sigma_{\text{let}}; (\textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ u)\ \tau \longrightarrow \textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ (u\ \tau)}$$

$$\textsc{S\_LetCApp}\ \frac{}{\Sigma_{\text{let}}; (\textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ u)\ \gamma \longrightarrow \textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ (u\ \gamma)}$$

$$\textsc{S\_LetCast}\ \frac{}{\Sigma_{\text{let}}; (\textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ u) \rhd \gamma \longrightarrow \textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ (u \rhd \gamma)}$$

$$\textsc{S\_LetFlat}\ \frac{}{\begin{array}{l} \Sigma_{\text{let}}; \textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ (\textbf{let}\ \overline{x' : \sigma' = e'}\ \textbf{in}\ u) \longrightarrow \\ \textbf{let}\ \overline{x : \sigma = e}, \overline{x' : \sigma' = e'}\ \textbf{in}\ u \end{array}}$$

$$\textsc{S\_LetCase}\ \frac{}{\begin{array}{l} \Sigma_{\text{let}}; \textbf{case}\ (\textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ u)\ \textbf{of}\ \overline{p \to u'} \longrightarrow \\ \textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ (\textbf{case}\ u\ \textbf{of}\ \overline{p \to u'}) \end{array}}$$

$$\textsc{S\_PrimLet}\ \frac{}{\Sigma_{\text{let}}; \otimes\ (\textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ u) \longrightarrow \textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ (\otimes\ u)}$$

$$\textsc{S\_PrimCast}\ \frac{\Gamma \vdash_{\mathrm{co}} \gamma \ :\ T \sim T}{\Sigma_{\mathrm{let}}; \otimes\, \overline{v}\ (v' \rhd \gamma) \longrightarrow \otimes\, \overline{v}\ v'} \qquad \textsc{S\_Prim}\ \frac{\Sigma_{\mathrm{let}}; \overline{e} \longrightarrow \overline{e'}}{\Sigma_{\mathrm{st}}; \otimes\, \overline{e} \longrightarrow \otimes\, \overline{e'}}$$

$$\textsc{S\_PrimDelta}\ \frac{\otimes\, \overline{v} \longrightarrow_{\delta} v'}{\Sigma_{\mathrm{st}}; \otimes\, \overline{v} \longrightarrow v'} \qquad \textsc{S\_Proj}\ \frac{\Sigma_{\mathrm{let}}; e \longrightarrow e'}{\Sigma_{\mathrm{let}}; \pi_i^k\ e \longrightarrow \pi_i^k\ e'}$$

$$\textsc{S\_ProjKPush}\ \frac{\begin{array}{l} K : \forall \overline{a : \kappa}. \forall \Delta. \overline{\sigma} \to (T\ \overline{a}) \in \Gamma \\ \Psi = \mathsf{extend}(\mathsf{context}(\gamma); \overline{\rho}; \Delta) \\ \overline{\tau'} = \Psi_2(\overline{a}) \\ \overline{\rho}' = \Psi_2(dom\ \Delta) \\ \text{for each } e_i \in \overline{e} \\ \qquad e_i' = e_i \rhd \Psi(\sigma_i) \end{array}}{\Sigma_{\mathrm{st}}; \pi_i^k\ (K\ \overline{\tau}\ \overline{\rho}\ \overline{e} \rhd \gamma) \longrightarrow \pi_i^k\ (K\ \overline{\tau'}\ \overline{\rho}'\ \overline{e'})}$$

$$\textsc{S\_ProjLet}\ \frac{}{\Sigma_{\mathrm{let}}; \pi_i^k\ (\mathbf{let}\ \overline{x : \sigma = e}\ \mathbf{in}\ v) \longrightarrow \mathbf{let}\ \overline{x : \sigma = e}\ \mathbf{in}\ (\pi_i^k\ v)}$$

$$\textsc{S\_ProjMatch}\ \frac{K_k\ \Delta_k\ \overline{x : \sigma} \to x_i}{\Sigma_{\mathrm{let}}; \pi_i^k\ (K_k\ \overline{\tau}\ \overline{\rho}\ \overline{e}) \longrightarrow e_i[\overline{\rho}/\Delta_k]}$$

C.4 Metatheory

This section contains the proofs that our step reduction rules preserve typing, and that an evaluator following our step reduction rules never gets stuck.

C.4.1 Preservation

Theorem: C.4.1 (Preservation). *If* $\Gamma \vdash_{\mathrm{tm}} e\ :\ \tau$ *and* $\Sigma_{\mathrm{let}}; e \longrightarrow e'$ *then* $\Gamma \vdash_{\mathrm{tm}} e'\ :\ \tau$

Proof. The proof for the original parts of System FC can be found in [69], we will only present the additional cases needed by our extensions.

» Case S_LetApp. By inversion of the typing rules we have:

- $\Gamma \vdash_{\mathrm{tm}} \mathbf{let}\ \overline{x : \sigma = e}\ \mathbf{in}\ u\ :\ \sigma \to \tau$
- $\Gamma, \overline{x : \sigma} \vdash_{\mathrm{tm}} u\ :\ \sigma \to \tau$
- $\Gamma \vdash_{\mathrm{tm}} e'\ :\ \sigma$

We can show:

- $\Gamma, \overline{x : \sigma} \vdash_{\mathrm{tm}} u\ e'\ :\ \tau$, by T_App.
- $\Gamma \vdash_{\mathrm{tm}} \mathbf{let}\ \overline{x : \sigma = e}\ \mathbf{in}\ (u\ e')\ :\ \tau$, by T_LetRec.

» The cases for S_LetTApp, S_LetCApp, and S_LetFlat, are analogous to the above case.

» Case S_LetCase. By inversion we have:

- $\overline{\Gamma \vdash_{\text{alt}} p_i \to u'_i \ : \ T\,\overline{\tau'} \to \tau}$
- $\Gamma \vdash_{\text{tm}} \textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ u \ : \ T\,\overline{\tau'}$
- $\Gamma, \overline{x : \sigma} \vdash_{\text{tm}} u \ : \ T\,\overline{\tau'}$

We can show:

- $\Gamma, \overline{x : \sigma} \vdash_{\text{tm}} \textbf{case}\ u\ \textbf{of}\ \overline{p \to u'} \ : \ \tau$, by T_Case.
- $\Gamma \vdash_{\text{tm}} \textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ (\textbf{case}\ u\ \textbf{of}\ \overline{p \to u'}) \ : \ \tau$, by T_LetRec.

» The cases for S_ProjKPush, S_ProjLet, and S_ProjMatch are analogous to the cases: S_KPush, S_LetCase, and S_CaseMatch.

» Case S_PrimCast. By inversion we have:

- $\Gamma \vdash_{\text{tm}} \otimes\,\overline{v} \ : \ T \to \tau$
- $\Gamma \vdash_{\text{tm}} (v' \triangleright \gamma) \ : \ T$
- $\Gamma \vdash_{\text{tm}} v' \ : \ T$

We can show:

- $\Gamma \vdash_{\text{tm}} \otimes\,\overline{v}\,v' \ : \ \tau$, by T_App.

» Case S_PrimLet. By inversion we have:

- $\Gamma \vdash_{\text{tm}} \otimes \ : \ T \to \tau$
- $\Gamma \vdash_{\text{tm}} \textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ u \ : \ T$
- $\Gamma, \overline{x : \sigma} \vdash_{\text{tm}} u \ : \ T$

We can show:

- $\Gamma, \overline{x : \sigma} \vdash_{\text{tm}} \otimes\,u \ : \ \tau$, by T_App.
- $\Gamma \vdash_{\text{tm}} \textbf{let}\ \overline{x : \sigma = e}\ \textbf{in}\ (\otimes\,u) \ : \ \tau$, by T_LetRec.

» Case S_Var. By inversion we have:

- $x : \tau \in \Gamma$.
- $\Gamma \vdash_{\text{bind}} x : \tau \leftarrow e$.
- $\Gamma \vdash_{\text{tm}} e \ : \ \tau$.

» Case S_CaseDef. By inversion we have:

- $\Gamma \vdash_{\text{alt}} \_ \to u_i \ : \ \sigma \to \tau$
- $\Gamma \vdash_{\text{tm}} u_i \ : \ \tau$

» The remaining cases: S_LetRec, S_Proj, S_Prim, and S_PrimDelta, are all by induction.

$\square$

C.4.2 Progress

The progress theorem roughly states that if an expression is not a *value*, always one of the step-reduction rules applies so that an evaluator for expressions can make *progress*. Values, and their types, *value types*, are defined by the grammar:

$$\begin{array}{lll} v & ::= & \lambda x : \sigma.e \mid \Lambda a : \kappa.e \mid \lambda c : \phi.e \mid K\,\overline{\tau}\,\overline{\rho}\,\overline{e} \mid \otimes\,\overline{v} \\ \xi & ::= & \sigma_1 \to \sigma_2 \mid \forall a : \kappa.\sigma \mid \forall c : \phi.\sigma \mid T\,\overline{\sigma} \end{array}$$

We recapitulate the canonical forms lemma:

Lemma: 4.2.1 (Canonical forms). *Say $\Sigma \vdash_{\text{tm}} v \ : \ \sigma$, where Σ is a closed context and v is a value. Then σ is a value type. Furthermore,*

1. *If $\sigma = \sigma_1 \to \sigma_2$ then v is $\lambda x : \sigma_1.e$ or $K\,\overline{\tau}\,\overline{\rho}\,\overline{e}$ or $\otimes\,\overline{v}$.*
2. *If $\sigma = \forall a : \kappa.\sigma'$ then v is $\Lambda x : \kappa.e$ or $K\,\overline{\tau}\,\overline{\rho}\,\overline{e}$.*
3. *If $\sigma = \forall c : \phi.\sigma'$ then v is $\lambda c : \tau_1 \sim \tau_2.e$ or $K\,\overline{\tau}\,\overline{\rho}\,\overline{e}$.*
4. *If $\sigma = T\,\overline{\tau}$ then v is $K\,\overline{\tau}\,\overline{\rho}\,\overline{e}$ or $\otimes\,\overline{v}$.*

The canonical forms lemma tells us that the shape of a value is determined by its type. Additionally, we can only make progress when we are working in a consistent context, which is defined by:

Definition C.1 (Consistency). *A context Γ is consistent if ξ_1 and ξ_2 have the same head form whenever $\Gamma \vdash_{\text{co}} \gamma \ : \ \xi_1 \sim \xi_2$.*

We can now recapitulate, and prove, our progress theorem:

Theorem: 4.2.1 (Progress). *Assume a closed*[1]*, consistent, context Γ. If $\Gamma \vdash_{\text{tm}} e_1 \ : \ \tau$ and e_1 is not a value v, a coerced value $v \rhd \gamma$, or a let-abstracted version of either, then there exists an e_2 such that $\Sigma_{\text{let}}; e_1 \longrightarrow e_2$.*

Proof. By induction on e_1. Assume e_1 is not a value, coerced value, nor a let-abstracted version of either.

» Case $e_1 = x$. Because we are operating in a closed context, x refers to a let-bound variable, and S_Var applies.

» Case $e_1 = e\ e'$. By induction, e is either a value v, a coerced value $v \rhd \gamma$, a let-expression, or takes a step.

1. In the first case, by canonical forms, v is either an abstraction (which beta reduces by S_Beta), a constructor application (which means that e_1 is a value), or a primitive application. When e is a primitive application, e' is either a value (which means that e_1 is a value), a coerced value, a let-expression, or e_1 takes a step by the primitive congruence rule (S_Prim).

[1] A context is closed if it does not contain any expression variable bindings.

  - When e' is a coerced value ($v \rhd \gamma$), by the well-formedness check, primitives only operate on unparametrised data types and γ is a witness of the syntactical equality $T \sim T$, and can be safely removed by (S_PrimCast).
  - When e' is a let-expression, the let-binders are moved out of the application by PrimLet.

2. In the second case, we have a coercion between a value type τ, and $\tau_1 \to \tau_2$. By consistency, then τ must be $\sigma_1 \to \sigma_2$ and the S_Push rule applies.
3. In the third case, the let propagation rule for applications, S_LetApp, applies.
4. In the last case e_1 takes a step by the application congruence rule S_-EApp.

» Case $e_1 = e\ \tau$ and $e_1 = e\ \gamma$ are analogous to the previous case modulo primitives. By the well-formedness check, primitives are monomorphic.

» Case $e_1 = e \rhd \gamma$. By induction, either e is a value v, a coerced value $v \rhd \gamma$, a let-expression, or takes a step.

1. In the first case, then e_1 is a coerced value.
2. In the second case, $(v \rhd \gamma') \rhd \gamma$ steps to $v \rhd (\gamma' ⨾ \gamma)$ by S_Comb.
3. In the third case, the let propagation rule for casts, S_LetCast, applies.
4. In the last case, the congruence rule for casts, S_Coerce, applies.

» Case $e_1 = \textbf{let}\ \overline{x : \tau = u}\ \textbf{in}\ e$. By induction, e is either a value v, a coerced value $v \rhd \gamma$, a let-expression, or takes a step.

1. In the first case, then e_1 is a let-abstracted value.
2. In the second case, then e_1 is a let-abstracted coerced value.
3. In the third case, the let-bindings are grouped into one let-expression by S_LetFlat.
4. In the last case, the congruence rule for let expressions, S_LetRec, applies.

» Case $e_1 = \textbf{case}\ e\ \textbf{of}\ \overline{p \to u}$. By induction, either e is a value v, a coerced value $v \rhd \gamma$, a let-expression, or takes a step.

1. In the first case, by canonical forms, v is either a constructor application and the case-decomposition reduces by S_CaseMatch or S_-CaseDef, or v is a primitive application, and e reduces by the rule S_PrimDelta.
2. In the second case, we have a coercion γ between a value type τ, and $T\ \overline{\sigma}$. By consistency, then τ must be $T\ \overline{\sigma'}$ and the KPush rule applies.
3. In the third case, the let-bindings are moved out of the case-decomposition by S_LetCase.
4. In the last case, the congruence rule for case-decompositions, S_Case, applies.

- Case $e_1 = \pi_i^k\ e$ is analogous to the previous case.

$\square$

D

Preservation of the rewrite rules

This appendix proves that the rewrite rules of chapter 4 are type and semantics preserving.

D.1 Type preservation

For our proofs we will make use of the substitution lemma defined in [68]:

Lemma: D.1.1 (Term substitution). *Say* $\Gamma_1 \vdash_{\text{tm}} e' \ : \ \sigma'$. *If* $\Gamma_1, x : \sigma', \Gamma_2 \vdash_{\text{tm}} e \ : \ \sigma$, *then* $\Gamma_1, \Gamma_2 \vdash_{\text{tm}} e[x \mapsto e'] \ : \ \sigma$.

We need to show that the rewrite rules preserve the types of the expressions, and ensure that binders added to the global environment Γ are correctly typed. We state our type preservation theorem as follows:

Theorem: D.1.1 (Type preservation). *Given that for every expression binding in* Γ, $f_i : \sigma_i = e_i$, $\Gamma \vdash_{\text{tm}} e_i \ : \ \sigma_i$ *holds. If* $\Gamma \vdash_{\text{tm}} e \ : \ \tau$, *and* (Γ', e') *are the new environment and the new expression after applying a rewrite rule to e, then* $\Gamma' \vdash_{\text{tm}} e' \ : \ \tau$, *and for every* $f_i : \sigma_i = e_i$ *in* Γ', $\Gamma' \vdash_{\text{tm}} e_i \ : \ \sigma_i$ *holds.*

Proof. Many of the rewrite rules are a direct implementation of the operational semantics of System FC and are thus type preserving by theorem C.4.1, which is proven in the previous appendix.

We thus present the cases that are *not* a direct implementation of the operational semantics:

» Case CaseTApp. By inversion of the typing rules we have:

– $\Gamma \vdash_{\text{tm}} \textbf{case}\ e\ \textbf{of}\ \overline{p \to u} \ : \ \forall a : \kappa . \tau$

$\overline{}$

– $\Gamma \vdash_{\text{alt}} p_i \to u_i \ : \ T\ \overline{\tau'} \to \forall a : \kappa . \tau$

- $\overline{\Gamma, \Delta', \overline{x : \sigma'} \vdash_{\text{tm}} u_i \; : \; \forall a : \kappa . \tau}$
- $\overline{\Gamma \vdash_{\text{ty}} \sigma \; : \; \kappa}$

We can show:

- $\overline{\Gamma, \Delta', \overline{x : \sigma'} \vdash_{\text{tm}} u_i \; \sigma \; : \; \tau}$, by T_TApp.
- $\overline{\Gamma \vdash_{\text{alt}} p_i \to (u_i \; \sigma) \; : \; T \; \overline{\tau'} \to \tau}$, by T_AltCon.
- $\Gamma \vdash_{\text{tm}} \textbf{case } e \textbf{ of } \overline{p \to (u \; \sigma)} \; : \; \tau$, by T_Case.

» Cases CaseCApp and CaseCast are analogous to CaseTApp

» Case LamApp. By inversion we have:

- $\Gamma \vdash_{\text{tm}} \lambda x : \sigma . e \; : \; \sigma \to \tau$
- $\Gamma, x : \sigma \vdash_{\text{tm}} e \; : \; \tau$
- $\Gamma \vdash_{\text{tm}} u \; : \; \sigma$

We can show:

- $\Gamma \vdash_{\text{bind}} x : \sigma \leftarrow u$, by T_Bind.
- $\Gamma \vdash_{\text{tm}} \textbf{let } \{x : \sigma = u\} \textbf{ in } e \; : \; \tau$, by T_LetRec.

» Case CaseApp. By inversion we have:

- $\Gamma \vdash_{\text{tm}} \; \textbf{case } e \textbf{ of } \overline{p \to u} \; : \; \sigma \to \tau$
- $\overline{\Gamma \vdash_{\text{alt}} \; p_i \to u_i \; : \; T \; \overline{\tau'} \to (\sigma \to \tau)}$
- $\overline{\Gamma, \Delta', \overline{x' : \sigma'} \vdash_{\text{tm}} u_i \; : \; \sigma \to \tau}$
- $\Gamma \vdash_{\text{tm}} \; e' \; : \; \sigma$

We can show:

- $\overline{\Gamma, , \Delta', \overline{x' : \sigma'}, x : \sigma \vdash_{\text{tm}} u_i \; x \; : \; \tau}$, by T_App.
- $\overline{\Gamma, x : \sigma \vdash_{\text{alt}} p_i \to (u_i \; x) \; : \; T \; \overline{\tau'} \to \tau}$, by T_AltCon.
- $\Gamma, x : \sigma \vdash_{\text{tm}} \textbf{case } e \textbf{ of } \overline{p \to (u \; x)} \; : \; \tau$, by T_Case.
- $\Gamma \vdash_{\text{bind}} x : \sigma \leftarrow e'$, by T_Bind.
- $\Gamma \vdash_{\text{tm}} \textbf{let } \{x : \sigma = e'\} \textbf{ in } (\textbf{case } e \textbf{ of } \overline{p \to (u \; x)}) \; : \; \tau$, by T_LetRec.

» Case BindNonRep. By inversion we have:

- $\Gamma, \Gamma_{\overline{b}}, x_i \; : \; \sigma_i \; \vdash_{\text{tm}} \; u \; : \; \tau$, where $\Gamma_{\overline{b}}$ represents all let-binders except $x_i : \sigma_i$.

We can show:

- $\Gamma, x_i : \sigma_i \vdash_{\text{tm}} \textbf{let } \{b_1; ...; b_{i-1}; b_{i+1}; ...; b_n\} \textbf{ in } u \; : \; \tau$, by T_LetRec.
- $\Gamma \vdash_{\text{tm}} (\textbf{let } \{b_1; ...; b_{i-1}; b_{i+1}; ...; b_n\} \textbf{ in } u)[e_i / x_i] \; : \; \tau$, by the substitution lemma (lemma D.1.1).

» Case LiftNonRep. By inversion we have:

- $\Gamma, \Gamma_{\overline{b}}, x_i \; : \; \sigma_i \; \vdash_{\text{tm}} \; u \; : \; \tau$, where $\Gamma_{\overline{b}}$ represents all let-binders except $x_i : \sigma_i$.

We can show:

- $\Gamma \vdash_{\text{tm}} f : \overline{\tau'} \to \sigma_i$, by the definition of free variables, and the repeated application of the T_Abs rule.
- $\Gamma \vdash_{\text{tm}} f\ \overline{z} : \sigma_i$, by T_App.
- $\Gamma, x_i : \sigma_i \vdash_{\text{tm}} \textbf{let}\ \{b_1; ...; b_{i-1}; b_{i+1}; ...; b_n\}\ \textbf{in}\ u : \tau$, by T_LetRec.
- $\Gamma \vdash_{\text{tm}} (\textbf{let}\ \{b_1; ...; b_{i-1}; b_{i+1}; ...; b_n\}\ \textbf{in}\ u)[(f\ \overline{z})/x_i] : \tau$, by the substitution lemma (lemma D.1.1).

» Case TypeSpec. By inversion we have:

- $\Gamma \vdash_{\text{tm}} f\ \overline{e} : \forall a : \kappa \to \tau.$
- $\Gamma \vdash_{\text{tm}} f : \overline{\tau'} \to \forall a : \kappa \to \tau$
- $\overline{\Gamma \vdash_{\text{tm}} e_i : \tau'_i}$
- $\Gamma \vdash_{\text{ty}} \sigma' : \kappa$

We can show:

- $\Gamma, \overline{x : \tau'} \vdash_{\text{tm}} (\Gamma@f)\ \overline{x}\ \sigma : \tau$, by: the precondition that globally bound expression are correctly typed, repeated use of T_App, followed by T_TApp.
- $\Gamma \vdash_{\text{tm}} \lambda\overline{x : \tau'}(\Gamma@f)\ \overline{x}\ \sigma : \overline{\tau'} \to \tau$, by repeated use of T_Abs.
- $\Gamma \vdash_{\text{tm}} f'\ \overline{e} : \tau$, by repeated use of T_App.

» The cases CoSpec, NonRepSpec, and CastSpec, are analogous to the above case.

» Case CaseCase. By inversion we have:

- $\Gamma \vdash_{\text{tm}} \textbf{case}\ e\ \textbf{of}\ \{p_1 \to u_1; ...; p_n \to u_n\} : T\ \overline{\tau'}$
- $\Gamma \vdash_{\text{tm}} e : T'\ \overline{\tau''}$
- $\overline{\Gamma \vdash_{\text{alt}} p_i \to u_i : T'\ \overline{\tau''} \to T\ \overline{\tau'}}$
- $\overline{\Gamma, \Delta', \overline{x : \sigma'} \vdash_{\text{tm}} u_i : T\ \overline{\tau'}}$
- $\overline{\Gamma \vdash_{\text{alt}} p'_i \to u'_i : T'\ \overline{\tau'} \to \tau}$

We can show:

- $\overline{\Gamma, \Delta', \overline{x : \sigma'} \vdash_{\text{tm}} \textbf{case}\ u_i\ \textbf{of}\ \overline{p' \to u'} : \tau}$, by T_Case.
- $\overline{\Gamma \vdash_{\text{alt}} p_i \to \textbf{case}\ u_i\ \textbf{of}\ \overline{p' \to u'} : T'\ \overline{\tau''} \to \tau}$, by T_AltCon.
- $\Gamma \vdash_{\text{tm}}$ $\textbf{case}\ e\ \textbf{of}\ \{p_1 \to \textbf{case}\ u_1\ \textbf{of}\ \overline{p' \to u'}; ...; p_n \to \textbf{case}\ u_n\ \textbf{of}\ \overline{p' \to u'}\} : \tau$, by T_Case.

» Case InlineNonRep is correct by the precondition on the lemma, that for every expression binding $f_i : \sigma_i = e_i$ in Γ, $\Gamma \vdash_{\text{tm}} e_i : \sigma_i$ holds.

» Case, the third variant of CaseCon. By inversion we have:

- $\Gamma \vdash_{\text{alt}} p_0 \to u_0 : T\ \overline{\tau'} \to \tau.$
- $\Gamma, \Delta', \overline{x : \sigma'} \vdash_{\text{tm}} u_0 : \tau.$

We can show:

- $\Gamma \vdash_{\text{tm}} u_0 : \tau$, by the definition of free variables.

□

D.2 Semantics preservation

We need to show that the transformations do not change the meaning of expression, that they preserve the semantics. We state our semantics preservations theorem as follows:

Theorem: D.2.1. *Let $e \longrightarrow^{*} e'$ be the repeated application of the step reduction rules (appendix C) on the expression e to get a new expression e'. Let e_t be the result of applying one of the rewrite rules to e, then there exists an e' such that $e \longrightarrow^{*} e'$ and $e_t \longrightarrow^{*} e'$.*

Proof. Many of the rewrite rules are a direct implementation of the operational semantics of System FC, and thus semantics preserving by definition.

We thus present the cases that are *not* a direct implementation of the operational semantics:

» CaseCase transformation:
  - The precedent is: $\textbf{case}\ (\textbf{case}\ e\ \textbf{of}\ \overline{p \to u})\ \textbf{of}\ \overline{p' \to u'}$.
  - The antecedent is: $\textbf{case}\ e\ \textbf{of}\ \overline{p \to \textbf{case}\ u\ \textbf{of}\ \overline{p' \to u'}}$.

  We show that the transformation is semantics preserving by showing that both the precedent and the antecedent reduce to the same alternative in $\overline{p' \to u'}, u_j$. We demonstrate the reduction steps for those situations where the subjects and alternatives always reduce to a constructor application, which we indicate by K^{*}.
  The precedent reduces to:

$$1.\ \cfrac{2.\ \cfrac{1.\ \cfrac{\textbf{case}\ (\textbf{case}\ e\ \textbf{of}\ \overline{p \to u})\ \textbf{of}\ \overline{p' \to u'} \longrightarrow^{*} \textbf{case}\ (\textbf{case}\ K^{*}\ \textbf{of}\ \overline{p \to u})\ \textbf{of}\ \overline{p' \to u'}}{\textbf{case}\ (\textbf{case}\ e\ \textbf{of}\ \overline{p \to u})\ \textbf{of}\ \overline{p' \to u'} \longrightarrow^{*} \textbf{case}\ u_i\ \textbf{of}\ \overline{p' \to u'}}}{\textbf{case}\ (\textbf{case}\ e\ \textbf{of}\ \overline{p \to u})\ \textbf{of}\ \overline{p' \to u'} \longrightarrow^{*} \textbf{case}\ K^{*}_i\ \textbf{of}\ \overline{p' \to u'}}}{\textbf{case}\ (\textbf{case}\ e\ \textbf{of}\ \overline{p \to u})\ \textbf{of}\ \overline{p' \to u'} \longrightarrow^{*} u'_j}$$

  The antecedent reduces to:

$$1.\ \cfrac{2.\ \cfrac{1.\ \cfrac{\textbf{case}\ e\ \textbf{of}\ \overline{p \to \textbf{case}\ u\ \textbf{of}\ \overline{p' \to u'}} \longrightarrow^{*} \textbf{case}\ K^{*}\ \textbf{of}\ \overline{p \to \textbf{case}\ u\ \textbf{of}\ \overline{p' \to u'}}}{\textbf{case}\ e\ \textbf{of}\ \overline{p \to \textbf{case}\ u\ \textbf{of}\ \overline{p' \to u'}} \longrightarrow^{*} \textbf{case}\ u_i\ \textbf{of}\ \overline{p' \to u'}}}{\textbf{case}\ e\ \textbf{of}\ \overline{p \to \textbf{case}\ u\ \textbf{of}\ \overline{p' \to u'}} \longrightarrow^{*} \textbf{case}\ K^{*}_i\ \textbf{of}\ \overline{p' \to u'}}}{\textbf{case}\ e\ \textbf{of}\ \overline{p \to \textbf{case}\ u\ \textbf{of}\ \overline{p' \to u'}} \longrightarrow^{*} u'_j}$$

  Here:
  1. Indicates the repeated application of the S_Case rule, followed by either the S_CaseMatch or S_CaseDef rule.
  2. Indicates the repeated application of the S_Case rule.

  The cases where the subjects and alternatives reduce to: coerced values, let-expressions, or primitive applications, are straightforward and similarly tedious.

» The proofs for the other transformations are similarly straightforward and tedious.

□

Acronyms

A	ALU	arithmetic logic unit
	ANF	administrative normal form
	ASIC	application-specific integrated-circuit
	AST	abstract syntax tree
B	BSV	BlueSpec SystemVerilog
C	CλaSH	CAES language for synchronous hardware
	CPU	central processing unit
	CSE	common sub-expression elimination
D	DSL	domain specific language
	DSP	digital signal processing
F	FFT	fast fourier transform
	FIFO	first-in-first-out buffer
	FIR	finite impulse response
	FPGA	field programmable gate array
G	GADT	generalized algebraic data type
	GHC	Glasgow Haskell compiler
	GPU	graphics processing unit
H	HDL	hardware description language
	HLS	high-level synthesis
L	LHS	left hand side
	LUT	look-up table
M	MAC	multiply-and-accumulate
R	RHS	right hand side
	RTL	register-transfer level
S	S.A.C.	stand alone complex
	SMxV	sparse matrix-vector multiplication
T	TRS	term rewrite system
V	VHDL	VHSIC hardware description language (HDL)
	VHSIC	very high speed integrated circuit
	VLSI	very-large-scale integration

Bibliography

[1] A. Acosta. ForSyDe tutorial. `https://forsyde.ict.kth.se/trac/wiki/ForSyDe/Haskell/ForSyDeTutorial`, January 2011. (Cited on page 31).

[2] V. Akella and G. Gopalakrishnan. From Process-Oriented Functional Specifications to Efficient Asynchronous Circuits. In *Proceedings of the 5th International Conference on VLSI Design*, pages 324–325. IEEE, January 1992. doi: 10.1109/ICVD.1992.658072. (Cited on page 7).

[3] P. J. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2008. ISBN 978-0-12-088785-9. (Cited on page 16).

[4] C. Baaij. CλasH: From Haskell To Hardware. Master's thesis, University of Twente, December 2009. URL `http://essay.utwente.nl/59482/`. (Cited on page 13).

[5] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics. Elsevier Science, 1985. ISBN 978-0-08-093375-7. (Cited on page 25).

[6] J. M. Bell, F. Bellegarde, and J. Hook. Type-Driven Defunctionalization. In *Proceedings of the 2nd International Conference on Functional Programming (ICFP)*, pages 25–37, 1997. doi: 10.1145/258948.258953. (Cited on page 98).

[7] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware Design in Haskell. In *Proceedings of the 3rd International Conference on Functional Programming (ICFP)*, pages 174–184. ACM, 1998. ISBN 1-58113-024-4. doi: 10.1145/289423.289440. (Cited on pages 10, 26, and 27).

[8] T. Bollaert. Catapult Synthesis: A Practical Introduction to Interactive C Synthesis. In P. Coussy and A. Morawiec, editors, *High-Level Synthesis*, pages 29–52. Springer Netherlands, 2008. ISBN 978-1-4020-8587-1. doi: 10.1007/978-1-4020-8588-8_3. (Cited on page 4).

[9] E. C. Brady. IDRIS —: Systems Programming Meets Full Dependent Types. In *Proceedings of the 5th Workshop on Programming Languages Meets Program Verification (PLPV)*, PLPV '11, pages 43–54, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0487-0. doi: 10.1145/1929529.1929536. (Cited on page 62).

[10] R. K. Brayton, A. L. Sangiovanni-Vincentelli, C. T. McMullen, and G. D. Hachtel. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, Norwell, MA, USA, 1984. ISBN 0898381649. (Cited on page 91).

[11] M. M. T. Chakravarty, G. Keller, and S. P. Jones. Associated Type Synonyms. In *Proceedings of the 10th International Conference on Functional Programming (ICFP)*, ICFP '05, pages 241–253, New York, NY, USA, 2005. ACM. ISBN 1-59593-064-7. doi: 10.1145/1086365.1086397. (Cited on pages 60 and 75).

[12] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(4):473–491, April 2011. ISSN 0278-0070. doi: 10.1109/TCAD.2011.2110592. (Cited on page 6).

[13] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach. An Introduction to High-Level Synthesis. *IEEE Design & Test of Computers*, 26(4):8–17, 2009. ISSN 0740-7475. doi: 10.1109/MDT.2009.69. (Cited on page 3).

[14] Design, Automation & Test in Europe (DATE) conference. University Booth 2011 Programme. `http://www.date-conference.com/date11/node/4165`, March 2011. (Cited on page 141).

[15] R. A. Eisenberg and S. Weirich. Dependently Typed Programming with Singletons. In *Proceedings of the 2012 Haskell Symposium*, Haskell '12, pages 117–130, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1574-6. doi: 10.1145/2364506.2364522. (Cited on page 62).

[16] R. A. Eisenberg, D. Vytiniotis, S. Peyton Jones, and S. Weirich. Closed Type Families with Overlapping Equations. In *Proceedings of the 41st Symposium on Principles of Programming Languages (POPL)*, pages 671–683, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2544-8. doi: 10.1145/2535838.2535856. (Cited on page 75).

[17] S. Frankau. *Hardware Synthesis from a Stream-Processing Functional Language*. PhD thesis, University of Cambridge, July 2004. (Cited on page 72).

[18] M. Gerards, J. Kuper, A. Kokkeler, and B. Molenkamp. Streaming reduction circuit. In *Proceedings of the 12th conference on Digital System Design (DSD)*, pages 287–292, Los Alamitos, CA, USA, August 2009. IEEE Computer Society. ISBN 978-0-7695-3782-5. doi: 10.1109/DSD.2009.141. (Cited on pages 136 and 137).

[19] D. R. Ghica. Geometry of Synthesis: A structured approach to VLSI design. In *Proceedings of the 34th annual Symposium on Principles of Programming Languages (POPL)*, pages 363–375. ACM, 2007. ISBN 1-59593-575-4. doi: 10.1145/1190216.1190269. (Cited on page 22).

[20] D. R. Ghica and A. Smith. Geometry of Synthesis II: From Games to Delay-Insensitive Circuits. In *Proceedings of the 26th Conference on the Mathematical Foundations of Programming Semantics (MFPS)*, volume 265 of *Electronic Notes in Theoretical Computer Science*, pages 301–324. 2010. doi: 10.1016/j.entcs.2010.08.018. (Not cited).

[21] D. R. Ghica and A. Smith. Geometry of Synthesis III: Resource Management Through Type Inference. In *Proceedings of the 38th Symposium on Principles of Programming Languages (POPL)*, pages 345–356. ACM, 2011. ISBN 978-1-4503-0490-0. doi: 10.1145/1926385.1926425. (Cited on pages 22 and 24).

[22] D. R. Ghica and A. Smith. Verity – The Geometry of Synthesis. `http://www.veritygos.org`, January 2013. (Cited on pages 22 and 67).

[23] D. R. Ghica, A. Smith, and S. Singh. Geometry of Synthesis IV: Compiling Affine Recursion into Static Hardware. In *Proceedings of the 16th International Conference on Functional Programming (ICFP)*, pages 221–233. ACM, 2011. ISBN 978-1-4503-0865-6. doi: 10.1145/2034773.2034805. (Cited on pages 22, 25, and 52).

[24] A. Gill. Type-Safe Observable Sharing in Haskell. In *Proceedings of the 2nd Haskell Symposium*, pages 117–128. ACM, September 2009. ISBN 978-1-60558-508-6. doi: 10.1145/1596638.1596653. (Cited on pages 10 and 27).

[25] A. Gill. kansas-lava: Kansas Lava is a hardware simulator and VHDL generator. `http://hackage.haskell.org/package/kansas-lava`, November 2011. (Cited on page 26).

[26] A. Gill, T. Bull, G. Kimmell, E. Perrins, E. Komp, and B. Werling. Introducing Kansas Lava. Submitted to The International Symposia on Implementation and Application of Functional Languages (IFL), November 2009. URL `http://ittc.ku.edu/~andygill/papers/kansas-lava-ifl09.pdf`. (Cited on pages 26, 27, 136, and 159).

[27] A. Gill, T. Bull, A. Farmer, G. Kimmell, and E. Komp. Types and Type Families for Hardware Simulation and Synthesis. In R. Page, Z. Horváth, and V. Zsók, editors, *Trends in Functional Programming*, volume 6546 of *Lecture Notes in Computer Science (LNCS)*, pages 118–133. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-22940-4. doi: 10.1007/978-3-642-22941-1_8. (Cited on pages 26 and 27).

[28] Haskell.org. Template Haskell Wiki. `http://www.haskell.org/haskellwiki/Template_Haskell`, October 2013. (Cited on pages 27, 31, 144, and 145).

[29] R. Herveille. CORDIC core. `http://opencores.org/project,cordic`, September 2014. (Cited on page 147).

[30] R. Herveille. I2C controller core. `http://opencores.org/project,i2c`, September 2014. (Cited on pages 142, 147, and 159).

[31] J. C. Hoe and Arvind. Hardware Synthesis from Term Rewriting Systems. In *Proceedings of the 10th International Conference on VLSI*, pages 595–619, 1999. (Cited on page 21).

[32] J. Hughes. Generalising Monads to Arrows. *Science of Computer Programming*, 37 (1-3):67–111, May 2000. ISSN 0167-6423. doi: 10.1016/S0167-6423(99)00023-4. (Cited on page 159).

[33] IEEE Standard. Verilog Hardware Description Language, 2005. (Cited on page 18).

[34] IEEE Standard. VHDL Language Reference Manual, 2008. (Cited on page 16).

[35] IEEE Standard. SystemVerilog – Unified Hardware Design, Specification, and Verification Language, 2012. (Cited on page 20).

[36] X. Jin. Implementation of the MUSIC Algorithm in CλaSH. Master's thesis, June 2014. URL `http://essay.utwente.nl/65225/`. (Cited on page 135).

[37] E. Kmett. Lenses, Folds and Traversals. `http://lens.github.io`, September 2014. (Cited on pages 144 and 165).

[38] M. Kooijman. Haskell as a higher order structural hardware description language. Master's thesis, University of Twente, December 2009. URL `http://essay.utwente.nl/59381/`. (Cited on pages 13 and 59).

[39] J. Kuper and R. Wester. N Queens on an FPGA: Mathematics, Programming, or Both? In *Communicating Process Architectures (CPA)*, United Kingdom, August 2014 (to appear). Open Channel Publishing. (Cited on page 136).

[40] S. Mac Lane. *Categories for the Working Mathematician*. Number 5 in Graduate Texts in Mathematics. New York, 1971. ISBN 0387900357. (Cited on page 143).

[41] G. Martin and G. Smith. High-Level Synthesis: Past, Present, and Future. *IEEE Design & Test of Computers*, 26(4):18–25, 2009. ISSN 0740-7475. doi: 10.1109/MDT.2009.83. (Cited on pages 4 and 6).

[42] C. Mcbride and R. Paterson. Applicative Programming with Effects. *Journal of Functional Programming*, 18(1):1–13, January 2008. ISSN 0956-7968. doi: 10.1017/S0956796807006326. (Cited on pages 57 and 165).

[43] M. C. McFarland, A. C. Parker, and R. Camposano. The high-level synthesis of digital systems. *Proceedings of the IEEE*, 78(2):301–318, 1990. ISSN 0018-9219. doi: 10.1109/5.52214. (Cited on page 3).

[44] G. H. Mealy. A Method for Synthesizing Sequential Circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955. ISSN 1538-7305. doi: 10.1002/j.1538-7305.1955.tb03788.x. (Cited on pages 30 and 59).

[45] A. Megacz. Hardware Design with Generalized Arrows. In A. Gill and J. Hage, editors, *Implementation and Application of Functional Languages*, volume 7257 of *Lecture Notes in Computer Science*, pages 164–180. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-34406-0. doi: 10.1007/978-3-642-34407-7_11. (Cited on page 160).

[46] N. Mitchell and C. Runciman. Losing Functions without Gaining Data. In *Proceedings of the 2nd Symposium on Haskell*, pages 13–24. ACM, September 2009. ISBN 978-1-60558-508-6. (Cited on page 99).

[47] A. Mycroft and R. Sharp. A Statically Allocated Parallel Functional Language. In *Proceedings of the 27th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 1853 of *Lecture Notes in Computer Science (LNCS)*, pages 37–48. Springer-Verlag, 2000. ISBN 978-3-540-67715-4. doi: 10.1007/3-540-45022-X_5. (Cited on pages 21, 22, and 66).

[48] A. Niedermeier. *A Fine-Grained Parallel Dataflow-Inspired Architecture for Streaming Applications*. PhD thesis, Univ. of Twente, Enschede, August 2014. (Cited on page 136).

[49] R. S. Nikhil. Bluespec: A General-Purpose Approach to High-Level Synthesis Based on Parallel Atomic Transactions. In Philippe Coussy and Adam Morawiec, editor, *High-Level Synthesis - From Algorithm to Digital Circuit*, pages 129–146. Springer Netherlands, 2008. (Cited on page 20).

[50] OpenCores. WISHBONE System-on-Chip Interconnect. `http://opencores.org/opencores,wishbone`, June 2010. (Cited on page 142).

[51] B. O'Sullivan, J. Goerzen, and D. Stewart. *Real World Haskell*, chapter Monads. O'Reilly Media, Inc., 2008. ISBN 978-0-596-51498-3. URL `http://book.realworldhaskell.org/read/monads.html`. (Cited on pages 144 and 165).

[52] R. Paterson. A New Notation for Arrows. In *Proceedings of the 6th International Conference on Functional Programming (ICFP)*, ICFP '01, pages 229–240, New York, NY, USA, 2001. ACM. ISBN 1-58113-415-0. doi: 10.1145/507635.507664. (Cited on pages 68 and 159).

[53] R. Paterson. Arrows and Computation. In Jeremy Gibbons and Oege de Moor, editor, *The Fun of Programming*, pages 201–222. Palgrave, 2003. (Cited on page 68).

[54] S. Peyton Jones and A. Santos. Compilation by Transformation in the Glasgow Haskell Compiler. In *Functional Programming, Glasgow 1994*, Workshops in Computing, pages 184–204. Springer London, 1995. ISBN 978-3-540-19914-4. doi: 10.1007/978-1-4471-3573-9_13. (Cited on page 99).

[55] R. Pope and B. Yorgey. first-class-patterns: First class patterns and pattern matching, using type families. `http://hackage.haskell.org/package/first-class-patterns`, July 2013. (Cited on pages 28 and 29).

[56] F. Pottier and N. Gauthier. Polymorphic Typed Defunctionalization. In *Proceedings of the 31st Symposium on Principles of Programming Languages (POPL)*, pages 89–98. ACM, 2004. ISBN 1-58113-729-X. doi: 10.1145/964001.964009. (Cited on page 98).

[57] J. C. Reynolds. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of the 25th ACM National Conference*, pages 717 – 740. ACM Press, 1972. (Cited on page 98).

[58] I. Sander and A. Jantsch. System Modeling and Transformational Design Refinement in ForSyDe. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(1):17–32, January 2004. (Cited on pages 30 and 159).

[59] R. Sharp and A. Mycroft. The FLaSH Compiler: Efficient Circuits from Functional Specifications. Technical report, AT&T Research Laboratories Cambridge, 2000. (Cited on page 21).

[60] T. Sheard and S. Peyton Jones. Template meta-programming for Haskell. In *Proceedings of the 2002 Workshop on Haskell*, pages 1–16. ACM, 2002. ISBN 1-58113-605-6. doi: 10.1145/581690.581691. (Cited on pages 27 and 31).

[61] J. R. Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. Technical report, Pittsburgh, PA, USA, 1994. URL `http://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf`. (Cited on page 136).

[62] M. Sulzmann, M. M. T. Chakravarty, S. Peyton Jones, and K. Donnelly. System F with Type Equality Coercions. In *Proceedings of the 2007 International Workshop on Types in Languages Design and Implementation (TLDI)*, pages 53–66, New York, NY, USA, 2007. ACM. ISBN 1-59593-393-X. doi: 10.1145/1190315.1190324. (Cited on pages 74, 75, and 79).

[63] S. Sutheland, S. Davidmann, and P. Flacke. *SystemVerilog for Design*. Springer US, 2nd edition, 2006. ISBN 978-0-387-36495-7. doi: 10.1007/0-387-36495-1. (Cited on page 18).

[64] The GHC Team. The GHC Compiler, version 7.6.3. `http://haskell.org/ghc`, April 2013. (Cited on page 12).

[65] The GHC Team. GHC API. `http://www.haskell.org/ghc/docs/latest/html/libraries/ghc/index.html`, September 2014. (Cited on pages 74 and 75).

[66] F. Van Nee. To a new hardware design methodology: A case study of the cochlea model. Master's thesis, March 2014. URL `http://essay.utwente.nl/64835/`. (Cited on page 135).

[67] J. E. Volder. The CORDIC Trigonometric Computing Technique. *IRE Transactions on Electronic Computers*, EC-8(3):330–334, September 1959. ISSN 0367-9950. doi: 10.1109/TEC.1959.5222693. (Cited on page 147).

[68] S. Weirich, D. Vytiniotis, S. Peyton Jones, and S. Zdancewic. Generative Type Abstraction and Type-level Computation. In *Proceedings of the 38th Annual Symposium on Principles of Programming Languages (POPL)*, pages 227–240, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0490-0. doi: 10.1145/1926385.1926411. (Cited on pages 74, 75, and 191).

[69] S. Weirich, J. Hsu, and R. A. Eisenberg. System FC with Explicit Kind Equality. In *Proceedings of the 18th International Conference on Functional Programming (ICFP)*, pages 275–286, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2326-0. doi: 10.1145/2500365.2500599. (Cited on pages 74, 75, 78, 80, 82, 86, 177, 181, and 185).

[70] R. Wester and J. Kuper. A space/time tradeoff methodology using higher-order functions. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–2, USA, 2013. IEEE Computer Society. (Cited on page 136).

[71] R. Wester and J. Kuper. Design space exploration of a particle filter using higher-order functions. In *Reconfigurable Computing: Architectures, Tools, and Applications*, volume 8405 of *Lecture Notes in Computer Science*, pages 219–226. Springer Verlag, London, United Kingdom, 2014. (Cited on page 135).

[72] R. Wester and J. Kuper. Deriving stencil hardware accelerators from a single higher-order function. In *Communicating Process Architectures (CPA)*, United Kingdom, August 2014 (to appear). Open Channel Publishing. (Cited on page 135).

[73] R. Wester, D. Sarakiotis, E. Kooistra, and J. Kuper. Specification of APERTIF Polyphase Filter Bank in CλaSH. In *Communicating Process Architectures (CPA)*, pages 53–64, United Kingdom, August 2012. Open Channel Publishing. (Cited on page 135).

[74] H. Xi, C. Chen, and G. Chen. Guarded Recursive Datatype Constructors. In *Proceedings of the 30th Symposium on Principles of Programming Languages*, pages 224–235, New York, NY, USA, 2003. ACM. ISBN 1-58113-628-5. doi: 10.1145/604131.604150. (Cited on page 75).

[75] B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving Haskell a Promotion. In *Proceedings of the 8th Workshop on Types in Language Design and Implementation (TLDI)*, TLDI '12, pages 53–66, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1120-5. doi: 10.1145/2103786.2103795. (Cited on pages 61, 74, and 75).

[76] S. Zefirov. HHDL: Hardware Description Language embedded in Haskell. `http://hackage.haskell.org/package/HHDL`, December 2011. (Cited on pages 28 and 29).

[77] Z. Zhang, Y. Fan, W. Jiang, G. Han, C. Yang, and J. Cong. AutoPilot: A Platform-Based ESL Synthesis System. In P. Coussy and A. Morawiec, editors, *High-Level Synthesis*, pages 99–112. Springer Netherlands, 2008. ISBN 978-1-4020-8587-1. doi: 10.1007/978-1-4020-8588-8_6. (Cited on page 4).

[78] G. Érdi. A Brainfuck CPU in FPGA. `https://github.com/gergoerdi/brainfuck-cpu-fpga`, January 2013. (Cited on page 28).

List of Publications

[CB:1] **C. P. R. Baaij**, M. Kooijman, J. Kuper, M. E. T. Gerards, and E. Molenkamp. Tool Demonstration: CLasH - From Haskell to Hardware. In *Proceedings of the 2nd Symposium on Haskell*, page 3. ACM, 2009. doi: 10.1145/1596638.1667736.

[CB:2] G. J. M. Smit, J. Kuper, and **C. P. R. Baaij**. A mathematical approach towards hardware design. In P. M. Athanas, J. Becker, J. Teich, and I. Verbauwhede, editors, *Dagstuhl Seminar on Dynamically Reconfigurable Architectures*, number 10281 in Dagstuhl Seminar Proceedings. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, December 2010.

[CB:3] A. Niedermeier, R. Wester, K. C. Rovers, **C. P. R. Baaij**, J. Kuper, and G. J. M. Smit. Designing a dataflow processor using CλaSH. In *28th Norchip Conference*, page 69. IEEE Circuits and Systems Society, November 2010. doi: 10.1109/NORCHIP.2010.5669445.

[CB:4] A. Niedermeier, R. Wester, **C. P. R. Baaij**, J. Kuper, and G. J. M. Smit. Comparing CλaSH and VHDL by implementing a dataflow processor. In *Proceedings of the Workshop on PROGram for Research on Embedded Systems and Software (PROGRESS)*, pages 216–221. Technology Foundation STW, November 2010.

[CB:5] J. Kuper, **C. P. R. Baaij**, M. Kooijman, and M. E. T. Gerards. Exercises in architecture specification using CλaSH. In *Proceedings of Forum on Specification and Design Languages (FDL)*, pages 178–183. ECSI Electronic Chips & Systems design Initiative, September 2010. doi: 10.1049/ic.2010.0149.

[CB:6] M. E. T. Gerards, **C. P. R. Baaij**, J. Kuper, and M. Kooijman. Hiding State in CλaSH Hardware Descriptions. In *Preproceedings of the 22nd Symposium on Implementation and Application of Functional Languages (IFL)*, pages 107–119. Utrecht University, August 2010.

[CB:7] **C. P. R. Baaij**, M. Kooijman, J. Kuper, W. A. Boeijink, and M. E. T. Gerards. CλaSH: Structural Descriptions of Synchronous Hardware using Haskell. In *Proceedings of the 13th Conference on Digital System Design (DSD)*, pages 714–721. IEEE Computer Society, September 2010. doi: 10.1109/DSD.2010.21.

[CB:8] J. Kuper, **C. P. R. Baaij**, M. Kooijman, and M. E. T. Gerards. Architecture Specifications in CλaSH. In T. J. Kaźmierski and A. Morawiec, editors, *System Specification and Design Languages*, volume 106 of *Lecture Notes in Electrical Engineering (LNEE)*, pages 191–206. Springer New York, December 2011. doi: 10.1007/978-1-4614-1427-8_12.

[CB:9] M. E. T. Gerards, **C. P. R. Baaij**, J. Kuper, and M. Kooijman. Higher-Order Abstraction in Hardware Descriptions with CλaSH. In P. Kitsos, editor, *Proceedings of the 14th Conference on Digital System Design (DSD)*, pages 495–502. IEEE Computer Society, August 2011. doi: 10.1109/DSD.2011.69.

[CB:10] **C. P. R. Baaij**, J. Kuper, and L. Schubert. SoOSiM: Operating System and Programming Language Exploration. In G. Lipari and T. Cucinotta, editors, *Proceedings of the 3rd International Workshop on Analysis Tools and Methodologies for Embedded and Real-time System (WATERS)*, pages 63–68, 2012.

[CB:11] R. Wester, **C. P. R. Baaij**, and J. Kuper. A two step hardware design method using CλaSH. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 181–188. IEEE Computer Society, August 2012. doi: 10.1109/FPL.2012.6339258.

[CB:12] B. N. Uchevler, K. Svarstad, J. Kuper, and **C. P. R. Baaij**. System-Level Modelling of Dynamic Reconfigurable Designs using Functional Programming Abstractions. In *14th International Symposium on Quality Electronic Design (ISQED)*, pages 379–385. IEEE, March 2013. doi: 10.1109/ISQED.2013.6523639.

[CB:13] **C.P.R. Baaij** and J. Kuper. Using Rewriting to Synthesize Functional Languages to Digital Circuits. In Jay McCarthy, editor, *Trends in Functional Programming (TFP), Provo, UT, USA, May 14-16, 2013, Revised Selected Papers*, volume 8322 of *Lecture Notes in Computer Science (LNCS)*, pages 17–33. Springer-Verlag, 2014. ISBN 978-3-642-45339-7. doi: 10.1007/978-3-642-45340-3_2.

This thesis

```
@phdthesis{baaij2015:thesis,
  author={Baaij, Christiaan P.R.},
  title={Digital Circuits in CλaSH -- Functional
    Specifications and Type-Directed Synthesis},
  school={Universiteit Twente},
  address={PO Box 217, 7500AE Enschede, The Netherlands},
  year={2015},
  month={jan},
  day={23},
  number={CTIT Ph.D.-thesis series No. 14-335},
  issn={1381-3617},
  isbn={978-90-365-3803-9},
  doi={10.3990/1.9789036538039}
}
```

CλASH

כדי לעשות את זה קשה, זה לא כמו בספר

ISBN 978-90-365-3803-9

9 789036 538039