MASTER'S THESIS

TIMED TYPES FOR SYNCHRONOUS HARDWARE

EXPRESSING TEMPORAL BEHAVIOUR THROUGH THE TYPE SYSTEM OF A FUNCTIONAL LANGUAGE

Author:
Gerald Otter

Committee:
Dr. Ir. Jan Kuper
Ir. Christiaan Baaij
Dr. Ir. Philip Hölzenspies
Ir. Bert Molenkamp

Computer Architecture for Embedded System (CAES)
Faculty of EEMCS
University of Twente

August 15, 2013

Abstract

As shown through the development of the functional hardware description language CλaSH, functional languages are well suited to describe synchronous hardware. Functional languages allow creation of extensive type systems. The type systems of functional languages can be used to verify vastly different properties of expressions. Even though CλaSH is not the first functional hardware description language, none of the existing functional hardware description languages conscript the aid of the type system to verify properties specific to hardware designs. Even though hardware description languages allow verification of certain properties during simulation, static analysis through the type system is only performed similarly to general purpose languages.

In this thesis, a type system is developed which allows verification of temporal properties of synchronous hardware specifications. Aside from providing a formal definition of the type system through typing rules and grammar, an implementation is discussed which shows the practical feasibility of expressing temporal behaviour through the type system. We show that descriptions of temporal behaviour through the type system leads to concise specifications, where the computational aspect of hardware definitions is separated from the temporal aspect. We provide an analysis of the constraint-based approach used to create the type system, and determine what the pitfalls are when trying to include time in the type system.

From the provided analysis we conclude that expressing time as part of the type system certainly has its advantages. Specifications including time-dependent behaviour can be made more concise, though it does come at a cost of some impredictability when creating compositions. However, the constraint-based type system discussed is not ideal for reasoning about the properties of time-dependent specifications, and is difficult to extend. As a result, more research is definitely needed in order to more completely asses the possibility of expressing time through the type system.

Acknowledgements

This thesis would not have been possible without the help of many. Foremost I would like to thank my first supervisor Jan Kuper. Aside from ample constructive criticism with regards to the content of this thesis, he has been paramount in guiding me through the forest of newly aquired knowledge. Not only that, he has also generously given me personal advice when I needed it, even it was not always what I wanted to hear.

I would also like to thank Christiaan Baaij. Christiaan has diligently read most of the texts I have produced over the past six months, which were admittedly difficult to interpret most of the time. Without his help, even when disregarding CλaSH's existence, this thesis would not be what it is today. His experience with code generation and hardware synthesis has provided me with the necessary criticism to relate my ideas to practice. Without his capacity to point out errors in logic and reasoning, the text before you would certainly be more difficult to decipher.

I feel lucky to have had Philip Hölzenspies join in providing me with criticism during the final stretch of writing this thesis. Without his help and strong criticism, the type system might not have existed at all, let alone in its current incarnation. His advice showed me the value of my work, regardless of its results.

Over the past ten years, many fellow students have influenced the way I see the world and react to it. In particular I would like to thank Mark de Ruiter, for the interesting conversations we had, even though we perhaps should have been working on our theses instead. In that same vein, I would like to thank the people of the CAES group, for providing a fun and relaxed working environment, although I could have used a less relaxed environment at times. I also thank my good friends from across the globe for their continuing support.

Finally, I thank my immediate family. Thank you for supporting me, regardless of the goals I set for myself. Thank you, my dearest mother and brother, and despite everything, my loving father.

Contents

Chapter 1

Introduction

Hardware description languages are used to describe digital logic and electronic circuits. Time is a primary attribute of hardware, and as such Hardware Description Languages (HDLs) such as Very High Speed Intergrated Circuit (VHSIC) Hardware Description Language (VHDL)[33] and Verilog[44] have the ability to express behaviour as a function of time. Even though VHDL and Verilog have the ability to specify behaviour as a function of time, they do *not* have the ability to verify the correctness of specifications which include time-dependent behaviour. Verification of VHDL code which introduces time-dependent behaviour is done manually via simulation. After specification of functionality, the specification is simulated in order to test whether or not the resulting hardware representation behaves as intended by the designer. Even though verification through simulation can be automated by languages such as Property Specification Language (PSL)[16], verification still solely depends on the input from the designer.

Functional programming languages such as Haskell[26], ML[32] and many others, have traditionally been used in areas where verification is important. The functional paradigm uses side-effect free functions, i.e. the result of a function only depends on the argument(s) of the function. These functions are called "pure" functions. Such functions can be represented using a formal system called the λ-calculus. In pure functions, results may only depend on the argument of a function, without modifying the arguments in the process. Compositions of pure functions are easier to reason about as a result, as functions do not affect each other aside from through their direct argument(s) and result(s). In functional languages, the type system is used to reason about the effects of function application and abstraction. The type system provides the proof that certain expressions are indeed well-typed, or conversely, provide the proof that certain expressions are not well-typed. Expressions which are not well-typed are rejected by the compiler, as correct behaviour cannot be guaranteed.

Functional HDLs such as CλaSH[6,28], the Formal System Design (ForSyDe)[42] language and Lava[9] can transform a description in the form of a functional language to hardware, often by first transforming the functional specification to VHDL. The translation to VHDL is made in order to create actual hardware, by using the synthesis tools that already exist for VHDL. Although creation of hardware directly from functional descriptions is possible, the implementation costs are high enough to warrant usage of an intermediate language such as VHDL. Unlike traditional HDLs, functions are considered values in functional HDLs. Even though functions do not have straightforward bit-representations, first-class functions are certainly useful in the domain of hardware description, as demonstrated by CλaSH, ForSyDe and Lava. As functions are values, the type system used in CλaSH and other functional HDLs also defines the types of functions. Since the types of functions and the type of data used by functions is known at compile-time, guarantees can be given about the correctness of expressions in these languages. The type system of CλaSH allows polymorphism and higher-order functions, which enables increased code-reuse and greater conciseness than traditional HDLs.

The most common method to reason about time-dependent behaviour in hardware design is through synchronous hardware design. Synchronous hardware uses two elements, namely memory elements and combinational logic. The memory elements in a circuit are updated in synchrony according to a clock, making its behaviour time-dependent. This contrasts with combinational logic, which has time-independent behaviour. The synchronous approach makes it possible to more easily reason about concurrency, as time-dependent behaviour is limited to certain time-frames, called clock cycles. Nevertheless, current mainstream HDLs do not support verification of compositions of components which have time-dependent behaviour. Synchronous languages, such as Lustre[22] and Esterel[8], do support verification of time-dependent compositions. However, current synchronous languages do not give the developer insight into how this verification is performed, nor explicitly show the time-dependent behaviour of components in a concise manner. This is different for many functional (hardware description) languages, where the type system is often an integral part of the language. The type system of Haskell for instance, gives the developer an intuitive interface through which properties of programs are made explicit. These properties are then used in the verification process, which determines whether the stated properties of programs actually hold.

In this thesis, we aim to extend the existing functional HDL CλaSH with verification mechanisms similar to those used in synchronous languages. In short, the question permeating through the pages of this thesis is:

> "How can we express time-dependent behaviour of synchronous hardware through the type system."

To do so, we create a type system which, as opposed to existing synchronous languages, allows us to reason about time-dependent behaviour using the λ-calculus. As naming would suggest, synchronous languages are especially useful for specifying synchronous hardware designs. Synchronous hardware design, where memory elements are updated synchronously via a clock, makes it possible to reason about the timing characteristics of hardware components, without needing to take all the physical details (such as the propagation delay) into account. CλaSH already enables the designer to specify such time-dependent behaviour through specification of sequential logic. Memory elements are updated according to a clock, while combinational logic produces results immediately when provided with input(s). Even though function application is used in CλaSH to compose combinational logic, the same operation cannot be used to compose sequential logic. To compose pure functions with stateful functions, a formalism in the Haskell language called "arrows"[25] is used[20] in CλaSH. The use of arrows requires extensive knowledge of functional programming, which is not very common for hardware designers. Aside from extending CλaSH with verification of time-dependent behaviour, we also introduce an easy to use method to compose sequential logic with combinational logic.

1.1 Scope

To be able to confidently answer the research question posed in the previous section, we set the following objectives for this research:

1. To investigate how time can be represented in the type system from a syntactical point of view.
2. To investigate how temporal effects within a single clock domain can be expressed as part of the type system.
3. Providing formal semantics of the type system through which type soundness may be proven. Proving type soundness is out of the scope of this thesis however.
4. To develop an implementation, which shows the practical feasibility of expressing time as part of the type system.

1.2 Thesis Structure

As part of this thesis a comparison between CλaSH and two other languages is conducted, namely Lustre[22] and ForSyDe[42]. As the comparison has no direct relation to the objectives of this thesis, it is added as an appendix. First, various form of the λ-calculus are discussed first to provide the necessary background information. Second, we introduce the syntax of our extension to CλaSH through examples, after which we show how the time-dependent behaviour of compositions is inferred by the type system. After doing so, we introduce typing rules to formalise the relation between types and expressions of the underlying λ-calculus. Next, the implementation of a prototype implementation of the type system is discussed. This prototype implementation is slightly different from the type system discussed earlier. Afterwards we discuss the results of our work in the conclusion, before finally ending with a discussion of future work. Finally, we end this thesis with a conclusion.

Chapter 2

Background

In this chapter, we briefly present the background needed to read the remainder of this thesis. We briefly discuss various forms of the λ-calculus, which we use to reason about functional hardware descriptions. First, we discuss the untyped λ-calculus. Second, we discuss the addition of a simple type system, leading to the simply typed λ-calculus. Finally, we discuss the type system of Damas-Milner, which includes polymorphism and constraints, both of which are important for the development of our own type system.

2.1 The untyped λ-calculus

The untyped λ-calculus is a minimal algebraic structure, which is used to represent function abstraction and function application. Within the untyped λ-calculus, every value is considered a function. Church has shown through Church-numerals that numbers can be encoded in terms of function application and abstraction. Function application and abstraction also make it possible to encode booleans, pairs and lists. All this is possible despite the untyped λ-calculus comprising of just three terms:

- variable
- abstraction
- application.

These terms can be used to create expressions as indicated by the following grammar:

$$
\begin{array}{llll}
\mathrm{e} & ::= & & \textit{expressions:} \\
 & | & \mathrm{x} & \textit{(variable)} \\
 & | & \lambda \mathrm{x.e} & \textit{(abstraction)} \\
 & | & \mathrm{e\ e} & \textit{(application)}
\end{array}
$$

Using the above grammar, we can define functions through abstraction, and apply functions to other functions. For instance, the identity function is a combination of abstraction and a variable: $\lambda x.x$. Application to an expression e then allows a reduction step: $(\lambda x.x)e$ can be reduced to simply e. This reduction step is called β-reduction, which represents the computational aspect of the λ-calculus. Various reduction strategies exists such as *normal order reduction*, *call-by-name*, *call-by-value* and *call-by-need* which specify the exact conditions under which a reduction can take place. For a more detailled account see "The lambda calculus: Its syntax and semantics"[7]. Whenever an expression can be reduced, it is called a redex, short for *reducible expression*. Whenever an expression cannot be reduced, it has reached its *normal form*. The normal form of an expression depends on the reduction strategy used.

We can imagine expressions as representing a tree structure. For instance, we can apply the identity function to the identity function as expressed by $(\lambda x.x)(\lambda y.y)$. This expression represents a tree of function abstraction and application, as shown by figure 2.1. This structure is called the abstract syntax tree.

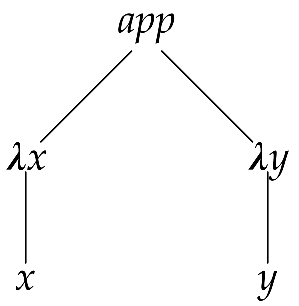

Figure 2.1: Tree structure of the expression $(\lambda x.x)(\lambda y.y)$

The untyped λ-calculus is a rewriting system. Redexes are rewritten by replacing free occurences of variables in an expression with another expression. For instance, the expression $e = (\lambda x.e_1)\ e_2$ can be rewritten as $e = [x \mapsto e_2]e_1$, where $[x \mapsto e_2]$ replaces all free occurences of x in e_1 with e_2. A free occurence of a variable is one which is not bound by any abstraction. In e, the variable x used in e_1 refers to the binder λx, and as such may be replaced by e_2 when applying the substitution $[x \mapsto e_2]e_1$. When replacing free occurences of x in e_1, we have to be careful of *shadowing*, where e_1 itself contains an abstraction of x again. The inner λ-abstraction "shadows" the outer λ-abstraction. As a result, not every occurence of x may be replaced within e_1, but only the free occurences of x.

As shown, the untyped λ-calculus provides a minimal amount of structure through its grammar. In the next section we discuss the simply-typed λ-calculus, which provides additional structure through its *type system*.

2.2 Simply-typed λ-calculus

Within the untyped λ-calculus, a number of expressions are considered valid, but when reduced using β-reduction will never reach a normal-form. A normal form is a state where an expression can no longer be rewritten; the computation is finished. For instance, the expression $\Omega = (\lambda x.xx)(\lambda x.xx)$ will never evaluate to a normal-form. The expression $Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$, named Curry's Y-combinator, allows us to define basic recursion, which also has the ability to never evaluate to a normal-form.

To disallow these non-normalising functions, the simply typed λ-calculus introduces a type system, where every term has a well-defined type. The simply typed λ-calculus is strongly normalizing. Strongly normalizing implies that every valid expression will reduce to a normal form without exception. In practice, this is often considered too strict, which is why a fixpoint-combinator can be introduced as a *primitive*. A primitive is, like the terms of the untyped λ-calculus, part of the definition of the language.

Since every valid expression reduces to a normal form, every valid expression is known to be terminating, given finite inputs. As a result, the simply-typed λ-calculus is *not* turing-complete. We introduce the grammar of the simply-typed λ-calculus below, or $\lambda^{\rightarrow}$ for short.

Definition 1: Grammar of $\lambda^{\rightarrow}$

e	::=		*expressions:*	τ	::=		*types:*
	$\mid$	x	*(variable)*		$\mid$	$\tau \rightarrow \tau$	*(function type)*
	$\mid$	λ x:τ.e	*(abstraction)*		$\mid$	T	*(base type)*
	$\mid$	e e	*(application)*				
	$\mid$	c	*(constant)*	T	::=		*base types:*
	$\mid$	**let** x = e **in** e	*(let binding)*		$\mid$	Bool	*(Boolean type)*
					$\mid$	Int	*(Integer type)*
c	::=		*primitive literals*				
	$\mid$	true	*(Boolean true)*	Γ	::=		*contexts:*
	$\mid$	false	*(Boolean false)*		$\mid$	$\emptyset$	*(empty context)*
	$\mid$	n $\in \mathbb{Z}$	*(Integer literals)*		$\mid$	Γ,x:T	*(term variable binding)*

This definition shows the addition of a type τ to the abstraction expression. Variables introduced through λ-abstraction need to have a type annotation, in order for the system to be able to construct the correct function type. Types have one constructor, namely $\rightarrow$, which constructs a function type from two other types. Types need to be finite, meaning construction of a type must always terminate. As a result, every type must, at some point, use base types in its definition.

$\lambda^{\rightarrow}$ can not express polymorphism. For instance, the general form of the identity function in the untyped λ-calculus ($\lambda x.x$), is not valid in $\lambda^{\rightarrow}$, as x must have a type. Whenever x has a type, such as in $\lambda x : Int.x$, we can derive the type of the expression. The λ-abstraction introduces the $\rightarrow$ constructor, which allows us to create function types using base types. The function $(\lambda x : Int.x)$ has type $Int \rightarrow Int$, since the type of the result of the identity function has the same type as the function argument. We will revisit polymorphism in section 2.3.

The type system of $\lambda^{\rightarrow}$ is not complete without typing rules. Typing rules express the relation between types and expressions, and in particular define which expressions are *valid* and which are not. The nature of the typing rules for $\lambda^{\rightarrow}$ also makes it possible to *(re)construct* the the types from (partial) type information. We will explain the typing rules of $\lambda^{\rightarrow}$ next, where we we also explain the meaning of the context Γ as introduced by definition 1.

Typing rules

In this section we present the typing rules of $\lambda^{\rightarrow}$, using the commonly used Gentzen style of the sequent calculus[19]. For every valid expression e in $\lambda^{\rightarrow}$, a single rule expresses the relation between the expression and the type of the expression. Using these rules, a proof can be constructed for any valid expression e in $\lambda^{\rightarrow}$, by (de)-constructing the expression using the typing rules. When a proof cannot be constructed, then the expression is considered invalid.

We use τ and ρ to range over types. As is customary in type theory, we use Γ to represent the *context* or *type environment*. Γ is a set of pairs, which expresses the relation between *variable names* and *types*, as shown by definition 1. The type environment is needed in order to make judgements based on the context an expression is used in. For instance, the expression $x + 1$ cannot be typed when nothing is known about x. We use Γ to provide the context in which we can make a statement about $x + 1$. If $\Gamma = \{x : Int, + : Int \rightarrow Int \rightarrow Int\}$ and we assume that the literal 1 has type *Int*, then the entire expression would have type *Int* as well. However, we can only make this judgement when we know the context Γ in which the statement $x + 1$ is used.

In the typing rules, statements about the context are made explicit by use of the *turnstile* ($\vdash$) symbol. For instance, $\Gamma \vdash x : \tau$ indicates that the type of x is τ, which is derived from the context Γ. The turnstile indicates derivability; we can derive $x : \tau$ from Γ. As shown by the typing rules of definition 2, typing rules consist of of two parts: the *premise* of the rule, and the *conclusion*. The premise is separated from the conclusion by a horizontal line, with the premise(s) above the line, while the conclusion is written below the line.

Definition 2: Typing Rules for $\lambda^{\rightarrow}$

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\ \text{T-Var} \qquad \frac{}{c : T}\ \text{T-Const}$$

$$\frac{\Gamma, x : \rho \vdash e : \tau}{\Gamma \vdash \lambda x : \rho . e : \rho \rightarrow \tau}\ \text{T-Abs} \qquad \frac{\Gamma \vdash e_1 : \rho \rightarrow \tau \quad \Gamma \vdash e_2 : \rho}{\Gamma \vdash e_1\, e_2 : \tau}\ \text{T-App}$$

$$\frac{\Gamma \vdash e_1 : \rho \quad \Gamma, x : \rho \vdash e_2 : \tau}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \tau}\ \text{T-Let}$$

As shown, both the rules T-Var and T-Const are trivial. Given a literal “true”, “false” or a natural number, we can conclude that it would have either the type *Bool* or *Int*. The rule T-Var is slightly less trivial. The conclusion $\Gamma, x : \tau \vdash x : \tau$ defines that, whatever the context Γ is, when we add $x : \tau$ to it, we can conclude that x does indeed have type τ. While it may seem that both of these rules can always be applied because there is a lack of a premise, it is really only the case for T-Const. We can only derive $x : \tau$ in T-Var when $x : \tau$ is part of the context Γ. This means that the statement $x + 1$, with the empty context $\varnothing$, does not have a derivable type. The variable x must be introduced by λ-abstraction, as defined in the following rule, T-Abs.

The rule T-Abs shows the relation between abstraction and the function type constructor $\rightarrow$. The context Γ is extended with $x : \rho$ when deriving the expression e where x is abstracted from. From the extended context Γ we then derive e has type τ. When the entire premise $\Gamma, x : \rho \vdash e : \tau$ holds, we derive from the *non-extended* Γ, that $\lambda x : \rho . e$ has type $\rho \rightarrow \tau$. With the T-Abs rule, we show how abstraction *introduces* a function type.

Conversely, the T-Abs rule eliminates the $\rightarrow$ type constructor. The rule T-App has two premises, where both e_1 and e_2 are derived to have the types $\rho \rightarrow \tau$ and ρ respectively. Here, when we mention ρ, we mean the *same* type in both premises. This means that the T-Abs rule can only be applied when the left-most expression e_1 of application has a function type. Moreover, the type of the right-most expression e_2 must match the argument of the type of e_1. When these premises hold, we may derive $e_1\, e_2 : \tau$ from the same context Γ.

Finally, the T-Let rule defines two premises must hold, in order to derive **let** $x = e_1$ **in** e_2 to have type τ. First, we use the context Γ to derive the type of e_1 to be ρ. Secondly, we add the type binding $x : \rho$ to Γ in order to derive the type of e_2. The type of x must be added to Γ, as x is (probably) referenced within e_2, so in order to derive the type of e_2, the context must have that information available. When both these premises hold, then the entire let-statement has type τ. As the simply typed λ-calculus has no polymorphism the use of the let-binding is limited. In the next section we give an introduction to let-polymorphism and constraint-based typing.

2.3 Polymorphic λ-calculus

In the simply-typed λ-calculus, no polymorphic functions can be defined. A polymorphic function can be seen as a *family* of functions, of which an instance is chosen depending on the context which it is used in. As a trivial example, consider the identity function $\lambda x.x$. The identity function in the untyped λ-calculus does not have a notion of types, and can therefor be applied to any value. This is not the case in the simply-typed λ-calculus, but we can define the set of all functions which *share* the definition of the identity function under type-erasure. Both the functions $(\lambda x : \mathit{Int}.x) : \mathit{Int} \to \mathit{Int}$ and $(\lambda x : \mathit{Bool}.x) : \mathit{Bool} \to \mathit{Bool}$ share the same definition. While this makes intuitive sense, it is not a trivial problem for a type system to solve. Wand defines[46] the problem of polymorphism as "Given a term of the untyped λ-calculus, to find all terms of the typed λ-calculus which yield the given term when the type information on bound variables is deleted." In effect, when we have an expression of the untyped λ-calculus, we try to find the *family* of valid types belonging to that expression.

To show how this is done, we first extend the grammar of $\lambda^{\to}$. Secondly, we introduce polytypes, which allow us to reason about polymorphic types. Using polytypes, we define what constitutes the most general type of an expression, known as the principal type. Finally, using principal types and polytypes, we show how the process of type-checking is split in two by using *constraints*. To do so, we focus on the well-known "Damas-Milner"[13] type reconstruction algorithm, which is also known as the "Hindley-Milner" or "Damas-Hindley-Milner" type reconstruction algorithm.

Principal Types

We focus on principal types first. We extend the grammar of $\lambda^{\to}$ with type variables. This leads to the grammar we call λ_{DM}, which is shown by definition 3. The rest of the grammar is identical to $\lambda^{\to}$ from definition 2. As we are introducing polymorphism to the simply typed λ-calculus, we need a mechanism to introduce polymorphic functions. For this we use *let-polymorphism* as introduced[32] by Milner. There, let-bindings are used to introduce polymorphic functions, which can be used multiple times within the body of the let-binding, regardless of possible conflicting types between each instance. We discuss the idea of let-polymorphism in greater detail later in this section.

Definition 3: Grammar of λ_{DM} (Extended from definition 1).

$$\begin{array}{lcll} \tau & ::= & \alpha & \textit{(type variable)} \\ & | & \ldots & \\ \\ e & ::= & \lambda x.e & \textit{(abstraction)} \\ & | & \ldots & \end{array}$$

Aside from using α for type variables, we also use β when two type variables are distinctly different. Type variables in λ_{DM} are *monomorphic*, meaning a type variable can only refer to a single type. Moreover, without further modification, all type variables are *free* variables, as they are not bound to abstraction like term variables in the λ-calculus. Type variables are used in *polytypes*[1], shown by definition 4. Polytypes make it possible to introduce *polymorphism* using monomorphic type variables.

Definition 4: Polytypes

A polytype σ is defined by universal quantification over a finite number of type variables:

$$\sigma = \forall \alpha_0, \alpha_1, \ldots, \alpha_n.\tau \quad (n \geq 0)$$

, where the type variables α_0 to α_n are considered bound within τ.

Polytypes consist of universal quantification over type variables. The type variables used by universal quantification are bound within the polytype σ. In the polytype $\sigma = \forall\varnothing.\tau$, the polytype σ is equal to the type τ. Whenever a substitution of the form $[\alpha \mapsto \rho]$ is applied to a polytype σ, the resulting type is called an *instance* of σ. Similarly, a *set* of substitutions S may be applied to σ.

We can use polytypes to define polymorphic functions. For instance, the identity function $\lambda x.x$ is polymorphic when it has the associated polytype $\sigma = \forall\alpha.\alpha \to \alpha$. When applied to a value, for instance $1 : Int$, the substitution $[\alpha \mapsto Int]$ can be applied to σ, leading to the type instance $\sigma = \forall\varnothing.Int \to Int$.

As shown, we can create an instance of a polytype by applying a substitution to a polytype. Given the identity function, there exist many different substitutions to create type instances of the associated polytype. For instance, the identity function, when

[1] Some authors, such as Milner, prefer to use the term *type scheme*. Here, we use the term polytype.

associated with polytype $\sigma = \forall \alpha . \alpha \rightarrow \alpha$, can be instantiated using the substitutions $[\alpha \mapsto \mathit{Bool}]$, $[\alpha \mapsto \mathit{Int}]$, or even $[\alpha \mapsto (\mathit{Int} \rightarrow \mathit{Bool})]$ and other variations. Given that many substitutions exists, the principal type is defined informally as the type which is least constricting, but is still a valid type. That is, the principal type τ of an expression e, is the type to which any set of substitutions may be applied, assuming the set of substitutions lead to a valid type τ' of e.

Definition 5: Principal Types

Given a type τ, associated with a polytype $\sigma = \forall \alpha_0, \alpha_1 \ldots \alpha_n . \tau$ of an expression e, then τ is the principal type of e iff:

1. the type τ holds under the empty context, that is, $\varnothing \vdash e : \tau$;
2. $\forall \tau' . \varnothing \vdash e : \tau' \Rightarrow \exists S . \tau' = S\tau$, that is, for all valid types τ' of e, there exists a set of substitutions S, such that τ can be turned into τ'.

In some of the literature the binary relation $\sqsubseteq$ is defined as well. This relation is used to express an ordering in polytypes. Given two polytypes $\sigma' \sqsubseteq \sigma$ defines σ' to be at least as constricting as σ. For instance, given $(\sigma' = \forall \alpha . \mathit{Int} \rightarrow \alpha)$ and $(\sigma = \forall \alpha, \beta . \beta \rightarrow \alpha)$, the relation $\sigma' \sqsubseteq \sigma$ holds. Similarly, $\sigma \sqsubseteq \sigma$ holds, as σ is at least as constricting as itself.

Definition 6: Typing Rules for λ_{DM}

$$\frac{\Gamma(x) = \sigma}{\Gamma \vdash x : \sigma}\ \text{DM-Var} \qquad \frac{}{\varnothing \vdash c : T}\ \text{DM-Const}$$

$$\frac{\Gamma, x : \rho \vdash e : \tau}{\Gamma \vdash \lambda x : \rho . e : \rho \rightarrow \tau}\ \text{DM-Abs} \qquad \frac{\Gamma \vdash e_1 : \rho \rightarrow \tau \quad \Gamma \vdash e_2 : \rho}{\Gamma \vdash e_1\, e_2 : \tau}\ \text{DM-App}$$

$$\frac{\Gamma \vdash t : \sigma \quad \alpha \notin \mathit{ftv}(\Gamma)}{\Gamma \vdash e : \forall \alpha . \sigma}\ \text{DM-Gen} \qquad \frac{\Gamma \vdash e : \sigma \quad \sigma' \sqsubseteq \sigma}{\Gamma \vdash e : \forall \alpha . \sigma'}\ \text{DM-Inst}$$

$$\frac{\Gamma \vdash e_1 : \sigma \quad \Gamma, x : \sigma \vdash e_2 : \tau}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \tau}\ \text{DM-Let}$$

Using principal types, we define the typing rules of the Damas-Milner type system in definition 6. Most of the rules of the Damas-Milner type system are straightforward. The rules DM-Abs and DM-App are exactly the same as the rules T-Abs and T-App from page 9. However, instead of binding the type of a variable x to τ in the rule DM-Var, the type of a variable is bound to a polytype. This is a conservative extension, as polytypes allow expression of not-polymorphic types. This does *not* mean higher order polymorphic functions are allowed, as abstraction has no notion of a polytype. Instead, polymorphic functions can only be introduced by the rule DM-Let. DM-let defines that, given an expression e_1, associated with polytype σ, e_1 may be used in e_2, but only if it leads to e_2 having a monotype τ. This is called *let-polymorphism*, as introduced earlier.

Two other rules are needed in order to both *introduce* and *eliminate* polymorphic type variables. The rules DM-Gen and DM-Inst do just that. The rule DM-Gen allows introduction of universal quantification, but only if the type variable α used in universal quantification is bound in the context Γ. This is indicated by the meta-function *ftv*, which determines the *free type variables* of its argument. The rule DM-Inst allows instantiation of a polytype (e.g. apply a valid substitution to a polytype), provided the new polytype σ' is at least as constricting as the original polytype σ. By carefully allowing polytypes in only part of the typing rules of DM, polymorphic types can *never* be substituted with other polymorphic types[37]. This form of polymorphism is called predicative polymorphism, as opposed to the impredicative polymorphism of for instance System F[21].

Type-checking using Polytypes

Principal types make it difficult to check the types of expressions using a syntax-directed approach. In a syntax-directed approach, type-checking can be done *solely* by applying typing rules. When no typing rule applies, the expression being checked does not have a valid type. When the entire expression is checked, the entire expression has a valid type, provided the type system is sound.

However, to type-check expressions containing polytypes, that approach is not as straightforward. It is difficult to express the concept of principal types, and how to derive a set of substitutions for that principal type, using a syntax-directed approach. The typing rules of Damas and Milner "do not provide an easy method for finding, given a context Γ and expression e, a polytype σ such that $\Gamma \vdash e : \sigma$."[13] Milner provides the W algorithm to find a polytype σ, given a context Γ and an expression e. The algorithm, which is presented in full detail later, uses Robinson's Unification[39] algorithm to find the most general type possible for every expression and declaration. Algorithm W was later reformulated by Damas and Milner in "Principal Type-schemes for Functional Programs"[13]. Essentially, algorithm W interleaves generation and unification of *type constraints* to find the most general type of a given expression.

We now discuss algorithm W (definition 7). Given a context Γ and expression e, the algorithm returns a set of substitutions S and a type τ, such that $S\Gamma \vdash \tau$. Depending on the expression, the algorithm defines a different procedure to derive the principal type of the expression. In the case of a variable, all type variables bound in the polytype of

Definition 7: Algorithm W

$$
\begin{aligned}
W(\Gamma, x) &= \text{let } \Gamma(x) = \forall \alpha_1, \ldots, \alpha_n . \tau' \\
& \text{ and } \beta_i \notin \Gamma \\
& \text{ in } (\{\}, [\beta_i / \alpha_i]\tau') \\
W(\Gamma, e_1\, e_2) &= \text{let } (S_1, \tau_1) = W(\Gamma, e_1) \\
& \text{ and } (S_2, \tau_2) = W(S_1\Gamma, e_2) \\
& \text{ and } V = \mathit{unify}(\{S_2\tau_1 = \tau_2 \rightarrow \beta\}) \\
& \text{ and } \beta \notin \Gamma \\
& \text{ in } (V \cup S_2 \cup S_1, V\beta) \\
W(\Gamma, \lambda x.e) &= \text{let } (S, \tau) = W((\Gamma, x : \beta), e) \\
& \text{ and } \beta \notin \Gamma \\
& \text{ in } (S, S\beta \rightarrow \tau) \\
W(\Gamma, \textbf{let } x = e_1 \textbf{ in } e_2) &= \text{let } (S_1, \tau_1) = W(\Gamma, e_1) \\
& \text{ and } (S_2, \tau_2) = W((S_1\Gamma, x : \forall \alpha_1, \ldots, \alpha_n . \tau_1, e_2) \\
& \text{ and } \{\alpha_1, \ldots, \alpha_n\} = \mathit{ftv}(\tau_1) \backslash \mathit{ftv}(S_1\Gamma) \\
& \text{ in } (S_2 \cup S_1, \tau_2)
\end{aligned}
$$

x are replaced by fresh type variables. Type variables are replaced using α-conversion, and as such the set of substitutions returned is empty.

When one expression e_1 is applied to another expression e_2, the set of substitutions S_1 and type τ_1 of e_1 is derived by applying the W algorithm recursively. The resulting substitutions are then applied to Γ, in order to derive the set of substitutions S_2 and most general type τ_2 of e_2. Finally, the unification algorithm is used to find a set of substitutions V, such that $S_1\tau_1 = \tau_2 \rightarrow \beta$, where β is a fresh type variable. The unification algorithm, which is not shown here, accepts a constraint of the form $\tau = \tau'$, and results in a set of substitutions such that $V\tau = \tau'$, if such a set exists. The constraint $S_1\tau_1 = \tau_2 \rightarrow \beta$ makes it clear that the type of e_2 has to agree with the type of the first argument of e_1. If this is the case, then $e_1\, e_2$ is defined to have type β, which is subject to the resulting substitutions V from the unification algorithm.

In the case of a λ-abstraction, a fresh type variable β is introduced as the type of the argument x. The context Γ, together with the body e of the λ-abstraction, is supplied to W to define the set of substitutions S and principal type τ, such that the type of $\lambda x.e$ is $S\beta \rightarrow \tau$.

Lastly, in the case of a let-binding, the most general type of the binding is found. The substitutions S_1, together with type τ_1, are then used to modify the context Γ, which is used to derive the most general type of e_2. The polytype used to append Γ to derive e_2 binds the variables which are unique in $ftv(\tau_1)$, when compared to the type variables of $ftv(S_1\Gamma)$. If this were not done, then referencing x in e_2 would create fresh variables for type variables which are already substituted in Γ.

2.4 Conclusion

The type systems discussed in this chapter show how we can reason about properties of (certain expressions of) the λ-calculus. The type system of Damas-Milner shows that, in order to reason about more complex types, type-checking can be done in two steps. As a result, the type system of Damas-Milner is not complete by only giving the typing rules. The typing rules are used to (re)construct types, but do not have the ability to determine principal types. For this the W algorithm is introduced. We consider both algorithm W and the typing rules to make up the entire type system of Damas-Milner.

In the same vein, we define our own type system in two steps in the next chapters as well. Like Damas-Milner, the typing rules lead to a number of constraints. Instead of defining the constraint generation in a separate algorithm however, we use the same approach as HM(X)[34], which is considered[38] an extension of the type system of Damas-Milner. Discussion of this type system is out of the scope of this thesis however, as we have no specific interest in the general polymorphism described by the HM(X) type system.

Chapter 3

Specification of Time-dependent Behaviour

In the previous chapter, we discussed different type systems of the λ-calculus. As stated in the introduction, the primary goal of this thesis is to provide a type system, which can be used to specify and verify time-dependent behaviour. To make it easier to digest the definition of the type system, we split discussion of the type system in two parts. First, we focus on the specification of time-dependent behaviour. We do this by discussing how the type system of Haskell is currently used in CλaSH, after which we discuss how our type system can be used to more concisely specify time-dependent behaviour than currently possible. Secondly, we focus on the grammar and typing rules of our type system in the next chapter. The order in which these two chapters are read, is not very important. As such, for readers more interested in the definition of the type system, the next chapter may be read before this one.

The type systems introduced in the previous chapter were not specifically created for hardware design. To show how traditional type systems are used in the context of hardware design, we give a brief introduction as to how the type system of Haskell is currently used in CλaSH. CλaSH leverages the language Haskell, which means it adopts the syntax of the language, together with the type system and other parts of the Glasgow Haskell Compiler (GHC) in order to generate VHDL code. As such an understanding of the Haskell language is preferred in order to read this chapter, though the basic syntax of the language is discussed as well. If such understanding is limited or non-existent, O'Sullivan et al. have written[35] an excellent, freely available introduction to Haskell and its syntax.

After the brief introduction to the type system of CλaSH and how CλaSH can be used to define time-dependent behaviour, we introduce a different method of specifying the same behaviour. Following, we discuss how type reconstruction is used to derive the time-dependent behaviour of compositions. Finally, we discuss sequences, and how sequences are reasoned about in the type system.

3.1 Specification of Combinational Logic

In CλaSH, combinational logic is represented through pure functions. Like combinational logic, pure functions only affect the values they are applied to; they are side-effect free. For instance, the combinational logic of figure 3.1 is represented as a pure function.

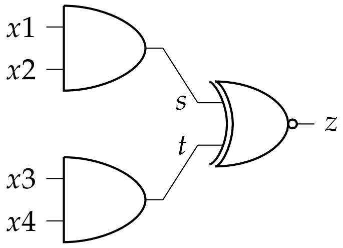

Figure 3.1: A circuit consisting of a few logic gates.

CλaSH is a structural HDL. The structure of the circuit is represented by the expression of the language. The component from figure 3.1 can be structurally defined in CλaSH through the function *combLogic*.

Code Snippet 1: Combinational logic in CλaSH (a).

```
combLogic x1 x2 x3 x4 = z
  where
    s = hwand x1 x2
    t = hwand x3 x4
    z = hwnot (hwxor s t)
```

As shown by the variables used in *combLogic* and figure 3.1, there is an immediate relationship between the code and the structure of the circuit the code represents. The defined function is named *combLogic* and accepts four arguments. These arguments are used in the **where** clause, which allows us to decompose the structure of a single component into multiple sub-components. Decomposition of the structure in smaller parts makes it easier to see the structure of the circuit. The **where** clause introduces variables *s* and *t*, which relate to values transferred between logic gates, while the variable *z* represents the output of the circuit. The definition of *combLogic* also shows us the influence

Haskell has on CλaSH. As function identifiers like *and*, *not* and *xor* are already defined by the Haskell language, CλaSH introduces different identifiers like *hwand*, *hwnot*, et cetera, for these functions.

In the definition of *combLogic*, the **where** clause creates *local* definitions. We need to be able to detect when something is defined locally, and when something is defined globally, as *s* and *t* should not exist within the same scope as *combLogic*. The definitions of *s* and *t* only make sense within the scope of *combLogic* after all. The *off-side rule* allows us to determine when a certain scope ends. In the definition of *combLogic*, the indentation of the **where** close defines that *s*,*t* and *z* belong to a different scope than *combLogic* itself.

Types in CλaSH

Functions such as *combLogic* have known types, even if we do not write them down explicitly. The type system of CλaSH can, similar to the type systems of the previous chapter, reconstruct the types of expressions. The type system of CλaSH can infer, from the definition of *combLogic*, that the function only works for values of the type *Bit*. The *Bit* type is represented by a single wire in a circuit, since a *Bit* only has two values.

Types can also be used in the definition of circuits directly. Including types in the definition increases clarity and conveys the intent of the designer to the type system. For instance, the definition of *combLogic* given earlier, is equivalent to the definition of code snippet 2.

Code Snippet 2: Combinational logic in CλaSH (b).

```
combLogic :: Bit → (Bit → (Bit → (Bit → Bit)))
combLogic x1 x2 x3 x4 = (hwnot (hwxor s t))
  where
    s = hwand x1 x2
    t = hwand x3 x4
```

The :: operator is used to inform the type system of the relation between the identifier *combLogic* and the type $Bit \rightarrow (Bit \rightarrow (Bit \rightarrow (Bit \rightarrow Bit)))$. This operation is called ascription and is used to give the type system additional information. In functional languages functions are values as well, and as such are also represented by the type system. The type constructor → constructs a function type from two other value types. In functional languages, including CλaSH, functions are first class, meaning functions are also values. In the definition of *combLogic*, the type $Bit \rightarrow (Bit \rightarrow (Bit \rightarrow (Bit \rightarrow Bit)))$

represents that, after applying *combLogic* to a valid value of type *Bit*, the function will return another function of type $Bit \rightarrow (Bit \rightarrow (Bit \rightarrow Bit))$. After the function is applied four times to valid values, the function returns a value of type *Bit*. The type constructor → is right associative. This means that a type as $Bit \rightarrow Bit \rightarrow Bit$ is interpreted as $Bit \rightarrow (Bit \rightarrow Bit)$.

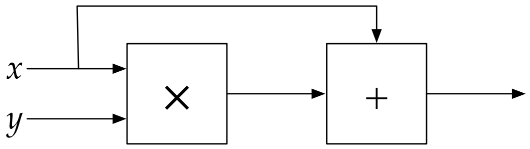

Figure 3.2: Multiplication and addition combined

Although CλaSH allows us to create compositions using combinational logic, not every composition is considered correct. When the types of a composition do not match, the composition is rejected and a type error is raised. For instance, using the multiplication operation and addition operation, we combine the two as a function named *mulAdd*. This is shown below by code snippet 3 to create the circuit of figure 3.2. We can not combine the definition of *mulAdd* and *combLogic* in *composition*, without raising a type error. As a result, the circuit of figure 3.3 does not work, as it requires the type of *z1* to be equal to both the type of *x1* from the definition of *combLogic*, as well as the type of *x* from the definition of *mulAdd*.

Code Snippet 3: Composition of combinational logic in CλaSH.

```
mulAdd :: Int → Int → Int
mulAdd x y = (x ∗ y) + x
composition z1 z2 z3 z4 = (t1, t2)
  where
    t1 = mulAdd z1 z2
    t2 = combLogic z1 z2 z3 z4
```

Ignoring function types, we can relate other value types to the number of wires used in a circuit. When these types do not match a type error is raised, which prevents us from trying to create a circuit with a mismatch in the number of wires in connections. This also explains why we left out the type of *composition* here; as a type error is raised for *composition*, the expression is not well-typed.

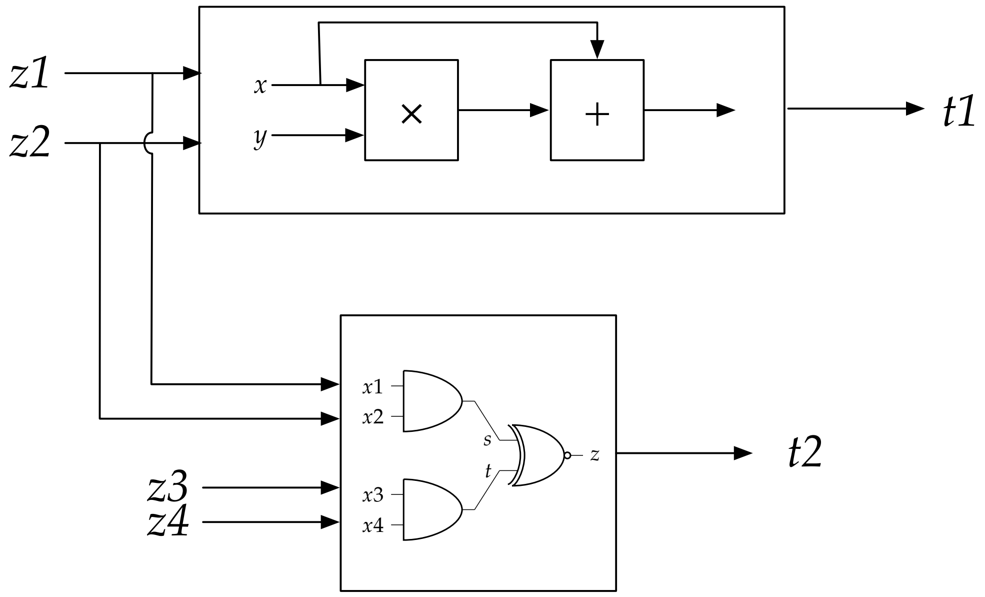

Figure 3.3: Combination of figures 3.1 and 3.2.

Polymorphism

The specification of a piece of digital logic is sometimes generic enough to be used in multiple ways. The addition operation can for instance be used to add numbers regardless of their bit-representation, at least a language perspective. When the addition operation is translated to VHDL, a proper adder may be chosen. Due to the type system, we know the type of the numbers which are to be added, and as such a specific adder can be chosen. Polymorphism makes it possible to specify and *reuse* such specifications. As an example of how code reuse is useful, consider the following code snippet, which represents the circuit of figure 3.4.

Code Snippet 4: A polymorphic specification

```
commonCode f g h x y = z3
  where z1 = f x y
        z2 = g z1
        z3 = h z2 y
```

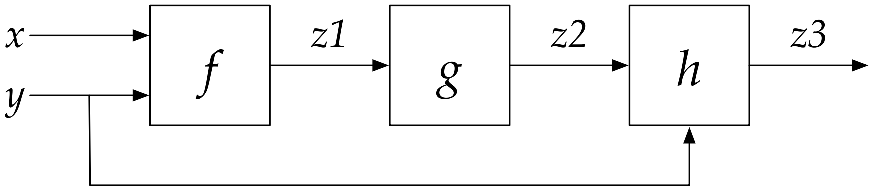

Figure 3.4: A polymorphic circuit, as specified by code snippet 4

The code of snippet 4 only specifies in what way the functions *f*,*g* and *h* interact with each other. As the definition of *commonCode* uses the functions *f*,*g* and *h* as arguments, the *commonCode* function needs to be applied to three different functions before the specification can be applied to bit-representable values. For instance, we can use *commonCode* as

```
specificCodeA = commonCode (+) id (+)
```

or as

```
specificCodeB = commonCode (*) shiftL (+)
```

, depending on the intented usage of the *commonCode* pattern.

Both of those specifications are correct. However, not every combination of functions leads to a correct specification. When we use *commonCode* as in

```
incorrectCode = commonCode (+) (+) (+)
```

, then the second $(+)$ function does not match with the function type which *g* is expected to have. This is because there exists a *relation* between the functions *f*, *g* and *h* within the context of the function *commonCode*. We use *type variables* to reason about these relationships. Like value variables, type variables are referentially transparent. For instance, the sub-expressions of *commonCode* have the following types:

```
f  :: a → b → c
g  :: c → d
h  :: d → b → e
z1 :: c
z2 :: d
z3 :: e
```

The type variables express the relation between the types of the expressions used in *commonCode*. As *f* provides the input to *g*, the type of the input to *g* is necessarily the same as the type of the output of *f*. The types of the remaining expressions are similarly limited.

Polymorphic types have a limitation however. Since polymorphic types represent a range of possible representations, we can not generate hardware from a polymorphic description. Thus, the final compilation must have fully determined types.

Combinational Logic and Time

In reality it takes time for outputs of combinational logic to become stable after a change occurs at the inputs of the circuit. However, in synchronous hardware design this delay cannot be observed. When the input changes, the output changes in the same time-frame. Even though verification of time-dependent behaviour is not useful for combinational logic, the distinction between combinational logic and sequential logic still has to be made. For this reason, the relation between inputs and outputs in combinational logic is made explicit by naming the moments at which values should exist. To do so, we introduce time variables. We use variables since we do not yet know at which specific time the input changes, but we would like to reason about the relation between input and output in terms of time.

Through the type system, we already have access to type information about the values used in expressions. From the *combLogic* definition we know that *combLogic* is a function. However, we also know that *x1* to *x4* are values with type *Bit*. Similarly, we know the output value is of type *Bit* as well. If we assume the output changes at moment *t*, then the input values must have stabilised at moment *t* as well. We represent that as part of the type ascription by annotating each datatype with the same time variable as follows:

```
combLogic :: Bit⟨t⟩ → Bit⟨t⟩ → Bit⟨t⟩ → Bit⟨t⟩ → Bit⟨t⟩
combLogic x1 x2 x3 x4 = ...
```

In the above example we have defined every value to be available at the same moment in time, as is expected of combinational logic in synchronous hardware designs. In synchronous hardware design we can always combine as much combinational logic as we wish, without having any impact on its *modelled* time-dependent behaviour. By adding more combinational logic, the time-dependent behaviour of the circuit stays the same, even though in reality, the performance suffers.

3.2 Specification of Sequential Logic

Unlike combinational logic, sequential logic can not be described directly by pure functions. Pure functions relate the output directly in terms of the input(s). In sequential logic however, the current output does not solely depend on current input(s), but also on past input(s). In this section we discuss how sequential logic is currently specified in CλaSH. Next, we introduce how we can use time variables to specify time-dependent behaviour. To do so, we focus on individual memory elements before defining a pipelined circuit. Following pipelining, we discuss type (re)construction of compositions. Finally, we discuss bounded sequences.

So far we have only shown how CλaSH represents combinational logic using pure functions. Since pure functions are side-effect free, pure functions alone cannot be used to recall information from the past. As a result, sequential logic in CλaSH is first defined as a pure function with a *specific* type of type. The pure function is then transformed into a component through a primitive function called *lift*, a process which is explained in greater detail later. In this specific type of pure function, only combinational logic is defined, while sequential logic is solely represented via its inputs and outputs.

Code snippet 5 shows the pattern which must be used to specify sequential logic. As before, there is a immediate relationship between the code presented in code snippet 5 and the circuit of figure 3.5.

Code Snippet 5: Sequential logic in CλaSH

```
sequential :: State a → b → (State a, c)
sequential (State s) x = (State s′, out)
  where s′   = ...
        out = ...
```

The definition of *sequential* shows the type of the pure function used to describe sequential logic. The first argument named *s* represents the input value from sequential logic of the *current clockcycle*. To distinguish between the inputs of sequential logic, and inputs of the component itself, the *State* keyword is used. The *State* keyword is CλaSH specific, and only has meaning in this specific type of function type. In Haskell, *State* is a normal type constructor, whereas in CλaSH it is a reserved word. To supply sequential logic with external input the second argument, named *x*, is used. Using the variables *s* and *x*, the output *out* and the value of *s* in the *next* clockcycle can be determined. As a result, the input and output are synchronised to a clock, even though the specification of sequential logic does not specify the clock signal directly. Instead, the relation between the input and the output of sequential logic is defined within the context of a single clock cycle.

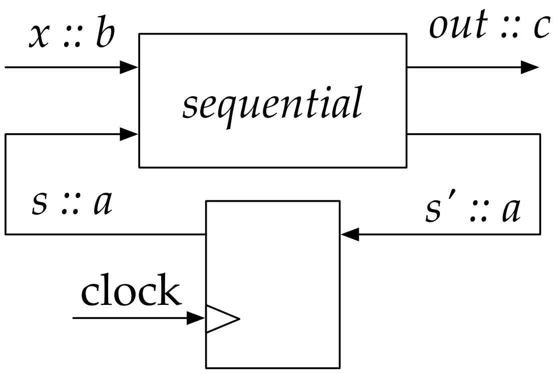

Figure 3.5: Sequential logic.

Individual Memory Elements

Although the definition of sequential logic in CλaSH is flexible enough to specify feedback, we focus on a more simplified example first. Code snippet 6 shows how a single memory element can be defined in CλaSH. Like before, there exists an immediate relationship between the code and the circuit it represents. In this example, a single type variable is enough to fully describe its behaviour.

Code Snippet 6: A single memory element in CλaSH

```
singleMem :: (State a) → a → (State a, a)
singleMem (State s) x = (State s′, out)
  where s′  = x
        out = s
```

The same behaviour is modelled more concisely in our type system. As shown, the definition of *singleMem* delays the input *x* with one cycle of the clock. Delaying a value with one clockcycle implies some form of transformation in the time domain. Instead of specifying the input and output of sequential logic separately, we directly encode this transformation in the type of the function. As shown by code snippet 7, the intent of the specification which uses time variables is clear from the type of the function alone. The input exists at time t, and the output exists one cycle later, at $t + 1$.

Code Snippet 7: A single memory element defined using a time variable.

```
singleMem′ :: a⟨t⟩ → a⟨t + 1⟩
singleMem′ x = x
```

Instead of introducing temporal transformations at the term level, we introduce such transformations as part of the type. A memory element behaves as the identity function in the value domain, and as a linear transformation in the time-domain. Variables on the term level are referentially transparent; they can be replaced by the same value throughout the expression without changing the behaviour of the function. This is true when applying a positive shift to the time-domain of a value as well. When a value exists at some point in time t, it can always be stored to be available at $t + a$, where a is any natural number. Specifying the time-dependent behaviour as part of the function type leads to a clean separation between *what* ought to be done by a function, and *when* it ought to be done.

Memory elements defined in this way have limitations however. Memory elements which have some conditional behaviour cannot be expressed solely by transformations in the time domain. To describe such memory elements, either a language primitive needs to be added, or the needed type of memory element should be detected from contextual information during translation to VHDL or during hardware synthesis.

Similarly to CλaSH, the clock is not explicitly defined. Even so, a relation between the time variable t, the expression $t + 1$ and the clock exists, as shown by figure 3.6.

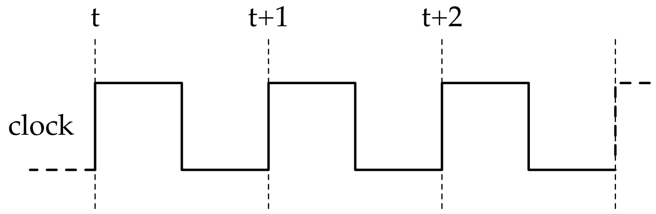

Figure 3.6: Relation between the clock and time variables

Time variables represent specific moments in "real" time. However, time is discrete and as such we cannot distinguish between two moments in time which occur between t and $t + 1$. In synchronous logic, a period of time needs to pass in order for signals to stabilise. The moment in time t represents the first moment when its input is stable. The time expression $t + 1$ as part of the output defines when this output will stabilise, in relation to its input.

Similarly to the definition of *singleMem*, we can define two memory elements by changing the offset, as shown by code snippet 8.

Code Snippet 8: Two memory elements defined using a time variable.

$$\begin{aligned}&\mathit{doubleMem} :: a\langle t\rangle \rightarrow a\langle t+2\rangle\\&\mathit{doubleMem}\ x = x\end{aligned}$$

We compose *doubleMem* with *singleMem* to create *tripleMem*. The type system can infer the correct type of the composition. As a result, the type of

$$\mathit{tripleMem} = \mathit{doubleMem} \circ \mathit{singleMem}$$

is inferred to be $a\langle t\rangle \rightarrow a\langle t+3\rangle$.

Even though the hardware specifications so far have always used the same time variable, time variables are not shared between specifications. Time variables only exist within the scope of the specification where they are introduced. As a result, the time variable used in *singleMem* is different from the time variable used in *doubleMem*, even though they use the same identifier.

Pipelining

The definition of a single memory element is straightforward in CλaSH as is, so it may be hard to see exactly how adding time-dependent behaviour can be advantageous based on the examples from the previous section. To show how adding time-dependent behaviour to the type system can lead to more concise specifications, consider the following circuit.

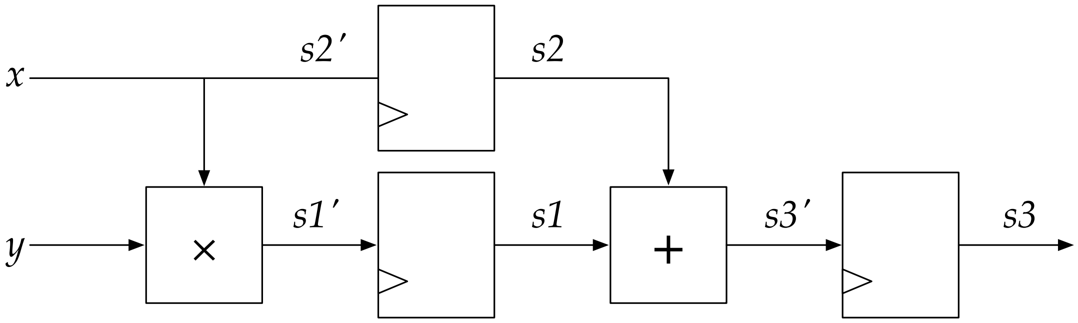

Figure 3.7: A pipelined circuit

CλaSH is able to represent the circuit as a single pure function as shown by code snippet 9. The *State* keyword, together with the structure of the function type, allows us to transfer values between two clockcycles. The variables $s1, s2, s3$ represent the outputs from the registers, as opposed to the variables $s1', s2', s3'$, which represent the inputs of the registers. This means that, given a definition of $s1'$ at some clockcycle t, then $s1$ has the same value as $s1'$ at some clockcycle $t + 1$. This relation is straightforward, but when many registers are added, the function definitions soon become complex.

Code Snippet 9: Definition of a pipeline in CλaSH.

```
pipeline :: State (Int, Int, Int) → (Int, Int) → (State (Int, Int, Int), Int)
pipeline (State (s1, s2, s3)) (x, y) = (State (s1′, s2′, s3′), s3)
  where s1′ = x ∗ y
        s2′ = x
        s3′ = s1 + s2
```

Fortunately, CλaSH makes it possible to create better structural descriptions by lifting pure functions such as *pipeline* to the component level. The component level, which is based on arrows[24], allows us to interpret the pure function *pipeline* as an impure function with embedded state. Although a complete introduction to component-based programming in CλaSH is out of the scope of this thesis, we can show the advantages and disadvantages of the approach. Code snippet 10 shows the structural representation of the circuit from figure 3.7. The *lift* function, “lifts” a pure function to the component level. Unfortunately, to do so, the *initial* content of the memory element has to be specified when using the lifting function. As shown by the schematic of figure 3.7, no initial values are needed for this circuit to function properly. Without initial values, there is a latency of two clockcycles, before the circuit produces a meaningful result.

Code Snippet 10: Definition of a pipeline in CλaSH using arrows.

```
pipelineC = proc (x, y) → do
  s1 ← singleMem ‘lift‘ 0 -≺ x + y
  s2 ← singleMem ‘lift‘ 0 -≺ x
  s3 ← singleMem ‘lift‘ 0 -≺ s1 + s2
  returnA -≺ s3
```

Initial values have to be supplied regardless of the method used to create sequential logic. Even the pure function *pipeline*, has to be lifted to the component level, before translating it to VHDL. As shown by *pipelineC*, the component level has a different syntax when compared to the functions shown earlier. Usage of the language is more difficult as a result, due to the added syntax to describe interactions between components and the complexity of the underlying arrows model.

When we compare this with the code of snippet 11, we see that specification using time variables at the type level is a more concise way to specify time-dependent behaviour.

Code Snippet 11: Definition of a pipeline using time variables.

```
pipeline′ :: Int⟨t⟩ → Int⟨t⟩ → Int⟨t + 2⟩
pipeline′ x y = (x ∗ y) + x
```

Even though the presentation is fairly concise, the definition of *pipeline′* is not equal to CλaSH’s definition. Given the definition of *pipeline′*, the compiler cannot determine where *exactly* memory elements ought to be inserted. Based on the supplied function type, we know that the entire expression $(x * y) + x$ has type $Int\langle t+2\rangle$, while x and y have type $Int\langle t\rangle$. We know two memory elements are needed in order to make the transformation from t to $t+2$ possible, but we cannot determine where these elements ought to be inserted. Using retiming[29], the optimal placing of memory elements can be determined. However, that still does not give the ability to define the circuit of figure

3.7.

To do so, individual functions representing memory elements can be added. Since our language does not support function overloading, nor the mechanisms to use a function twice, two functions need to be defined in order to fully define the circuit of figure 3.7. If a single *delay* function is added, such as in *pipeline''* below, the time-dependent behaviour of the circuit is defined more precisely.

Code Snippet 12: Definition of a pipeline using time variables and a delay.

$$
\begin{aligned}
&\mathit{pipeline}'' :: \mathit{Int}\langle t\rangle \to \mathit{Int}\langle t\rangle \to \mathit{Int}\langle t+2\rangle\\
&\mathit{pipeline}''\ x\ y = ((\mathit{delay}\ y) * x) + x\\
&\mathit{delay} :: \mathit{Int}\langle t\rangle \to \mathit{Int}\langle t+1\rangle\\
&\mathit{delay}\ x = x
\end{aligned}
$$

In *pipeline''*, y is explicitly delayed by one clock cycle before it is multiplied by x. Then, assuming $(+)$ represents an adder made with combinational logic, x must be delayed *before* using it in the multiplication. By expressing time as part of the type, in combination with retiming, we can choose the degree of specification.

The difficulty here, is to determine when the specification of some time-dependent behaviour should be subject to retiming. This is not part of this thesis however, which is why we do not go into detail here.

Type Reconstruction

One of the most important aspects of our type system, is that variables are referentially transparant and only refer to single values. This worked out for the examples *pipeline'* and *singleMem'* earlier. In those examples, compositions were created from composing unary functions, or by composing combinational logic. The type system can reconstruct the type of any valid composition of well-typed functions. As an example, consider the *delay* and *sel* functions from code snippet 13.

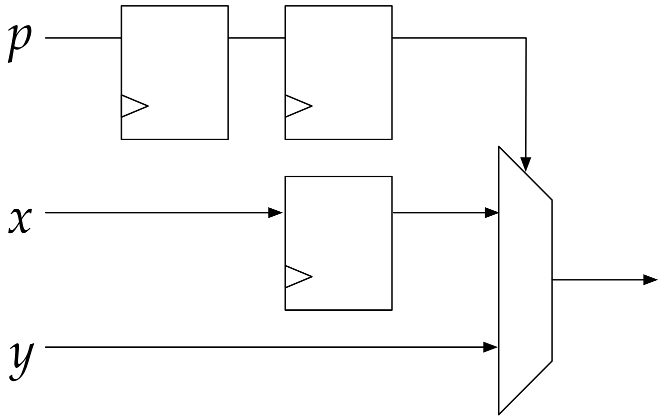

Figure 3.8: Schematic representation of *sel*.

Code Snippet 13: Delayed selection.

$$sel :: Bool\langle t\rangle \to Int\langle t+1\rangle \to Int\langle t+2\rangle \to Int\langle t+2\rangle$$
$$sel\ p\ x\ y = \textbf{if}\ p\ \textbf{then}\ x\ \textbf{else}\ y$$

$$delay :: Int\langle t\rangle \to Int\langle t+3\rangle$$
$$delay\ z = z$$

As shown in the code, the value p needs to be delayed twice, in order to be used together with the value y. The value x needs to be delayed once. We combine the definition of *sel* with *delay* as follows:

$$comp\ w\ q = sel\ q\ w\ (delay\ w)$$

Through the application of *sel* q to w, w is equal to x. Moreover, $w = z$ in *delay*, and $z = y$ through application of *sel*. As a result, $y = x$, which is seemingly impossible from the definition of *sel*. The type of *comp* can reflect this however, by adding additional memory elements. This is shown in figure 3.9.

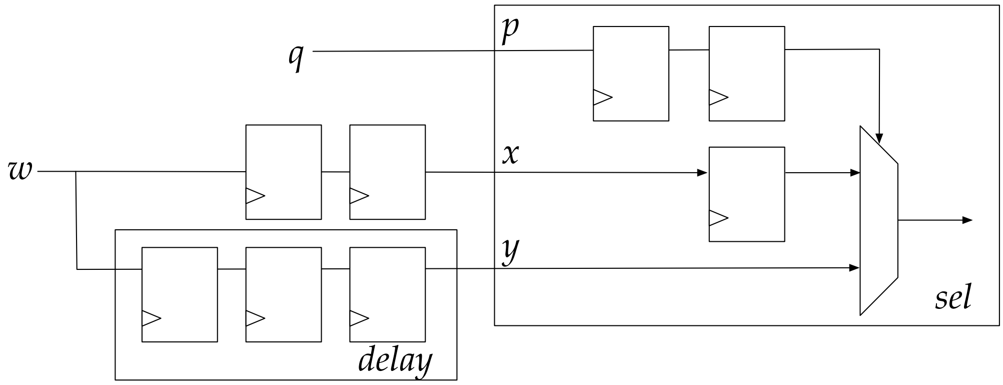

Figure 3.9: Schematic representation of *sel*.

As shown, two memory elements are added beyond those already available in *delay* and *sel*. This is done in order to equate *w* from *comp* with both *x* and *y* from *sel*. As a result, the type of *comp* would be

$$
\begin{aligned}
&comp :: Int\langle t\rangle \rightarrow Bool\langle t+1\rangle \rightarrow Int\langle t+3\rangle\\
&comp\ w\ q = sel\ q\ w\ (delay\ w)
\end{aligned}
$$

There exist an odd side-effect of this automatic derivation however. Due to how the typing rules are defined in the next chapter, functions can only be constructed where the arguments are ordered in time. That is, a function type as $Int\langle t\rangle \rightarrow Int\langle t+2\rangle \rightarrow Int\langle t+1\rangle \rightarrow Int\langle t+2\rangle$, while seemingly valid, can not be constructed. This side effect is most obvious when we derive the type of *comp′* below, in which the arguments *w* and *q* are flipped:

$$
\begin{aligned}
&comp' :: Bool\langle t\rangle \rightarrow Int\langle t\rangle \rightarrow Int\langle t+3\rangle\\
&comp'\ q\ w = sel\ q\ w\ (delay\ w)
\end{aligned}
$$

Since arguments need to be ordered, the only way to derive a valid type from *comp′* is to add one additional register between the input to *comp′* and the input of *sel*.

Sequencing

Using multiple input values which originate from the same wire is used frequently in synchronous hardware design, espcially in the area of Digital Signal Proccessing (DSP).

We can not define such behaviour using only the syntax we have introduced earlier in this chapter, as explained in the previous section. Consider the circuit of figure 3.10, where two consecutive values are added.

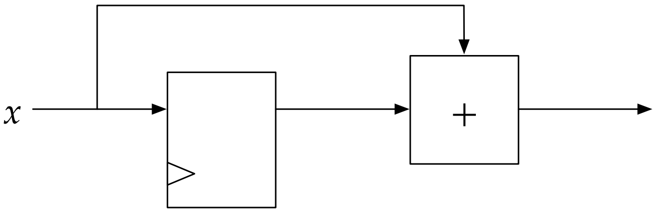

Figure 3.10: Adding two consecutive values.

In this schematic, x is not referentially transparant, as x refers to the wire, which represents multiple values. As shown in the previous section, this is not allowed in our descriptions, which makes it impossible to define this circuit directly through function abstraction and application. Instead, we allow finite sequences of values.

Code Snippet 14: Summing two consecutive values.

$$sum2 :: Int\langle t, t+1\rangle \to Int\langle t+1\rangle$$
$$sum2\langle x1, x2\rangle = x1 + x2$$

In *sum2*, the first argument consists of two sequential values. The type of first argument consists of two time variables, which match with the two identifiers *x1* and *x2*. By matching the identifiers with the time expressions, we show how *x1* relates to *x2*. When this function is applied to another value, the correct VHDL could, in principle at least, be generated.

The *sum2* function from above could similarly be represented as a binary function. However, it can not be used in compositions in the same fashion as *sum2*. As shown in the previous section, when *sum2′* is applied a single value twice, additional registers would be added to maintain referential transparency.

Code Snippet 15: Summing two consecutive values with a binary function.

$$sum2' :: Int\langle t\rangle \to Int\langle t+1\rangle \to Int\langle t+1\rangle$$
$$sum2'\ x1\ x2 = x1 + x2$$

Sequences are really just a form of syntactic sugar. In the next chapter, we show how functions of the form *sum2′* are converted to the equivalent of *sum2*. Using a function with type

$$foo :: a\langle t\rangle \rightarrow a\langle t+1\rangle \rightarrow ... \rightarrow a\langle t+n\rangle \rightarrow b\langle t+n+a\rangle$$

, we can derive the following structure including memory elements.

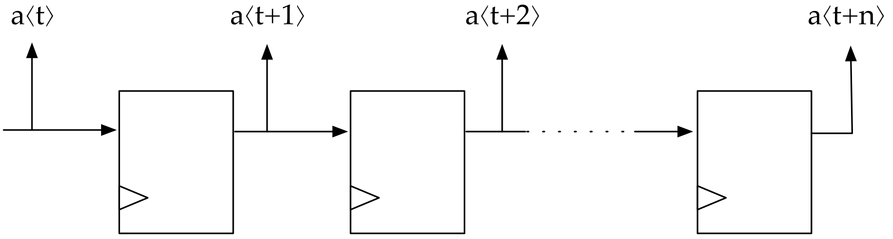

Figure 3.11: Structure generated by sequence usage.

The above structure only works for bit-representable values. In a language where functions are not bit-representable, sequences can not be used for functions.

3.3 Conclusion

In this chapter we discussed the specification of time-dependent behaviour in a functional language using the type system. One of the main advantages of the approach shown, is the fact that variables refer to *individual values*. This is different from most, if not all, existing functional hardware description languages. In existing functional hardware description languages like Lava and ForSyDe, streams are used. In those languages, variables refer to streams of values.

In our approach, variables refer to individual values. This is not without cost however, as variables need the ability to be moved through time. Without the ability to be moved through time, variables which refer to individual values can not be used to describe time-dependent behaviour. As a result from this ability, compositions of functions have a side effect. By composing functions, additional memory elements can be added to the system. This side-effect can be impredictable if one is not aware of its existence, which makes it less appealing to use in a hardware description language.

However, a great benefit of the expressing time-dependent behaviour in the way we showed, is that the definition of *what* needs to be done is separated from *when* it ought to be done. This distinction makes it easier to reason about the time-dependent behaviour, without being concerned with what the function does, or vice versa. By providing the “when” as part of the type, the time-dependent behaviour of compositions can be inferred, and checked by the system if the designer supplies the expected type.

Chapter 4

A Temporal Type System

In the previous chapter, we have (informally) shown how representing time as part of the type system gives us the ability to reason about various forms of hardware and their compositions. However, informal reasoning about temporal representation does not give any guarantees with regards to the feasibility of expressing time as part of the type system.

In this chapter, we define a grammar and a set of typing rules, which make a convincing case for the feasibility of expressing temporal behaviour as part of the type system. To do so, we first define expressions and timed types. Secondly, we introduce *polytemporal types*, based on polytypes as introduced in section 2.3. Third, we define constraints as per HM(X)[34], which are used in the typing rules of our type system. Finally, we show a number of derivations and briefly discuss the expressiveness of the given type system.

4.1 Grammar of λ_t

The grammar of our language, which we call λ_t, is very similar to the languages discussed in chapter 2. Expressions e consists of the usual terms of the λ-calculus: variables, abstraction and application. We chose to also represent constants and the meta-function δ. The δ meta-function is used to introduce delays within function types. Introduction of delays is severly restricted however, which is why we explain its mechanics later in this chapter, together with reasoning as to *why* it is severly restricted.

The constants which we allow are either of the *Boolean* variety, or the *Integer* variety. Having the ability to define different types of constants allows us to easily show that our type-system also includes regular type-checking over base types. Furthermore, even though the system currently has no term which directly represents choice, the system could be easily extended with it when needed due to the existence of Boolean types.

Definition 8: Expressions

$$
\begin{array}{lcll}
e & ::= & & \text{expressions:} \\
 & | & x & \textit{(variable)} \\
 & | & \lambda x.e & \textit{(abstraction)} \\
 & | & e\, e & \textit{(application)} \\
 & | & c & \textit{(constant)} \\
 & | & \delta(e, n \in \mathbb{N}_0) & \textit{(delay)}
\end{array}
\qquad
\begin{array}{lcll}
\text{c} & ::= & & \text{constants:} \\
 & | & \text{true} & \textit{(Boolean true)} \\
 & | & \text{false} & \textit{(Boolean false)} \\
 & | & \text{n} \in \mathbb{Z} & \textit{(Integer literals)}
\end{array}
$$

We define the set of simple types τ below. If needed, we also use the letter ρ to range over simple types. Like the simply typed λ-calculus, there exist only two ways in which a simple type can be constructed. A function type can be constructed using the $\rightarrow$ type-constructor, while a *timed type* can be constructed by combining a *time expression* ϵ and a base type T.

Definition 9: Types

$$
\begin{array}{lcll}
\tau & ::= & & \text{simple types:} \\
 & | & \tau \rightarrow \tau & \textit{(function type)} \\
 & | & T\langle \epsilon \rangle & \textit{(timed type)}
\end{array}
\qquad
\begin{array}{lcll}
\text{T} & ::= & & \text{base types:} \\
 & | & \mathit{Int} & \textit{(Integer type)} \\
 & | & \mathit{Bool} & \textit{(Boolean type)}
\end{array}
$$

$$
\begin{array}{lcll}
\epsilon & ::= & \epsilon + (a \in \mathbb{N}_0) & \textit{(time expression)} \\
 & | & t & \textit{(time variable)}
\end{array}
$$

Base types are similar to those used in the simply-typed λ-calculus, and are constructed by one of the type constants *Bool* or *Int*. A base type tells us something about the form or shape of a value; either something is a number or a boolean value. Time expressions consists of a *variable* moment in time t, and one or more *concrete* offsets to t. Using τ we can define the type of a memory element which holds an integer as: $\mathit{Int}\langle t \rangle \rightarrow \mathit{Int}\langle t+1 \rangle$.

We can now define the behaviour of δ informally as well. The δ keyword accepts values with function types. When the δ function is applied to a function with some offset n, then it adds the offset to every time expression occuring on the right-hand side of the type constructor $\rightarrow$. For instance, given the identity function

$$\lambda x.x : \mathit{Int}\langle t \rangle \rightarrow \mathit{Int}\langle t \rangle.$$

We apply $\delta(\lambda x.x, 1)$, which results in the function

$$\delta(\lambda x.x, 1) : \mathit{Int}\langle t \rangle \rightarrow \mathit{Int}\langle t+1 \rangle.$$

We discuss the full implications of this behaviour later in this chapter.

Time variables, like type variables in Damas-Milner, are *monomorphic* and range over the set of natural numbers $\mathbb{N}_0$. To describe hardware, we need to be able to define hardware components which are *polytemporal*, e.g. work for clock cycles $0, 1, \ldots, n$. For this we introduce a *polytemporal type* in definition 10, similar to polytypes of Damas-Milner.

Definition 10: Polytemporal types

A *polytemporal type* σ, written as

$$\sigma = \forall t.\tau \quad (ftv(\tau) = \{t\})$$

, consists of universal quantification over a single time variable t, which is bound in the simple type τ. Moreover, there are no free time variables in τ other than t.

Every value is defined in terms of some time variable t. This means that, unlike other type systems which include some form of polymorphism, types are *always* polytemporal. Literals are values which exist at every moment in time, expressed by their type: $42 : \forall t.T\langle t\rangle$. Similarly, variables are polytemporal as well. Without any restriction, all values can exist at any $t \in \mathbb{N}_0$, which is not exactly useful. Restrictions on time variables are needed to *limit* the existence of values. For instance, by applying a function which represents a single memory element to a constant value, the *resulting* value can only exist at every clockcycle > 0. To be able to restrict existence of values, we introduce *constraints* below.

4.2 Constraints

Constraints consists of existential quantification of time variables, which are used as binders for the time variables used in time expression constraints. Time expression constraints can be used to restrict a certain time expression in terms of another. Time expression constraints are inequalities with good reason; usage of equalities would restrict the expressiveness of the type system more than is needed. This is explained later in greater detail.

It may seem strange to have both existential quantification of time variables and universal quantification in the same type system. However, the constraints C are created separately from polytemporal types, and as a result, needs its own binders. This approach is not new; HM(X) uses the same approach to reason about polymorphism.

Definition 11: Constraints

We define the *constraint* C as follows:

$$\begin{array}{lllr}
C & ::= & & \text{constraints:} \\
 & | & \exists t.C & \textit{(existential constraint)} \\
 & | & C \wedge C & \textit{(composition)} \\
 & | & \epsilon \leqslant \epsilon & \textit{(time expression constraint)} \\
 & | & \mathrm{T} = \mathrm{T} & \textit{(base type constraint)} \\
 & | & \tau \equiv \tau & \textit{(type equivalence constraint)} \\
 & | & \text{true} & \textit{(satisfied constraint)} \\
 & | & \text{false} & \textit{(non-satisfied constraint)}
\end{array}$$

, which can be used to define both type constraints as time constraints.

In the constraints we have inequalities on time expressions, and equalities on base types. Restrictions on base types are not strictly needed to show the merit of our type system with regards to reasoning about temporal behaviour. We have included restrictions on base types to show that the combination of time expressions and base types can indeed be reasoned about within the same system. Additionally, the type equivalence constraint requires that types must be equal under α-conversion. These different forms of constraints are needed in order to reason about the time-dependent behaviour, and will be explained in more detail later.

Other constraints include the trivial constraints "true", "false" and composition of constraints through logical conjuction ($\wedge$). Composition of constraints including existential quantification needs to result in a well-formed constraint. Even though the conjuction $\exists t_1.C_1 \wedge \exists t_2.C_2$ is correct from the above definition, it becomes tedious to make sure every time variable is quantified and in scope. As a result, we define a constraint to be well-formed if every existential quantification is performed first, before introducing additional constraints. In the typing rules presented later in this chapter, we assume that every constraint is well-formed. When two constraints as $\exists t_1.C_1 \wedge \exists t_2.C_2$ are composed, all existential quantification is moved to the head of the constraint.

Together with the context Γ, constraints are used in type judgements of the form $C; \Gamma \vdash e : \forall t.\tau$. The type judgment ought to be read as: we can derive from the context Γ that the expression e has the simple type τ, for every t such that the constraints in C hold.

Without typing rules it is difficult to create an example which shows how constraints can be used to reason about the existence of values. To do so, we postpone showing a full type derivation here, and focus on how constraints can be used. Suppose we have

Definition 12: Constraints

We define the context Γ as follows:

Γ	::=		contexts:
	$\mid$	$\emptyset$	(empty context)
	$\mid$	$\Gamma, x : \tau$	(type binding).

a function *register* with the polytemporal type

$$register : \forall t.Int\langle t\rangle \to Int\langle t+1\rangle.$$

Now, without any constraint, this function can be used to transform any *Int* of any $t \in \mathbb{N}_0$ to an *Int* at $t+1$. When *register* is applied to a value, as in $v = register\,42$, the resulting value v exists one clockcycle later than the value *register* is applied to. All this can be expressed without any constraint, as the value simply has type

$$v : \forall t.Int\langle t+1\rangle.$$

However, when we apply another function (called *foo*) to v, the constraint that v can only exists at $t \geqslant 1$ must be adhered to. Suppose *foo* has type:

$$foo : \forall t'.Int\langle t'\rangle \to Int\langle t'+1\rangle \to Int\langle t'+2\rangle.$$

When applied to v, as in $v' = foo\,v$, the resulting function has type

$$foo\,v : \forall t'.Int\langle t'+1\rangle \to Int\langle t'+2\rangle.$$

However, this does *not* work for every t', as the first argument of *foo* does not exist at $t' = 0$. By adding the constraint

$$\exists t.\exists t'.t+1 \leqslant t'$$

, we can be sure that, when the constraint holds, t' is not equal to 0. If it were, then there would have to exist a t such that $t+1 \leqslant 0$. As $t \in \mathbb{N}_0$, this is obviously incorrect.

It may seem odd at this point why we chose inequalities over equalities, as equalities would lead to the same result. However, consider the function resulting from *foo* v. If we were to apply it to v again, as in $v'' = (foo\,v)\,v$, then the resulting constraint

$$\exists t.\exists t'.t+1 = t' \wedge t+1 = t'+1$$

could never hold. Yet, this application is *valid*. Memory elements can be used to delay a value indefinitely. With inequalities we can express this as part of the constraint:

$$\exists t.\exists t'.t + 1 \leqslant t' \wedge t + 1 \leqslant t' + 1.$$

From this constraint we can derive that, in order for the composition of v'' to work, a memory element has to be added for the *second* v, before applying *foo* v to it. We can derive this by solving the constraints for the minimal value of t and t'. From the constraints we derive $t = 0$ and $t' = 1$. Plugging these into the known types of *foo* and *register*, allows us to determine when a memory element needs to be added. For instance, v exists at 1 if $t = 0$, and *foo*'s second argument is expected at $t' + 1$, which is 2 if $t' = 1$. The difference between v and the type of the second argument of *foo*, indicates that something must be added here in order to make the composition work. This also makes sense intuitively. The type of *foo* indicates that it *expects* the second argument to exist one clock cycle later than the first. When applied to the same value twice, something must be done to one of the values in order to make the composition work.

4.3 Typing Rules

Now that we introduced constraints, we can introduce the typing rules which give us the ability to (re)construct and verify expressions of λ_t. Before doing so however, we provide a relation between two simple types below, in order to simplify the definition of typing rules. In particular, this type relation is used to create the constraints which are introduced by function application.

Definition 13: The type relation $\preceq$

We define the precedence relation $\preceq$ as

$$\tau_1 \preceq \tau_2 = \begin{cases} \epsilon_1 \leqslant \epsilon_2 \wedge \mathrm{T}_1 = \mathrm{T}_2 & \text{if } \tau_1, \tau_2 = T_1\langle\epsilon_1\rangle, T_2\langle\epsilon_2\rangle \\ \tau_1 \equiv \tau_2 & \text{if } \tau_1, \tau_2 = \tau_{11} \to \tau_{12} \wedge \tau_{21} \to \tau_{22} \\ \text{false} & \text{otherwise} \end{cases}$$

, where τ_1 temporally precedes τ_2 and the base types of τ_1 and τ_2 are equal.

The $\preceq$ relation distinguishes three cases to do so. In the first case, a relation between two timed types is created. As explained in the previous section, relations between time expressions are inequalities.

The second case defines type equality under α-conversion. This case is used to define the precedence relation for function types. To show why we use equivalence here, instead of defining a "real" precedence relation for function types, we show the two most obvious ways to represent function precedence, and why they do not work.

First, we define the relation $\tau_1 \preceq \tau_2 = \tau_{11} \preceq \tau_{21} \wedge \tau_{12} \preceq \tau_{22}$, where $\tau_1 = \tau_{11} \rightarrow \tau_{12}$ and $\tau_2 = \tau_{21} \rightarrow \tau_{22}$. In this case, τ_1 must precede τ_2. When τ_2 represents the type of a function which is expected, then τ_1 represents the type of a function which is actually supplied. By applying the type inequality pointwise for every timed type in τ_1 and τ_2, we state that the supplied function may expect the supplied values *earlier* than the expected function. Even though the constraints would reflect that condition if it occurs, by using this precedence relation additional memory elements could be inserted through reckless use of higher order functions.

The same precedence relation could be inverted. However, this is also problematic, as there is no guarantee that the supplied function results in a value, before that actual value is needed. Again, the constraints would reflect that condition, and more memory elements could be inserted if this condition occurs. However, as we already have a certain amount of automatic introduction of memory elements, we restrict the type of higher order functions to match *exactly* with the expected function type.

Finally, the third case is used for cases where types do not match. Even though this case might seem superfluous when compared to the second, the third case is needed in order to restrict usage of sequences, which we discuss later in this chapter.

The Typing Rules TT-Var, TT-Const & TT-Gen

Finally, we define and discuss the typing rules. We start by the rule TT-Var, which is similar to the rules involving variables discussed in chapter 2.

Definition 14: Typing rules of λ_t (a)

$$\frac{\Gamma(x) = \tau}{\text{true}; \Gamma \vdash x : \tau}\ \text{TT-Var} \qquad \frac{}{C; \Gamma \vdash c : \tau}\ \text{TT-Const}$$

$$\frac{C; \Gamma \vdash e : \tau \quad \mathit{ftv}(\tau) = t}{\exists t.C; \Gamma \vdash \forall t.\tau}\ \text{TT-Gen}$$

When $\Gamma(x) = \tau$, then we can conclude from Γ, without constraint, that $x : \tau$. We use this to define the type of x in Γ_0 as follows:

$$\frac{\Gamma_0(x) = \mathit{Int}\langle t_0 \rangle}{\text{true};\Gamma_0 \vdash x : \mathit{Int}\langle t_0 \rangle.} \ (1)$$

As shown, the TT-Var rule does not introduce universal quantification over t_0. We apply TT-Gen, which transforms an expression e with type τ, to a polytemporal type. A restriction on TT-Gen is that all free variables in τ are equal to t, in order to allow universal quantification over t. Since the definition of τ does not include a binder for time variables, all time variables in τ are free. This means that, after applying this rule, $e : \forall t.\tau$, no time variables other than t are used in τ.

Additionally, we add the constraint $\exists t.$, without modifying the existing constraint. Existential quantification in the constraint provides us with a binder to add inequalities of time expressions. It is however difficult to see that, if there exists a t such that the constraints in C hold, e has the type $\forall t.\tau$. As mentioned earlier, the universal quantification is subject to the constraints in C. If the constraints in C do not hold for a particular t, then e is not well-typed for that particular t. If we assume $C = \text{true}$, then any $t \in \mathbb{N}_0$ can be chosen.

We use the definition of Γ_0 from (1), to derive the polytemporal type of x.

$$\frac{\text{true};\Gamma_0 \vdash x : \mathit{Int}\langle t_0\rangle}{\exists t_0.\text{true};\Gamma_0 \vdash x : \forall t_0.\mathit{Int}\langle t_0\rangle}\ \text{(2)}$$

The Typing Rule TT-Abs

Next, we define TT-Abs. This rule is rather involved, for two reasons:

1. polytemporal types only consist of a single time variable, so the types of x and e need to be changed via α-conversion to use a new time variable;
2. if x is used within e, then the constraints the types of x and e are subject to are known, yet the exact relation between x and y cannot be determined from their types alone.

Definition 15: Typing rules of λ_t (b)

$$\frac{C;\Gamma \vdash x : \forall t_1.\tau_1 \quad C;\Gamma \vdash e : \forall t_2.\tau_2 \quad t_3 \notin \mathit{vars}(C,\Gamma)}{\exists t_3.C \wedge t_3 \leqslant t_1 \wedge t_2 \leqslant t_3 + a;\Gamma \vdash \lambda x.e : \forall t_3.[t_3/t_1]\tau_1 \to [t_3 + a/t_2]\tau_2}\ \text{TT-Abs}$$

Given that x is used within e, we can determine the exact difference in the time expressions of x and e based on the constraints they are subject to. From this difference we can, in principle, determine an offset a such that a fresh time variable $t_3 \leqslant t_1$ and $t_2 \leqslant t_3 + a$. When this is the case, we say the resulting abstraction *encloses* x and e.

Similar to Damas-Milner, we cannot determine from types alone what the new type of $\lambda x.e$ ought to be. Like algorithm W in the case of Damas-Milner, we need an algorithm which finds the smallest a such that $\lambda x.e : \forall t_3.[t_3/t_1]\tau_1 \rightarrow [t_3 + a/t_2]\tau_2$ encloses $x : \forall t_1.\tau_1$ and $e : \forall t_2.\tau_2$. We discuss the options of solving for a in the next section.

Using the derivation of (2), we derive the type of the identity function $\lambda x.x$. Since we derive the identity function here, $e = x$, which results in two identical premises for the rule TT-Abs. For simplicity' sake, we only include the single premise here.

$$\frac{\exists t_0.\text{true};\Gamma_0 \vdash x : \forall t_0.Int\langle t_0\rangle}{\exists t_1.\exists t_0.t_1 \leqslant t_0 \wedge t_0 \leqslant t_1 + a;\Gamma_0 \vdash \lambda x.x : \forall t_1.Int\langle t_1\rangle \rightarrow Int\langle t_1\rangle} \tag{3}$$

We take $a = 0$ and simplify this expression for presentation purposes by removing the constraint $t_1 \leqslant t_0 \wedge t_0 \leqslant t_1 + a$, which implies $t_1 = t_0$. The resulting type judgement reads:

$$\exists t_0.\text{true};\Gamma_0 \vdash \lambda x.x : \forall t_0.Int\langle t_0\rangle \rightarrow Int\langle t_0\rangle.$$

In the example of (3), e is equal to x, which greatly simplifies the derivation. After introducing the remaining typing rules we will show another derivation, where e and x are different.

The Typing Rule TT-Delta

To introduce delays, TT-Delta applies a positive shift to time expression of the right hand side of a function type.

Definition 16: Typing rules of λ_t (c)

$$\frac{C;\Gamma \vdash e : \forall t.\tau \rightarrow \rho}{C;\Gamma \vdash \delta(e,n) : \forall t.\tau \rightarrow [t+n/t]\rho} \quad \text{TT-Delta}$$

As a result, function types always have an increasing delay as more → type constructors are nested. Given a function type, the TT-Delta rule only delays the type of the result, whether that is another function type or a timed type. Using the result of (3), we derive the type of a function which represents a single memory element as follows.

$$\frac{\exists t_0.\text{true};\Gamma_0 \vdash \lambda x.x : \forall t_0.Int\langle t_0\rangle \rightarrow Int\langle t_0\rangle}{\exists t_0.\text{true};\Gamma_0 \vdash \delta(\lambda x.x, 1) : \forall t_0.Int\langle t_0\rangle \rightarrow Int\langle t_0 + 1\rangle} \tag{4}$$

The Typing Rule TT-App

Functions can be applied to other values. In the rule TT-App, we define that e_1 must have a function type, and e_2 has a type which is unrelated to e_1.

Definition 17: Typing rules of λ_t (d)

$$\frac{C_1; \Gamma_1 \vdash e_1 : \forall t_1.\tau_{11} \to \tau_{12} \quad C_2; \Gamma_2 \vdash e_2 : \forall t_2.\tau_2}{C_1 \wedge C_2 \wedge \tau_2 \preceq \tau_{11}; \Gamma_1 \cup \Gamma_2 \vdash e_1\, e_2 : \forall t_1.\tau_{12}} \text{ TT-App}$$

This is contrary to how other languages define function application, where $\tau_2 = \tau_{11}$. Instead, we state that $\tau_2 \preceq \tau_{11}$. Intuitively speaking, we can apply a function to a value when the value exists, as long as the value exists before the function "expects" it to exist. If the value exists one or more clockcycles before the function expects it to exist, we can use memory elements to delay the value to the point where the function expects it.

To show how this typing rule works, we take the derivation of Γ_0 from (1) and (2) to similarly derive the type of y. We then apply the function of (4) to y, leading to a set of constraints as follows.

$$\frac{\exists t_0.\text{true};\Gamma_0 \vdash \delta(\lambda x.x, 1) : \forall t_0.\mathit{Int}\langle t_0\rangle \to \mathit{Int}\langle t_0+1\rangle \qquad \dfrac{\dfrac{\Gamma_2(y) = \mathit{Int}\langle t_2\rangle}{\text{true};\Gamma_2 \vdash y : \mathit{Int}\langle t_2\rangle}}{\exists t_2.\text{true};\Gamma_2 \vdash y : \forall t_2.\mathit{Int}\langle t_2\rangle}}{\exists t_0, t_2.t_2 \leqslant t_0 \wedge \mathit{Int} = \mathit{Int};\Gamma_0 \cup \Gamma_2 \vdash \delta(\lambda x.x, 1)y : \forall t_0.\mathit{Int}\langle t_0+1\rangle} \; (5)$$

Other Derivations

To show a more involved example, we use the application $\delta(\lambda x.x, 1)\, y$ from (5) to create a function of y. As shown in (5), one register is applied to y. As a result, the abstraction of $\lambda y.\delta(\lambda x.x, 1)\, y$ would have the same type as the expression in derivation (4). To create a more meaningful derivation, we create another shifted identity function $\delta(\lambda z.z, 1)$, which is derived similarly to (4), to create the following expression:

$$\lambda y.\delta(\lambda z.z, 1)\; (\delta(\lambda x.x, 1)\, y),$$

which delays the argument y of type *Int* twice.

To keep the typing rules readable, we define the following shorthands, which are derived from (1) to (5).

$$\begin{aligned} C_2 &= \exists t_0, t_2.t_2 \leqslant t_0 \wedge \mathit{Int} = \mathit{Int} && \text{derived from (4)} \\ C_3 &= \exists t_3.\text{true} && \text{derived from (2)} \\ e_3 &= \delta(\lambda z.z, 1)\; (\delta(\lambda x.x, 1)\, y) && \text{derived from (5) and (2)} \\ \Gamma_4 &= \Gamma_0 \cup \Gamma_2 && \text{derived from (5)} \end{aligned}$$

The constraint C_3 is associated with $\delta(\lambda z.z, 1)$, which has type $\forall t_3.Int\langle t_3\rangle \to Int\langle t_3 + 1\rangle$. We use the above definitions to derive the following:

$$\frac{C_3; \Gamma_3 \vdash \delta(\lambda z.z, 1) : \forall t_3.Int\langle t_3\rangle \to Int\langle t_3 + 1\rangle \qquad C_2; \Gamma_4 \vdash \delta(\lambda x.x, 1)y : \forall t_0.Int\langle t_0 + 1\rangle}{C_1 \wedge C_2 \wedge Int\langle t_0 + 1\rangle \preceq Int\langle t_3\rangle; \Gamma_4 \vdash e_3 : \forall t_3.Int\langle t_3 + 1\rangle} \text{(6)}$$

In (6) we defined two memory elements, and applied them to y. We use this definition to show how the abstraction of y leads to the (correct) type $\forall t_4.Int\langle t_4\rangle \to Int\langle t_4 + 2\rangle$. We define $C_4 = C_1 \wedge C_2 \wedge Int\langle t_0 + 1\rangle \preceq Int\langle t_3\rangle$ and apply TT-Abs to it as follows.

$$\frac{C_4; \Gamma_4 \vdash e_3 : \forall t_3.Int\langle t_3 + 1\rangle \qquad \exists t_2.\text{true}; \Gamma_2 \vdash y : \forall t_2.Int\langle t_2\rangle}{\exists t_4.C_4 \wedge t_4 \leqslant t_2 \wedge t_3 \leqslant t_4 + a; \Gamma_4 \vdash \lambda y.e_3 : \forall t_4.Int\langle t_4\rangle \to Int\langle t_4 + a + 1\rangle} \text{(7)}$$

We rewrite the constraints from the conclusion of (7) for presentation purposes.

$$\begin{array}{ll} t_4 \leqslant t_2 & \wedge \\ t_2 \leqslant t_0 & \wedge \\ t_0 + 1 \leqslant t_3 & \wedge \\ t_3 \leqslant t_4 + a & \end{array}$$

To find a manually and create a minimal amount of memory elements, we take $t_4 = 0$, $t_2 = 0$ and $t_0 = 0$. This results in $1 \leqslant t_3 \wedge t_3 \leqslant t_4 + a$. We take $t_3 = 1$, which results in $a = 1$. Placing $a = 1$ into the result of (7) gives $\lambda y.e_3 : \forall t_4.Int\langle t_4\rangle \to Int\langle t_4 + 2\rangle$.

The (complex) derivation of (7) shows that types of compositions, which use functions with time-dependent behaviour, can be reconstructed.

4.4 Solving Constraints

In the previous section we have shown that we can derive the polytemporal type of expressions and a constraint which must hold for the expression to have a valid type. Following the discussion of the previous section, we need an algorithm such that:

1. we can derive the offset needed for a single time variable to enclose two others;
2. we can derive the possible placement of registers when hardware is generated from the type-checked expression;

Instead of defining our own algorithm, we use an Satisfiability Modulo Theories (SMT)-solver such as Z3[14] or Yices[15] to solve our constraints directly. SMT-solvers accept first-order logic extended with arithmetic statements, and can be used to find a set of integers which satisfy our constraints. Given that we can find a set of integers which satisfy our constraints, we can determine the difference in time between time variables. By examining the difference between time variables we can determine where registers need to be inserted. The set of constraints following from derivation (7), are used to derive $t_0 = t_2 = t_4 = 0, t_3 = 1$. From the derivations (1) to (7), we know the type of every subexpression in $\lambda y.e_3$. From these types, and plugging in the values for t_0 to t_4 from above, we can derive where differences in existence occur. Thus, we can derive where memory elements need to be inserted.

4.5 Sequences

As explained in the previous chapter, referential transparency does not allow us to access multiple values from the same wire. Instead, we expand the grammar of definition 8.

Definition 18: Types (b)

$$\begin{array}{lclr} \tau & ::= & & \text{expressions:} \\ & | & \text{T s} & \textit{(sequence type)} \\ & | & \dots & \dots \end{array} \qquad \begin{array}{lclr} \text{s} & ::= & & \text{sequence:} \\ & | & \epsilon, \text{s} & \textit{(time expression)} \\ & | & \langle\!\langle\rangle\!\rangle & \textit{(empty sequence)} \end{array}$$

Sequences behave like lists in functional languages, though here these "lists" are not first class. Addition of sequences to the grammar alone does not make sequences first class. As the precedence relation of definition 13 is not modified, any usage of sequences in function application will lead to the "false" constraint. To introduce sequences, we define the following rule.

Definition 19: Typing rules of λ_t (e)

$$\frac{C;\Gamma \vdash e : \forall t.T\langle t\rangle \rightarrow T\langle t+1\rangle \rightarrow \ldots \rightarrow T\langle t+n\rangle \rightarrow \rho}{C;\Gamma \vdash seq(e) : \forall t.T\langle\!\langle t, t+1, \ldots, t+n\rangle\!\rangle \rightarrow \rho} \quad \text{TT-Seq}$$

The TT-Seq rule allows us to define a sequence based on a n-ary function. Given an expression e, if that expression represents a function with a number of strictly temporally increasing elements of the same base type T, then we can construct a sequence out of those arguments. Since the precedence relation does not allow us to apply a function with sequences to a value, we define a separate rule named TT-SApp.

Definition 20: Typing rules of λ_t (f)

$$\frac{C_1; \Gamma_1 \vdash e_1 : \forall t_1.T\langle\langle t_1, \ldots, t_1 + n\rangle\rangle \to \rho \quad e_2 : C_2; \Gamma_2 \vdash \forall t_2.\tau_2}{C_1 \land C_2 \land \tau_2 \preceq T\langle t_1\rangle; \Gamma_1 \cup \Gamma_2 \vdash e_1\, e_2 : \forall t_1.\rho} \text{ TT-SApp}$$

This rule defines that, given a function which accepts a sequence, it can be applied to a single value. The sequence type only informs the type system that we wish to extract multiple values from the same source. Since the original function e from TT-Seq is well-typed, the function definition is correct.

4.6 Conclusion

The type system we presented is fairly limited. Delays can only be inserted using the δ operator. The δ operator acts on values with a function type, which makes it a little cumbersome when using n-ary functions. In case the result of a binary function exists later than the second argument, a second δ-operation needs to be applied to the *partially applied* binary function. This is mostly an operational concern however, as application of the δ operation with the appropriate offset to a function can be automated.

A downside of the δ approach is that it may be hard to give the end-user proper feedback if an error occurs involving a delayed function. Because of the δ keyword, there exists a disconnect between how delays are *presented* (see the previous chapter), and how delays are *created*. While a transformation between the two representations probably exists, the disconnect between the two representations might cause problems when errors need to be reported.

An obvious solution might be to allow δ to be applied to *any* value. Unfortunately, if this were the case then we cannot be sure that only functions which have an increasing delay can be constructed. For instance, a binary function could be created where a delay is only applied to the first argument, which would result in a function without an increasing delay. Even though constraints give us the ability to enforce construction of function types according to certain rules, it is hard to provide a meta-function which allows the user to communicate function types without exposing the constraint-based system to the user.

The constraint-based system, as explored in HM(X)[34], makes it difficult to see for which t an expression is well-typed. The polytemporal type σ does not have enough information available to determine whether an expression is well-typed from σ alone. As a result, we cannot use typing rules alone to determine whether or not an expression is well-typed. Moreover, to allow abstraction, we need to determine an offset a such that a single time variable encloses both the expression and the variable used in the abstraction. This is similar to Damas-Milner, where an additional algorithm, W, is needed to determine the principal type of an expression.

Despite the apparent complexity, the resulting type-system can only realistically model pipelining and sequences. A major advantage of the type system however, is that the type system only allows variables to refer to single values or functions. As a result, and due to referential transparency, whenever a variable is used we can mentally replace it with some placeholder value to see what the function does.

Even though we introduced polytemporal types, we cannot use a single function definition multiple times. To use a single function twice within a description, let-bindings or where-clauses need to be introduced. Aside from the inability to use the same function twice within the same expression, we also have not determined a way to “freshen” the time variables used in the type of a function. Even if the time variables of the function type itself were fresh, the same time variables are also used within constraints. Without further analysis, it is difficult to see whether freshening of the time variables used within a function type is sufficient.

The given type system is far from complete. Without a soundness proof, it is difficult to see what situations we have missed in our analysis. However, the given type system allows us to gain a better understanding on how to express time as part of the type system. The lack of higher order functions which are flexible with regards to their temporal constraints is unfortunate. Currently, we define higher order functions to require exactly the same types under α-conversion of time variables. This reduces the usefulness of higher order functions.

Chapter 5

Implementation

In this chapter, we discuss the implementation of the prototype typechecker. The prototype checker was originally made to explore the feasibility of expressing time as part of the type system. As such, not every feature of the discussed type system is implemented here. Moreover, the implementation is different from the type system presented in the previous chapter in a number of ways. We discuss the difference at the end of this chapter.

The prototype is made using the Haskell language. As such, knowledge of Haskell is required to interpret the code examples. Not all the code is shown however. In this chapter we discuss the implementation, but do not provide a full coverage of all source code used to create it. If needed, code examples are provided and explained.

To discuss the implementation, we first provide a general overview on how the typechecker is implemented. Secondly, we introduce the relevant parts of the grammar used by the typechecker. Finally, we introduce the modified unification algorithm and the method we use to derive and verify timing constraints from expressions.

5.1 Introduction

Before going into details, we first provide a general overview on how we created the typechecker. In our typechecker, we support both base types, such as *Bool* and *Int*, as well as time expressions. In the implementation these are separated before unification, as shown by figure 5.1. As mentioned in earlier chapters, we use constraint-based typechecking. Constraint-based typechecking makes it possible to *infer* the type of a term, at least if it has a valid type. The algorithm we use is a modified version of the algorithm discussed in "Types and Programming Languages"[37], and as such does not behave exactly the same as the type system presented in chapter 4. A single constraint generation algorithm is used to generate both the constraints of base types, as well as the constraints for time expressions. As such, we can infer the time-dependent behaviour from terms. The unification algorithm is slightly different for time expressions and base types however, which is why the implementation of these two are separated.

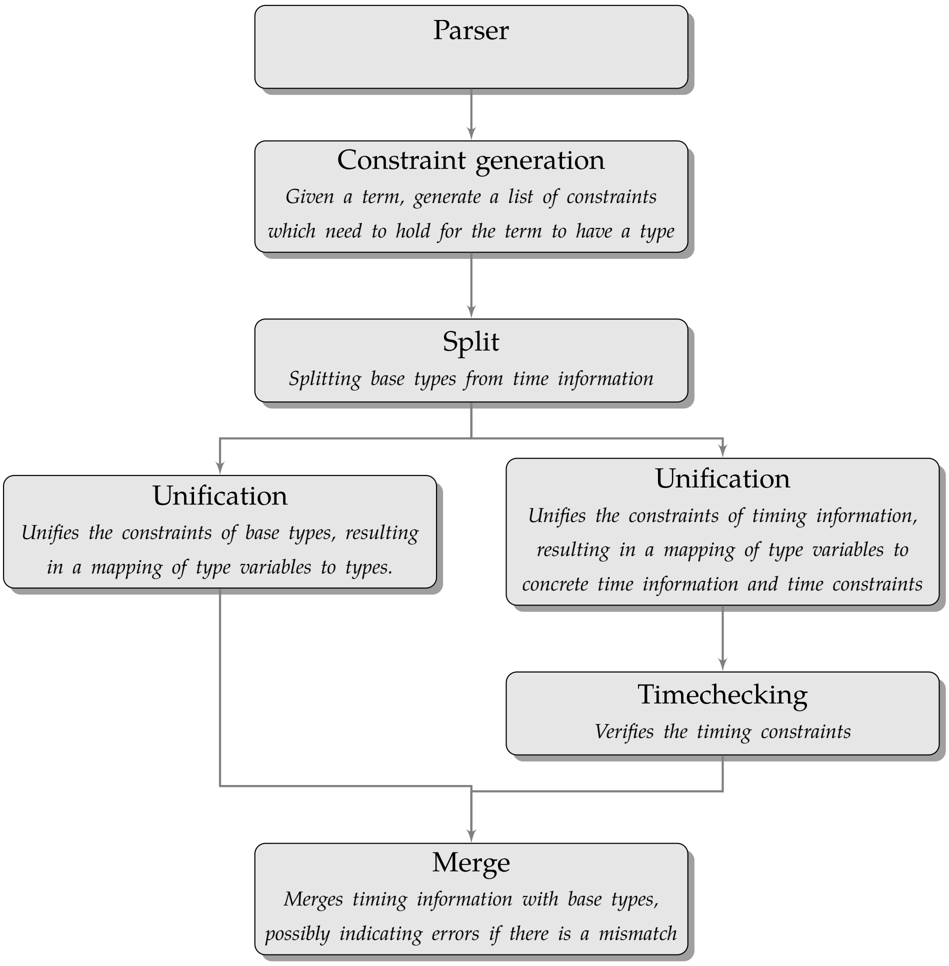
Parser
Constraint generation
Given a term, generate a list of constraints which need to hold for the term to have a type
Split
Splitting base types from time information
Unification
Unifies the constraints of base types, resulting in a mapping of type variables to types.
Unification
Unifies the constraints of timing information, resulting in a mapping of type variables to concrete time information and time constraints
Timechecking
Verifies the timing constraints
Merge
Merges timing information with base types, possibly indicating errors if there is a mismatch

Figure 5.1: Overview of the implementation of the typechecker.

As seen above, the entire typechecking part is split in two parts. Validation and inference of time expressions is independent from regular typechecking. This approach has some advantages:

- Regular typechecking can be used on the base types, something which is well understood already.
- When merging the results from type and timechecking we can verify that the results from timechecking have the same structure as the ones from typechecking. This provides a simple error detection method, increasing the confidence in correctness.

- If the timechecking algorithm is independent of typechecking the same approach may be used to augment other typecheckers with time information.
- Extending the timechecking part is easier.

Before going into details on how these parts of the typechecker are implemented, we first discuss the implemented grammar the typechecker uses.

5.2 The Grammar

In our implementation we do not parse any source code. As such, it is easier to use DeBruijn indexing instead of named variables. If we were to use named variables, we would have to first check which variable belongs to what binder. When using DeBruijn indices we immediately know which variable refers to which binder. As an added bonus, we also do not need to concern ourselves with shadowing.

DeBruijn indices are integers which refer to binders. For instance, the code

```
λx.λy.x y
```

would become

```
λ.λ.1 0
```

When using DeBruijn indices, variables are no longer named. Instead, variables are numbered. As shown, the DeBruijn index indicates how many binders need to be skipped to find the binder which the variable belongs to. To represent functions, the three terms of the λ-calculus are represented in the grammar as indicated by code snippet 16. The grammar is represented in the language Haskell using Algebraic Data Types (ADTs). ADTs allow us to define the grammar concisely.

Code Snippet 16: Implementation of Term (1)

```
type DeBruijn = Int
data Term = TmVar DeBruijn        -- Variable
          | TmAbs String Term     -- Abstraction
          | TmApp Term Term       -- Application
```

We define a *Term* as consisting of either a *variable* (indicated by the type constructor *TmVar*), *abstraction* (indicated by *TmAbs*), and *application* (indicated by *TmApp*). Using these terms we build expressions of the λ-calculus. For instance, term $\lambda x.\lambda y.x\, y$ is represented by the algebraic datatype above as follows:

```
TmAbs "x" $ TmAbs "y" $ TmApp (TmVar 1) (TmVar 0)
```

Strictly speaking, the *String* identifiers are not needed. For debugging purposes it is often handy to have names to identify individual variables however. So far, the definition of *Term* is very minimal. In the next section we extend the definition of our language to include type representation.

Implementation of Types

As mentioned before, we have separated unification of base types and time expressions. To increase code reuse, we have defined types as a parametric datatype. This gives us the ability to use the shared structure of function types, variable types and constants between time expressions and base types.

Code Snippet 17: Implementation of MetaType

```
data MetaType a = Arrow (MetaType a) (MetaType a)
                | Var VariableIndex
                | Constant a

type VariableIndex = Int
```

As shown by code snippet 17, types in our typechecker are either function types, represented by the *Arrow* type constructor, type variables or constant. Type variables are represented by integers and used for type-inference, as shown later. The type constructor *Constant* refers to all types except type variables or function types. For instance, the type *Int* is a constant type, as is the time expression $\langle t+1 \rangle$. The definition of *Metatype* allows us to define types such as $Int\langle t+1 \rangle$ and *separate* types into base types and time expressions via a transformation function.

To show how *MetaType* is used, we start by defining a function consisting of base types. Using the definition of *BaseType* from code snippet 18, we represent a function with type $Bool \rightarrow Int \rightarrow Int$ as follows:

```
Arrow (Constant Bool) (Arrow (Constant TyInt) (Constant TyInt))
```

The above expression has type *MetaType BaseType*, which indicates that this type only contains base type information.

Code Snippet 18: Implementation of Base Types

```
data BaseType = TyBool
              | TyInt
```

Similarly, we use the definition of *MetaType* to create functions consisting solely of time expressions. Before introducing the algebraic datatype of time expressions however, we need to be able to introduce time variables. For this we extend the definition of *Term* as indicated by code snippet 19.

Code Snippet 19: Implementation of Term (2)

```
data Term = TmVar DeBruijn       -- Variable
          | TmAbs String Term    -- Abstraction
          | TmApp Term Term      -- Application
          | TmTAbs String Term   -- Time Abstraction
```

Similarly to regular function abstraction, time abstraction introduces a time variable. For example, given a function with type $Int\langle t\rangle \rightarrow Int\langle t+1\rangle$, the time variable t has to be introduced. The *TmTAbs* term is used to introduce time variables. We then use DeBruijn indices to create time expressions, as shown by the definition of *TimeExpr* in code snippet 20. The DeBruijn index refers to the time variable introduced by *TmTAbs*. Time expressions use both a time variable, as well as an offset. The offset is added to the time variable. Since we only support addition, the addition operation is implictly applied to the offset and the time variable.

Code Snippet 20: Implementation of time expressions

```
data TimeExpr = TimeExpr DeBruijn Offset
              | TimeLiteral Offset

data TimedType = TimedType BaseType TimeExpr

type Offset = Int
```

However, even though we can introduce time variables using *TmTAbs*, we still cannot ascribe terms with types. Ascription of terms with types is essential in our type system, as it is the only way in which time-dependent behaviour can be specified. To introduce ascription, we extend the definition of *Term* for the third (and final) time.

Code Snippet 21: Implementation of Term (3)

```
data Term = TmVar DeBruijn           -- Variable
          | TmAbs String Term        -- Abstraction
          | TmApp Term Term          -- Application
          | TmTAbs String Term       -- Time Abstraction
          | TmAs Term TimedType      -- Ascription
```

Using time abstraction, ascription and time expressions, we can give types to terms. For instance, we can ascribe a function f with type $Int\langle t\rangle \to Int\langle t\rangle \to Int\langle t+2\rangle$ as

```
TmTAbs "t" $
  TmAs f     $
        Arrow (TimedType TyInt (TimeExpr 0 0))
              (Arrow (TimedType TyInt (TimeExpr 0 0))
                     (TimedType TyInt (TimeExpr 0 2)))
```

In this piece of code, we first introduce the time variable t, before ascribing the function f with a type. The type, as shown earlier, consists of a time expression and base type, together with the proper DeBruijn indices to make every time expression refer to the same time variable.

Now that we have defined a (minimal) definition of types and terms, we can discuss constraint generation. The constraint generation defines what constraints a term is subject to in order for the term to have a valid type. As such, constraint generation expresses the relation between types and terms, although it does not define how the constraints which terms are subject to are verified.

5.3 Constraint Generation

We use the grammar as introduced in the previous section to generate constraints. We represent a constraint as a relation between two types, as shown by code snippet 22.

Code Snippet 22: Constraints

```
type Constraint a = (MetaType a, MetaType a)
```

The generated constraints do not show the ordering of computations; they are equalities. As an example, consider the delay function

```
delay :: Int⟨t⟩ → Int⟨t + 1⟩
delay x = x
```

Here, x has both the type $Int\langle t\rangle$ and $Int\langle t+1\rangle$. To determine whether this is valid, we need to be able to distinguish between the x which is used as an input, and x which is used as an output for the function *delay*. The code of

```
delay' :: Int⟨t + 1⟩ → Int⟨t⟩
delay' x = x
```

should be invalid, to maintain causality. Using *delay* leads to the constraint $Int\langle t\rangle = Int\langle t+1\rangle$, while using *delay′* leads to the constraint $Int\langle t+1\rangle = Int\langle t\rangle$. In order to consider the first constraint valid, while denying the second constraint, we only allow function definitions in the form of *delay*, meaning only *increasing* the delay is allowed within a function definition. Even though this may seem like a big restriction, it is done with good reason. When we restrict ourselves to constraints which represent equalities, we can use the constraint generation as introduced[37] by Pierce. This does not mean the ordering of constraints is not important. Later in this chapter, when we discuss the modified unification algorithm, we show how we can derive the ordering of function application based on the (unique) time variables functions use. Even though we do not need inequalities to reason about constraints such as those described by *delay* and *delay′*, we do need them when inferring the time-dependent behaviour of compositions of time-dependent functions.

Code Snippet 23: Constraint generation of abstraction

```
constraints :: Term → State TypeState (MetaType BoundedType)
constraints tm = case tm of
  TmAbs str tm →
    do nvlam ← freshTypeVar
       nvtot ← freshTypeVar
       ctx   ← gets context
       addTypeVarBinding (str, nvlam)
       ty    ← constraints tm
       modify (λs → s {context = ctx})
       addTypeConstraints [(nvtot, Arrow nvlam ty)]
       return nvtot
```

The constraint generation algorithm allows us to infer the types of terms. To do so, we need the ability to create *unique* type variables, which represent the (unknown) types of terms. For this purpose we use the *State Monad*, which holds the in which terms are used, and additional information to generate unique type variables. When a new, unique type variable is needed, *freshTypeVar* is used to generate the integer which represents the unique type variable. As an example, consider the code of snippet 23. This code snippet defines a function *constraints*, which generates the type constraints a term is subject to. A **case**-expression is used to match the type constructor of *Term* and allows us to define specific constraint generation rules for each specific *Term*.

In code snippet 23, the constraints of the *TmAbs* term are generated. First, a new type variable is generated, which represents the type of the binder. Given a term $\lambda x.tm$, *nvlam* represents the type of x. The type of x is added to the context by adding the mapping between x and *nvlam* through the *addTypeVarBinding* function. This makes it possible for term variables to use the type of the binder they refer to.

Following the context mapping, a type variable is generated which represents the type of the entire $\lambda x.tm$ expression. Before adding the constraint however, the constraints *tm* is subject to are generated. After generating the constraints of *tm* through *constraints tm*, the resulting type *ty* is used to define the constraints the term *TmAbs str tm* is subject to. For abstraction the constraint is $nvtot = nvlam \rightarrow ty$, since the type of $\lambda x.tm$ represents a function which, when applied to an argument with type *nvlam*, will return something with type *ty* of *tm*.

It is important to update the context *ctx* carefully however, as *DeBruijn* indices are also used as indices of the context. To make sure the binding of the type variable to *str* is only used within *tm*, the context is stored in *ctx* before generating the constraints of *tm*. After generating the constraints of *tm*, the context is restored.

We similarly generate the constraints for *TmApp*, as shown by code snippet 24.

Code Snippet 24: Constraint generation of application

```
TmApp t1 t2 →
  do ty2 ← constraints t2
     ty1 ← constraints t1
     nv  ← freshTypeVar
     addTypeConstraints [(ty1, Arrow ty2 nv)]
     return nv
```

In our type system, only functions may be applied to values. As a result, when generating the constraints of an application like *t1 t2*, the term *t1* must have a function type, which first accepts a value with the type of *t2*. Without inspecting the exact type of *t1* and *t2*, we do not know what the resulting type of the application is. For this purpose, a fresh variable named *nv* is created. The constraint application is subject to is $ty1 = ty2 \to nv$.

Referencing variables is not subject to additional constraints. Variables are introduced by abstraction, and as such the type of the variables is known through the context. Code snippet 25 shows how the types of variables are found, by looking at the DeBruijn index and finding the binding to the variable as introduced by abstraction.

Code Snippet 25: Constraint generation of variables

```
TmVar k →
  do ty ← deBruijn2Var typeVarBind k
     return ty
```

The type variables introduced so far are different from time variables. Later, in the unification step, all type variables are substituted depending on the type of unification used. When unifying base types, a mapping is created between type variables and base types. In the case of time expressions however, type variables are mapped to time expressions. This means that type variables are used for both unification of base types, as unification of time expressions. Unlike type variables, time variables are explicitly introduced by time abstraction, as shown by code snippet 26.

Code Snippet 26: Constraint generation of time abstraction

```
TmTAbs str tm →
  do nv ← freshTimeVar
     addTimeVarBinding (str, nv)
     constraints tm
```

Time variables are scoped by the *TmTAbs* type constructor, and as such is unique between two different scopes, regardless of the identifier used. We use integers to represent time variables. When a new time variable is generated, the last generated identifier for time variables is used and increased with one. Since the constraint generation algorithm processes terms in a specific order, we use the identifiers to reconstruct the ordering of function application during unification. This is important, as we would not be able to distinguish between a step *forward* in time (when comparing two types) and a step *backwards* in time.

As mentioned earlier, ascription is the only method through which terms can be bound to a certain type. As such, ascription is the only method through which time-dependent behaviour can be introduced. Using the term *TmAs*, we first generate the constraints of the term which is ascribed with a type, as shown by code snippet 27.

Code Snippet 27: Constraint generation of Ascription

```
TmAs tm ty →
  do tytm ← constraints tm
     ty' ← bindType ty
     addTypeConstraints [(tytm, ty')]
     return tytm
```

After ascribing a term with a type, we process the provided type *ty* by *bindType*. We process the type *ty* to replace the DeBruijn index from *ty* with the actual numeric time variable it is bound to. For instance, given the previously introduced term

```
TmTAbs "t" $
  TmAs f    $
          Arrow (TimedType TyInt (TimeExpr 0 0))
                (Arrow (TimedType TyInt (TimeExpr 0 0))
                       (TimedType TyInt (TimeExpr 0 2)))
```

, the *TmTAbs* term causes the constraint generation to generate a new time variable. Suppose the unique time variable which is generated has the value 42, then the DeBruijn index in *TimeExpr* 0 0 is changed to *TimeExpr* 42 0. After replacing the DeBruijn index in the time expression, we add the type constraints such that the type of *tm* equals the rewritten type expression *ty*′. This method may seem error-prone. For the purposes of clarification we do not distinguish between time expressions which use DeBruijn indexing and time expressions which do not. In the actual prototype however, this distinction is made. For the remainder of the discussion of the implementation this is not needed however, which is why we chose to use a single representation for time expressions here.

To enable the definition of more then just trivial examples, we also define the term *TmIf*, which represents a multiplexer[1].

Code Snippet 28: Constraint generation of Choice

```
TmIf t1 t2 t3 →
  do ty1 ← constraints t1
     ty2 ← constraints t2
     ty3 ← constraints t3
     ntv ← simpleTimeVar
     let booltype = Constant $ TimedType TyBool (TimeExpr ntv 0)
     addTimeVarBinding ("ifbind", booltype)
     addTypeConstraints [(ty2, ty3)]
     addTypeConstraints [(ty1, booltype)]
     return ty2
```

First, the constraints of the predicate is determined, after which the constraints for both branches are determined. To make sure the predicate is actually a *Bool* type, without

[1] We forgo extending the *Term* datatype for simplicity's sake

restricting the time-dependent behaviour of *t1*, we create a free time variable and use it to create a timed type as $Bool\langle tfree + 0\rangle$. We then constrain $ty1 = booltype$ and $ty2 = ty3$.

Example of Constraint Generation

To show what type of constraints are generated, we present the generation of constraints for a specific function. We use the functions defined in code snippet 29 to derive a set of constraints. The functions of code snippet 29 represent the circuit of figure 5.2. The constraints generated by the algorithm are used by the unification algorithm, which is explained in the next section. The unification algorithm also introduces additional registers to make the composition work, using the method of composition as introduced in chapter 3. Although the code definition is fairly trivial, it allows us to show how a specification is verified, and how placement of memory elements is derived.

Code Snippet 29: Functions used in constraint generation.

$delay :: Int\langle t0\rangle \rightarrow Int\langle t0+3\rangle$

$delay\ x = x$

$sel :: Bool\langle t1\rangle \rightarrow Int\langle t1\rangle \rightarrow Int\langle t1+1\rangle \rightarrow Int\langle t1+2\rangle$

$sel\ p\ x\ y =$ **if** p **then** x **else** y

$comp\ pc\ xc = sel\ pc\ xc\ (delay\ xc)$

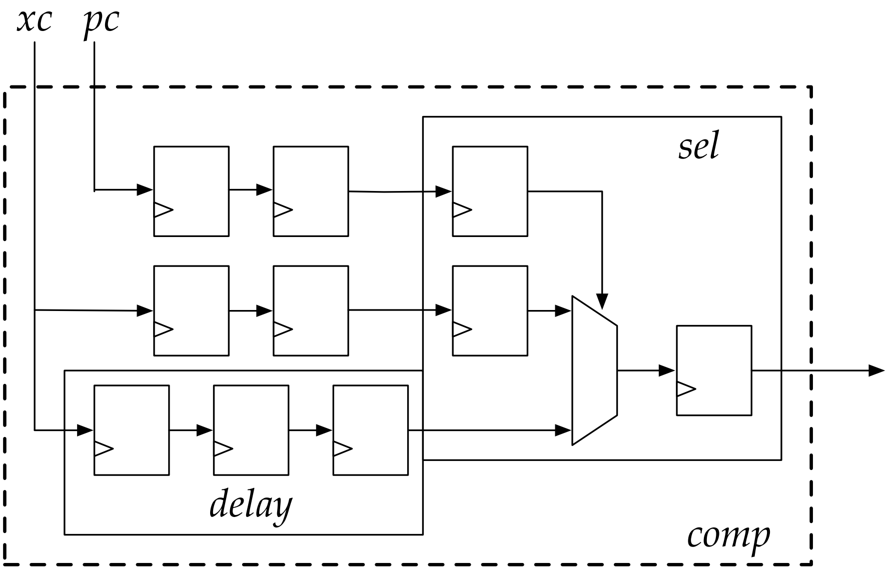

Figure 5.2: Composition of *sel* and *delayI* to create *comp*.

The code from code snippet 29 is represented by the Abstract Syntax Tree (AST) of code snippet 30. The term *comp* is passed to the *contraints* function as explained in the previous section.

Code Snippet 30: Example of constraint generation (2).

```
delayI = TmTAbs "t0" $
           TmAs  (TmAbs "xd" (TmVar 0)) $
                   Arrow    (Constant $ TimedType TyInt (TimeExpr 0 0))
                            (Constant $ TimedType TyInt (TimeExpr 0 3))
sel =      TmTAbs "t1" $
           TmAs (TmAbs "p" $ TmAbs "x" $ TmAbs "y" $
                  (TmIf  (TmVar 2) (TmVar 1) (TmVar 0)))
                  (Arrow (Constant $ TimedType TyBool (TimeExpr 0 0))
                  (Arrow (Constant $ TimedType TyInt (TimeExpr 0 0))
                  (Arrow (Constant $ TimedType TyInt (TimeExpr 0 1))
                           (Constant $ TimedType TyInt (TimeExpr 0 2)))))
comp = TmAbs "pc" $ TmAbs "xc" $
           TmApp (TmApp (TmApp sel (TmVar 1))
                              (TmVar 0))
                   (TmApp delayI (TmVar 0))
```

The constraints which are generated by the constraint generation algorithm include many type variables. We choose to represent type variables by *Xn*, where *n* is the unique identifier for the type variable. Similarly, we choose to represent the time variables by *tn*. The constraints generated by the algorithm for *comp* are the following:

$$\begin{array}{ll}
X1 = X0 \rightarrow X3 & X3 = X2 \rightarrow X15 \\
X7 = Bool\langle t2 \rangle & X13 = X2 \rightarrow X14 \\
X8 = X0 \rightarrow X13 & X8 = Bool\langle t1+0 \rangle \rightarrow Int\langle t1+0 \rangle \\
 & \quad \rightarrow Int\langle t1+1 \rangle \rightarrow Int\langle t1+2 \rangle \\
X8 = X7 \rightarrow X10 & X10 = X9 \rightarrow X12 \\
X5 = X2 \rightarrow X6 & X14 = X6 \rightarrow X15 \\
X9 = X11 & X12 = X11 \rightarrow X9 \\
X5 = X4 \rightarrow X4 & X5 = Int\langle t0 \rangle \rightarrow Int\langle t0+3 \rangle
\end{array}$$

As shown, the ascription introduces the constraints $X8 = Bool\langle t1+0 \rangle \rightarrow Int\langle t1+0 \rangle \rightarrow Int\langle t1+1 \rangle \rightarrow Int\langle t1+2 \rangle$ and $X5 = Int\langle t0 \rangle \rightarrow Int\langle t0+3 \rangle$ for *sel* and *delayI* respectively. The constraint $X7 = Bool\langle t2 \rangle$ is introduced by the *TmIf* rule. To provide a mapping between type variables and types, we must unify the constraints, which we discuss in the next section.

5.4 Unification

We chose to implement a single unification algorithm, which is flexible enough to allow unification of time expressions, as well as unification of base types. The similarities between both forms of unification are big enough to warrant a flexible definition. Moreover, at the point of creating the implementation, the method of checking the time expressions was still in development. As such, a flexible form of unification helped with development, as it allows us to easily modify the unification algorithm. The unification algorithm is shown by code snippet 32. Before applying *unify* to the set of constraints, the constraints are split in a base type part, as well as a part which only contains time expressions. The unification algorithm only works on constraints which are instances of the *Substitutable* typeclass, shown by code snippet 31.

Code Snippet 31: Substitutable typeclass

```
class Eq a ⇒ Substitutable z a where
  add   :: Substitution a → z → z
  apply :: ([Substitution a] → [Substitution a]) → z → z
  match :: a → a → z → z
```

Code Snippet 32: Unification

```
unify :: (Show a, Substitutable z a) ⇒ [Constraint a] → z → z
unify [ ] r = id r
unify (c : cs) r = case c of
  (Constant t1, Constant t2) → unify cs $ match t1 t2 r
  (Arrow s1 s2, Arrow t1 t2) → unify ((s1, t1) : (s2, t2) : cs) r
  (tyS@(Var idS), tyT)
     | tyS ≡ tyT              → unify cs r
     | ¬ $ idS ‘isFVIn‘ tyT   → let cs′ = (tyS ↦ tyT) cs
                                    r′  = add (tyS, tyT) $ apply (tyS ↦ tyT) r
                                in unify cs′ r′
  (tyS, tyT@(Var idT))        → unify ((tyT, tyS) : cs) r
  (t1, t2                )    → error "not unifiable "
```

In the substitutable typeclass, *a* represents the type of constraints, which is either a base type or a time expression. Depending on the type of constraints, a different record can be used to build the list of substitutions. The list of substitutions provides a mapping between type variables and constant types. The different operations of the *Substitutable* typeclass are explained when we discuss the unification algorithm. However, we point out that these operations, together with the polymorphic container type *z*, allow us to modify the resulting substitutions by the unification algorithm.

As shown, *unify* accepts two arguments. One is a set of constraints, the other being a record type which is defined by the *Substitutable* typeclass. To clarify the algorithm, we first focus on how it is implemented for base types, before showing how it is implemented for time expressions. The base type instance of the *Substitutable* typeclass is defined by code snippet 33.

Code Snippet 33: BaseTypes instance

```
type Substitution a = (MetaType a, MetaType a)
data BaseRecord = BaseRecord [Substitution BaseType]
instance Substitutable BaseRecord BaseType where
  add x (BaseRecord xs) = BaseRecord (x : xs)
  apply f (BaseRecord xs) = BaseRecord $ f xs
  match x y tr
    | x ≡ y = tr
    | otherwise = error "x /= y"
```

The *unify* function has five distinct cases of constraints which are handled. First, the constraint between two constant types are compared. For this, the *match* function is used, which is part of the *Substitutable* typeclass. The functionality of *match* depends on the type of constraints which are unified. In the case of base types, *match* just checks for equality. When base types are not equal, an error is raised to indicate that the checked term is not well-typed.

The second case of *unify* handles function constraints, where a function type is constrained by another function type. When a function type is constrained by a non-function type, then the term is not well-typed. If it is constrained by another function type, the individual types which make up the function types are *added* to the existing set of constraints.

The third case of *unify* matches a type variable X_{idS} to an unknown type *tyT*. If the left hand side of the constraint is equal to the right hand side, then the constraint is discarded and unification continues for the remaining constraints. If they are not the same, then it is checked whether or not the type variable X_{idS} exists as a free type variable within *tyT*. Checking for free type variables are important. Consider the constraint $X0 = X1 \to X0$, where *X0* is not free in $X1 \to X0$. This constraint is not well-typed, as we do not support recursion.

If this is not the case, then the variable X_{idS} is replaced by *tyT* in the remaining constraints, as indicated by the $(\mapsto)$ function. Since we have made a substitution in the form of $X_{idS} \mapsto tyT$, we store the substitution. However, aside from applying the substitution to the remaining constraints, the substitution is also applied to the existing substitutions, which is what the *apply* function does in the case of base types. The way existing substutions are stored depends on the type of constraints which are being unified. As a result, the functions *add* and *apply* are part of the *Substitutable* typeclass. In the case of base types, substitutions are added to the record, while existing substitutions within the record are rewritten according to the new substitution. The fourth case of *unify* is the same as the third case, only with the right hand side and left hand side switched. Finally, the fifth case represents the case when terms are not well-typed.

Code Snippet 34: *TimeExpr* instance of *Substitutable*

```
data TimeRecord = TimeRecord
  {timeSubs :: [Substitution TimeExpr]
  ,timeVars :: [(TimeExpr, TimeExpr)]}
instance Substitutable TimeRecord TimeExpr where
  add x tr = tr {timeSubs = x : (timeSubs tr)}
  apply f tr = tr {timeSubs = f $ timeSubs tr}
  match x y tr =
    let maysub = matchConstantType (x, y)
    in case maysub of
      Just z → tr {timeVars = z : (timeVars tr)}
      Nothing → tr
```

The same algorithm is used for time expressions. The code of snippet 34 shows the *TimeExpr* instance of *Substitutable*. The unification algorithm uses *TimeRecord* to store substitutions. The *timeSubs* stores the substitutions, while *timeVars* is used to verify time expressions. To create substitutions, first the unification algorithm finds the relations between time expressions which are needed to verify the time-dependent behaviour.

These relations are stored in *timeVars*. After the time-expressions are verified, the *original* set of constraints is modified. The resulting constraints are then unified *again*, after which *timeSubs* contains the final substitutions. Adding the substitutions and applying them to time variables is straightforward. However, matching time expressions is more complex, as shown by code snippet 35. The result of *matchConstantType* are added to *timeVars*, which is used to verify the time expressions.

Code Snippet 35: Matching function for time expressions

```
matchConstantType c =
  case c of
    (TimeExpr t1 o1, TimeExpr t2 o2)
      | t1 ≡ t2    → Nothing
      | otherwise → Just (TimeExpr t1 o1, TimeExpr t2 o2)
    (TimeLiteral o1, TimeLiteral o2)
      | o1 ≡ o2    → Nothing
      | otherwise → error "cannot unify literals o1 o2"
    (c1@(TimeExpr t1 o1), c2@(TimeLiteral o2))
                  → Just (c2, c1)
    (c1@(TimeLiteral o1), c2@(TimeExpr t2 o2))
                  → matchConstantType c2 c1
```

When matching time expressions, four different cases can be distinguished. Either a time expression is constrained to another time expression, a literal is constrained to another literal, a literal is constrained by a time expression, or vice versa. The first case filters the time expressions which are needed to verify the time-dependent behaviour. For instance, when typechecking the function $f\ x\ y = y$ with type $Bool\langle t\rangle \rightarrow Bool\langle t+1\rangle \rightarrow Bool\langle t+2\rangle$, the result would be the equality of $Bool\langle t+1\rangle = Bool\langle t+2\rangle$. This equality is created as y on the left hand side of the expression has type $Bool\langle t+1\rangle$, while y on the right hand side of the expressions has type $Bool\langle t+2\rangle$. Since these time variables are the same, we can determine that this is indeed allowed, as function definitions can only have *increasing* delays. However, in a constraint with different time variables, such as $Bool\langle t0+1\rangle = Bool\langle t1+2\rangle$, we cannot determine if this is actually allowed behaviour. For the purpose of checking the time constraints at a later point, the constraint is recorded and added to *timeVars*, as shown by code snippet 34.

The second case of *matchConstantType* checks literals. In the prototype typechecker, these literals are only compared for equality. For the purpose of showing that the principle of checking time expressions actually work this is enough, though it could be extended to allow delays on values which exist at a specific time.

The last two cases of *matchConstantType* check time expressions which are constrained to literals, and vice versa. When time expressions are constrained to literals, the constraint is added with the left hand side containing the literal, and the right hand side containing the time expression.

Example of Unification

Given the constraints generated earlier, repeated below for convenience, we unify them. We only show the full unification of base types here. To show unification of time expressions, we first need to generate the time variables which are checked seperately.

$$
\begin{array}{ll}
X1 = X0 \rightarrow X3 & X3 = X2 \rightarrow X15 \\
X7 = \mathit{Bool}\langle t2 \rangle & X13 = X2 \rightarrow X14 \\
X8 = X0 \rightarrow X13 & X8 = \mathit{Bool}\langle t1 + 0 \rangle \rightarrow \mathit{Int}\langle t1 + 0 \rangle \\
 & \quad \rightarrow \mathit{Int}\langle t1 + 1 \rangle \rightarrow \mathit{Int}\langle t1 + 2 \rangle \\
X8 = X7 \rightarrow X10 & X10 = X9 \rightarrow X12 \\
X5 = X2 \rightarrow X6 & X14 = X6 \rightarrow X15 \\
X9 = X11 & X12 = X11 \rightarrow X9 \\
X5 = X4 \rightarrow X4 & X5 = \mathit{Int}\langle t0 \rangle \rightarrow \mathit{Int}\langle t0 + 10 \rangle
\end{array}
$$

After splitting these constraints into base types and time expressions, we can pass them to the unification algorithm. In the case of base types, the resulting mapping is

$$
\begin{array}{ll}
X0 = \mathit{Bool} & X1 = \mathit{Bool} \rightarrow \mathit{Int} \rightarrow \mathit{Int} \\
X2 = \mathit{Int} & X3 = \mathit{Int} \rightarrow \mathit{Int} \\
X4 = \mathit{Int} & X5 = \mathit{Int} \rightarrow \mathit{Int} \\
X6 = \mathit{Int} & X7 = \mathit{Bool} \\
X8 = \mathit{Bool} \rightarrow \mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{Int} & X9 = \mathit{Int} \\
X10 = \mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{Int} & X11 = \mathit{Int} \\
X12 = \mathit{Int} \rightarrow \mathit{Int} & X13 = \mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{Int} \\
X14 = \mathit{Int} \rightarrow \mathit{Int} & X15 = \mathit{Int}
\end{array}
$$

The constraint generation algorithm defined the type of *comp* to be equal to *X1*. As shown, the resulting type of *comp* is $\mathit{Bool} \rightarrow \mathit{Int} \rightarrow \mathit{Int}$, which is as expected. The remainder of the constraints define the types of each intermediate expression which makes up *comp*. Given these types, we can reason about the types of every (partial) function and other values.

For time variables this is slightly different, as we have not checked whether time expressions are actually correct yet. However, *timeVars* result in a list of constraints, which are shown below.

$$\begin{aligned}
\langle t1 + 1 \rangle &= \langle t0 + 3 \rangle \\
\langle t1 \rangle &= \langle t0 \rangle \\
\langle t2 \rangle &= \langle t1 \rangle
\end{aligned}$$

These constraints are used to verify the complete time-dependent behaviour of *comp*, and provide placement information for memory elements. In the next section we will use the constraints listed above.

5.5 Checking Time Constraints

Multiple methods could be used to verify the time-dependent behaviour using the constraints of *timeVars*. We chose to use first order logic to express the constrained time expressions, and use SMT solvers to find integer representations for the time variables used in time expressions. Using the integer representations for time variables we can then rewrite the time expressions of compositions in terms of a single time variable and its offsets.

We chose to use the Z3[14] solver, as it is the only SMT solver which supports unbounded integers. A complete discussion about SMT solvers is out of the scope of this thesis. However, De Moura and Bjørner define SMT as follows: "A SMT problem is a decision problem for logical first order formulas with respect to combinations of background theories such as: arithmetic, bit-vectors, arrays, and uninterpreted functions."

Since the Z3 solver is not written in Haskell, we use the library *Data.SBV*[2] to handle the interfacing with Z3. *Data.SBV* provides the Haskell language with the *Symbolic* monad, which allows us to create statements in first order logic. The library then allows us to use the Z3 solver to find integer representations for time variables. The usage of SMT solvers in Haskell or other functional languages is not new. Liquid Types[41] for instance are used to extend Hindley-Milner type inference using SMT-solvers.

As an example, we use the constrained time expressions from the previous section to create a statement in first order logic, which is checked by the Z3 solver.

$$\begin{aligned}\langle t1+1\rangle &= \langle t0+3\rangle\\ \langle t1\rangle &= \langle t0\rangle\\ \langle t2\rangle &= \langle t1\rangle\end{aligned}$$

First, the proper ordering is determined. Since time variables are created in order, we know that the time variable *t1* must occur later than *t0*. Similarly, *t2* must occur later than *t1*. Using the constrained time expressions from above, we can then define the following statement using first order logic:

$$\begin{aligned}\exists t0.\exists t1.\exists t2.\quad & t1+1 \geqslant t0+3\\ & t1 \geqslant t0\\ & t2 \geqslant t1\\ & t0 \geqslant 0\\ & t1 \geqslant 0\\ & t2 \geqslant 0\end{aligned}$$

We create this statement by using the *Data.SBV* library, and ask Z3 to find integer representations for *t0*, *t1* and *t2*. We do this by using the *Symbolic* monad, as shown by code snippet 36.

Code Snippet 36: Using the *Symbolic* monad to create statements in first order logic.

```
representation =
  do t1 ← exists "t0"
     t2 ← exists "t1"
     t3 ← exists "t2"
     constrain $ t0      .⩾ 0
     constrain $ t1      .⩾ 0
     constrain $ t2      .⩾ 0
     constrain $ t1 + 1. ⩾ t0 + 3
     constrain $ t1      .⩾ t0
     constrain $ t2      .⩾ t1
     solve [ ]
```

Using the definition of *representation*, we use the *sat* function of *Data.SBV* to ask the Z3 solver to find a integer representation of *t0*, *t1* and *t2*. In our experience, Z3 automatically finds the minimal integer values, though this is not guaranteed. In the future, Z3 will also support optimization criteria, allowing us to guarantee the minimal integer values for time variables. If the time expressions are erroneous, the *sat* function will not return integer representations for *t0*, *t1* and *t2*.

In the case of verifying the behaviour of *comp* however, the *sat* function provides the following integer representations:

$$
\begin{aligned}
&\mathit{t0} = 0\\
&\mathit{t1} = 2\\
&\mathit{t2} = 2
\end{aligned}
$$

When we assume $t = 0$, we can rewrite the existing expressions of the original constraints as follows:

$$
\begin{array}{ll}
\mathit{X1} = \mathit{X0} \rightarrow \mathit{X3} & \mathit{X3} = \mathit{X2} \rightarrow \mathit{X15}\\
\mathit{X7} = \mathit{Bool}\langle t+2\rangle & \mathit{X13} = \mathit{X2} \rightarrow \mathit{X14}\\
\mathit{X8} = \mathit{X0} \rightarrow \mathit{X13} & \mathit{X8} = \mathit{Bool}\langle t+2\rangle \rightarrow \mathit{Int}\langle t+2\rangle\\
 & \quad \rightarrow \mathit{Int}\langle t+3\rangle \rightarrow \mathit{Int}\langle t+4\rangle\\
\mathit{X8} = \mathit{X7} \rightarrow \mathit{X10} & \mathit{X10} = \mathit{X9} \rightarrow \mathit{X12}\\
\mathit{X5} = \mathit{X2} \rightarrow \mathit{X6} & \mathit{X14} = \mathit{X6} \rightarrow \mathit{X15}\\
\mathit{X9} = \mathit{X11} & \mathit{X12} = \mathit{X11} \rightarrow \mathit{X9}\\
\mathit{X5} = \mathit{X4} \rightarrow \mathit{X4} & \mathit{X5} = \mathit{Int}\langle t\rangle \rightarrow \mathit{Int}\langle t+3\rangle
\end{array}
$$

We can then use unification to generate the substitutions for time expressions. As a result, the following substitutions are created:

$$
\begin{array}{ll}
\mathit{X0} = \langle t+2\rangle & \mathit{X1} = \langle t+2\rangle \rightarrow \langle t+2\rangle \rightarrow \langle t+4\rangle\\
\mathit{X2} = \langle t+2\rangle & \mathit{X3} = \langle t+2\rangle \rightarrow \langle t+4\rangle\\
\mathit{X4} = \langle t+2\rangle & \mathit{X5} = \langle t+2\rangle \rightarrow \langle t+3\rangle\\
\mathit{X6} = \langle t+3\rangle & \mathit{X7} = \langle t+2\rangle\\
\mathit{X8} = \langle t+2\rangle \rightarrow \langle t+2\rangle & \mathit{X9} = \langle t+2\rangle\\
\quad \rightarrow \langle t+3\rangle \rightarrow \langle t+4\rangle & \\
\mathit{X10} = \langle t+2\rangle \rightarrow \langle t+3\rangle \rightarrow \langle t+4\rangle & \mathit{X11} = \langle t+3\rangle\\
\mathit{X12} = \langle t+3\rangle \rightarrow \langle t+4\rangle & \mathit{X13} = \langle t+2\rangle \rightarrow \langle t+3\rangle \rightarrow \langle t+4\rangle\\
\mathit{X14} = \langle t+3\rangle \rightarrow \langle t+4\rangle & \mathit{X15} = \langle t+4\rangle
\end{array}
$$

The type of the *comp* function, after we combine the above substitution with the base type substitutions, is defined to be $\mathit{X1} = \mathit{Bool}\langle t+2\rangle \rightarrow \mathit{Int}\langle t+2\rangle \rightarrow \mathit{Int}\langle t+4\rangle$, which reflects the time-dependent behaviour of *comp*. As mentioned before, we use flexible composition, and as such the type of *comp* reflects the described circuit of figure 5.2 on page 62.

Since all substitutions are written in terms of a single time variable, all time-dependent behaviour from every expression is known. Since all time-dependent behaviour from every expressions is known, it should *in principle* be possible to derive a circuit description from the type information.

5.6 Conclusion

Even though the implementation is not complete, we have shown that verification and inference of time-dependent behaviour is feasible. The implementation is not as elegant as it could be however. Since the implementation was mainly used as a tool to figure out how to verify and infer time-dependent behaviour, it was created without a detailled plan.

If a real implementation were to be made, the usage of SMT-solvers may not be needed. While SMT-solvers allow us to define statements of first order logic, solving these statements does come at a cost. The computational complexity of SMT-solvers is known to be non-polynomical, which might make them less suited for larger projects. Moreover, since the constraints on time expressions are fairly simple, other methods could be used to check the types of expressions and determine register placement.

The implementation shown here differs from the type system of chapter 4 in a few aspects. First, the implementation does not use a δ function to only allow construction of functions which include monotonic increase of delays. As such, the given implementation might not be as robust as the type system presented earlier. Instead, ascription is used to bind a type to a term.

Furthermore, in the type system of chapter 4, we do not use type variables, nor unification. These are artifacts from adopting Pierce's algorithm, and not strictly needed for an implementation. In addition, time variables are introduced through a separate term. In the type system introduced earlier, time variables are scoped using universal and existential quantification as part of either constraints or polytemporal types.

Chapter 6

Conclusion

This thesis has produced two tangible results. A type system has been created, which can express limited forms of time-dependent behaviour. A slightly different implementation has been created, showing the practical feasibility of the type system.

We have seen that the created type system is limited; we can only express pipelining and sequences. Despite the fact that it is limited, it at least shows that certain time-dependent behavior can be expressed as part of the type system. Specification of such behaviour using the type system has proven to be pleasant, albeit slightly counterintuitive at times. The function type shows the type of data, as well as when the data is expected. By only specifying time as part of the type system, the time-dependent behaviour is known directly when the type signature is inspected. This is a boon to documentation; instead of inspecting what the function does (temporally speaking) from the statements used to create the function, the behaviour is directly shown in its type.

Compositions have the ability to add memory elements. This can be considered a side-effect of function composition, and is a mixed blessing. Even though it is useful in creating compositions under the assumption that values are referentially transparent, the behaviour of compositions can be impredictable. We avoided adding the same side effect in usage of higher order functions, by restricting the types of higher order functions to be equivalent.

However, more complex behaviour is difficult to add. Even for something as simple as sequences, we needed to add additional rules to the type system. This makes the type system *hard* to extend. Without extending the type system to allow more complex behaviour, it is hard to see how such a type system could be useful for real implementations of hardware. The type of behaviour we currently support, namely pipelining and sequences, may not be very error-prone in practice. If it is not error-prone and the given type system can not be extended further, then dedicating an entire type system to making it less error-prone would be overkill.

It is also questionable to what degree the type system verifies the behaviour of compositions. Often, the primary purpose of a type system is to verify certain behaviour. Here, verification is only possible within the extent of the type signatures given by the designer. Without any type information, only descriptions of combinational logic are created, making temporal verification superfluous. However, the given type system does provide the user with a hands-on tool to express time-dependent constraints. While verification is not completely automated, if the user provides the parameters which need to be verified, then at the very least the provided information can be verified. This form of verification is different from the type PSL offers. In PSL, behaviour is verified during simulation, while type systems offer verification during compile-time. This in itself is a powerful feature.

As we have not provided a proof of soundness, it is hard to provide a definitive answer as to whether the given type system behaves correctly. This is a downside of creating a type system from scratch. Unfortunately, we did not expect the type system to be so unintuitive. Since the typing rules alone are not enough to determine whether something is well-typed, the context functions are used in is important. As the context influences the correctness of a type, it becomes more difficult to reason about the soundness of the given type-system. We can not take the correctness of the typing rules in chapter 4 at face value as a result.

On the whole, the usefulness of expressing time appears to be big enough to warrant further investigation. Even though the constraint-based approach is not successful in allowing specification of all sorts of hardware, other techniques might be more successfull. In this thesis we struggled to both define a type system, while at the same time trying to research how to define time as part of this type system. Perhaps existing type systems allow us to more easily research how time-dependent behaviour can be expressed (partly) as part of the type system. As a result, direct applications for the presented type system and implementation seem limited. Our approach shows that it is feasible to express time as part of the type system, but several difficulties need to be overcome in order to create a true temporal type system.

Chapter 7

Future Work

The given type system and implementation leave much to be desired. In this chapter we provide a few different options for future work. The idea of expressing time as part of the type system is interesting enough to produce future research.

Solving Constraints Without SMT-solvers

Currently we rely on SMT-solvers to both check whether the constraints hold, as well as finding suitable places to add registers. Use of SMT-solvers offers flexibility; during development it is useful to have a very powerful tool which can handle many situations. However, checking the generated constraints using SMT-solvers is overly complex for the situation we currently use them in.

In our current approach, we never eliminate constraints. It may be possible to eliminate constraints when applying λ-abstraction. When applying λ-abstraction (rule TT-Abs, def. 15, page 44), the resulting timed type encloses both x (the parameter) and e (the expression used to build the abstraction). If the time variables used in the constraints for both x and e are not used in an application involving the abstraction $\lambda x.e$, then these constraints should be removable. It is difficult to see a situation where constraint can not be eliminated, other than possibly through user defined constraints (see section 7). Barring those situations, if we are able to determine whether all constraints can be removed under abstraction, it would make solving of constraints trivial. The rule TT-App (rule TT-App, def. 17, page 45) is then the only way constraints can be added, as currently the only *other* way is through abstraction. As a result, the number of variables to solve for is extremely limited in the average case. Given that abstraction removes constraints, the set of variables used in the constraints is limited, as they can only be created by successive application inside a single abstraction. In turn, this makes simple iterative methods more feasible.

Different Approach to Expressing Time in a Type System

In this thesis, we used constraints as our method to express time as part of the type system. We chose to use constraints, as they greatly simplify the process of developing a type system. Due to constraints, we have effectively split the checking of types, from (re)constructing types. This makes it easier to create a separate algorithm, or in our case, forgo creation of the algorithm altogether.

By using a separate algorithm for checking constraints and solving for the number of registers needed, it becomes possible to "dynamically" add registers in order to uphold referential transparency. This means that function application has a side-effect, which is not apparent from the operation itself. A bigger context is needed in order to understand the implications of function application.

This is a situation which is not very desireable, as it also increases complexity for the designer. With a straightforward, rule-based system, it is easy to figure out why the system rejects or approves a certain expression. When a bigger context is needed in order to reconstruct and/or typecheck an expression, it can also become more complex to understand the decisions the type (re)construction algorithm makes. In the same vein, a possible soundness proof would also be harder to create.

To sidestep this issue, different types of type systems could be investigated. Subtyping could be investigated by forming a subtyping relation where $a\langle t+n\rangle$ is a subtype of $a\langle t\rangle$ for some $n \in \mathbb{N}_0$. The difficulty then is to define a subtyping relation between functions, which does not cause unreasonable effects.

Refinement types[18], where function arguments can have preconditions, and function results can have post conditions, could be used to express the time constraints of functions. Liquid types[41] have an implementation in the Haskell language through LiquidHaskell[1] and could be used to eliminate our own usage of SMT-solvers. However, LiquidHaskell also uses SMT-solvers internally, so using the refinement approach of other languages might be more interesting from a research point of view.

Let-bindings & Polymorphism

Currently, our type system does not support polymorphism as used in the language ML[32] or other languages. To introduce such polymorphism, let-polymorpishm as introduced by Milner could be introduced. First, a let binding like the one below should be created.

Definition 21: Typing rule for Let-bindings

$$\frac{C;\Gamma \vdash e_1 : \forall t_1.\tau_1 \quad C;\Gamma \vdash e_2 : \forall t_2.\tau_2}{C;\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2}\ \text{TT-Let}$$

However, in our type system as introduced in chapter 4, every different usage of x in e_2 uses the same time variables. Polymorphism could be added by changing the time variables at every usage. Perhaps changing the rule TT-Gen (def 14, page 43) to replace the time variable t in τ with a fresh variable would suffice. However, this change must also be reflected in the constraints which e_1 is subject to. It is difficult to see whether simply using α-conversion on the constraints would work out.

Similarly, we have not defined traditional polymorphism. The type system of CλaSH is pleasant to use for its use of higher order functions and polymorphism. The supplied type system could be expanded to include polymorphism of base types. To do so, the approach of HM(X) could be used, or the system could be extended with something similar to algorithm W.

Feedback

Even though we have described memory elements as part of the type system, we have not been able to describe the feedback operation. The feedback operation consists of a memory element which, contrary to pipelining, feeds a value back into the circuit that produced the value. For instance, the circuit of figure 7.1 uses the feedback operation to create a sum of multiple multiplications.

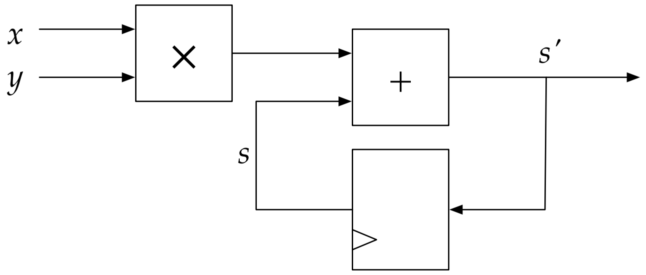

Figure 7.1: Circuit of multiply-accumulate.

Even though we have defined sequences as syntactic sugar, we are not able to use the same sequences to define feedback. If we define a single resulting value from the feedback operation to exist at time t, then it depends both on the input(s) from time t *plus* all other inputs which have had an effect on the content of the feedback register. An obvious type for a function which represents feedback would be

$$\mathit{feedback} :: \mathit{Int}\langle 0 \mathinner{..} t\rangle \rightarrow \mathit{Int}\langle t\rangle$$

, that is, the resulting value at t, would depend on all values from 0 to t.

The problem however, is how to define the transition from a type as $\mathit{Int}\langle 0 \mathinner{..} t\rangle$ to $\mathit{Int}\langle t\rangle$. Even if a way exists to describe such a transition, would it be able to accurately describe different forms of feedback? Additional research is needed to classify different forms of feedback, and finding properties of those different classes which can be expressed by a type system.

Multiple Clock Domains

In this thesis, we used a single clock for entire circuits. However, for many areas in DSP, multiple clock domains are used. We focus on clock domains which have different speeds, but are synchronised (e.g. the clocks do not drift).

Extending the current type-system with multiple clocks is difficult, as polytemporal types only allows a single time variable in its binder. Either multiple time variables need to be used in order to reason about different clocks, or the relation between different clocks can be mapped to time variables in a different set of constraints. The latter would surely make the type system very complex, making it hard to reason about the system and its soundness. The former is also difficult, as currently the single time variable used in a polytemporal type is necessary for expressing our typing rules (page 43).

User defined Constraints

The constraints of our type system are currently used to reason about the compositions of time-dependent behaviour. However, time constraints could also possibly be used to verify whether restrictions imposed on the circuit from the environment are adhered to. For instance, two functions could be defined, both with time variables in a different scope, together with a global constraint that the time variables used by those two functions are always equal. This would then create an additional constraint which has to be adhered to no matter how these functions are used.

Inferred Register Placement

Currently we do not provide an algorithm which, given a set of terms and types belonging to those terms, can determine where memory elements ought to be placed. In this thesis we only discussed placement of memory elements very briefly, only to indicate that it should *in principle* be possible. Since the type system shows possibilities with regards to retiming, an approach which determines whether retiming is intended by the designer would be more than welcome.

Multiple Time Variables

We only use a single time variable to define a polytemporal type. A single time variable is needed in order to be able to reason about the effects of time expressions under abstraction and application. However, having more than one time variable would open up many possibilities to describe more general temporal functions. Unfortunately, having multiple time variables makes it hard to reason about functions and their temporal constraints. As such, this option should preferably be researched together with a different approach to expressing time as part of the type system.

Appendix A

Language Comparison

This chapter was written as part of a literature study to become familiar with the domain of functional hardware description languages as well as synchronous languages. Although there are many functional hardware description languages, we only discuss ForSyDe[42]. Just as there are many functional hardware description languages there are also many synchronous languages. Having no previous experience with any of the synchronous languages available we discuss Lustre[23], as there also exists an extension named Pollux[40], which allows translation from a Lustre specification to a hardware implementation.

However, before comparing both the CλaSH language to Lustre and ForSyDe, an introduction to CλaSH itself is needed.

A.1 CλaSH

CλaSH is a language which leverages the language Haskell[26]. Leveraging means the host language and its toolchain are modified to fulfil a different role. In the case of CλaSH it means that the back-end of the GHC compiler is replaced to allow VHDL generation. This does mean that updates to the GHC compiler can not automatically be transferred to CλaSH.

Leveraging has some upsides however, as the entire chain of parser, lexer, type-checker, et cetera does not have to be written from scratch, but may be inherited from the host language. Since the guest language is bound to the host language it creates a dependency on the host language, which makes it much harder to deviate from the semantics and syntax of the host language. When deviation from the host language is required, it may be harder to facilitate this deviation when compared to a language which was created from scratch.

Following this short introduction on how CλaSH relates to Haskell we will introduce the basic concepts of the language and which hardware designs it allows us to describe.

Language basics

Since CλaSH uses Haskell it is also defined in terms of Haskell. Haskell is a strongly and statically typed language, which is a type of language which can catch errors in reasoning early. This is especially nice for both rapid development and development of critical systems.

> ◇ **Strong & Static Typing**
> While the definition of a strong type system versus a weak type system and a static type system versus a dynamic type system is not fully agreed upon, O'Sullivan et al. define[35] a strong type system as follows.
> A strong type system is able to detect a class of errors in reasoning at compile-time by limiting forms of expressions. When a type system is more restrictive in what form of expressions are allowed we can say it is strong *in comparison* to another type system which is less restrictive. In short, in a strong type system "types cannot go wrong"[32], as mentioned by Milner.
> If we take Haskell as an example for a strong type system, then C[27] would be an example of a weak(er) type system. For instance, it is perfectly valid C to interpret data with a certain type as a completely unrelated type by using *void* pointers. As such *void* can be considered a hole in the type system.
> This is also related to the ability to coerce (cast) a value. In Haskell this is not possible; we may only restrict a type, not convert one type to the other without any restriction like C allows. For instance, the type *a* may be restricted to *Int*, but an *Int* may not be restricted to *Char* as this implies conversion. For this purpose Haskell offers primitive functions which do allow certain conversions, for instance between an *Integer* value and a *Integral* value.
> Unlike strong typing, static typing does not give additional safety directly, but merely indicates that all expressions have types, and that these types are known at compile-time. This has the added advantage of catching errors before any code is executed, while being less flexible in dealing with run-time type changes.

CλaSH is a synchronous language which describes circuits in a structural way. Each component within a system is described in terms of subcomponents and can be described down to the gate level. By synchronous we mean that all stateful elements are updated simultaneously. Moreover, the validity of values is limited to a certain timeframe.

CλaSH is structural as opposed to behavioural. This means that the structure of an expression in the CλaSH language must in some form represent the expected structure of the generated hardware. This is different from the behavioral approach where functionality is described in terms of its behavior. When functionality is described in terms of behaviour we do not care what the resulting structure is, as long as the expected behaviour remains intact. As a result a structural language is *more* restricted when compared to a behavioural language.

CλaSH is excellent at describing structural forms of hardware, at least when we limit ourselves to considering combinational logic. For instance, the circuit shown in figure A.1 can easily be described in CλaSH, as code snippet 37 shows. As shown, CλaSH's representation of the circuit is both concise and accurate in representing combinational logic. Of course this is not the only representation, as CλaSH is flexible in how to define a circuit, shown by code snippets 38 and 39.

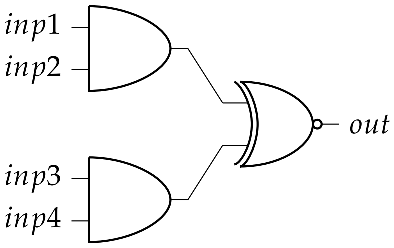

Figure A.1: A simple combinational circuit

Code Snippet 37: CλaSH definition of a simple circuit (1)

```
simpleCircuit :: Bit → Bit → Bit → Bit → Bit
simpleCircuit inp1 inp2 inp3 inp4 = hwnot (hwxor s t)
  where
    s = hwand inp1 inp2
    t = hwand inp3 inp4
```

As can be seen in code snippet 39, CλaSH also supports vector types, which have type-level bounds on their size. Currently the length on the type-level is encoded using the D_x type and with d_x at the term level. Here, x is an integer representing the number of elements a vector may have. Since Haskell at the point of the development of CλaSH

Code Snippet 38: CλaSH definition of a simple circuit (2)

```
simpleCircuit :: (Bit, Bit, Bit, Bit) → Bit
simpleCircuit (inp1, inp2, inp3, inp4) =
  hwnot $ hwxor (hwand inp1 inp2) (hwand inp3 inp4)
```

Code Snippet 39: CλaSH definition of a simple circuit (3)

```
simpleCircuit :: Vector D4 Bit → Bit
simpleCircuit v = hwnot $ hwxor s t
  where
    s = hwand (v ! d2) (v ! d3)
    t = hwand (v ! d0) (v ! d1)
```

did not have type-level integers it was not possible to encode these directly into the type. Instead McBride shows[31] how to implement 'fake' type-level integers using type families and multi-parameter type classes. Unfortunately this has the added effect of not being as easy to use when compared to other aspects of the Haskell type system, since relatively trivial errors can not be reported in a concise manner. This is one of the downsides of the leveraging approach mentioned earlier; we cannot modify the Haskell type system without creating our own branch of the GHC compiler. A recent update to GHC allows integer literals on the type-level, which will be used in the upcoming version of the CλaSH compiler. On the other hand moving the compiler to a language which supports dependent types could be considered. The Idris[10] language has the ability to represent true vector types at the cost of losing top-level type inference. The leveraging approach can easily be adapted to languages such as Idris which, like Haskell, also uses a simplified normal form. This is important as in general a simplified language is easier to translate than a more complex language, by sheer complexity.

While type-level vectors are possible in Haskell, it does not mean all work on type-level vectors is necessarily complete, as it is currently very hard, if not impossible, to make Haskell's pattern matching work with vector types. For instance, it would be very nice if we could rewrite the code from code snippet 39 to something like the code shown in code snippet 40.

One option to handle this would be to use the *IsString* typeclass.

Code Snippet 40: CλaSH definition of a simple circuit (4)

```
simpleCircuit :: Vector D4 Bit → Bit
simpleCircuit⟨v0 : v1 : v2 : v3⟩ =
  hwnot $ hwxor (hwand v0 v1) (hwand v2 v3)
```

◇ **Typeclasses in Haskell**

Like many other languages the language Haskell allows a form of polymorphism. Polymorphism is generally used to increase code reuse. The language Haskell allows polymorphic specifications by making it possible to act on data without knowing the exact form of data. This is done by introducing type variables. When a type is variable it is often denoted *a*, or any other lower case letter representing the type. This is a form of unrestricted polymorphism, where no information whatsoever is needed of the type. One example where this is used is in functions acting on lists. For instance, we can always take the first element of a list, as such functions work on the "structure" of the type.

In practice however it is often desirable to create a restricted form of polymorphism. Typeclasses have the ablity to restrict polymorphism such that whatever form the type may take, it must implement the functions described by the typeclass in order to be well-typed.

For instance, if we consider the *Show* typeclass, then we must implement a function of the form *show* :: *a* → *String*, which transforms the type *a* to *String*. We can then implement this typeclass for a type which does not have a String representation and use this type in every function which has a restriction on polymorphism in the form of *Show*. In those types of function the type signature would look like *Show a* ⇒ *a* → *a*, which implies that for this function to have a valid type, *a* must be a valid instance of the typeclass *Show*.

The problem with using the *IsString* approach is that it does not use compile-time checks, so one could write this kind of code:

```
select1 :: Vector D2 Bit → Bit
select1 "<v1,v2,v3,v4>" = v1
```

and it would not cause any issues until the code is simulated. Considering CλaSH's safety properties given by its type-system we would not expect this to be possible, as the type of $\langle v1, v2, v3, v4 \rangle$ would not be *Vector D2 Bit*. The problem could be easily avoided though if we were to create a preprocessor which solely transforms this string representation `"<v1,v2,v3,v4>"` to an actual representation which the Haskell type-system can reason about. While this may solve the problem it will cause issues when error messages are to be displayed, as the Haskell type system has no knowledge about the extensions we have added.

State representation

State representation is particularly difficult with CλaSH, as compositition using stateful components is not similar to regular function application. In "Higher-Order Abstraction in Hardware Descriptions with ClaSH"[20] an approach to represent state and components is introduced, by using the arrow construction used in Haskell. A detailed introduction to arrows is out of the scope of this study, though "Generalising monads to arrows"[25] may be referenced when an introduction is required. The arrow notation is more difficult to understand, making it less suitable for hardware designers without a functional programming background. Gerards et al. mention[20] that all state functions which can be defined in CλaSH must have the following type signature:

```
state → input → (state, output)
```

Using this we can create a function, defined in code snippet 41, which simply sums up all values.

Code Snippet 41: Sum as a stateful function in CλaSH

```
sum :: Int → Int → (Int, Int)
sum s i = (s′, s′)
  where s′ = s + i
```

As shown, in this case the sum is both the output as well as the state to be remembered until the next cycle. In other cases these might be distinctly different.

The decoupling of state and logic via the *State* datatype is necessary due to Haskell being a pure functional language. Since functions are primarily considered to be pure, side-effects such as statefulness can never be part of pure functions. But when we see functions as representations of physical hardware components this becomes troublesome, as we would normally expect the state to be *encapsulated* in the definition of the component. The *sum* function from earlier would then have the type $sum :: Int \to Int$, with no state visible.

Stateful functions have to be defined using a specific structure of a function. This structure can then be mapped to a component arrow, of which the definition is listed in code snippet 42.

Code Snippet 42: Component type declaration as used in[20]

```
newtype Comp i o = C {
  exec :: i → (o, Comp i o)
}
```

The details pertaining to the exact syntax of code snippet 42 are not that important, but what can be seen is a definition of a component which contains a function which, when provided with input, produces output and a new component representation. The notion of a continuation has already been used in the past to support Functional Reactive Programming[17,24,45] where arrows are also used as a basis for implementation. Functional Reactive Programming is based on streaming and continually varying values, where functions represent behaviours which act on these continually varying values.

Components in CλaSH can be created through a process called lifting using specialized function primitives. Lifting refers to the change of perspective, where the arrow perspective is considered a higher level, component-based perspective. Lifting then lifts a function from a low-level perspective where the state is an explicit function argument, to a high-level, component based perspective.

When lifted to the component level components can be used to compose bigger components through arrow syntax. This lifting operation is also responsible for assigning initial state to the component. Gerards et al. define a multiply-accumulate component as shown in code snippet 43, which clearly shows the decoupling between initial state and the function definition.

Ideally we would like to define this component in one function, with perhaps adding the initial state as an optional parameter. Given these components one can use the arrow syntax to easily control which component is connected to which component. The syntax to use arrows was created[36] by Paterson after the actual arrow implementation

Code Snippet 43: Stateful multiply-accumulate in CλaSH

```
mac acc (x, y) = (acc′, acc′)
  where acc′ = acc + x ∗ y

macA = mac ‘lift‘ 0
```

itself. Without going into details here, suppose we want to define a component which combines two *mac* components in a bigger , arrow-based component. This component, listed in code snippet 44, sums the results of both *mac* components.

Code Snippet 44: Composing stateful multiply-accumulate components

```
twoMacAcc = proc (w, x, y, z) → do
  r1 ← mac ‘lift‘ 0 ≺ (w, x)
  r2 ← mac ‘lift‘ 0 ≺ (y, z)
  returnA ≺ r1 + r2
```

There, the arguments of the component are listed as a quadruple, with its inputs listed after the **proc** keyword. There is a lot more to be said about arrows and its syntax, but that is outside of the scope of this study. “A new notation for arrows”[36] can be used if more information is required.

High-level Description

The descriptions we have seen so far were either pure functions like in code snippet 37, or component based through arrows. Aside from these it is also possible to describe structures from a high-level perspective by merely indicating what shape the structure must take. For instance, instead of describing a sum of a vector of integers through applying a function on each element manually , one can use higher-level functions such as *foldr*, *map* and *zipWith* which allow for more concise code. Additionally the same code is also faster to develop. These functions are only available as primitives in CλaSH through *vfoldr*,*vmap* and *vzvipWith*. As such it is currently impossible to create custom high-level descriptions directly.

One example on how these high-level descriptions may be used is to sum values of a vector, as shown by code snippet 45.

CλaSH provides the following high-level descriptions:

Code Snippet 45: Sum defined over a vector of integers

```
sumV :: Vector D4 Int → Int
sumV = vfoldr (+) 0
```

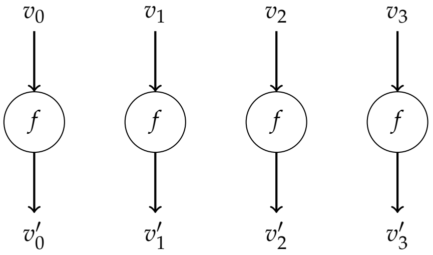

Figure A.2: Structural interpretation of *map*

- *map*, which is used to map a function on a vector of values.
- *vfoldr*, which is used to fold a vector of values into a single value. Other uses are also possible depending on the function passed to the fold, e.g. a function which constructs a new vector.
- *zipWith*, which takes a vector of tuples and uses a function to combine both elements of the tuple into a single value.

If a more complete description of the semantics of these functions in Haskell is required "Real World Haskell"[35] or "Learn you a Haskell"[30] may be referred to. For a more detailed look at the semantics of these functions in CλaSH, "CλaSH: From Haskell to Hardware"[6] may be referred to.

As mentioned before, these descriptions have a structural interpretation. For instance, *map* represents the structure of the circuit of figure A.2. Both *fold* and *zipWith* also have structural interpretations, as shown by figures A.3 and A.4.

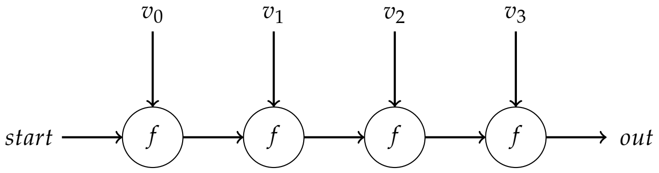

Figure A.3: Structural interpretation of *vfoldr*

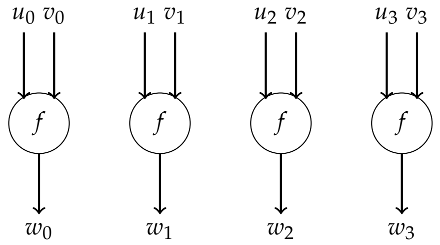

Figure A.4: Structural interpretation of *zipWith*

With most important features of CλaSH explained we can now introduce the Lustre language, after which we will compare CλaSH to it.

A.2 Lustre

The Lustre[23] language is not used solely as a hardware description language. It is used for modelling and code generation for safety critical systems. Even though an extension in the form of Pollux[40] exists to generate hardware from these descriptions, we focus on the Lustre language itself, as more material is available to base a comparison to CλaSH on. Moreover, we do not focus on the process of hardware generation, but merely on the semantical value of descriptions themselves. Lustre was chosen as it has a number of interesting features which make it a natural match for hardware description languages, such as being synchronous and data-flow oriented. It also allows decompositional design through nodes.

As explored in the previous section, the synchronous approach works well to create an abstract notion of time and can be used in combination with functional languages. Dataflow[3] within Lustre is considered a high level programming language which enables modelling of systems using flows of data which are transformed by nodes in the flow network. According to Halbwachs et al. the combination of a synchronous model and data flow languages was not investigated before, but the Lustre language combined the two by "proposing primitives and structures which restrict data flow systems to only those which can be implemented as bounded memory automata-like programs"[23].

Data representation

Within Lustre, data is represented in the form of a *flow*, which is a pair made of

- A possibly infinite sequence of values of a given type;
- A clock, which represents the instances of time when values exist (c.q. are valid)

Since Lustre is a synchronous language, values of the clock do not correspond to any real moments in time but to abstract moments in time. Lustre tries to model cyclic behaviour and allows operations to directly act on flows. From the entire program description and operations on flows the basic clock can be derived, from which all other clocks in the system can be derived. The basic clock is simply the smallest possible distinction between two moments in time. Temporal operators may be used to create slower clocks, which will be explained shortly.

When defining operations on data, Lustre always acts on the entire flow of data, e.g given a flow X and Y, the following equation:

$$Z = X + Y$$

is equal to

$$\forall n \in \mathbb{N}_0 . z_n = x_n + y_n$$

, with n being instances of time where both x and y exist.

Temporal operators

Besides having basic arithmetic operations on flows such as addition and multiplication, Lustre also has a number of temporal operations which specifically modify flows without operating directly on the data of a flow:

- The temporal operator **pre**, shown below, allows us to define basic recursion.

  $$y = \mathbf{pre}\ (x)$$

  This would define a new flow, named y, which is simply the right shifted version of x. This also means y_0, the 0^{th} value of the flow y, would have no valid value. Invalid values or values which have no real value are called *nil* within Lustre. The resulting flow would be

  $$y = (nil, x_0, x_0, x_1, \ldots, x_n).$$

- To avoid *nil* values regular values can be injected into the start of a flow through using the $\rightarrow$ operator, demonstrated below.

$$y = 0 \to (x + \textbf{pre}\ (y))$$

This defines a flow named y, with y_0 having value 0 and with every subsequent value being the sum of x until that moment. The resulting flow would be

$$y = (0, x_0, x_0 + x_1, \ldots, \textstyle\sum_0^n x_n).$$

- Lustre also supports multiple clockrates through different operators, such as the **when** operator. The **when** operator maps a flow of booleans to a flow of values, sampling only the values on moments when the boolean flow is *true*. The code below shows the mapping of the flow x to a clock which is twice as slow as the previous clock.

$$\begin{aligned} &b = true \to (\neg\ \textbf{pre}\ (b)) \\ &y = x\ \textbf{when}\ b \end{aligned}$$

This defines a flow b with values $(b_0, b_1, \ldots, b_n) = (true, false, true, ...)$ and transforms $(x_0, x_1, \ldots, x_n)$ to $(x_0, x_2, \ldots, x_n)$ with a clock which is twice as slow as the basic clock.

- Upsampling is also possible with the exception of upsampling the basic clock. The **current** operator interpolates a flow of values to a flow of boolean values with a clock faster than its own. For instance, taking the flow $x = (x_0, x_2, \ldots, x_n)$ like above, applying it to the flow $(true, true, \ldots, true)$ gives a flow with values: $(x_0, x_0, x_2, x_2, \ldots, x_n, x_n)$.

Program structure

Of course, only being able to define expressions using the statements introduced in the previous subsection is not very useful for larger projects. To decompose functionality, Lustre introduces the concept of "nodes". Within Lustre a flow network is a composition of nodes, in which nodes may be reused. Code snippet 46 shows the basic definition of a node.

In the description two outputs are defined, s and t. The t output is defined as a sum of the previous value of t, combined with the addition of x and y. The output s is defined to be 0 when the reset input is toggled, otherwise it will sum the previous value of s together with x and y. In this example s represents a partial summation, while t represents a total summation.

Reusing nodes is also straightforward, as it is much like function application. For in-

Code Snippet 46: Node definition in Lustre

```
$\textbf{node}\ \mathit{COUNTER}\ (x, y : \mathit{int}; \mathit{reset} : \mathit{bool})\ \mathit{returns}\ (s, t : \mathit{int});$
$\textbf{let}$
  $s = 0 \rightarrow \textbf{if}\ \mathit{reset}\ \textbf{then}\ 0\ \textbf{else}\ \textbf{pre}\ (s) + x + y;$
  $t = 0 \rightarrow \textbf{pre}\ (t) + x + y;$
$\textbf{tel}$
```

stance, the node from code snippet 47 uses the *COUNTER* from before to sum three consecutive sums of x and y and the total sum. In that example the reset is switched every three cycles, while x and y are summed in s only when the reset is *false*.

Code Snippet 47: Node reusing in Lustre

```
$\textbf{node}\ \mathit{SUM3}\ (x, y : \mathit{int})\ \mathit{returns}\ (s, t : \mathit{int});$
$\textbf{let}$
  $\mathit{rst} = 0 \rightarrow \textbf{if}\ (\textbf{pre}\ (\mathit{rst}) = 5)\ \textbf{then}\ 0\ \textbf{else}\ \textbf{pre}\ (\mathit{rst}) + 1;$
  $(s, t) = \mathit{COUNTER}\ (x, y, \mathit{rst}\langle 3);$
$\textbf{tel}$
```

From observing code snippet 47 we can conclude the following:

1. it would be nice to have integer references to previous points in time, so we would be able to write something as

   $$\mathit{rst} = \mathit{repeat}\ 3\ \mathit{true} \rightarrow (\neg\ \textbf{pre}\ (3, \mathit{rst}))$$

   , which is a lot clearer and to the point. The *repeat* operation would then result into a flow $(\mathit{true}, \mathit{true}, \mathit{true}, \mathit{nil}, \mathit{nil}, \ldots, \mathit{nil})$, after which $\neg$ **pre** $(3, \mathit{rst})$ defines the remaining cyclic behaviour by generating the next three values as the inverse of the last known value of the flow.
2. Compositions of nodes only show the inputs and outputs of the composition. As a result, when composing nodes, the original definitions of the nodes have to be consulted to reason about the behaviour of the composition.

Lustre also allows assertions for runtime verification and code generation optimization. For instance the assertion

```
assert ¬ (x and y);
```

will trigger a run-time assertion error if x and y are both true. Lustre will also use this to help with code generation, though details on how this is done are not listed. While run-time assertions are nice to have, for critical applications it is not desirable to merely have error signalling without error handling. This is especially the case in the context hardware architectures where performance is required.

Arrays and recursion

To allow high-level constructs arrays were introduced to Lustre at version 4. Basically these work like vectors in CλaSH and VHDL. The sizes of these types have to be known at compile-time, as Lustre simply expands each array depending on how many elements it has. This is not really a problem, unless one wants to reuse a design and parameterization is required, as Lustre does not support it.

Halbwachs and Raymond show[22] some examples and details concerning array syntax, which we will briefly repeat here. For instance, code snippet 48 shows a ripple carry adder using an array of four full adders.

Code Snippet 48: Ripple carry adder in Lustre

```
node ADD1 (a, b, c_i : bool) returns (s, c_o : bool);
let
  s   = a xor b xor c_i;
  c_o = (a and b) or (b and c_i) or (c_i and a);
tel
node ADD4 (A, B : bool^4) returns (S : bool^4, carry : bool)
var C : bool^4;
let
  (S [0], C [0])           = ADD1 (A [0], B [0], false)
  (S [1..3], C [1..3])     = ADD1 (A [1..3], B [1..3], C [1..3]);
  carry                    = C [3];
tel
```

The $ADD1$ node is a straightforward implementation of a full adder, which is then used in the ripple carry definition. The ripple carry is less straightforward, especially the second equation as it applies an array to a node which only accepts single flows, not arrays of flows. Lustre allows this sort of behavior, as the compiler can reason about definitions such as these since it only uses compile-time size definitions. Halbwachs and Raymond call[22] this polymorphism, but since this does not allow any length of arrays to be used, it is a limited kind of polymorphism.

Lustre only supports a very specific kind of recursion which is based on being able to determine when the recursion terminates at compile-time. This is related to totality in programming, a practice where partial functions are not allowed. Partial functions are undefined for a subset of their domain, which is needed to support general recursion. Recursion in Lustre is defined by a function which has a certain number of inputs and one constant which is know at compile-time. This constant needs to be part of the terminating condition and is needed to unroll the recursion at compile-time. Code snippet 49 shows the basic principle.

Code Snippet 49: Halting recursion in Lustre

```
node REC (const x : int; y : int) returns (z : int)
let
  with x = 0
    then y
    else y + REC (x − 1, y);
tel
```

The above code simply multiplies one input with the other by using the plus operator, e.g. rewriting multiplication as a series of addition. It is important that x is a constant value here, that is, there is one value which x represents, and this value may not change during the process defined in this node. Since this is a constant value it is known at compile-time and as such can be used to unroll the recursion.

We can also define a recurrence relation which captures the essence of recursion. A sequence of the natural numbers can be defined using the → and **pre** operators:

$$c = 0 \rightarrow \textbf{pre}\ (c) + 1;.$$

Comparison of CλaSH and Lustre

With the basic concepts of both CλaSH as Lustre explained we can now compare the two languages. At first glance both languages seem to have little in common, as Lustre is not really a hardware description language, even though it can be used to model hardware. However, we can look at its syntax and paradigm and compare those.

Lustre is built around flows, making distinct moments in time explicit. It also allows us to refer to previous values of a flow and as such is a stream-based approach. This can be considered a more behavioural approach when compared to CλaSH. Within CλaSH one would have to explicitly store and move values around to achieve the same effect as Lustre, as shown in code snippet 50, as CλaSH is a structural language where we name wires or values.

Code Snippet 50: Sum of the last three consecutive values in CλaSH

```
add3 :: (Int, Int) → Int → ((Int, Int), Int)
add3 (s1, s2) i = ((s1′, s2′), o)
  where
    s1′ = i
    s2′ = s1
    o = i + s1 + s2
```

Of course, there are also alternatives, such as constructing a single register and using arrow notation to combine two of these into a final component. This does not increase the readability however, as it is hard to keep track of how values are used. The structural approach makes it easy to reason about the resulting hardware representation, yet it is hard to discern what the structure is from this piece of code.

Lustre's syntax makes this more concise, as shown in code snippet 51, though the lack of relative indexing of flows is still not very elegant. The Lustre code is more concise and actually explains how *add*3 should behave in time relative to a certain point, where the CλaSH code is still about manipulating values and "where" they should go.

Code Snippet 51: Sum of the last three consecutive values in Lustre

```
node ADD3 (i : int) returns (o : int);
let
  o = pre (i) + pre (pre (i)) + i;
tel
```

Aside from the flow aspect, Lustre also offers multiple clocks, although all of these are assumed to be synchronized. The same could be implemented in CλaSH, although support for this is currently not available. The **when** and **current** statements make it easy to upsample or downsample a given clock, which is not implemented concisely in CλaSH in its current state.

Of course, CλaSH is a language which is specifically designed for hardware description, while Lustre only allow hardware generation through Pollux. Even so, Lustre gives some nice insights into state, especially how state can be considered a temporal phenomenon. Lustre can easily do this because, while it does not allow cyclic definitions entirely, it does allow cycles when a **pre** operator is present. A description such as

```
x = 0 → pre (x) + pre (x);
```

would not be possible in CλaSH without using some form of flow or a state machine which represents such a flow. It is unfair however to view Lustre's recursive definitions as superior to CλaSH's arrow-based components, as it is perfectly possible to define sequential logic in CλaSH. However, as CλaSH is a functional language, a more "pure" recursive specification would be preferable, which is exactly what Lustre offers.

Additionally, both Lustre and CλaSH allow bounded vectors or arrays. In Lustre it is very easy to create slices, also known as subvectors, from vectors, while it remains a bit cumbersome in CλaSH, due to the nature of the host language Haskell.

However, one major advantage CλaSH has over Lustre is the type system. Since CλaSH allows type inference, it is possible for a designer to query the type system; the type system can aid the designer. Type inference allows the compiler to derive types from the structure of expressions themselves. In Lustre, when an error is encountered it is hard to find out exactly what went wrong, as the Lustre verification only points us in the right direction. It never concisely points out what the actual error is. In CλaSH the verification *is* the type system, and as such provides the programmer with a hands-on tool to interface with the compiler, leading to a reduction in time spent to find bugs or faulty logic.

A.3 ForSyDe

The ForSyDe[42] language, like CλaSH, uses the language Haskell to provide a new language te develop embedded systems. At first the language was focussed on simply modelling systems in general and simulating the descriptions using the GHC interpreter. With the addition of VHDL generation[4] it became a language in which one can model and synthesize hardware descriptions. The hardware description capabilities are the parts which this section will focus on.

ForSyDe is a language which currently only supports synchronous systems, approach used by many other hardware description languages as well. Many hardware description languages use a synchronous model of time as it is relatively simple to simulate. It also is an easy model from the designer's perspective. ForSyDe, unlike CλaSH, uses the Embedded Domain Specific Language (EDSL) approach to avoid the effort needed to create a compiler, type-checker, et cetera from scratch. A Domain Specific Language (DSL) is, as its name implies, a language designed specifically with one domain in mind. A EDSL however, is a specific kind of DSL, which uses a host language in which the guest language is *embedded*. Embedding a language has some advantages, like using existing syntax, tools of the host language, and not needing to develop an entire compiler from scratch. The EDSL approach does not offer many advantages over the leveraging approach used by CλaSH, a property which will be discussed later in more detail. For now it is enough to know that leveraging binds a language to a specific *implementation*, while the EDSL approach binds a language to a specific *language*. In practice GHC is the defacto standard compiler for Haskell, and as such there is not much difference in dependence on either the language or the specific implementation.

Processes

Within ForSyDe, systems are modelled using an abstraction called processes. These processes can be combined through networks. This approach is very similar to dataflow approaches discussed before. Perhaps unsurprisingly, ForSyDe uses Haskell's list structure to represent signals. A process can then be seen as a function which transforms one signal to another signal. However, as functions operate on entire signals, it is harder for a compiler to discern what *elements* of the signal are used. This makes it harder to do optimalizations based on these descriptions, since functions which are referential transparent with regards to the elements of a signal offer many ways to optimize code, a quality which is often associated with Haskell.

The fact that signals are needed in ForSyDe can be seen in how processes are defined. ForSyDe supplies a number of process constructors which take a function as input, an approach often used in Haskell. First, they define a signal in the style of lists with :– as the Signal constructor

```
data Signal a = a :– Signal a |NullS
```

, which is used in the definition of the *mapSY* constructor, shown by code snippet 52.

Code Snippet 52: The *mapSY* process constructor in ForSyDe

$$mapSY :: (a \to b) \to Signal\ a \to Signal\ b$$
$$mapSY\ \_\ NullS = NullS$$
$$mapSY\ f\ (x :\!-\ xs) = f\ x :\!-\ (mapSY\ f\ xs)$$

The process resulting from *mapSY* clearly behaves the same as a regular *map* function over lists as is commonly used in Haskell.

Aside from *mapSY*, ForSyDe also supplies the following process constructors:

- *zipWithSY*, which is analogous to *zipWith* from Haskell; it takes two *Signal*s and uses a function to combine these *Signal*s into one.
- $delaySY_k$, which delays a *Signal* with k discrete time steps. It can be used to create feedback, for instance to create the *sourceSY* constructor.
- *sourceSY*, which generates an infinite series of outputs.
- *scanlSY*, which acts much like the the *scanl* of Haskell. When thinking of streams it makes sense to use *scanl* to encapsulate state. A stream of values together with a function which acts on two individual elements of the stream can then be used to 'store' intermediate results. Of course, since these intermediate results only depend on the previous values of the Signal and the function applied to it it is still a pure function.
- *mooreSY*, a constructor for a Moore machine; a process from which the output only depends on the internal state.
- *mealySY*, a constructor for a Mealy machine; a process from which the output depends on both the internal state and the input signals.
- *unzipSY*, which is analogous to the *unzip* function in Haskell.

These process constructors can be composed to create new, more complex processes, as in code snippet 53.

This is equivalent to the following Haskell code using lists:

$$mac\ xs\ ys = scanl\ (+)\ 0\ (zipWith\ (*)\ xs\ ys)$$

Sander and Jantsch list[42] more detailed examples, but this example should be enough to explain the principle.

Code Snippet 53: Stateful multiply-accumulate in ForSyDe

```
macSY :: Num a ⇒ Signal a → Signal a → Signal a
macSY x y = scanlSY (+) 0 prod
  where
    prod = zipWithSY (∗) x y
```

Deep and shallow embedding

While not mentioned before, the only way to generate VHDL from a ForSyDe model is to use the so-called “deeply embedded” approach. The regular ForSyDe model does not contain the information needed to generate VHDL code. This is due to the fact the language is embedded in the host language. Since it is embedded in the host language, it is restricted by the host language as well. This restriction has certain downsides, as one would normally not be able to *observe sharing* within a certain program, functionality which is needed to detect cycles in a program. Cycles are often needed in hardware descriptions to describe feedback, so observable sharing is necessary. Fortunately Lava[9,11] faced the same problems at an earlier stage and solved this through observable sharing[12].

In order to use this a non-backwards compatible change had to be made, which is referred to as the “deeply embedded” approach. This approach heavily depends on using Template Haskell, which makes the AST of the program c.q. hardware description accessible from within the language itself. The usage of Template Haskell introduces several problems, which will be looked at in detail in the comparison with CλaSH. The gist of it is however that to use functionality many Haskell programmers take for granted, such as polymorphism and higher order functions, functions that depend on them need to have a specific type signature. These functions must then be passed into Quasi-Quatation[43] blocks, all of which only hinder readability and do nothing to aid the designer.

Program structure

To provide more structure to the hardware design, the designer has the ability to define circuit definitions using hierachical structures. These structures have a strong relation with the structures used in VHDL, probably both for reasons of familiarity as well as ease of translation. ForSyDe supplies the following hierachical structures:

- A *Port* is almost identical to VHDL’s port clause, aside from the fact that in ForSyDe it is not allowed to mix input and output ports; two separate ports have to be made per system, provided they indeed have both in and output ports. The port structure serves as an interface between the outside world and the system. However, there wasn't a lot of concrete information available as to what the actual use of this structure is. Since it is so closely related to VHDL it seems natural to use these interfaces to mix regular VHDL code with generated VHDL code. These ports are named *InPort* and *OutPort* and consist of a list of tuples containing the name and the type of each input or output.

- A *Circuit* is defined as a function which transforms an *InPort* to an *OutPort*. The circuit definition then simply consists of applying functions to the specific input ports of the circuit.
- The circuits defined can then be turned into a *Block*, which is a white-box which can not be connected directly to other system structures. A *Block* can be instantiated, or transformed into a *BlockIns* structure. This structure is a black-box, but its inputs and outputs can be used to connect to other parts of the system. A *Block* and *BlockIns* are similar to entity-architecture pairs and component in VHDL.

Arrays and Vectors

Vectors in CλaSH are not much different from those used in ForSyDe as the library introduced[5] by Acosta was used as an inspiration for CλaSH as mentioned in "Clash: From Haskell to Hardware"[6]. As such it is not very interesting to go into details, as any comparison will simply state they are very similar and offer the same advantages and disadvantages. We already discussed the advantages and disadvantages of this approach in the context of CλaSH in a previous section.

Comparison of CλaSH and ForSyDe

CλaSH and ForSyDe seem to have a lot in common. They both use Haskell, albeit in different ways. The methods used to translate the model is different, as well as the model itself and the way this model is presented to the designer.

ForSyDe is a language which makes streaming explicit and allows the designer to create processes which act on these streams. CλaSH on the other hand does not deal with streams at all and is more structural in its approach, allowing the designer to define a circuit in terms of *wires* or *individual values*, not in *streams* of values. CλaSH is able to describe a circuit structurally because it is able to transform a program description from the outside of the language it is created in. This is not the case for ForSyDe, as it can only analyze whatever the language Haskell and its GHC implementation allow it to analyze.

As mentioned before, CλaSH leverages Haskell and as such is bound tightly to a specific implementation of GHC. In CλaSH the translation is done outside the programming environment of Haskell, so it can look from the outside in and has access to Haskell's Core language to transform definitions from the Haskell source straight to VHDL. This is different from the ForSyDe approach, which uses Template Haskell to access the AST of the program from within the limits of the language it is embedded in. This limits on the translation of *user defined* descriptions:

- When defining a function and passing it to a process constructor the number of parameters must equal the number of arguments; currying is not allowed.
- The signature of the function supplied to the process constructor has to be made explicit, regardless of the function.
- Pattern matching is limited to literals, variables and wildcards; no pattern matching on data constructors is allowed.
- **where** and **let** clauses are not allowed.
- Polymorphism is only supported for instances of a certain typeclass. In "Hardware Synthesis in ForSyDe"[4] it is mentioned it was only possible for *Int* and *Bool* at the time, with options to expand it. Nevertheless, it still makes polymorphism less clear since it changes the type for for instance the *mapSY* from

  ```
  mapSY :: (a → b) → Signal a → Signal b
  ```

  to

  ```
  hdMapSY :: (HDPrimType a, HDPrimType b) ⇒ HDFun (a → b) →
  HDSignal a → HDSignal b.
  ```

  The details of this notation are not that important, as they are only needed by the underlying system to enable transformation to VHDL.
- Passing a function to a process constructor requires the use of Template Haskell.

Most of these limitations are quite severe when considering how these features are used in Haskell itself. Not having **where** or **let** clauses is undesirable, since it is considered idiomatic Haskell. The same could be said for currying, except the value of currying depends on what the semantics of currying are in the context of hardware description languages. Within CλaSH currying is used for parameterization, something which seems natural at first, but might benefit from a more strict separation between parameters and inputs. One of the bigger issues is the use of Template Haskell and Quasi Quotation.

Quasi Quotation is a shorthand method to *splice* existing code into Haskell at compile-time. A full overview of Quasi Quotation is out of the scope of this study, but giving an example will most likely clear up any misconceptions.

To show how this influences the syntax we can define a simple component which just adds one to its input, as shown in code snippet 54.

Code Snippet 54: Definition of hdPlus1 in ForSyDe

```
hdPlus1 :: HDSignal Int → HDSignal Int
hdPlus1 = hdMapSY doPlus1
  where doPlus1 = $(mkHDFun [d | doPlus1 :: Int → Int
                                 doPlus1 a = a + 1 |])
```

The definition of *doPlus1* consists of a splicing (*$()*) operation and a Quasi Quotation operation (*[d | |]*). The Quasi Quotation operation takes the definition of the function, extracts its AST at compile-time, and the splicing operation injects it in the proper point, also at compile-time. There are some downsides with this approach, which make it not really suited for a hardware description language.

Firstly, it can give uninformative error messages when there is an error in the definition of the function which is being spliced into code, or when there is an type error between two Template Haskell code fragments, although this has been greatly improved in recent GHC versions. Secondly, it makes it harder to inspect program elements from the interpreter, as the quoted part has the type *QExp* when evaluated in the interpreter. Thirdly, and most importantly, it adds extra syntax for no apparent reason, at least as far as the hardware designer is concerned. Without knowing why these Quasi Quotations exists, which would be common as it is considered advanced Haskell, it will be very hard to ignore all this syntactic 'noise'. While the reasoning given is clear, it will make it harder for a designer to adopt this language.

The CλaSH implementation is much simpler in comparison and, with this simplicity in mind, can be both interpreted as a structural representation, naming the *wires*, or a behavioral implementation, naming the *values*:

```
plus1 x = x + 1.
```

Of course one can get by without even naming anything due to partial application and *point-free*[1] programming through a simple $plus1 = (+1)$.

A.4 Conclusion

After discussing the various languages we can conclude that each of these languages have their advantages and disadvantages. CλaSH offers a seemingly seamless translation of Haskell to VHDL when the idiomatic style of Haskell is preferred. This is the major downside of the ForSyDe approach, which cannot use common features as pattern matching, top-level type inference and other features considered idiomatic Haskell. The fact that it depends on Quasi-Quotation and Template Haskell is also a downside in that, for the uninitiated at least, it is not clear *why* the splicing and quotation operations are needed.

Unlike Lustre, ForSyDe and CλaSH do not offer protection against errors in synchronization. While Lustre does not have a type system like Haskell-based languages, it at least offers the ability to verify whether synchronization and composition behaves as expected. The downside of this approach is that it is stream-based, which may make it harder for hardware designers to reason about circuits without developing a feeling for the system first. The fact that it was not developed with hardware in mind does not help, as it never tries to relate its concepts to the concepts hardware designers are familiar with.

While ForSyDe has the power of the type system, it does nothing to exploit it. The type system in ForSyDe mainly serves as a tool of limitation pertaining to making the model work in Haskell, not a tool of limitation to aid the developer in actually creating hardware descriptions. Lustre has such a tool, but does not give the developer the hands-on it requires to actually have it aid development. There, it is only used as a tool for catching errors.

Like ForSyDe, Lustre takes the streaming approach. From a hardware perspective this is not desirable, as it does not match well with the mental model most hardware designers have.

While sequences of values are important for how execution of computations behave, generally designers want to decompose a problem, solve the decomposed problems and compose solutions. With sequences of values the compositions of solutions may

[1] Point-free programming refers to the practice of not mentioning the arguments in a function definition. It is used when composing functions without actually labelling the data

influence each other, with effects of these compositions largely unclear or hidden in the specific structure of expressions.

As a result, it appears that CλaSH has a more intuitive and simple model of expression, even though it does not handle state as elegantly as its counterparts. CλaSH does have the advantage of flexibility due to its leveraging approach, making it easier to introduce specific features which could make its type system more helpful in aiding the designer with hardware design.

Bibliography

[1] Liquidhaskell web page. http://goto.ucsd.edu/~rjhala/liquid/haskell/blog/about/. Accessed: 2013-08-11.

[2] Data.SBV library description. http://hackage.haskell.org/packages/archive/sbv/0.9.7/doc/html/Data-SBV.html. Accessed: 2013-06-16.

[3] W.B. Ackerman. Data flow languages. *Computer;(United States)*, 2, 1982.

[4] Alfonso Acosta. Hardware synthesis in forsyde. *Sweden: KTH/ICT/ETS*, 2007.

[5] Alfonso Acosta. Forsyde tutorial, 2008.

[6] Christiaan Baaij. Clash, from haskell to hardware. Master's thesis, University of Twente, 2009.

[7] Hendrik Pieter Barendregt. *The lambda calculus: Its syntax and semantics*, volume 103. North Holland, 1985.

[8] Gérard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of computer programming*, 19(2): 87–152, 1992.

[9] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: hardware design in haskell. In *ACM SIGPLAN Notices*, volume 34, pages 174–184. ACM, 1998.

[10] Edwin C Brady. Idris—: systems programming meets full dependent types. In *Proceedings of the 5th ACM workshop on Programming languages meets program verification*, pages 43–54. ACM, 2011.

[11] K. Claessen and G. Pace. An embedded language framework for hardware compilation. *Designing Correct Circuits*, 2, 2002.

[12] K. Claessen and D. Sands. Observable sharing for functional circuit description. *Advances in Computing ScienceâĂŤASIANâĂŹ99*, pages 78–78, 1999.

[13] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM, 1982.

[14] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[15] Bruno Dutertre and Leonardo De Moura. The yices smt solver. *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, 2:2, 2006.

[16] Cindy Eisner and Dana Fisman. *A Practical Introduction to PSL (Series on Integrated Circuits and Systems)*. Springer-Verlag New York, Inc., Secaucus, NJ, 2006.

[17] Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, 1997. URL http://conal.net/papers/icfp97/.

[18] Tim Freeman and Frank Pfenning. *Refinement types for ML*, volume 26. ACM, 1991.

[19] Gerhard Gentzen. Untersuchungen über das logische schließen. i. *Mathematische zeitschrift*, 39(1):176–210, 1935.

[20] M. Gerards, C. Baaij, J. Kuper, and M. Kooijman. Higher-order abstraction in hardware descriptions with clash. In *Digital System Design (DSD), 2011 14th Euromicro Conference on*, pages 495–502. IEEE, 2011.

[21] Jean-Yves Girard. Une extension de lâĂŹinterpretation de gödel à lâĂŹanalyse, et son application à lâĂŹelimination des coupures dans lâĂŹanalyse et la théorie des types, 1971, 1971.

[22] N. Halbwachs and P. Raymond. A tutorial of lustre. *available by anonymous ftp from imag. imag. fr as le/ftp/pub/LUSTRE/tutorial. ps*, 1993.

[23] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[24] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Arrows, robots, and functional reactive programming. *Advanced Functional Programming*, pages 1949–1949, 2003.

[25] J. Hughes. Generalising monads to arrows. *Science of computer programming*, 37(1): 67–111, 2000.

[26] S.L.P. Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[27] Brian W Kernighan, Dennis Ritchie, Stanley B Lippman, and Josee Lajoie. C programming. *Language*, 2009.

[28] M. Kooijman. Haskell as a higher order structural hardware description language. 2009.

[29] Charles E Leiserson and James B Saxe. Optimizing synchronous systems. In *Foundations of Computer Science, 1981. SFCS'81. 22nd Annual Symposium on*, pages 23–36. IEEE, 1981.

[30] M. Lipovaca. *Learn You a Haskell for Great Good!: A Beginner's Guide*. No Starch Press, 2011.

[31] Conor McBride. Faking it simulating dependent types in haskell. *Journal of functional programming*, 12(4-5):375–392, 2002.

[32] Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

[33] Zainalabedin Navabi. *VHDL: Analysis and modeling of digital systems*. McGraw-Hill, Inc., 1997.

[34] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *TAPOS*, 5(1):35–55, 1999.

[35] B. O'Sullivan, D.B. Stewart, and J. Goerzen. *Real World Haskell*. O'Reilly Media, 2009.

[36] R. Paterson. A new notation for arrows. In *ACM SIGPLAN Notices*, volume 36, pages 229–240. ACM, 2001.

[37] Benjamin C Pierce. *Types and programming languages*. The MIT Press, 2002.

[38] Benjamin C Pierce. *Advanced topics in types and programming languages*. The MIT Press, 2005.

[39] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, 12(1):23–41, 1965.

[40] Frederic Rocheteau, Imag-lgi Grenoble, and Nicolas Halbwachs. Pollux: a lustre based hardware design environment. 1994.

[41] Patrick M Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. *ACM SIGPLAN Notices*, 43(6):159–169, 2008.

[42] I. Sander and A. Jantsch. System modeling and transformational design refinement in forsyde [formal system design]. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 23(1):17–32, 2004.

[43] T. Sheard and S.P. Jones. Template meta-programming for haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16. ACM, 2002.

[44] Donald E Thomas and Philip R Moorby. *The Verilog® Hardware Description Language*, volume 2. Springer, 2002.

[45] Z. Wan and P. Hudak. Functional reactive programming from first principles. In *ACM SIGPLAN Notices*, volume 35, pages 242–252. ACM, 2000.

[46] Mitchell Wand. A simple algorithm and proof for type inference. *Fundamenta Informaticae*, 10(2):115–121, 1987.