DEPENDENT TYPING IN HARDWARE DESCRIPTION LANGUAGES

David Lewis

Bachelor of Engineering
Software

Department of Electronic Engineering
Macquarie University

November 22, 2015

Supervisor: Dominic Verity

Acknowledgements

Dom, for his generosity, calm and insight.

My mother, for her love, support and eagle-eyed proofing.

STATEMENT OF CANDIDATE

I, David Lewis, declare that this report, submitted as part of the requirement for the award of Bachelor of Engineering in the Department of Electronic Engineering, Macquarie University, is entirely my own work unless otherwise referenced or acknowledged. This document has not been submitted for qualification or assessment at any academic institution.

Student's Signature:

Date:

ABSTRACT

In recent years there has been growing interest in the modernisation of the ageing industry standards in Hardware Description Languages (HDLs). The proliferation of embedded systems and targeted hardware optimisation is encouraging a reexamination of the design process and driving increased demand for formal verification of hardware systems. In response to these needs, a body of research has grown around the application of functional programming techniques to the hardware design process. Recent developments in Dependently Typed (DT) languages extend and enhance the capabilities of functional languages in a manner that is highly suited to the implementation of an Embedded Domain Specific Language (EDSL) for hardware design. In an iterative design process, efficiency can be gained by reducing iteration time or number of cycles. Both of these aims can be achieved by increasing expression and information flow, allowing designers to make more informed decisions regarding later stages of the process, and providing directly applicable constraints on future iterations. Dependently Typed Languages allow a designer to more richly express information about the type of data managed in a model circuit, are more natively robust against common errors, and provide formal means of proving the correctness of an implementation. This thesis explores the implementation of Oz, a Hardware Description Language embedded in Idris, a modern DT language.

Contents

List of Figures

List of Listings

Chapter 1

Overview

1.1 Introduction

By far the most common Hardware Description Languages (HDLs) for specifying and designing hardware are VHDL and Verilog. In some ways, these languages are powerful tools. They include dimensions of both structural and behavioural specification – the ability to describe a design both in terms of what it does and its component parts. However, while powerful, these tools are not precise, and over the last two decades they have stagnated in relation to modern programming languages and practices.

Both VHDL and Verilog were developed ad-hoc in response to specific use cases. At the inception of these HDLs and their associated compilers, simulating complex circuits was a formidable computational challenge. To alleviate this, certain parts of the toolchain had testing mechanisms baked in, and were optimised for simulation rather than compilation. As a result, only a subset of the languages are synthesisable to hardware, with no concrete separation between the two interpretations of the design. While simulation can still be computationally taxing, it is no longer to a degree where it could be considered reasonable to pollute compiler design with simulation considerations.

As a result of the haphazard development of the languages, they are both ambiguous and too permissive, allowing non-physical designs without any warning. Synthesis of a design to actual circuitry is almost trial and error when venturing away from a poorly documented body of best practice. The languages themselves form an impediment to the implementation of solid engineering discipline.

Classic HDLs also have lagged behind modern programming languages in expressiveness and style. Support for higher-order design is limited to the inaptly named 'generic' construct which allows only linear expansion. In many cases we would like to express a generic design that has some other more complex recursively defined substructure – a tree parameterised by depth, for example. The lack of expressive features of this kind has led to poor coding practises across the industry.

1.2 Motivation

The obvious manifestation of these issues in industry is wastage of time and effort in the design process. The tools prevent engineers from making informed decisions and effectively testing and verifying designs. This thesis presents a continuation on the body of research toward embedding an HDL in a modern programming language, taking particular note of the following key areas of concern.

1.2.1 Expressiveness

Expressiveness is a quality of code that fuses terseness, readability and reusability. It values code that concisely expresses an idea, and provides an appropriate level of abstraction or specificity for the domain. An expressive language also values higher-order design and tends to prefer well-founded, enforced rules over code-contracts and best practice. These properties are useful in any language, and simply form a baseline of user-friendliness for any prospective system.

Expressiveness may come into tension with other non-functional requirements. Type safety and static checking require a lot of information, most of the onus for which is put on the user. As an example, Python is considered highly expressive and user friendly, but the weak type system and lack of safety make it ill-suited to many applications.

1.2.2 Verification

Reliability and correctness is a key concern in hardware systems which may be impossible to update in-place should a bug be found. Areas of concern include automotive, aeronautical and industrial applications, where unverified hardware systems may have critical safety impacts.

Unfortunately, current industry standard HDLs provide very little support for formal verification.[13] In 2008 a new standard for VHDL was released which partially incorporates a Property Specification Language (PSL). The PSL allows a designer to encode the intended properties of a circuit in a more abstracted manner. However, this system does not deliver a mechanism to check these properties – rather, it opens the door for such mechanisms to be included in later compilation stages.

Recently, promising research has been made showing how hardware may be verified by first order logic.[6] Conventionally, many formal verification techniques have relied on either exhaustion or propositional logic – extraction properties and success criteria and submitting them to a SAT-solver.[17] These techniques quickly become computationally infeasible and circuit sizes and complexity increase. First order logic adds predicates and quantification to the logical language, which brings huge classes of verification problems into tractability. In particular it helps with the verification of generic circuits by induction, allowing whole families of circuits to be verified by parametricity.

1.2.3 Information Propagation

Hardware design is typically an iterative process, which can be made more efficient in two ways – reducing the number of cycles, or reducing the length of the cycles. In both cases the key is to increase information flow through the stages of the process.

By increasing the amount of backward-propagation, more informed design decisions may be made regarding the impacts in later stages. This can come in the form of finding more effective ways to translate the input requirements of one process into success criteria for the preceding processes. Through enriching the type system, these success criteria can be encoded directly in the type of a circuit representation.

A key example of how this can aid in the hardware design process is in the annotation of timing information. At the end of a long development iteration, it may be discovered that some key pipeline in a circuit is too long. We could increase the efficiency of a successive iteration by encoding new timing constraints in the circuit model.

1.3 Representing Circuits

A core prerequisite for the creation of a domain specific language is a full understanding of the data structures involved in representing elements of the domain. This thesis addresses the design of synchronous logical circuits. That is, circuits which respond to clock pulses produced by some external source.

1.3.1 Circuits as Graphs

There is an obvious and natural way to codify these logical circuits as graphs. Logic gates or other physical primitives form the nodes, with wires forming the edges. Simple circuits may be entirely combinatorial, mapping a series of inputs to outputs through a ladder of gates. However, any complexity will be achieved through feedback. Viewed as a graph, this is equivalent to introducing cycles.

Circuit graphs have certain properties motivated by the physical nature of the components forming the vertices and the intent of the designer. For one, the graph may be considered directional, in the sense that there is an implication of 'information flow' along the edges of the graph. Additionally, cycles in the graph must contain a delay – a memory element that will be implemented as a latch.

During a given clock cycle, we can unlink graph cycles by treating the output of the latch as a static value. Necessarily, this will turn the directed graph into a Directed Acyclic Graph (DAG), in which any node has an associated dependency tree. The leaves of this tree may be external ports, genuine static values, or the unrolled latches.

In VHDL and Verilog, the DAG is represented by top level non-blocking signal assignments, which may not contain cycles. The delays are assignments occurring inside blocks guarded by a timing constraint, which models the process of taking the appropriate signal and feeding it back into the latch.

1.3.2 Structural and Behavioural

Structural specification of circuits has a direct mapping to the components and wires of a circuit. An HDL using this style enshrines implementation details in the abstractions of the language. A language in which the user specifies nodes and edges of the circuit graph directly is structural.

Behavioural specification of circuits adds a level of abstraction to the interface of an HDL. An example might be a `case` statement or a conditional block. These may have fairly obvious implementations in circuitry, but this information is not directly expressed in the language. A language with a set of combinators by which subgraphs – each being a signal transformer abstracting some hardware operation – can be composed into larger systems is behavioural.

1.3.3 Deep and Shallow

When building an embedded domain specific language, a key design decision is the level of embedding of the domain model. This decision impacts the design of the language and the API with which it communicates with other tools in the domain toolchain.

In a deep embedding, we use classes or data types to represent the domain objects. This is useful since all the domain elements can be manipulated as values in the host language, typically providing maximum flexibility. However, such a DSL often requires more 'scaffolding' to both create models and interact with them.

In a shallow embedding, we instead use features of the host language to encode domain objects. This can be more readily set up, as it reuses existing functionality. However, inspecting and manipulating models requires introspective and reflective features in the host language which are often complex.

1.4 Prior Art

In Chapter 2 we discuss the work that has been done in this area. Bringing modern programming language features to hardware description has most successfully been achieved by embedding in a modern host language, as an Embedded Domain-Specific Language (EDSL).

The Lava family of languages provide expressive, usable interfaces for circuit specification. In 1998 the original Lava[3] paper was presented, describing how well-suited modern functional programming languages are to specifying circuits. Advanced features such as type polymorphism and higher order functions can be used to drastically improve the expressiveness of an HDL. Lava advocated for structural representation of circuits in the base library, with layers of behavioural abstraction on top.

In a follow-up paper expanding on this research, Claessen[5] did a great deal of work exploring various implementation options. The key challenge identified at that time was around how to represent the circuit structure.

Expressing cyclic structures through recursion is challenging in functional languages. In order for a program traversing or manipulating such structures to terminate, it is necessary to reify cycles into an explicit structure. Claessen's thesis favours a particular method of achieving this known as Observable Sharing, described in Chapter 2. It has proven to be incompatible with the functionality and design ethos of dependently typed languages.

Where Lava addressed verification, it relies on methods of exporting internal structures to SAT solving libraries for testing by exhaustion. These methods rapidly become computationally infeasible for larger designs, and may often be replaced by contemporary native verification features in dependently typed languages.

As an alternative to Lava, CλashCλash[1] and Π-ware[16] have been developed more recently. These both include ways to represent feedback cycles explicitly. These languages have also started to explore modern forms of verification and property propagation that are a key target of this research. The most advanced of these is Π-ware, which brings to bear the tools available in the dependently typed language Agda. Π-ware returns to a behavioural specification paradigm, in which the user defines circuits as signal transformers rather than representing the wires themselves as first class objects. However, Agda is intended primarily a theorem-proving engine, and designer Pizani Flor states that Π-ware still requires significant improvements in terms of ease-of-use.

1.5 Dependent Types

In this document we aim to discover the utility of dependently typed languages in the development of an embedded hardware description language.

In dependently typed languages, types can depend on terms. That is, types are first class language constructs, and may be manipulated in the same manner as values. These richer types allow more of the semantics of a program to be explicitly encoded at the type level. The type checker may then confirm at compile time that the implementation complies with that specification.

In Chapter 3 we give a brief summary of the salient features of Idris, a modern dependently typed language.

1.5.1 Application to HDLs

Functional programming languages with dependent types lend themselves neatly to the domain of representing circuits. As a data-flow rather than control-control flow language, we can improve the expressiveness and accuracy of an HDL by including richer types in function signatures.

Dependent types also give us extremely flexible generic types. We can define abstract circuit structures that are indexed by values or parameterised by types. This can drastically improve code re-use which is a key failing of VHDL and Verilog. Where VHDL generics can do very basic linear generics – variable width arithmetic for example – a

dependently typed language could provide dynamic tree or heap structures in hardware.

Finally, the type checker plays an important role in verifying properties of circuits. This relationship stems from the Curry-Howard Correspondence[10,18] which states that types are equivalent to propositions and programs equivalent to proofs. Dependent types allow the types to be sufficiently precise to make this concept extremely powerful.

A dependent type system can fully host the mechanics of first order logic – propositions, predicates and quantification. Any first-order theorem about a circuit design may be asserted over a circuit representation, and statically checked for correctness. In many cases this obviates the need for exhaustive property checking and has the advantage of hosting the properties natively for use in higher-order design.

Chapter 2

Background and Related Work

In this chapter we survey in greater detail the existing work done to modernise HDLs and the technology we use to further this work.

2.1 Lava

2.1.1 Lava 98

The initial version of Lava was created and presented in 1998 as an abstract hardware description library.[3] Lava 98 has a structural specification via shallow embedding, in which an EDSL uses the mechanics of the host language to encode domain objects. Circuits are plain Haskell functions with circuit semantics being encoded in function application and pattern matching, and wires are represented using references. In a sense the abstract syntax tree (AST) of Haskell is used to encode the circuit as a whole.

The classic example in many Lava papers is a simple half adder, which would be represented as follows:

```
halfAdd :: Circuit m => (Bit, Bit) -> m (Bit, Bit)
halfAdd (a, b) =
    do carry <- and2 (a, b)
       sum <- xor2 (a, b)
       return (carry, sum)
```

In this context, the type variable `m` is an instance of the `Circuit` class. Circuit is one of a hierarchy of classes that subclass Monad, providing a computational context in which values can reside. The mechanics are beyond the scope of this paper, but the monad carries the information involved in symbolically defining the nodes of the circuit graph. Functions over the Circuit monad can be chained together to share the same circuit context, which is syntactically represented through the `do` block.

Extensions of the Circuit monad would then contain further mechanics required by whatever purpose the abstract circuit was put to. Example interpretations in Lava 98 included simulation, verification and compilation to VHDL. This demonstrated that the

interpretations can be disjoint from one another and loosely coupled to the representation over which they operate. Freed from these interpretation concerns, the representation may focus on providing a succinct and precise circuit description, and a flexible interface by which to inspect it. Unfortunately, this pattern has been largely lost in later versions of Lava.

2.1.2 Lava 2000

In 2001, Claessen[5] presented a paper expanding on Lava, and examining several implementation strategies.

Observable Sharing

The major impact of the Lava 2000 paper was the transition from the monadic-declarative to a more direct style. The construction of sequential circuits in Lava 98 required a manner of explicit plumbing sorely lacking readability and reusability. In addition, the automatic unrolling of generic circuits to arbitrary sizes was not well addressed.

Hence, Claessen proposed a non-conservative language extension termed 'observable sharing.' In Lava 2000, circuits with feedback are declared using infinitely recursive structures. These circuits cannot be evaluated directly for compilation or simulation, since such a program would not terminate. This extension exposes enough of the underlying mechanisms to allow a circuit traversal algorithm to notice when two nodes refer to the same underlying memory.

Combinators and Interfaces

Motivated by recurring use cases, Claessen introduced a more complete treatment of abstract combinators over arbitrary circuits. These did not have a particularly rigorous underpinning theory however, leading to some dysfunction in the library. Some combinators had unusually specific applications, or were provided without an obvious symmetrical counterpart.

Lava 2000 used Haskell's `List` type to encapsulate the input and output interfaces to a circuit. However, the intent of lists is to represent zero or more results from some operation. They do not provide enough information at the type level to safely specify a serial combinator, for example. This was a serious impediment to the type safety and compile-time verification of circuits.

Interpretations

Building on the representation/interpretation duality, Claessen presented some canonical interpretations. The 'standard' interpretation directly computes values from the shallow-embedded Haskell functions over some initial values. The 'symbolic' interpretation creates a more abstract structure that could be used for verification purposes. By rendering the

circuit representation into a series or propositions in the correct form, it could be exported to a SAT solver to produce property proofs.

2.1.3 Kansas Lava

Kansas Lava is a version of Lava developed at the University of Kansas between 2011 and 2014.[8] It follows the general pattern of the Lava 2000, but introduces several key innovations.

Deep and Shallow

Signals in Kansas Lava are a dual between a deep and shallow embedding of the circuit architecture. The shallow embedding consists of a Stream representing the values flowing over the wire, and a token representing the clock which the signal responds to. The clock token is to provide the potential for multiple clocks. However, several library components assume a single clock, so it is not usable in practice.

The deep embedding contains information pertaining to the compilation to VHDL. These are contained in an extensible datatype in the Kansas Lava library. In most cases, the translation of a signal is an operation of some kind on the right, assigned to a new signal identifier on the left. More complex cases like the clock-edge guarded assignments for feedback might generate multiple lines of VHDL.

Combinators over signals in Kansas Lava must treat both the deep and shallow embedding. For example, a plus binary operation would produce a new stream in the shallow embedding by zipping together each input stream over the plus operation. It would then descibe the binary operation statement by invoking the appropriate member of the deep embedding datatype. The following helper function creates such a dual embedding from a plain Haskell function and an entity name.

```
primXS2 :: forall a b c i . (Rep a, Rep b, Rep c) =>
                            (X a -> X b -> X c) ->
                            String ->
                            Signal i a -> Signal i b ->  Signal i c
primXS2 f nm (Signal a1 ae1) (Signal a2 ae2) =
    Signal (S.zipWith f a1 a2) (entityD2 nm ae1 ae2)
```

There are two major issues with the model, one semantic and one stylistic. The semantic issue is that there is no guarantee of semantic coherence between the shallow and deep embedding. In all versions of Lava, and indeed in hardware description in general, there must exist some assumed semantic equivalence between representation and physical hardware. However, the surface area of this interface should be kept as small as possible to ensure trust in the correctness of modelling and verification. In Kansas Lava this contract is not necessarily maintained, as it is trivial to declare signal types that operate differently in each embedding.

The stylistic issue harks back to the conceptual divide of earlier Lava versions between representation and interpretation. Kansas Lava eschews such a divide, and presents a tight coupling to simulation and compilation by the shallow and deep embedding respectively. This can make interpreting circuits for other purposes such as symbolic verification much less elegant.

Probes

During simulation of larger circuits for debugging, it may be beneficial to inspect intermediate values taken by wires internal to a circuit. In earlier versions of Lava this would require explicitly piping these values to the output edge of the circuit. Kansas Lava introduced probes,[7] a system by which probes could be explicitly attached to wires, and the information surfaced during simulation.

This kind of feature is a useful reference when considering questions of unit testing and integration testing in hardware design. Ideally, reusable subcircuit designs are unit tested and verified for correctness and composed into a larger whole. As the system grows, integration testing is necessary to ensure that system invariants across multiple subcircuits remain intact. In particular we may wish to examine the effects of malformed input or faults in early stages on the later parts of a system.

Sized Types

More recent versions of GHC have included extensions that mimic dependent types in limited ways. Sized types allows for indexing types over natural numbers, and simple arithmetic over these type level values.

Kansas Lava makes liberal use of sized types to represent the width of multi-bit channels. The implementation is quite expressive and adds a level of type safety to signal passing that was not present in prior iterations of Lava. However, the system runs into trouble expressing address widths where it is necessary to take a logarithm of a type-level value.

Stable Names

Stable names is a compiler enhancement to GHC that provides a Haskell interface to some deeper representational objects. With stable names it is possible to get a pointer to an object that is testable for equality and robust to garbage collection and memory optimization. This is a canonicalisation of the observable sharing extension described by Claessen and implemented in earlier Lava versions.

2.2 Π-ware

Π-ware[16] is one of the most recent EDSLs for hardware description, and is of particular interest as it is hosted in the dependently typed language Agda. Π-ware is based on

Coquet, a predecessor built in the theorem proving engine Coq, and the basic structure of the library is quite similar.

2.2.1 Representation

Π-ware describes circuits as transformers. These can be composed in various ways according to the constructors in Listing 1 This comprises a *deep embedding* of the circuit into the host language.

```
data ℂ′ where
  Nil    : ℂ′ zero zero
  Gate   : (g# : Gates#)  → ℂ′ (|in| g#) (|out| g#)
  Plug   : ∀ {i o}        → (f : Fin o → Fin i) → ℂ′ i o

  DelayLoop : ∀ {i o l} (c : ℂ′ (i + l) (o + l)) {p : comb′ c} → ℂ′ i o

  _⟩⟩′_  : ∀ {i m o}      → ℂ′ i m   → ℂ′ m o   → ℂ′ i o
  _|′_   : ∀ {i₁ o₁ i₂ o₂} → ℂ′ i₁ o₁ → ℂ′ i₂ o₂ → ℂ′ (i₁ + i₂) (o₁ + o₂)
  _|+′_  : ∀ {i₁ i₂ o}    → ℂ′ i₁ o  → ℂ′ i₂ o  → ℂ′ (suc (i₁ ⊔ i₂)) o
```

Listing 1: The ℂ′ type in Π-ware

Circuits defined in this system form a tree of composed subcircuits, where the Nil, Gate and Plug constructors form the base cases.

`Nil` is a useful base case for recursively defined generic circuits.

`Gate` instantiates a member of some closed family of primitive circuits. This family generates the category of circuits that can be built. The most obvious of these is a basic boolean logic set of `not`, `and`, `or`, and `xor`. These have a trivial mapping to implementation, but it is possible to define primitives at any level of abstraction.

`Plug` forms a 'rewiring' between circuits. This constructor takes advantage of the totality checker in Agda to enforce correctness rules. Because it is a total function from the output side to the input side, a given output may not be driven by multiple inputs or remain disconnected.

`DelayLoop` explicitly introduces a cycle whereby part of the output of a circuit is looped back to the input. The external interface of such a circuit is the unlooped segments of the input and output.

The remaining constructors, $\rangle\rangle'$, $|'$ and $|+'$ perform serial, parallel and selective composition respectively. Selective composition chooses the signal path based on the first bit of the input, and hence forms a primitive conditional.

The circuit type is largely a structural specification, where the composition of circuit elements maps directly to the data flow in simulation or compilation. However, at a deeper level, the implementation of the primitives defined by the `Gates#` data type is assumed. In the simplest case, these form a functionally complete set such as {`and, or, not`} or {`nand`} but may equally be circuit fragments from a problem domain such as cryptography

or signal processing. Since the library is essentially implementation agnostic over these primitives, their invocation is a behavioural specification mechanism.

2.2.2 Combinatorial vs Sequential

Π-ware demarcates a strong divide between combinatorial and sequential circuits. This is achieved through the `comb'` predicate. This distinction is useful when embedding the semantics of simulation into the host language. Combinatorial circuits may be considered as pure functions, whereas sequential circuits must use some mechanism to maintain state.

A quirk of the implementation of this dichotomy is the restriction that arguments to the `DelayLoop` circuit constructor be combinatorial. In Chapter 4 we will demonstrate that up to isomorphism this restriction does not alter the category of constructible circuits; it appears that this choice was made to simplify the structure of the library regarding verification over sequential circuits. However, it makes the interface somewhat more awkward to use, since it undermines composability and reusability of circuit fragments.

2.2.3 Semantics and Simulation

Π-ware provides a formal semantic description of the circuit combinators as a precursor to defining a simulation mechanism. For combinatorial circuits this leads to an excellent summary demonstrating how all combinatorial circuit reduce to plain functions.

The Π-ware paper does not provide a complete treatment for the semantics of sequential circuits, citing issues with Agda and time constraints. It gives the beginnings of a 'causal step semantics,' which depends on the entire history of the circuit state. If we consider every state as a consequence of some history, we encode reachability in our property proofs. This is a desirable attribute, since it can prevent explosive growth in the complexity of verification. Unfortunately, it's not clear from the paper if this approach is viable in general.

The simulation of sequential circuits in Π-ware proceeds according to the automaton model given by $\mathbb{C}\ i\ o \to i \to (o, \mathbb{C}\ i\ o)$. This pattern is optimised by the above-mentioned coersion of combinatorial circuits to functions.

Chapter 3

Dependent Types in Idris

In this paper we largely concern ourselves with the general purpose, dependently typed language Idris.[4] Inspired by and written in Haskell, Idris uses a Haskell-like syntax with certain extensions.

The main attraction of Idris is that it delivers the immense power of dependent types in a framework designed for general purpose programming. Previous attempts such as Coq and Agda have been plagued by usability issues and mainly remain of interest as theorem proving engines.[2]

In this chapter we will present an abridged introduction to Idris that addresses the language features used frequently in Oz. A more complete treatment is available in the official tutorial and documentation.[11]

3.1 Data Types

Data types in Idris can be specified in both the Haskell idiom and the generalised algebraic datatype (GADT) style. An example of the former is the following representation of natural numbers ubiquitous in Idris:

```
data Nat = Z | S Nat
```

This definition of the natural numbers is useful for representing the kinds of recursive structures we often encounter when working in Idris. Peano arithmetic is not very fast, but the compiler is intelligent enough to reduce `Nat` to a more efficient type at runtime.

An example of the GADT-style type declaration is the following vector implementation from Idris. This type is often used as example to demonstrate the basics of dependent types.

```
data Vect : Nat -> Type -> Type where
  Nil : Vect 0 a
  (::) : a -> Vect k a -> Vect (S k) a
```

We say that this type is *parameterised* by a Type and *indexed* over Nat. A fully realized member of this type might be a `Vect 3 Int`, we could construct a member of that type by `1 :: 2 :: 3 :: Nil`. Idris provides some syntactic sugar for list-like types with the constructors `(::)` and `Nil` so the same list can be declared by `[1, 2, 3]`.

Because types are first-class constructs, type aliasing is simple in Idris:

```
Position : Type
Position = Vect 3 Nat
```

3.2 Functions

Functions in Idris are implemented using pattern matching syntax similar to Haskell. A simple example is a safe *head* operation:

```
head : Vect (S n) a -> a
head (x :: xs) = x
```

In Idris, functions are typically exprected to be total. Partial functions are admissable, but need to be marked as such and will generate a compiler warning. In the head function above, the `Nil` constructor is not matched, but the type signature of this function ensures it can only accept a vector with at least one element. The type checker can determine that only the `(::)` constructor can produce such a vector, so this definition is accepted as total. Note that this obviates the need for bounds checking at runtime, since the type checker statically ensures that all calls to such methods are properly guarded. A further example of this is in the indexing function over a vector.

```
index : Fin n -> Vect n a -> a
index FZ (x :: xs) = x
index (FS k) (x :: xs) = index k xs
```

The `Fin n` type represents a finite set of size `n`. We know at compile time that a member of this set can serve as a valid index into a vector of size `n`. This cleanly and explicitly delegates bounds checking to the source of the index value, which must construct a valid `Fin n`.

3.2.1 Implicits, Auto and Predicates

In the prior examples, we have seen lower case elements in type signatures representing arbitrary inhabitants. These implicit arguments are filled in at compile time through the elaboration and unification process. Implicit arguments are those that are designed to be inferred from the calling context, and unification manages the essential transitivity of this behaviour.

Implicit arguments may have one of two additional keywords, `auto` or `default`. Auto implicits invoke Idris' proof search mechanism, which attempts to satisfy the goal using

some trivial proof tactics. Default implicits have a default value, which is often useful for calculating some intermediate result over an implicit. Consider the following example:

```
data NonEmpty : List a -> Type where
  IsNonEmpty : NonEmpty (x :: xs)

head : (l : List a) -> auto ok : NonEmpty l ->  a
head [] impossible
head (x :: xs) = x
```

NonEmpty is a predicate type – it's purpose is to serve as a proposition over some piece of information. Any instantiated element of this type is a proof of this proposition; in this case `IsNotEmpty l` is a proof that `l` is a non-empty list.

When used in conjunction with auto implicits, we can restrict function arguments to those that satisfy the predicate. The example uses the implicit *ok* to ensure that a head element exists. The impossible keyword is a type-checked assertion of an unreachable pattern, satisfiable by the NonEmpty predicate.

This mechanism allows us to carry around first-order logic at the type level, enabling powerful reasoning about the semantics of code at compile time.

3.2.2 Where Clauses

We have seen the *where* keyword in data type declarations, but it may also be invoked in function bodies to create a local scope for intermediate processing. The following function from the Idris library reverses a list.

```
reverse : List  a -> List  a
reverse = reverse' [] where
  reverse' : List  a -> List  a -> List  a
  reverse' acc []      = acc
  reverse' acc (x::xs) = reverse' (x::acc) xs
```

Idris can optionally infer the type of objects inside a where block if they can be completely determined by their application. However, it is considered good style to make types explicit except in the most trivial cases.

3.3 Type Classes

Type classes in Idris are implemented in much the same way as Haskell. They allow functions to operate over several data types. The `Show` class is a generic interface for providing a textual representation of a data type. In order to use the method `show`, a data type must be declared as an instance of `Show`.

```
class Show a where
  show : a -> String
```

```
instance Show Nat where
  show Z = "z"
  show (S k) = "s" ++ show k

> show (S Z)
"sz" : String
```

Type class requirements are expressed as a prefix to function signatures. This can be read as "under the condition that a is an instance of Show, take a Vect of 'a's and return a String."

```
showHead : Show a => Vect (S n) a -> String
showHead (x :: xs) = show x
```

Type classes can also be extended to provide additional interface that builds on the methods provided in the base class, as in Haskell.

3.4 Records

Records in idris are a way to collect several fields together and provide a consistent interface for them. The syntax differs in this case from Haskell. Recalling our type alias for `Vect 3 Nat` we can define an `Entity` as:

```
record Entity where
  constructor MkE
  name : String
  position : Position
```

This construct generates a data type containing the fields in the order they are declared. Getter and setter functions are generated for each field; setting may also be achieved by the record update syntax:

```
e : Entity
e = MkE "foo" [1, 2, 3]

> record { position = [3, 2, 1] } e
MkE "foo" [3, 2, 1]
```

3.5 Intermediate Values

Idris has several syntactic blocks for managing intermediate values and logic.

3.5.1 Let

Let expressions are for name binding. They can perform simple pattern matches on the left hand side, making them a useful tool for deconstructing intermediate values. Recalling the previous definition of an `Entity`:

```
manhattan : Entity -> Nat
manhattan e = let [x, y, z] = position e in x + y + z
```

3.5.2 Case

A *case* statement has the usual semantics, with some restrictions over the types of the branches. Each branch must match a value of the same type, and its return type must be fully realised within the current scope. This example from the Idris libraries returns all possible tails of a list, longest first:

```
tails : List a -> List (List a)
tails xs = (xs :: case xs of
  [] => []
  (_ :: xs') => tails xs')
```

3.5.3 With

The most technical intermediate value matching mechanism, *with* blocks perform a full nested pattern match in a function body.[15] The following example from the Idris tutorial is particularly illuminating. We wish to construct a binary representation of a natural number. We use a method akin to repeated division by two, where at each stage the next least significant bit is determined by the parity of the number. The implementation of the `parity` function is elided for simplicity.

```
data Parity : Nat -> Type where
   Even : Parity (j + j)
   Odd  : Parity (S (j + j))

parity : (n : Nat) -> Parity n

natToBin : Nat -> List Bool
natToBin Z = Nil
natToBin k with (parity k)
   natToBin (j + j)     | Even = False :: natToBin j
   natToBin (S (j + j)) | Odd  = True  :: natToBin j
```

The `with` block allows us to pattern match on a "view" of the argument k. An important aspect of this construct is that because types may depend on terms, the form of the pattern might change based on the branch of the with block we are executing. Optionally, the pattern may be restated in the new form to allow binding the results to a local name. As demonstrated in the example, the pattern can even be non-injective – usually we cannot match on an application of `+`, but here the whole expression has a canonical interpretation held by the `Parity` data type.

3.6 Conclusion

The examples demonstrate that Idris retains the readability of a modern functional programming language. Idris is an actively developed research language, but the volatility of the core has reduced now to the point that it can be considered a usable general purpose language. In practice, it is more difficult to write Idris code than Haskell, for example, as the type checking process is vastly more complex and satisfying totality and unification constraints puts additional onus on the user. Nevertheless, Idris is the first DT language where the usability issues are minor enough that they are outweighed by the tremendous benefits brought by the type system.

Chapter 4

System Model and Design

When designing a language to describe a regular family of objects like circuits, a robust place to start is with the underlying shape of the space they occupy in a categorical sense. This stems from the underlying principles of engineering; we wish to identify and codify high-level processes that are broadly applicable and repeatable. Category theory is this principle applied to mathematics. In this vein, the system model of Oz takes circuits to be approximately the morphisms of a traced monoidal category.[12] It is not within scope to demonstrate this assumption with any rigour – but we will show that it motivates an elegant and complete treatment of the design problem.

Oz is the union of a stronger formalism with the ability to precisely specify such an abstraction in a dependently typed language. It also extracts desirable features from the various predecessors discussed in Chapter 2 and recontextualises them into this system model. The key innovation is the fusion of Lava-style signal expression with Π-ware-style circuit combinators.

4.1 Design Considerations

4.1.1 Totality and Feedback

As discussed in Chapter 2, digital circuits that go beyond simple combinatorial logic and use feedback are inherently challenging to represent in functional languages. The resolution to this issue in Lava is to implement observable sharing, allowing reification of the infinite recursion up to a finite abstract graph.

This type of solution is not feasible in Idris. Idris does not supply the compiler features required to implement observable sharing, and indeed such a thing would violate the spirit of the language. A goal of a successful Idris program is that every function be total – that is, provably terminating for any input. Infinite structures exist in this paradigm, but operations over them have to be very carefully structured so that the type checker can infer that they always converge to a base case. Even assuming an extension like observable sharing, it would not be possible to satisfy the type checker that the multiply infinite graph representation of a complex circuit could necessarily be reduced to a tree.

Fundamental to the success of this project was finding a suitable alternative way to represent cycles that does not violate the underpinning aims of the language regarding totality and verifiability.

4.1.2 Correctness by Construction

The Curry-Howard Correspondence presents an equivalence of types with propositions and programs as proofs.[18] Taking functions as an example, we can view the type signature of a function as a lemma claiming the existence of a mapping between the inputs and the output. The program satisfying this type – this, the implementation of the function – is a proof of such a mapping by construction.

In the presence of dependent types, it is possible to more completely specify the semantics of a function. When we describe at the type level how the nature of the output depends on the inputs, we can make the meaning of the program more explicit. Then by the above correspondence, a type correct implementation of that function is provably correct at compile time. The more precise we make our types, the more powerful this concept becomes.

4.1.3 Expressivity

A key concern in the development of Oz was the expressivity of the language. Expressivity is a mixture of precision and succinctness. It is sometimes characterised by how direct the map is between the concepts envisioned by the user and the text of the code.

A key aspect of expressivity is the inclusion of an intuitive generic specification mechanism. The ability to specify complex generic circuits has been a major driving force for the research into modernising HDLs. Dependently typed language aid in this by allowing complex parameters for generic circuits to be carried at the type level. We wish to leverage these features to describe whole families of circuits in a simple, extensible way.

History has shown that improvements in expressivity largely arise through iteration, learning from and avoiding the pitfalls exposed in prior art. The detailed investigation of previous embedded HDLs in this report is an effort to incorporate this into Oz.

4.1.4 Separation of Concerns

One of the ideals of Lava 98 was that the abstract representation of a circuit would not be cluttered by artifacts of the intended interpretation. This addresses a serious flaw of VHDL and Verilog, which have intermixed elements of testing, simulation and compilation to the detriment of usability.

Oz aims to adhere to this original ideal and separate the concepts of representation and interpretation. The basic interpretations considered for this iteration are simulation, compilation (for example, to VHDL) and verification.

In particular we can serve this goal by choosing a deep embedding. As a shallow embedding, Lava uses the abstract syntax tree (AST) of plain Haskell to carry the model

circuit. This restricts how the circuit may be annotated with interpretation concerns. Using the rich algebraic data types of Idris, we can more easily build interpretation and traversal mechanisms.

4.1.5 Synchronous Circuits

At present, Oz is a language for describing synchronous digital circuits. In particular, it is assumed that there is a single universal clock. This does not allow for a completely general treatment of hardware design vis a vis the effects of propagation delay and necessity of timed subcircuits.

The design of Oz should allow for later extension with regard to timing, at least to the effect of admitting multiple clock signals. In Chapter 7 we address the various extensions envisaged for future work.

4.1.6 Output Target

For the initial proof of concept, Oz will not deliver a compilation to a target language. However, following the example of Lava and Π-ware, the representation will aim to be semantically compatible with VHDL.

This means that the primitive data types and operations provided in the representation should have a trivial mapping to counterparts in VHDL. Nevertheless, because hardware design has reasonably well-established core semantics, it is intended that Oz remain backend-agnostic in later iterations.

4.2 Outcomes

The design for Oz draws from prior efforts in the area of embedded HDLs, the powerful specification mechanisms in Idris, and the considerations outlined above. The design is summarised by the following:

- A Circuit is a data type representing a transformation of ordered data.
- A Signal is a typed wire with a value that can be completely determined from some context.
- A Pipeline collects together Signals with a common input context, forming a fundamental Circuit.
- A Bus describes an arrangement of wires for an input or output to a Circuit or Pipeline.

Each of Signal and Circuit may be recursively defined, forming a nested structure. In order to prevent illegal cyclic constructions, the totality checker is leveraged to ensure that in neither case are circular references permitted.

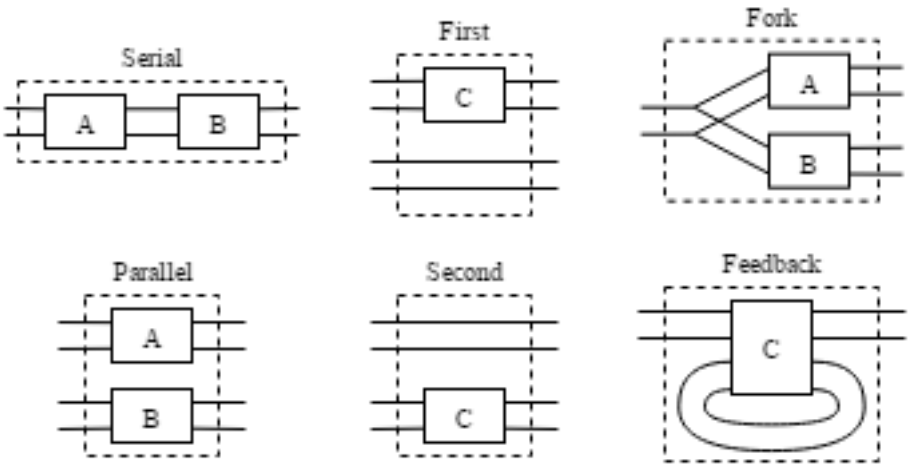

Figure 4.1: Circuit Constructors

4.2.1 Circuits

Like Π-ware, Oz has a deeply embedded behavioural specification type for circuits. It is here that we represent the categorial interpretation mentioned in the introduction to this chapter.

As a rather informal explanation, consider a category in which the objects are ordered clusters of data. Circuits are the morphisms between these clusters of data. Composition is provided by the Serial combinator, and we have an additional bifunctor given by the Parallel combinator. We do not demonstrate here that this definition has the requisite coherence conditions, but we have essentially a monoidal category. The feedback constructor imparts the trace mechanism of a traced monoidal category. In Oz we write a *syntax* for this category, the basic elements of which are shown in Figures 4.1 and 4.2.

In terms of design impacts, the most critical element is the feedback constructor. By making the mechanism of feedback explicit at this abstract level, we remove the need for cyclic references. In Chapter 7 we will discuss how this structure can be augmented with an algebra of rewrite rules for use in various verifaction or compilation processes.

4.2.2 Signals

The above design defines a set of rules for a top-level language, but does not address the Comb base case shown in Figure 4.2. The base case is a concrete semantic definition of a circuit from inputs to outputs. In Π-ware this is served by a selection of primitives – typically basic logical and arithmetic operations. However, this has serious ramifications for the expressiveness of the language.

In the design of Oz, we opt instead for an intermediate language resembling Lava, with a Signal type at the core. The obvious difference is that these circuits are purely combinatorial, since feedback is specified at the Circuit level. We can consider then the Signal type as a deeply embedded, simplified register transfer level (RTL) language.

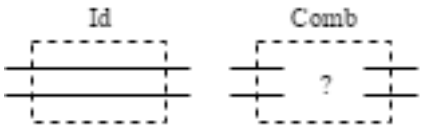

Figure 4.2: Circuit – Base Cases

Each time we define a Signal, it translates to a register assignment where the internals of the Signal represent the expression on the right hand side. A collection of signals that use the same set of inputs can be wrapped together to form a `Pipeline`, which takes it's name from the RTL parlance. We lift elements from the Signal level to the Circuit level by wrapping a `Pipeline` up as a Circuit.

4.2.3 Interfaces

The final part of the design specifies how interfaces to circuits may be declared. A key issue in Lava was the lack of a mechanism to specify the interfaces, and in many cases a `List` would be used, which sorely lacks type safety and expressiveness.

In our experimentation with Kansas Lava we found an idiom using Haskell's records to be useful when grouping signals into a manageable interface. In Oz, we raise this idiom into the language itself, defining a Bus type that describes a collection of typed wires.

Here dependent types play a major role in improving the expressiveness of Oz. We can use the Bus – an ordered collection of ports – as a parameter to our other types. By way of example, the circuit type may be parameterised by two Buses for the input and output edges. In this way we can deliver a strongly type safe serial composition combinator by requiring that the Bus elements carried in the type match exactly.

4.3 Requirements

The coarse-grained requirements elicited early in the project were stratified according to the MoSCoW model as follows.

Must Have	Representation	A deeply embedded representation of circuits using the precise dependent typing available in Idris.
Should Have	Simulation	The ability to realise circuits from their abstract representation and simulate over some set of inputs.
	Typed RTL	The representation should operate over a type universe matching the formally defined semantics of std_logic in VHDL.
Could Have	Array types	std_logic_array with relevant slicing, indexing and concatenation operations.
	Multiplexing	A switch-like construct for generalised conditionals.
Won't Have	Compilation	The representation must be built with a view to compilation to VHDL, but a concrete implementation will not be provided.
	User Tools	Following the example of Π-ware, the library should provide an API over which high-level tools may be built. The creation and inclusion of such tools is not in the scope of this project.

Chapter 5

Implementation

In this chapter we examine the implementation of Oz. We do not address the entirety of the source, but rather offer commentary on the key aspects relevant to this report. Appendix A provides a link to the full source and some additional context.

For this iteration we chose to stay close in many places to the semantics of a synthesisable subset of VHDL. Unfortunately this has in some places led to implementation quirks that stray from the design ethos of the project. We would like for the Signal language to have a more robust underlying design, but that remains work for a future iteration. Nevertheless, this proof of concept does demonstrate the essential feasibility of the general design outlined in Chapter 4.

As a hardware description language, the primary purpose of Oz is to provide a way to represent circuits. The interpretations of that representation should be separated from the core of the language where possible. The key interpretations considered during this project included:

- Verification - Proving circuit properties
- Simulation - Testing circuit behaviour
- Compilation - Outputting equivalent VHDL

Providing a viable interface for these interpretations has shaped the design of the library, but ultimately it was not feasible to attempt an implementation of all three. As a result, efforts were focused on delivering a simulation tool, and we present here a brief inspection of the implementation.

5.1 Wire Types

A key implementation decision was how to represent the data that makes up the signals in Oz. In Kansas Lava, for example, any type may be used as long as it has an instance of `Rep`, a type class encoding how to represent a type in raw bits. This can be useful in practice when using enumerated data types to represent instruction codes or the like. However, it does add a level of indirection when trying to reason about the data flowing through circuitry.

For this reason, we chose to use an explicit family of wire types corresponding to those available in VHDL's `std_logic` library. To represent these, we use a common pattern in dependently typed languages – the universe of types. As a consequence of the type theory,[14] it is not possible to pattern match over types directly. However, we often need to build a suite of tools polymorphic over some closed family of types. To achieve this, a data type describing this 'universe' is created, as demonstrated in Listing 2.

```
||| Types corresponding to those in the std_logic VHDL library
data SLType : Type where
  Bit : SLType
  Vector : Nat -> SLType
  Unsigned : Nat -> SLType
  Signed : Nat -> SLType
  Array : Nat -> (c : SLType) -> {auto tc : Logic c} -> SLType

||| A map to interpret SLTypes as Idris types
iSL : SLType -> Type
iSL Bit = Bool
iSL (Vector n) = Bits n
iSL (Unsigned n) = UInt n
iSL (Signed n) = SInt n
iSL (Array n t) = Vect (power 2 n) (iSL t)

||| A map providing the bit-width of an SLType
wSL : SLType -> Nat
wSL Bit = 1
wSL (Vector n) = n
wSL (Unsigned n) = n
wSL (Signed n) = n
wSL (Array n t) = (power 2 n) * (wSL t)
```

Listing 2: The SLType Universe

The mapping function `iSL : SLType -> Type` provides a way to multiplex over this family at runtime. We can invoke this function wherever we need to resolve the `SLType` to a native Idris type. We will see a good example of this is in the `Load` data type in the following section.

The mapping function `wSL : SLType -> Nat` details the bit width of each wire type. Several of the wire types carry their bit width as a parameter, but this function provides a uniform interface.

Idris' powerful type checker is leveraged to control the recursion in the type universe, ensuring that only 1-dimensional types can be carried in an array. The `Array` constructor to `SLType` builds a type corresponding to the `std_logic_array` type in VHDL. It accepts an implicit argument *tc* with type `Logic`.

```
data Logic : SLType -> Type where
  LogicBit : Logic Bit
  LogicVector : Logic (Vector r)
  LogicUnsigned : Logic (Unsigned r)
  LogicSigned : Logic (Signed r)
```

Listing 3: Logic Predicate

Shown in Listing 3, `Logic` is a predicate over `SLType`. The keyword *auto* invokes the proof seach mechanism in Idris' elaborator. When it encounters a predicate data type, it will try each constructor, attempting to build an appropriately typed argument. Consider the following declaration, a 32 element array of integers:

```
MemType : SLType
MemType = Array 5 (Unsigned 8)
```

It is trivial for the proof search mechanism to construct `LogicUnsigned` and fill *tc*. But `Logic` only provides proofs only for 1-dimensional `SLType`s. It is impossible to build an element of type `Logic (Array n t)` and hence the following declaration will statically fail:

```
MemType' : SLType
MemType' = Array 4 (Array 4 Bit)
```

5.2 Circuit Interfaces

Having defined the set of types representable in our circuit models, we require a way to describe the interfaces to those circuits. In Oz these interfaces are defined by the Bus data type given in Listing 4.

```
data Bus : Nat -> Nat -> Type where
  Nil : Bus 0 0
  (::) : (t : SLType) -> Bus n w -> Bus (S n) (wSL t + w)
  (||) : Bus l lw -> Bus r rw -> Bus (l + r) (lw + rw)
```

Listing 4: The Bus data type

This type is similar to a vector. It is parameterised by two natural numbers which contain the total number of wires and the total bit width respectively. Because it uses the *nil* and *cons* constructors we can use Idris list syntax to create elements of this type. The extra constructor `(||)` arises from the need to combine circuits in parallel In this situation, it is necessary to perform a join operation on their inputs and outputs. Typically we would use a concatenate function such as:

```
(++) : Bus l lw -> Bus r rw -> Bus (l + r) (lw + rw)
```

However, when the circuits are traversed for simulation or compilation, it is necessary to take the inverse of the concatenation. The `(++)` operation is not injective; it does not preserve enough information to permit inversion by pattern matching. The join constructor `(||)` preserves more information in the type, allowing the circuit combinators to split the Buses back into the original components.

Given in Listing 5, the Load data type carries data that matches the Bus interface. This type is used for simulation and defining static values in circuits. Load and Bus have a pattern similar to the heterogeneous vector type found in many functional programming languages. In the dependent type theory parlance, Load is an *ornamentation* of Bus.

```
data Load : Bus p w -> Type where
  Nil : Load []
  (::) : iSL t -> Load ts -> Load (t::ts)
  (||) : Load ts -> Load bs -> Load (ts || bs)
```

Listing 5: The Load data type

5.3 Circuits

With the plumbing defined for our implementation, we can move on to the implementation of the fundamental units of our HDL. The Circuit type represents an abstract circuit segment, mapping an ordered set of inputs to an ordered set of outputs. In Listing 6 we see the definition of Circuit in Oz. This data type is parameterised by two Buses, providing a type manifest for the input and output edges.

```
data Circuit : Bus i iw -> Bus o ow -> Type where
  Id : Circuit x x
  Comb : Pipeline i o -> Circuit i o

  Ser : Circuit a b -> Circuit b c -> Circuit a c
  Par : Circuit a b -> Circuit c d -> Circuit (a || c) (b || d)

  Fork   : Circuit a b -> Circuit a c -> Circuit a (b || c)
  First  : Circuit a b -> Circuit (a || c) (b || c)
  Second : Circuit a b -> Circuit (c || a) (c || b)

  Pack   : {b : Bus o ow} -> Circuit a b -> Circuit a [Vector ow]
  Unpack : {a : Bus i iw} -> Circuit a b -> Circuit [Vector iw] b

  Feedback : Circuit (a || b) c -> Load b -> Pipeline c b -> Circuit a c
```

Listing 6: Circuit Implementation

In this data type we see represented the six key combinators outlined in Chapter 4 as well as the base case constructors `Comb` and `Id`. It is important to note how much of the mechanics of these combinators is managed by the `Bus` parameters. These are matched as implicit arguments to the constructors providing enough information for the unifier to ensure type safe circuit combinations. Figure 5.1 reiterates these combinators, annotated by the implicit `Bus` parameters.

The additional `Pack` and `Unpack` constructors are here mainly as convenience methods. The provide a way to flatten or unflatten any representable types to an appropriate width bit vector. This is often useful when coercing data into a new channel format.

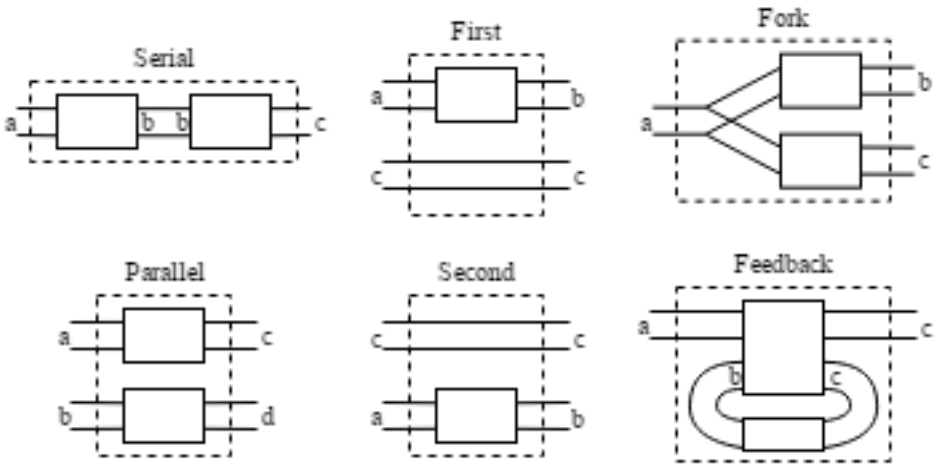

Figure 5.1: Circuit Constructors

Of particular note here is the `Feedback` constructor. Its arguments differ slightly from the abstraction given in the previous chapter. To match the design exactly we could have given the following definition:

```
Feedback' : Circuit (a || c) (b || c) -> Load c -> Circuit a b
```

However, the two versions of this combinator are strictly isomorphic. The expression `Feedback cir def loop` is the same as `Feedback' (Ser cir (Fork Id loop)) def`. In practice the implemented version is much more useful, since we often wish to have access to all the values the circuit generates, feeding back some subset of them.

```
data Signal : Bus p w -> SLType -> Type where
  ||| Base cases
  Literal : iSL t -> Signal i t
  Pin : (x : Fin i) -> Signal v (index x v)

  ||| Operations
  UnOp : Uop a b -> Signal i a -> Signal i b
  BinOp : Bop a b c -> Signal i a -> Signal i b -> Signal i c
  Mux : List (Signal i Bit, Signal i t) -> Signal i t -> Signal i t

  ||| Type operations
  Cast : (Logic (iSL a), Logic (iSL b)) =>
         {auto ok : wSL a = wSL b} ->
         Signal i a ->
         Signal i b
  SExtend : Signal i (Signed n) -> Signal i (Signed (n + m))
  Coerce : (Logic (iSL a), Logic (iSL b)) => Signal i a -> Signal i b

  ||| Indexing
  Concat : Signal i a -> Signal i b -> Signal i (Vector (wSL a) + (wSL b))
  Slice  : Logic (iSL a) => Signal i a -> Integer -> Signal i b

  ||| Arrays - Read (address, mem), Write ([enable, address, data], mem)
  Read  : {auto tc : Logic t} ->
          Signal i (Unsigned n) ->
          Signal i (Array n t) ->
          Signal i t
  Write : {auto tc : Logic t} ->
          Pipeline i [Bit, Unsigned n, t] ->
          Signal i (Array n t) ->
          Signal i (Array n t)
```

Listing 7: Signal Implementation

5.4 Signals

The Signal type is the mechanism by which we declare combinatorial subsections of circuitry. We present here a detailed breakdown of the source given in Listing 7.

Each signal constructor can be notionally translated to a non-blocking assignment statement in a typical RTL language. We don't need to remain anchored exactly to the syntax or expression available in VHDL or Verilog, but keeping these in mind ensures that we cover essential use cases.

The Bus parameter to the Signal type represents the context in which it arises – i.e. it is a typed manifest of signals that are in scope. In a circuit, wire values must arise from input pins or memory, or be statically defined. These are given by the `Pin`, `Read` and `Literal` constructors respectively. Consider the following expansion:

```
res : Signal i (Unsigned 8)
res = BinOp (Arith Plus) (Literal 1) (Pin 0)

sig_0 <= unsigned("00000001") --Literal 1
sig_1 <= ctxt_0               --Pin 0
res <= sig_0 + sig_1          --BinOp (Arith Plus) a b
```

For clarity, each constructor is split into its own assignment. A compiler would trivially improve this by inlining the two arguments to the addition. In the case of `Pin`, the compiler indexes into the `Bus` parameter and inserts the appropriate name. In Chapter 7 we discuss some potential enhancements to this interface, as the numeric references are not as readable as we might wish.

The `UnOp` and `BinOp` constructors take an operation argument that serves as a type predicate, ensuring type safe operation. For example, the operation `(Arith Plus)` can only be constructed when the signals involved are numeric types of equal bit-width. See Appendix A for the implementation of these operation predicates.

`Mux` provides a generalised conditional operator. It accepts a list of pairs of the form (predicate, result) and a default. It tests each predicate in turn until it finds a true value, and returns the associated result. If no predicate is true, it returns the default.

`Cast`, `SExtend` and `Coerce` provide ways to alter the type of values. `Cast` requires that the signals be the same bit width, wheras `Coerce` will either zero extend or truncate the input to from the output. `SExtend` provides sign extension, and hence has much tighter input type requirements.

`Concat` and `Slice` provide ways to join and index into signals. At present these operations are not as type safe as we would typically desire. Ideally, slice should use Idris' `Fin` type to specify a slice that is guaranteed to fall inside the width of the input signal. However, during development, `Fin` was undergoing a revamp in Idris itself, so the current implementation uses it sparingly. As a result, `Slice` takes an integer offset and uses unification to infer the width of the output, either of which might result in slices containing undefined bits.

Finally, we have the array interaction with `Read` and `Write`. `Read` indexes into an array signal as one might expect. `Write` accepts a `Pipeline` containing the information for the write, as well as the array to write to. `Write` is unusual in that it does not create a new signal, but alters the state of an existing one. This rather ungainly way of dealing with memory elements is borrowed from Kansas Lava, and does not fit nicely with the rest of the model. It is the main motivation for finding a better generalisation for the Signal language, as discussed in Chapter 7.

5.4.1 Pipelines

In order to lift our signal representation up to form circuit objects, we need a way to collect together a 'bundle' of signals that have a shared input context. The Pipeline type provides this structure, and organises the `Signal`s into compatibility with the Bus specification.

```
data Pipeline : Bus i iw -> Bus o ow -> Type where
  Nil : Pipeline i []
  (::) : Signal i t -> Pipeline i o -> Pipeline i (t :: o)
  (||) : Pipeline i l -> Pipeline i r -> Pipeline i (l || r)
```

Listing 8: Pipeline Implementation

Note that like `Bus` and `Load`, the `Pipeline` data type implements the *nil* and *cons* interface. This enables the use of Idris' list syntax when creating pipelines. As a simple example, consider the following rewiring pipeline:

```
swap : Pipeline [Bit, Bit] [Bit, Bit]
swap = [Pin 1, Pin 0]
```

5.5 Simulation

Simulation is an obvious and critically important use case for circuit representations. The full implementation including utility and library functions is available in Appendix A.

5.5.1 Signals and Pipelines

When simulating the output of a Signal or Pipeline, we must provide the expected context. The `Load` argument is parameterised by a matching `Bus` ensuring that it contains appropriately typed and ordered data. Listing 9 demonstrates some of the ways this input context is resolved into outputs. In the leaf cases, `Literal` simply ignores the input context and returns the contained value, and `Pin` indexes into the provided data `Load`. The remaining signals implement their functionality over recursive calls to `runSignal`, eventually resolving to one of the leaf cases. Because the Signal and Pipeline structures are required to be total, they *must* form a finite tree which can be resolved to an output.

```
runSignal : Signal i t -> Load i -> iSL t
runSignal (Literal x)    d = x
runSignal (Pin p)        d = index p d
runSignal (UnOp op a)    d = unOp op (runSignal a d)
runSignal (BinOp op a b) d = binOp op (runSignal a d) (runSignal b d)
runSignal (Mux [] def)   d = runSignal def d
runSignal (Mux ((p, c) :: xs) f) d = assert_total (if (runSignal p d)
  then (runSignal c d)
  else (runSignal (Mux xs f) d))
...

runPipeline : Pipeline i o -> Load i -> Load o
runPipeline [] d = []
runPipeline (s :: sb) d = ((runSignal s d) :: (runPipeline sb d))
runPipeline (l || r) d = ((runPipeline l d) || (runPipeline r d))
```

Listing 9: Signal and Pipeline Simulation

5.5.2 Circuits

Similarly to signals, we provide circuits with an input set by a `Load` fulfilling the `Bus` interface. However, circuits can potentially maintain some state if they are sequential. Emulating the approach used in Π-ware, we can simulate clock cycles by the automaton model. Considering the nested structure of the Circuit type as a tree, we perform a depth-first traversal. At each node we elicit the output and replace the node with an updated version of itself. Similarly to the Signal and Pipeline types, we rely on the totality conditions in Idris to ensure that traversing Circuits for simulation will terminate.

Listing 10 shows an abbreviated version of the circuit simulation code. In the case for `Comb` we simply provide the output of the wrapped `Pipeline` and a plain copy of the circuit since it has no state. The combinators `Ser` and `Par` are given as examples of how simulation is delegated to subcircuits. In case those circuits have state, we use them to build a replacement at the current node. The `Feedback` constructor demonstrates where state is actually updated. The new `Load` argument `fb'` captures the result of the feedback loop.

```
runCircuit : Circuit i o -> Load i -> (Circuit i o, Load o)
runCircuit c@(Comb b) inp = (c, runPipeline b inp)
runCircuit (Ser l r) inp =
  let (l', m)   = runCircuit l inp
      (r', out) = runCircuit r m
  in (Ser l' r', out)
runCircuit (Par t b) (inpT || inpB) =
  let (t', outT) = runCircuit t inpT
      (b', outB) = runCircuit b inpB
  in (Par t' b', (outT || outB))
...
runCircuit (Feedback c fb loop) inp =
  let (c', out) = runCircuit c (inp || fb)
      fb' = runPipeline loop out
  in (Feedback c' fb' loop, out)
```

Listing 10: Circuit Simulation

Chapter 6

Results and Discussion

Having discussed the design and implementation of Oz, we turn to its practical use and how it compares to its contemporaries. Oz is still a technical proof of concept with a number of rough edges to be smoothed. In this chapter we demonstrate some of the sucesses of the design and some areas that require further attention.

6.1 Basics

Let us examine a pair of basic circuit examples, outlined in Figure 6.1. These will demonstrate the basics of representing combinatorial and sequential logic in Oz.

The half adder is a basic building block for binary arithemtic. It has two input wires and two output wires, each carrying a single bit of information. This interface is captured in the type of the `halfAdder` implemetation in Listing 11. The internals are given by a `Pipeline` that produces the carry and sum signals by *and* and *xor* gates respectively. The `Pipeline` is raised to a `Circuit` by the `Comb` constructor.

```
halfAdder : Circuit [Bit, Bit] [Bit, Bit]
halfAdder = Comb [Pin 0 `and` Pin 1, Pin 0 `xor` Pin 1]

*Examples/Basics> snd $ runCircuit halfAdder [True, True]
[True, False] : Load [Bit, Bit]
```

Listing 11: Half Adder in Oz

Below the implementation we see the output of simulatng the half adder with some test data. Recall that `runCircuit` has type

```
runCircuit : Circuit i o -> Load i -> (Circuit i o, Load o)
```

We use `snd` to extract the second part of the pair which is the circuit output.

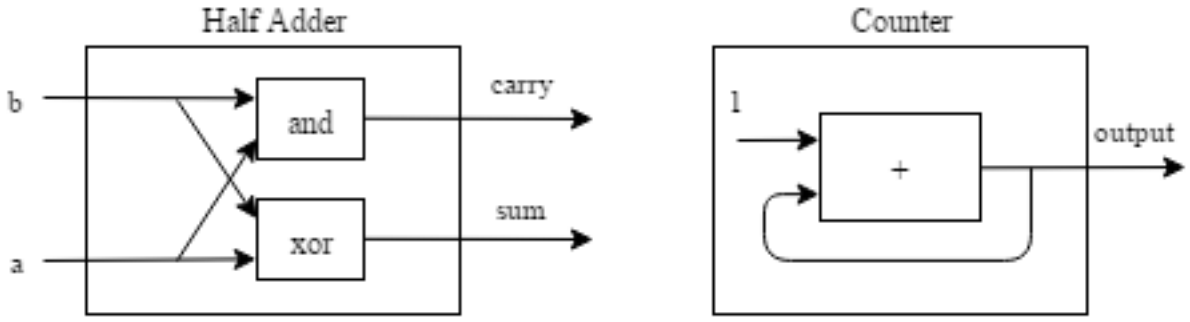

Figure 6.1: Basic Circuits

Listing 12 implements the counter – a signal generator which takes no inputs but produces output at each clock edge. The `Feedback` constructor takes an internal `Circuit`, a `Load` containing default state, and a `Pipeline` that can produce new state from the output. The type of the internal circuit can be inferred by unification over the types of the other two arguments. In this case it has type `Circuit ([] || [Unsigned 8]) [Unsigned 8]`.

```
counter : Circuit [] [Unsigned 8]
counter = Feedback (Comb [Pin 0 + Literal 1]) [0] [Pin 0]

*Examples/Basics> streamCircuit counter (replicate 10 [])
[[UI (MkBits 1)],
 [UI (MkBits 2)],
 [UI (MkBits 3)],
 [UI (MkBits 4)],
 [UI (MkBits 5)],
 [UI (MkBits 6)],
 [UI (MkBits 7)],
 [UI (MkBits 8)],
 [UI (MkBits 9)],
 [UI (MkBits 10)]] : List (Load [Unsigned 8])
```

Listing 12: Counter in Oz

Using a helper method defined in the Oz library, we can simulate the circuit over a series of cycles by passing a list of inputs. In this case the circuit accepts no input so we simply pass a `Nil` load. We see here some of the Idris internals. `UI` is a constructor for the deeply embedded unsigned integer type. It wraps an element of the native Idris `Bits` type, providing numeric operations. This output could stand to be more readable.

6.2 Stack

To motivate a more in depth analysis of some of the features of Oz, this section provides an example circuit mocked up in VHDL, Kansas Lava and Oz. Kansas Lava was chosen because it is modern, mature and has many of the same goals as Oz including expressivity and verifiability. Listing 13 shows the model circuit, a 32 element stack, implemented in VHDL. This is a simple design without any bounds checking – the stack pointer simply wraps at the boundary values.

Listing 14 is the Kansas Lava implementation of this stack, and Listing 15 is the Oz implementation. Figure 6.2 is a schematic diagram of the Oz implementation showing the encapsulation of the various subcircuits.

6.2.1 Circuit Interface

It is interesting to examine how the circuit interfaces are declared in each implementation. The stack has the following inputs:

- A two bit value that is interpreted as a signed integer to increment or decrement the stack pointer.
- A write enable bit.
- A 16 bit word of data.

and provides outputs:

- The updated stack pointer, to allow external bounds checking.
- A word read from the stack.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity stack is
port(
  clk    : in std_logic;                          -- Clock
  dsp    : in std_logic_vector(1 downto 0);       -- Stack pointer delta
  wEn    : in std_logic;                          -- Write enable
  dataIn : in std_logic_vector(15 downto 0);      -- Data to be pushed

  dataOut : out std_logic_vector(15 downto 0); -- Data popped
  spO     : out std_logic_vector(1 downto 0);  -- New stack pointer
);
end stack;

architecture bh of stack is

type mem_type is array (31 downto 0) of std_logic_vector(15 downto 0);
signal stack_mem : mem_type := (others => (others => '0'));
signal sp  : std_logic_vector(5 downto 0) := unsigned(31);
signal spN : std_logic_vector(5 downto 0);

begin

PROC : process(clk)
begin
    if (rising_edge(clk)) then
        spN <= signed(sp) + signed(dsp)

        if (wEn = '1') then
            stack_mem(spN) <= dataIn;
        end if;

        dataOut <= stack_mem(sp);
        sp <= spN;
        spO <= sp;
    end if;
end process;

end bh;
```

Listing 13: Stack in VHDL

In each implementation simple type aliases are used to refer to the word and stack pointer wire types. In Kansas Lava, we have used a design pattern where the inputs and outputs are enclosed in records. This allows us to declare a function from the input to the output that serves a similar function to `Pipeline` in Oz. However, this pattern is not obvious to a beginning user and not at all documented.

In Oz, the design enshrines this pattern in the language. The Bus parameters to each circuit codify exactly the type and ordering of the input and output. In the implementation given, all of the Buses are explicit which does hamper the expressivity of the code. There are two ways to alleviate this: the Bus objects may be aliased similarly to the Kansas Lava example, or the types of the intermediate circuits can be inferred by unification. Idris does not have top-level type inference, but will infer types inside a where block if they can be fully determined by their first application.

At present the unification and elaboration processes in Idris are still being tweaked, affecting both aliasing and inference in where blocks. Some difficulties were encountered when trying to simplify this code, and it was considered best to leave it in its present explicit and functional state. In future iterations of Oz, this will be a focus of usability improvement efforts.

```
data StackI = StackI {
    stkdSPI  :: Seq S2,
    stkDataI :: Seq (Enabled U16) }

data StackO = StackO {
    stkDataO :: Word,
    stkSPO   :: SP }

stack :: StackI -> StackO
stack inp = out where
    StackI {..} = inp
    out = StackO {..}

    spN = stkSPO + signed stkdSPI
    mem = writeMemory $ pack (en, pack (spN, val)) where
        (en, val) = unpack stkDataI
    stkDataO = asyncRead mem stkSPO
    stkSPO = register 0 spN
```

Listing 14: Stack implemented in Kansas Lava.

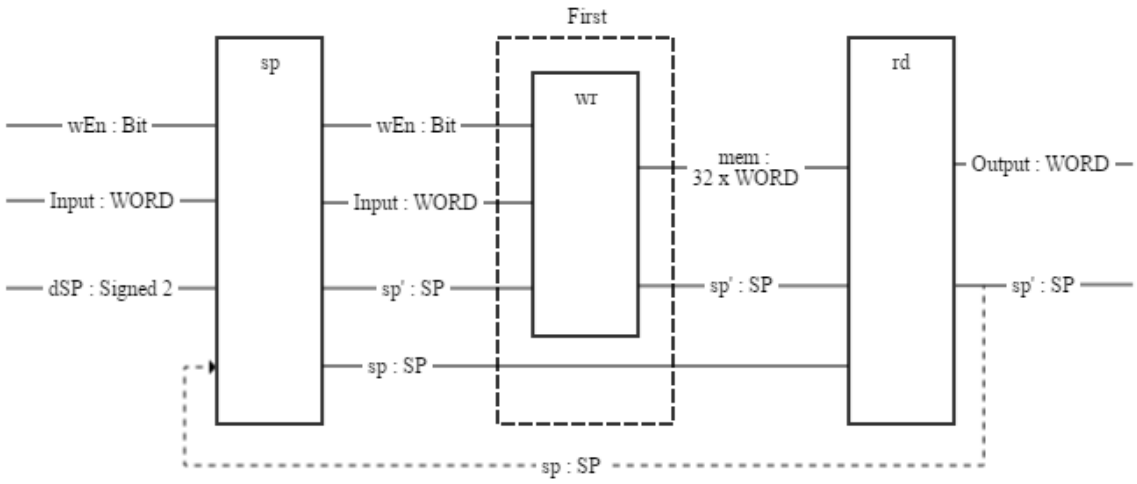

Figure 6.2: Stack Circuitry

```
||| The composed stack element
|||  In : wEn, input, dsp
||| Out : output, sp'
stack : Circuit [Bit, WORD, Signed 2] [WORD, SP]
stack = Feedback ((sp &> (First wr)) &> rd) [31] [Pin 1] where
  ||| Update the stack pointer
  sp : Circuit ([Bit, WORD, Signed 2] || [SP]) ([Bit, WORD, SP] || [SP])
  sp = Comb ([Pin 0, Pin 1, ((Pin 3) + (Cast (SExtend (Pin 2))))] || [Pin 3])

  ||| Write to array
  wr : Circuit [Bit, WORD, SP] [Array 5 WORD, SP]
  wr = Feedback
    (Comb [Write [Pin 0, Pin 2, Pin 1] (Pin 3), Pin 2])
    [replicate 32 0]
    [Pin 0]

  ||| Read from array
  rd : Circuit ([Array 5 WORD, SP] || [SP]) [WORD, SP]
  rd = Comb [Read (Pin 2) (Pin 0), Pin 1]
```

Listing 15: Stack implemented in Oz

6.2.2 Simulation

Simulation of each EDSL implementation is an interesting lens through which to view the languages. In Kansas Lava we simulate over Signal objects that carry internally a stream of values. But Signals also contain a deeply embedded representation of their VHDL counterpart to aid in compilation. In order to mock up signals for testing we need to use library functions that provide dummy deep embeddings.

```
testStack :: (Word, SP)
testStack = (takeS 7 stkData0, takeS 7 stkSP0) where
  Stack0 {..} = stack StackI {
    stkdSPI  = toS [     1,      1,        1,       0,      -1,      -1,      -1],
    stkDataI = toS [Just 4, Just 8, Just 175, Nothing, Nothing, Nothing, Nothing]
  }
```

```
*J1.Stack> testStack
( ? | 17 | 8 | 175 | 175 | 8 | 17 | ? .,
  0 |  1 | 2 |   2 |   1 | 0 | 31 | ? .)
```

In Oz, we have a clear separation between simulation and representation. In theory, we have a clearer, more expressive simulation API. Unfortunately Idris can't presently parse nested list syntax when it desugars to different types. Declaring a `List` of `Loads` for input is therefore a little bit cumbersome.

```
demo : List $ Load [WORD, SP]
demo = streamCircuit stack (
  [ True,  17,  1] ::          -- Push  17
  [ True,   8,  1] ::          -- Push   8
  [ True, 175,  1] ::          -- Push 175
  [False,   0,  0] ::          -- Peek 175
  [False,   0, -1] ::          -- Pop  175
  [False,   0, -1] ::          -- Pop    8
  [False,   0, -1] ::          -- Pop   17
   Nil
)
```

```
*Examples/Stack> demo
[[UI (MkBits   0), UI (MkBits  0)],
 [UI (MkBits  17), UI (MkBits  1)],
 [UI (MkBits   8), UI (MkBits  2)],
 [UI (MkBits 175), UI (MkBits  2)],
 [UI (MkBits 175), UI (MkBits  1)],
 [UI (MkBits   8), UI (MkBits  0)],
 [UI (MkBits  17), UI (MkBits 31)]] : List (Load [Unsigned 8, Unsigned 5])
```

Chapter 7

Future Work

This project has generated an interesting proof of concept with a great deal of promise. We have discovered substantial benefits to the use of dependent types in expressing a formal language for circuits. In this chapter we discuss some of the opportunities for further research.

7.1 Signal Formalisation

As outlined in Chapter 5, the present incarnation of Oz uses a Signal language that hews close to VHDL. This was simply an expedient decision to ensure that the language provided coverage of the typical design use cases. However, in the same spirit as the formalisation of the higher-level Circuit language, we would like to deliver a better mathematical model at the register transfer level of Signals.

As an example, we could consider the conversion constructors `Cast`, `Coerce`, `SExtend`. Clearly, sign extension is a special case at a level of abstraction vastly different to it's neighbouring cases. This is usually a hint that the abstraction has not sufficiently captured the rules of the domain or its use cases.

Similarly, `Read` and `Write` have an unintuitive interface that suggests they need to be revisted. The management of multi-dimensional values in a more general fashion would vastly increase the robustness of the Signal language.

7.2 Verification

Hardware verification is a central concern in the ongoing development of Oz and hardware design EDSLs more generally.[13] More work needs to be done in the representation and abstraction in Oz to cement link between the semantics of the hardware and the static verification in the type system.

Verification by first order logic would provide a massive advantage, rendering huge classes of verification problems tractable. In particular, classical methods relying on propositional logic and SAT-solvers have enormous trouble verifying generics. First order

logic admits predicates and quantification, which can encode the mechanics of generic constructs. This means that whole families of circuits can be verified in a single proof.

A large part of this effort involves implementing the formal structure of circuits. As a concrete example, we know that in Oz `Ser (First a) (Second b)` is equivalent to `Ser (Second b) (First a)`. This is shown diagramatically in Figure 7.1.

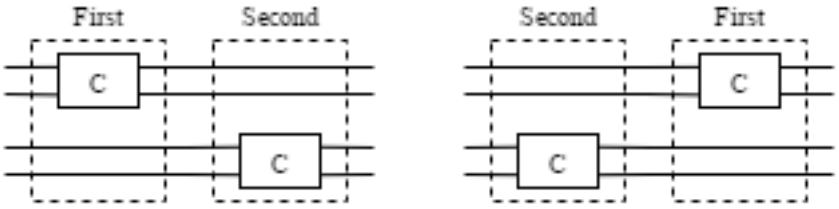

Figure 7.1: Sliding Rule

To implement these sorts of relationships in Idris, we need to devise a normalised circuit form that is agnostic to the construction apparatus. We then express the rules using the propositional equality operator in a manner similar to the following:

```
Sliding : norm (Ser (First a) (Second b)) = norm (Ser (Second b) (First a))
```

By the Curry-Howard Correspondence, a type correct implementation serves as a proof of this proposition. More work needs to be done to demonstrate that circuit structures in Oz do indeed form part of a traced monoidal category.[12] The referenced paper outlines a series of rules like the one above that may be applied in this category.

7.3 Toolchain Integration

One of the greatest motivating factors for the improvement and modernisation of the hardware design process is the woeful inefficiencies of the present toolchain. Hardware design operates through a long, multi-staged iterative process in which the various tools do not communicate well with one another. It is challenging for designers to make informed decisions about later stages; for example, how does an implementation choice at the gate level impact on the performance of firmware functions running on an embedded system? In some ways this is an unavoiable consequence of the fact that the process crosses multiple disciplines. Nevertheless, we contend that using the powerful abstraction available in DT languages, we can improve the flow of information in the design toolchain.

A key starting point to this would involve research into the flow of propagation delay data.[9] The types in Oz could be enriched to be fully aware of wires and placement information. There is promising research in this area applied to languages similar to Lava. Propagation delay has a major impact on low-level design, as we wish to minimize the delay across the pipeline inside a sequential circuit. Constraints can be elicited from the output of one design iteration into type-level predicates, ensuring the next iteration meets new, more precise success criteria.

Given such an enhancement, Oz could also be extended to interface fully with various other parts of the compilation pipeline. One such interaction could involve exposing the circuit rearrangement algebra to place and route routines, finding better approximations of the optimal circuit layout.

7.4 Timing

There are a number of other enhancements that can be considered in the area of timing. The most obvious is the ability to use multiple clock sources, and better encoding of reset signals. This could be achieved by attaching an extra parameter to the `Feedback` circuit constructor that carries information about what event should trigger the loop. In practice this is typically the rising or falling edge of a clock signal, but any enhancements in this area should carefully consider how to implement a generalised *guard*.

More advanced notions of timing may also be considered. One may wish to encode the notion of a Z-transform over circuit objects and model their behaviour in the frequency domain. This might be of particular interest as more timing information is preserved in the circuit model by the above extensions. Many of the operations involved in optimising for clock speed may prove useful for frequency domain manipulations.

Chapter 8

Abbreviations

ASIC	Application-Specific Integrated Circuit
AST	Abstract Syntax Tree
DT	Dependently Typed
EDSL	Embedded Domain-Specific Language
FPGA	Field-Programmable Gate Array
GADT	Generalised Algebraic Datatype
HDL	Hardware Description Language
RTL	Register Transfer Level
TT	Type Theory

Appendix A

Oz Source

Source for Oz is available online at https://github.com/davidwlewis/Oz

Bibliography

[1] C. Baaij, "CλaSH: From Haskell to Hardware," Master's thesis, University of Twente, 2009. [Online]. Available: http://eprints.eemcs.utwente.nl/17922/01/CLasHFromHaskellToHardware.pdf

[2] J. M. S. Bengt Nordström, Kent Petersson, *Programming in Martin-Löf's Type Theory*. Oxford University Press, 1990. [Online]. Available: http://www.cs.chalmers.se/Cs/Research/Logic

[3] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, "Lava: hardware design in haskell," in *ACM SIGPLAN Notices*, vol. 34, no. 1. ACM, 1998, pp. 174–184.

[4] E. BRADY, "Idris, a general-purpose dependently typed programming language: Design and implementation," *Journal of Functional Programming*, vol. 23, pp. 552–593, 9 2013. [Online]. Available: http://journals.cambridge.org/article_S095679681300018X

[5] K. Claessen, "Embedded languages for describing and verifying hardware," Ph.D. dissertation, Chalmers University of Technology and Göteborg University, 2001. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.110.5587

[6] K. Claessen, R. Hhnle, J. Mrtensson, and S. Ab, "Verification of hardware systems with first-order logic," Copenhagen, DIKU, University of Copenhagen, Denmark, Tech. Rep., 2002.

[7] A. Farmer, G. Kimmell, and A. Gill, "Whats the matter with kansas lava?" in *Trends in Functional Programming*. Springer, 2011, pp. 102–117.

[8] A. Gill, T. Bull, G. Kimmell, E. Perrins, E. Komp, and B. Werling, "Introducing kansas lava," in *Proceedings of the 21st International Conference on Implementation and Application of Functional Languages*, ser. IFL'09. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 18–35. [Online]. Available: http://dl.acm.org/citation.cfm?id=1929087.1929089

[9] R. Ho, K. Mai, and M. Horowitz, "The future of wires," *Proceedings of the IEEE*, vol. 89, no. 4, pp. 490–504, Apr 2001.

[10] W. A. Howard, "The formulae-as-types notion of construction," 1995.

[11] (2015, November) Idris documentation. Idris. [Online]. Available: http://docs.idris-lang.org/

[12] A. Joyal, R. Street, and D. Verity, "Traced monoidal categories," *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 119, pp. 447–468, 4 1996. [Online]. Available: http://journals.cambridge.org/article_S0305004100074338

[13] C. Kern and M. R. Greenstreet, "Formal verification in hardware design: A survey," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 4, no. 2, pp. 123–193, Apr. 1999. [Online]. Available: http://doi.acm.org/10.1145/307988.307989

[14] P. Martin-Löf, "An intuitionistic theory of types," 1973.

[15] C. MCBRIDE and J. MCKINNA, "The view from the left," *Journal of Functional Programming*, vol. 14, pp. 69–111, 2004. [Online]. Available: http://journals.cambridge.org/article_S0956796803004829

[16] J. Pizani Flor, "Pi-ware: An embedded hardware description language using dependent types," 2014.

[17] M. R. Prasad, A. Biere, and A. Gupta, "A survey of recent advances in sat-based formal verification," *International Journal on Software Tools for Technology Transfer*, vol. 7, no. 2, pp. 156–173, 2005.

[18] P. Wadler, "Propositions as types," *Unpublished note, http://homepages. inf. ed. ac. uk/wadler/papers/propositions-as-types/propositions-as-types. pdf*, 2014.