Dependent Chisel: Statically-checked hardware designs based on Chisel and Scala 3

Author

Yuchen Du, DTU

Supervisor

Alceste Scalas, DTU

Co-Supervisor

Martin Berger, University of Sussex
Martin Schoeberl, DTU

Master Thesis
Department of Applied Mathematics and Computer Science. 2023, Lyngby, Denmark

Abstract

Chisel is an embedded domain specific language (DSL) in Scala 2 for hardware design. However, some programming errors in Chisel (e.g. bit width mismatch for IO ports) are only found late in Chisel's code-to-hardware pipeline(even during FPGA onboard testing). This complicates the debugging process.

In this project we aim to utilise the new features of the Scala 3 type system to mitigate these issues. The goal is to design and implement a language that can perform more accurate checks and report errors earlier, to speed up the development cycle for hardware design.

Acknowledgements

I would like to thank Alceste Scalas who suggest the topic of this thesis. His guidance helped me concentrate on the most crucial elements and clearly express the ideas, allowing me to approach my research with a well organized and scientific methodology. I appreciate Martin Berger for helping me reflect on my own understanding of theoretical aspects as well as telling me the panorama of the current state of the art.

I am grateful to Guillaume Martres for his kindness and deep knowledge of Scala compiler, guiding me through resolving compiler issues that seemed insurmountable to me. His assistance gave me confidence while I was facing those daunting challenges. I also express my heartfelt gratitude to Christiaan Baaij and Oron Port for sharing their experiences and insights about their own hardware languages to provide me with a unique perspective.

Last but not least, I am indebted to my wife for her unwavering support to encourage me to take the challenges throughout my research journey.

Contents

1 Introduction

The modern semiconductor industry relies on hardware description languages for chip design to replace previous schematic diagram based approach. However, traditional hardware description languages like Verilog gradually becomes a bottleneck for engineers since locating low-level errors such as bit width mismatch errors is tedious. These errors are even found as late as board-level testing. In addition, several limitations of the language such as inconvenient module instantiation, functions with no parameter support, and limited parameterization capability prolong the development cycle. Below is a summary of the problems of Verilog:

- Difficult to debug: errors in Verilog code may be spotted only during board testing.
- Limited expressiveness.
- Bad tooling system: difficult to refactor code, etc.
- Lack of modularity: poor support of parametric modules.
- Lack of Standardization: There are several versions of Verilog.

Although High level synthesis(HLS) is more concise, it doesn't solve the problem either due to the mismatch between the sequential execution model of host language like C and the parallel nature of hardware design.

The industry and academics confirm these issues by developing several new languages like Chisel[2], SpinalHDL[7] and Clash[9], which provide better type safety, parameterization capability, and modularity. Additionally, these languages offer better support for verification, debugging, and simulation, which ultimately results in reduced design time and improved productivity.

However, Chisel has the same issues with Verilog regarding how to deal with size mismatches which is called Implicit truncation as will be described later. Here's a simple example below:

```
1  val io = IO(new Bundle {
2    val a = Input(UInt(16.W))
3    val b = Input(UInt(16.W))
4    val y = Output(UInt(10.W))
5  })
6
7  io.y := io.a + io.b
```

In line 7, the size(bit width) of the left hand side is 16, but it is truncated according to the size of *io.y* which is 10. This would cause potential nasty bugs during hardware design.

In this thesis ,we aim to explore to what extent can Scala 3 type system alleviate those problems, especially for the bit width mismatch problem.

1.1 Research questions and objectives

We observed that there're several problems in the current solution for the circuit design as mentioned in introduction section. To limit the scope of discussion, we have defined two benchmark criteria. Firstly, we examine the correctness of code and the ease of debugging. Secondly, we explore the potential trade-offs between the correctness of code and expressiveness since writing strict and safe code may sacrifice its expressiveness and lead to code repetitions.

Since there're many different aspects for correctness in circuit design which are verified using a variety of methods like theorem proving,model checking and unit testing, this thesis only focus on one specific correctness : bit width mismatch.

To clarify the topics of this thesis, we elaborate and list our objectives below:

- **O1:** How advanced type system can improve correctness, especially bit width mismatch and catch errors earlier
- **O2:** To what extent can we maintain compatibility between static code and dynamic code

In objective O2, static means that the bit width is specified by integers as type parameter which will be checked by compiler, while dynamic means the bit width is computed from runtime values in Scala 3 which is not checked by compiler. This will be discussed in detail in the Section 5 Implementation.

1.2 Thesis structure

First, we introduce some backgrounds about embedded domain-specific languages (eDSL), type system and hardware design languages in Section 2.

Then, we do a brief survey about the current state of the art to spot problems in current design in Section 3.

After sufficient backgrounds, a high level consideration of the design is shown before going into the details of the implementation in Section 4 and 5. Lastly, evaluation of various objectives is tested and we conclude about future work in Section 6 and 7.

2 Background

In this section, some background knowledge about embedded domain-specific languages (eDSL), type system and hardware design languages are briefly introduced as a foundation for later section. The advanced features of the Scala 3 language like implicit parameter and compile time operations(type level arithmetic) will be explained when introducing examples in Section 4.1 and Section 2.3.2

First, several methods for constructing embedded domain-specific languages are introduced. Since DSLs inherent features from host language like type system and modularity, there's lots of interaction between the host language and constructed AST. Thus understanding this interaction is helpful to have desired language features. Also, different methods have pros and cons.

Next, we consider the role of the type theory in circuit design. Nowadays type system becomes a standard and powerful tool for ensuring correctness. By leveraging a more advanced type system, we can encode circuit properties and constraints at the type level, enhancing static analysis and verification. For example, the type system can enforce that signals are connected correctly, and components have compatible data types and bit widths. An advanced type system enables designers to catch potential errors early in the development process, reducing the need for extensive debugging at later stage which could potentially save lots of manpower.

Furthermore, some background on programming languages for hardware doesn't harm, since different hardware description languages (HDLs) operate at different levels of abstraction, indicating various trade-offs between hardware efficiency and ease of development.This gives several implications while we aim to output FIRRTL[3].

2.1 Embedded domain specific languages

A domain-specific language (DSL) is a language for some specialized applications that possibly increase productivity for programmers, in contrast to a Turing-complete general-purpose programming language.

An embedded DSL is a DSL built on top of a host language, which is able to utilize features from the host language like module system and the type system to provide modularity and correctness. This method saves enormous efforts and avoids reinventing the wheels since designing a good language correctly with many features is a hard task.

Although embedded DSL is constrained by the syntax of the host language, it won't severely limit the semantic analysis capability since we have access to the AST(abstract syntax tree) for the 3 approachs which will be introduced later.

2.1.1 Deep vs Shallow Embedding

A embedded DSL(EDSL) is typically designed according to whether it's shallow or deep. In shallow embedding, EDSL structures are more directly mapped to the host language's constructs, which allows more utilization of the features from the host language. However, this approach limits the manipulation of EDSL programs as data. On the other hand, deep embedding represents EDSL constructs as data structures, allowing more manipulation of EDSL programs as data and offering more control over semantics and execution. Yet, this method is more tedious as it requires defining much more custom data structures.

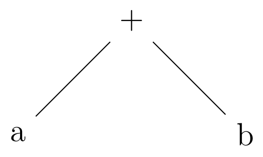
Actually, an EDSL can vary between being shallow and deep so that the "depth" of the language can be designed to achieve the best trade off in mind. For instance, Chisel utilizes the module system from Scala rather than inventing their own which can be deemed as being shallow, while data types like UInt can be deemed as being deep.

2.1.2 Different embeded DSLs

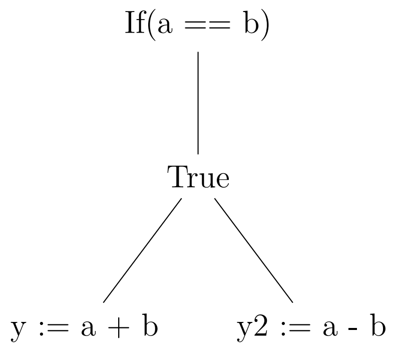
Theoretically, DSL, EDSL and generic programming languages all aim to construct tree data structure(AST) which is used as an intermediary representation in compilers. Thus, the problem comes down to efficiently construct the AST.

- Plain AST construction
- Free monad
- Declarative tree

The first method "Plain AST construction" is to directly write down the tree data structure. A simple expression like a+b can be written as a root tree "+" with two nodes "a" and "b".



However, directly constructing tree in this way can be too tedious for end users when the tree becomes more complex. Let's consider an AST where the if block contains two statement :



The first method to directly construct the AST might look like this:

```
If(a == b,
    Seq(y := a + b,y2 := a - b)
)
```

where the first parameter of If is the condition and the second parameter contains the statement to execute if the condition is True.

Free monad

Monad[28] is a functional programming technique to stage effectful computations and make them compose. Further, free monad[27] (or freer monad) is a technique to simplify the construction of monads. Some host languages like Haskell and Scala provide convenient syntax for writing monads make this method pleasant to use with "do notation" or "for yield" syntax. With monads, side effects are explicitly captured as pure data structures thus it's easier to do further analysis.

The construction of digital circuits doesn't involve side effects like reading and writing to standard input/output of the computer, but some of it's syntax like assignment makes it looks effectful. Since monad is a natural choice for effectful computation in functional programming, it's natural to explore the possibility to construct AST for digital circuits with monad.

Below is an potential monadic syntax:

```
If (a===b,
    for {
        _ <- y := a + b
        _ <- y2 := a - b
    } yield ()
)
```

the information for assignment $y := a + b$ is stored as the context provided by the monad. The type of the for block may be $FreeMonad[Unit]$. A more complicated example to show that free monadic DSL can be converted to AST is shown in Appendix A.2

Declarative tree:

We name the last method **"declarative tree"** since it mainly provides a way to conveniently construct tree data structure. It is adapted by Chisel and spinalHDL and also in our implementation. Some libraries[31] for graphical user interface also have similar API since those graphical view layout is also represented by tree. The DSL in this style may look like below:

```
If(a == b) {
      y := a + b
      y2 := a - b
    }
```

Readers might wonder how the code is converted to AST since it looks like ordinary Scala code. The tricks is discussed in the Implementation section and Appendix A.3.

We choose the last method as it is the most concise for users. It is "shallow" in the sense that many Scala syntax(like how statements are written) is reused, while it is also considered as "deep" because we have access to the AST during later stage.

2.1.3 Limitations of eDSL

Due to the nature of the embedded methodology, embedded DSLs are constrained by the syntax of the host language. In addition, it might be tricky to access the name of the variable, which requires either compiler plugin of the host language or meta programming features like macros to inspect the code. For example, spinalHDL and Chisel use compiler plugin to fill in the name for variables at compile time.

For instance, when referring to variable a in the code below we are unable to know the name unless its name is passed as an argument for the function:

```
val a = newInput[2]("a")
```

2.2 Type theory

Here we give an informal introduction of various type systems used or discussed in this thesis.

2.2.1 Dependent types

Dependent type is a powerful technique for extra type safety in software development, formal verification[23] and formalization of mathematics [24]. A language with dependent types is able to blur the boundary between types and values while still ensuring type safety at compile-time.

For example, dependent types can ensure the sorted property for a list of integers, or to ensure that a function is only called with values within a certain range.

Below is an example to define a function add1 that adds one to the input. The FinN type(which is a type family) is a wrapper for an integer which only has a single constructor called Fin, taking a natural number n and wrapping it as FinN n.

```
data FinN : Nat -> Type where
  Fin : (n : Nat) ->  FinN n

add1: FinN n -> FinN (n+1)
add1 (Fin n) = Fin (n+1)
```

In this way, we are confident from the type signature that the implementation of add1 must conform to it's type specification. The integer parameter for function *add*1 can be constant or runtime values, showing that dependent types blur the boundary between types and value, in contrast to the lightweight dependent types in Scala 3 as will be introduced next.

2.2.2 Lightweight dependent types in Scala 3

Scala 3 doesn't have full support of dependent types where types and values can freely interact, which is a design trade off to retain type inference since full dependent types require type annotation everywhere.

However, it still supports a subset of dependent type features like match types, dependent function type, and so called Compile-time operations. The last one is used in the thesis.

The Compile-time operations in Scala 3 allow certain constants or expressions to be evaluated at compile-time[30]. Those values can then be used as type annotations:

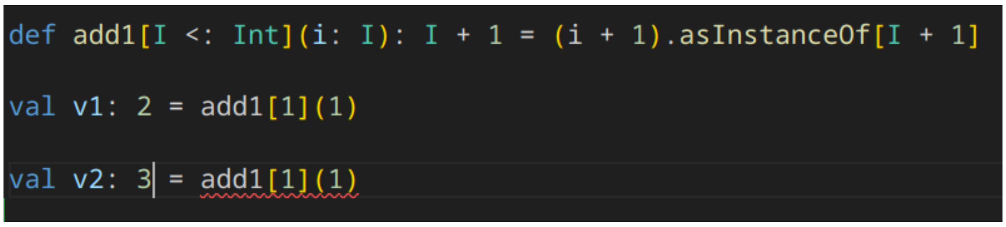
```scala
import Scala.compiletime.ops.int.*
import Scala.compiletime.ops.boolean.*

val conjunction: true && true = true
val multiplication: 3 * 5 = 15
```

In addition, the code below express that the value i must have the type I. If I=5, then the value i must also be 5, similar to the previous example.

```scala
def add1[I <: Int](i: I): I + 1 = (i + 1).asInstanceOf[I + 1]
```

The limitations of the example above is that the function add1 only works if the type parameter are constants rather than arbitrary expressions. While in full dependent type languages like idris or agda, arbitrary value expressions are as type parameters.

Another limitation is the mandatory use of *asInstanceOf*. Without it, the compiler won't be able to infer its precise type to be $I + 1$. Thus cares must be taken when defining those functions since it's not checked by the compiler, but it's safe when users invoke this as shown in the code below where v2 is found to have incorrect type:



```scala
def add1[I <: Int](i: I): I + 1 = (i + 1).asInstanceOf[I + 1]

val v1: 2 = add1[1](1)

val v2: 3 = add1[1](1)
```

Figure 1: type level annotation for integers

This feature can also be used to define a "safe" multiplication operation that won't cause overflow for big numbers, or Auto-ranging Fixed-Point[14] integers in spinalHDL. Let w be the bit width of the number, then the maximum bit width of the result is $2 * w$ which is reflected in it's type.

```scala
  case class Mul[w <: Int](a: Expr[w], b: Expr[w])
      extends Expr[2 * w]
```

The examples presented for type level arithmetic are not directly used in this thesis, but they are helpful for a better understanding of the Scala 3 type system. In our implementation, the bit width is fixed as below:

```scala
extension [w <: Int](x: Expr[w]) {
    def +(oth: Expr[w]): BinOp[w] = BinOp(x, oth, "+")
    def -(oth: Expr[w]): BinOp[w] = BinOp(x, oth, "-")
    def *(oth: Expr[w]): BinOp[w] = BinOp(x, oth, "*")
    def /(oth: Expr[w]): BinOp[w] = BinOp(x, oth, "div")
    def ===(oth: Expr[w]): Expr[1] = BinOp(x, oth, "==").asTypedUnsafe[1]
  }
```

2.2.3 Refinement types

Refinement types allow the specification of additional constraints or properties over types, which can be seen as a restricted form of dependent types. Examples include specifying that a string parameter must be a valid email address, or that a number must be within a certain range, by defining a refinement type "Positive Int" to represent integers greater than zero.

In Scala 3,refinement types are provided by several libraries, and iron[29] is used in this thesis. A simple example[29] expressing that the log function requires Positive input is given below:

```scala
import io.github.iltotore.iron.*
import io.github.iltotore.iron.constraint.numeric.*

def log(x: Double :| Positive): Double =
  Math.log(x) //Used like a normal `Double`
```

In the thesis, it's sometimes used to restrict the range of integers, for example, some parameter of parameterized modules are required to be positive as in Section 6.2.

2.3 Hardware description languages

Current hardware programming languages sit at different levels of abstraction, described by Gajski Kuhn chart[11], each level offers trade-offs between fine control of hardware details like manufacturing area of the chip and high level abstraction to reduce development time.

High-level synthesis(HLS)[10] focus on functionality behavior so it's suitable for algorithm development. However, HLS languages, like SystemC[13] or Bluespec[12] may generate inefficient low level hardware design consuming huge chip areas.

On the other hand, low-level HDLs, like Verilog and Chisel, belong to Register-Transfer Level(RTL)[11], provide a more detailed representation of the circuit. These languages allow for more precise control over circuit layout.

Furthermore, Understanding the semantics of Verilog could provide further insights. However, in the context of our current discussion, we shall refrain from diving into this topic which is currently not well defined yet and readers can consult[15][16]

2.3.1 Levels of hardware programming languages

We list each level of hardware programming languages related to our concern below and which is later compared in detail:

- High-level synthesis (HLS, or Behavioral-Level)

- Register-Transfer Layer(RTL)
- Logic Gate level (aka Gate-Level,netlist)

Logic gate level: Logic gates are the most basic building blocks that provide operations like AND,OR etc. Logic gates can be combined to create memorization abilities called registers(usually as D flip-flops), one of the key abstractions provided by RTL level.

RTL: Logic gate level is now combinational logic in the context of RTL. On top of that, RTL adds Sequential logic which is basically registers, providing an abstraction for stateful computation.

Thus, RTL is more convenient when building memory related features so that developers don't need to manually implement components like D-flip flops. Another abstraction provided by RTL is black box components where some peripherals like SRAMs are provided by the FPGA vendor and RTL designers shall utilize those black-boxed SRAMs [3] rather than synthesis them to logic gates, resulting in more effective design while using synthesis tool of this particular FPGA vendor. A more detailed comparison of combinatorial and sequential logic is listed in Table 1

	Sequential Logic(Registers)	**Combinational Logic**
Stateful	yes	no
Timing	Operates on clock edges	continuously
Loops	yes	no
Delay	yes	no
Implementation	flip-flops	gates and multiplexers
Examples	Counters, shift registers	Adders, multiplexers, decoders

Table 1: detailed comparision of RTL

HLS: Following the same analogy between Assembly language and high level programming languages like C or other object- oriented programming, RTL (Register-Transfer Level) design is analogous to Assembly, offering low-level control over hardware components and requiring manual allocation of registers. On the other hand, HLS (High-Level Synthesis) sit on more abstract level and may not require explicit register allocation.

Current HLS mostly operate on C or C++, thus it's easier to translate algorithms describes in C into HLS code. However, the sequential execution nature of C and C++ doesn't resonate well with the parallel nature of hardware design, so the generated RTL level code may not be not efficient.

In addition, A middle ground between RTL and HLS called dataflow languages like DFiant [4] claims to provide a good trade off between low-level hardware description and high-level programming which eliminate problems in HLS called tyranny of the clock [5](where all components of a digital system are synchronized to a global clock, restricts the design flexibility and hampers performance optimization.). Their clock-agnostic dataflow model eliminates explicit register placements and clock dependencies so it can be deemed as a new kind of HLS.

2.3.2 DSL at RTL level

Chisel and spinalHDL belong to RTL level since they require developers to explicitly write down registers. However they are more expressive than Verilog, just like macros in C makes C more powerful. They are also called circuit generators because developers can use them to generate code to avoid duplication. This concept of zero cost abstraction doesn't incur runtime performance cost

In this sense, Chisel like languages are similar to Rust[17], where none of Rust abstractions impose a global performance penalty[18]. This is in contrast to garbage collected languages which impose runtime overhead.

2.4 Verification of circuit design

The testing and verification of circuit design is crucial in hardware design since the chip manufacture is quite expensive. Unlike software engineering where developing new features is sometimes more important than fixing bugs, a defect in hardware design could cost millions like the Pentium FDIV bug.

The Verification is a complicated process and a simplified illustration is displayed in Fig 2 below:

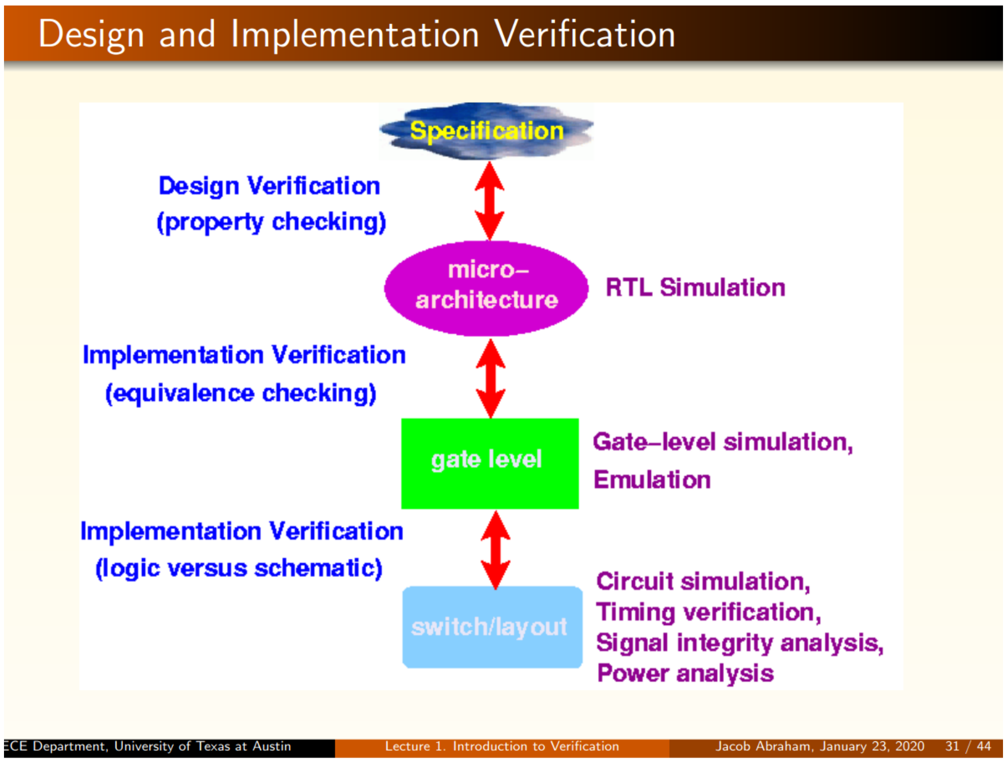

Design and Implementation Verification
Specification
Design Verification
(property checking)
micro–architecture
RTL Simulation
Implementation Verification
(equivalence checking)
gate level
Gate–level simulation,
Emulation
Implementation Verification
(logic versus schematic)
switch/layout
Circuit simulation,
Timing verification,
Signal integrity analysis,
Power analysis
ECE Department, University of Texas at Austin
Lecture 1. Introduction to Verification
Jacob Abraham, January 23, 2020 31 / 44

Figure 2: Overall verification process in hardware design, figure taken from [8]

To further clarify the process, the most relevant part is explained in more detail below:

- **Designing :** design the circuit in some hardware description language like Chisel, Verilog.
- **Running tests in simulation:** unit test and other kind of test to gather runtime data is performed in a simulation environment and the output is compared to the expected result. In addition, successfully booting some operating systems like Linux is also a common benchmark.
- **Formal Verification:** This step involves mathematically proving that certain properties of the design is true under all possible conditions. Formal verification techniques like model checking and theorem proving are used.
- **Hardware/FPGA Prototyping:** The design is loaded into special hardware (emulator or FPGA) for more extensive testing.

It's important to note that although many verification tasks is done after designing as discussed above, some checks can still be done during design which can be done to spot errors earlier [7] :

- Bit width mismatch
- Uninitialized/dangling registers
- Detect combinatorial loops
- Clock crossing violation

In this thesis, we only focus on the bit width mismatch problem since it can be checked by Scala 3 type system. Other checks can be done in FIRRTL compiler.

3 Related work

Here we briefly introduce the syntax and some aspects of Chisel and other hardware description languages.

3.1 Chisel

Chisel is a embedded DSL written in Scala which belongs to the "declarative tree" style DSL as discussed earlier in Section 2.1.2. Maybe the simplest circuit is an adder that can add two numbers:

```
1 class adder extends Module {
2   val io = IO(new Bundle {
3     val a = Input(UInt(16.W))
4     val b = Input(UInt(16.W))
5     val y = Output(UInt(16.W))
6   })
7   io.y := io.a + io.b
8 }
```

Users are required to extend the library interface *Module* provided by Chisel and declare the input and output interface in line 3 to 5. In this example, there're two inputs and one output whose bit width is 16. After that, the relation between input and output is specified as addition in line 7.

Further, the parameterized adder that supports variable bit width is shown below where size is a value parameter for the class:

```
1 class adder(size: Int)  extends Module {
2   val io = IO(new Bundle {
3     val a = Input(UInt(size.W))
4     val b = Input(UInt(size.W))
5     val y = Output(UInt(size.W))
6   })
7   io.y := io.a + io.b
8 }
```

The adder example in spinalHDL[7] is similar so it won't be described here.

3.1.1 Syntax of Chisel

It would be helpful to specify the syntax in a more formal way. A Chisel module is a modular hardware component that can be instantiated multiple times at different call site, defined as Scala classes that extend the Chisel Module class, allowing users to create flexible and concise hardware designs.

One of the prominent features of Chisel is parameterized modules as introduced before. The parameters are specified as class constructor arguments includes various types, such as integers, booleans, or other Chisel types. Due to the lack of formal syntax specification, an incomplete specification of Chisel syntax is given below:

```
modules = class MyModule(params...) extends Module {
  val io = IO(new Bundle {
    ioDecls...
  })
  circuitStmts
}

ioDecls = val identifier = Input(...) | Output(...)

circuitStmts =
    identifier:= expr  |
    when(expr) {circuitStmts} |
    when(expr) {circuitStmts} otherwise {circuitStmts} |
    Module(MyModule) |
    ...

expr = expr bop expr | uop expr

bop = +| − | ∗ | /

uop = ~
```

where params are parameters for Scala types where Chisel types like UInt shall not be used. Since it's built on top of Scala's object-oriented class system, Chisel ensures modular program design.

After users define their modules, one of them is the top level main module which can be instantiated by a Chisel library function to generate FIRRTL.

3.1.2 Compile stage of Chisel

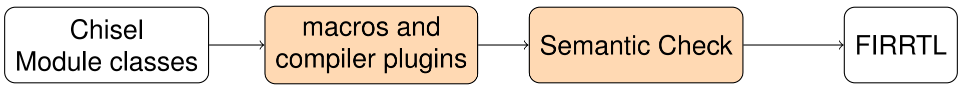


Figure 3: compile stage from Chisel to FIRRTL

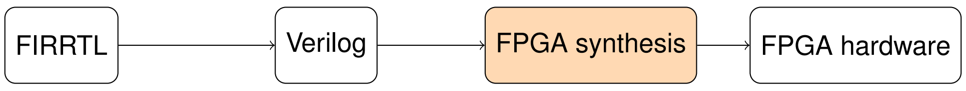


Figure 4: compile stage from FIRRTL to FPGA hardware

In Figure 3, user defined Chisel classes will go through compile stage transformations like inserting names for variables(which is for better debugging), like macros and compiler plugins. Then the code is compiled and executed to do some simple semantic checks before generating FIRRTL[3].

Further, In Figure 4 FIRRTL is compiled to verilog by the FIRRTL compiler[3] before finally burn into FPGA hardware to actually run in the process graph below:

Chisel have some static checks in compile stage to make sure different data types like UInt and SInt are not mixed. However, bit width is not statically checked.

3.2 Bit width check and implicit truncation

Chisel is unable to detect some errors due to a feature which we name "implicit truncation". It occurs when a value in the left hand side with bigger size is assigned to a value in the right hand side with smaller size or operations take arguments of different size, leading to the truncation of the value with bigger size.

For the example below:

```scala
class adderWidthTrunc extends Module {
  val io = IO(new Bundle {
    val a = Input(UInt(16.W))
    val b = Input(UInt(16.W))
    val y = Output(UInt(10.W))
  })

  io.y := io.a + io.b
}
```

the output io.y have smaller size than left hand side values. When generating FIRRTL, Chisel doesn't output any warnings or errors. The FIRRTL compiler(which converts FIRRTL to Verilog) doesn't check that either, so the below erroneous Verilog code is output:

```verilog
1  module adderWidthTrunc(
2    input         clock,
3    input         reset,
4    input  [15:0] io_a,
5    input  [15:0] io_b,
6    output [9:0]  io_y // size issue!
7  );
8    wire [15:0] _io_y_T_1 = io_a + io_b; // @[adder.Scala 21:16]
9    assign io_y = _io_y_T_1[9:0]; // @[adder.Scala 21:8]
10 endmodule
```

At line 4 to 6 of above, the bit width mismatch happens but developer got no error or warnings at all. As we see in Figure 4 this verilog code could continue to go into FPGA hardware which is quite hard to backtrack into the root cause in Scala code. This can be harmful because it can lead to unwanted loss of information or unexpected behavior. Furthermore, bugs caused by implicit truncation can be difficult to detect when it causes issues since the resulting design may appear to work correctly in some cases but fail in others. This intermittent behavior may cause some of the most tricky kind of bugs in software engineering.

In spinalHDL, the problem of implicit truncation seems to be dealt better since it is disabled by default, and the explicit annotation "resized" is required . The design below

```scala
class TopLevel extends Component {
  val a = UInt(8 bits)
  val b = UInt(4 bits)
  b := a
}
```

will throw width mismatch error at Scala runtime and preventing users to continue downward, and the fix below is required:

```scala
class TopLevel extends Component {
  val a = UInt(8 bits)
  val b = UInt(4 bits)
  b := a.resized
}
```

This method requires users to provide explicit casting by writing down **resized**. However, "resized" doesn't provide extra information about how the truncation is processed (which bits to keep and which bits to throw), so it merely adds extra burdensome by forcing users to put "resized" everywhere. Thus, It still incurs similar issues like Chisel.

In **Clash**[9] it's similar to spinalHDL which requires explicit annotation:

```haskell
f :: Signed 4 -> Signed 8
f x = resize x
```

4 Design

The goal of our design is to create a hardware description language with extra correctness checks by utilizing more advanced type systems for bit width correctness. We name it **dependent Chisel**, where bit width truncation in Section 3.2 in Chisel and spinalHDL is intentionally not supported. Instead, users would have to write very explicitly how to cast different sizes. We believe this small sacrifice will pay out fruitfully in the future since size mismatch errors will be easily find rather than hiding deep and only shown up randomly.

We list important aspects and decisions of our design below :

Parametric modules: We aim to support static size check for parametric modules which is formally verified by the Scala type checker, whereas spinalHDL only do size checking at Scala runtime. This is similar to formal verification versus testing. Formal verification can provide correctness under all possible conditions, while testing can provide partial correctness when there's a test case covering that.

Host language: Scala 3 is chosen as the host language since it adds lightweight dependent types as introduced in the Background section which makes it possible to have more compile time checks. In addition, since the original Chisel is built on top of Scala 2, it's natural to upgrade to Scala 3.

Syntax: Based on the success of Chisel and the promising ecosystem of the RISC-V microprocessor around it, we have opted to design the embedded DSL using a modified version of the Chisel syntax as described previously as declarative tree in Section 2.1.2. In this new syntax, certain redundancies, such as bundles, have been eliminated. As discussed in the Background section, this kind of syntax is expressive and intuitive for end users. It's also possible to use monadic DSL as introduced in section 2.1, so that many usage of Scala implicit parameters might be removed.

4.1 Appetizer:Adder

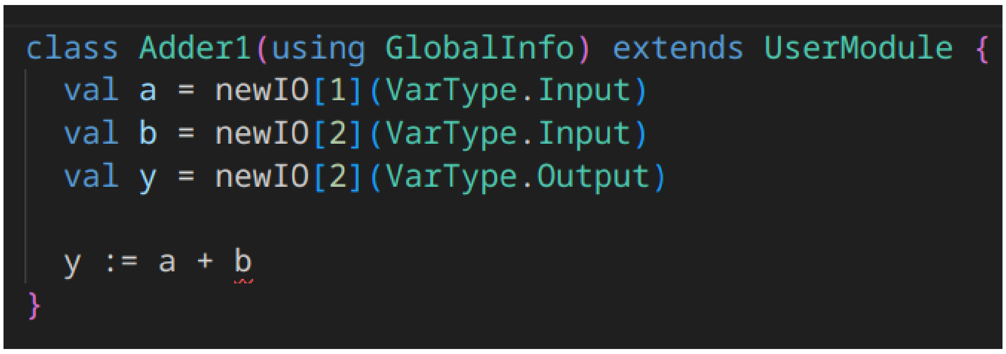
Here is a small examples introduced to warm up the reader without touching the implementation details. Below is a simple adder corresponding to the previous example in Section 3.2:

```scala
// dependent Chisel
class Adder1(using GlobalInfo) extends UserModule {
    val a = newInput[2]("a")
    val b = newInput[2]("b")
    val y = newOutput[2]("y")

    y := a + b
  }
```

Notable changes are that the width now becomes type parameter rather than a function parameter, and there's no need to wrap the input and output inside some extra boilerplate.

To mention a few implementation details, we require that the class has an implicit parameter $GlobalInfo$ which can be implicitly consumed by functions in the scope. Here *newInput* will take *GlobalInfo* as an implicit parameter. It might be possible in the future to remove this requirement since this adds some inconvenience for users.

To test the capacity of the type system, readers can modify the type parameter in line 2 to 4 in the above code to a different number other than 2. An error will show up in editor[26] or a compile error will appear when invoking sbt to compile the Scala project after this modification:



```scala
class Adder1(using GlobalInfo) extends UserModule {
  val a = newIO[1](VarType.Input)
  val b = newIO[2](VarType.Input)
  val y = newIO[2](VarType.Output)

  y := a + b
}
```

Figure 5: Errors shown in vscode with metals plugin for y := a + b

In addition, invoking the compiler via command line also show the same error during compile time. This shows that bit width errors can be caught at compile time.

5 Implementation

A detailed explanation of dependent Chisel from the syntax, semantics and the whole compilation process is presented in this section.

5.1 Compile stage

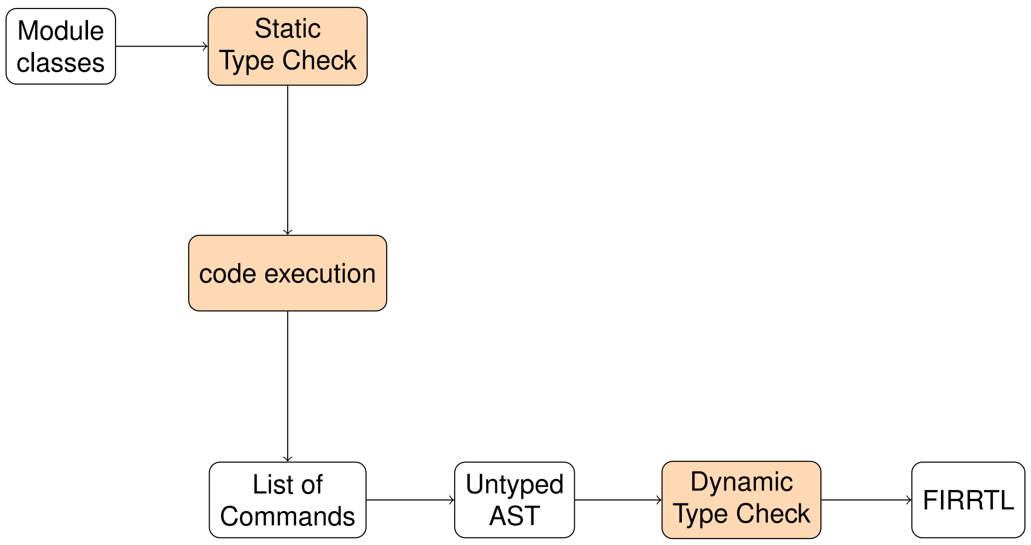


Figure 6: compile stage for dependent Chisel before FIRRTL

The overall process is described in Figure 6 and the relevant source code is[35]. The stages after FIRRTL is the same as Figure 4. Here, Module classes stand for Scala classes defined by end users which extend library interface $UserModule$, providing mechanisms to store newly defined inputs, outputs and assignments. One top level main class is required as the entrance of the whole design.

Scala type checker will then try to type check the code for those module classes. After type checking, the Scala code is executed to generate a list of commands where such list contains atomic assignments. In addition, the dynamic bit width check is also performed at this stage. This will be discussed in detail in Section 5.3.

The list of commands are then converted to abstract syntax tree by an algorithm which utilizes the stack data structure. More specifically, statements like assignments and control statements will be pushed into a list during class instantiation. For example, the control structure IfElse is implemented below:

```
def IfElse[w <: Int](
        b: Bool
    )(block: => Any)(block2: => Any): Unit = {
      pushBlk(Ctrl.If(b))(block)
      pushBlk(Ctrl.Else())(block2)
    }
```

Then this list of commands are converted to tree. Since it's not the core part of the thesis, The code and an example are placed in Appendix A.3 and online repository[33].

5.2 Syntax

A more formal description of the syntax could be helpful to understand the expressiveness and limit of our DSL. Unlike Chisel, we don't discriminate between IO declaration and circuit statements, bringing in more flexibility. However, we require an implicit parameter GlobalInfo.

```
modules = class userMod(using GlobalInfo) extends UserModule {
    circuitStmts
  }

circuitStmts = identifier:= expr |
    If(expr) {circuitStmts} |
    If(expr) {circuitStmts}{circuitStmts} |
    newMod(userMod) |
    newIO[width](...) |
    newInput[width](...) |
    newOutput[width](...) |
    newIODym(width,...)

expr = expr bop expr | uop expr

bop = +| — | * | /

uop = ~

width = integers
```

we might use newInput[width]() sometimes which is a short hand for newIO[width](VarType.Input)

A simple example of a **statically typed** adder with fixed width is below. This kind of definition is referred to as **static** because the bit width is given as type signature:

```
    class Adder(using GlobalInfo) extends UserModule {
    val a = newIO[2](VarType.Input)
    val b = newIO[2](VarType.Input)
    val y = newIO[2](VarType.Output)

    y := a + b
  }
```

the corresponding dynamically typed version which is similar to chisel, is given below. This kind of definition is referred to as **dynamic** because the bit width is given as value parameter for:

```scala
class AdderDym(using parent: GlobalInfo) extends UserModule {
    val a = newIODym(2, VarType.Input)
    val b = newIODym(2, VarType.Input)
    val y = newIODym(2, VarType.Output)

    y := a + b
  }
```

notice that the bit width is specified as value parameter rather than type parameter now.

5.2.1 Parameterized modules

In our new dependent Chisel, parameterized modules have two flavors. Specifically, they can also be parameterized by dependent type parameters, so we shall discuss this here in more detail.

Firstly, a simple **dynamic** parameterized adder is shown below, where the width parameter is a value parameter:

```scala
    class Adder(using GlobalInfo)(width: Int) extends UserModule {

    val a = newIODym(width, VarType.Input)
    val b = newIODym(width, VarType.Input)
    val y = newIODym(width, VarType.Output)

    y := a + b
  }
```

We already introduced the **statically typed** version before in Section 4.1. Here we aim to show the parameterized module with static guarantee:

```scala
class AdderTypeParmClass[I <: Int: ValueOf](using GlobalInfo) extends
   UserModule {
    val a = newInput[I]("a")
    val b = newInput[I]("b")
    val y = newOutput[I]("y")

    y := a + b
  }
```

The type class requirement ValueOf is added to type signature due to the limitations[19] of current Scala 3 compiler.

5.2.2 Mixing static and dynamic design

Sometimes parameters for some modules are determined during runtime execution, and some of them are statically typed. An example of calling an dynamic module from an static one is shown below :

```
1  class AdderDym(using parent: GlobalInfo) extends UserModule {
2
3      val a = newIODym(2, VarType.Input)
4      val b = newIODym(2, VarType.Input)
5      val y = newIODym(2, VarType.Output)
6
7      y := a + b
8    }
9
10 class AdderCallDym(using GlobalInfo) extends UserModule {
11
12     val a = newInput[2]("a")
13     val b = newInput[2]("b")
14     val y = newOutput[2]("y")
15
16     val m1 = newMod(new AdderDym)
17     m1.a := a
18     m1.b := b
19     y := m1.y.asTyped[2]
20   }
```

In line 19, the output y of dynamic module m1 is casted to a static bit width of 2. This casting is checked at runtime.

In line 17-18, no casting is needed since the left hand side is dynamically typed.

5.3 Semantical analysis : width checking

The syntax is about the grammatical structure of programs describing "correct" programs before doing context sensitive analysis at compile time, while the semantics[22] aims to endow "meaning" to programs where meaning usually means some kind of computation or checks. Here, We only focus on one aspect of the semantics: bit width checking, to ensure that signals with different sizes are mixed in a safe way.

In our design, there're two stages of width check for statically known size and runtime size, ensuring all the width mismatches will be spotted even when the design contains dynamic modules.

5.3.1 Static width checking

The first stage of width checking is performed by Scala 3 type checker. With the help from Scala 3, we are able to utilize the value of integers as type signature for stricter type checking.

For example, the below definition of a case class allows the compiler to distinguish VarTyped[1] and VarTyped[2]

```scala
case class VarTyped[w <: Int](name: String, tp: VarType) extends Var[w](name
  )
```

when there's a method like below:

```scala
def plus[w <: Int](x: Expr[w],oth: Expr[w]): BinOp[w] =...
```

the two parameters are required to have a the same size w.

5.3.2 Dynamic width checking

The second stage of bit width checking works like other type checking algorithm. The initial type information is stored in parameter *typeMap*. We define the signature of *getExprWidth* below:

```scala
def getExprWidth(
      typeMap: Map[Expr[?], Option[Int]],
      expr: Expr[?]
  ): Int
```

It retrieves the bit width of expressions and continues recursively, propagating the width through expressions. Aside from calculating the bit width of expressions, It also check whether the bit width are compatible between sub expressions. If not, an exception will be thrown.

checkCmdWidth is used to check assignments(AtomicCmds) and have the signature below :

```scala
def checkCmdWidth(
      typeMap: Map[Expr[?], Option[Int]],
      cmds: AtomicCmds
  )
```

It checks whether the bit width of the left hand side and the right hand side match. If not, a runtime exception will be thrown.

The code for bit width checking is in Appendix A.4 and also online at [34]

5.4 Other difference from Chisel and spinalHDL

Chisel and spinalHDL rely on Scala 2 macros and compiler plugins for various features of the library like automatic name generation for variables, whereas macros in Scala 3 are less powerful. However, in dependent Chisel, macros are not used which leads to the simplification of the codebase. It also makes it easier for further development.

6 Evaluation

We shall recap the objectives defined at the start of the thesis. Firstly, we consider how an advanced type system can enhance correctness by preventing bit width mismatches and identifying errors at an earlier stage. Secondly, we explore the degree to which we can preserve compatibility between static and dynamic code.

To check to what extent we have achieved the objectives, several examples are provided and evaluated in this section.

6.1 Case study : Adder

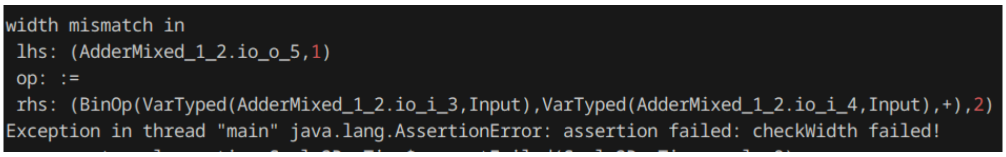
We have already seen a simple example in the Design Section. We shall now do some further evaluation on more complex designs.

6.1.1 Example 1: mixing static and dynamic modules

Before, we introduced a simple statically typed adder with fixed bit width in Section 4.2. Let's see another example showing mixed static and dynamic bit width check:

```scala
class AdderMixed(using GlobalInfo)(size: Int) extends UserModule {
    val a = newIO[2](VarType.Input)
    val b = newIO[2](VarType.Input)
    val y = newIODym(size, VarType.Output)

    y := a + b
  }
```

here the bit width of input a and b is fixed, but the bit width of y is dynamic. In this scenario, the bit width check will be deferred to Scala runtime. If *AdderMixed* is called with *AdderMixed*(1), then an error will be thrown while running the Scala code:



```
width mismatch in
 lhs: (AdderMixed_1_2.io_o_5,1)
 op: :=
 rhs: (BinOp(VarTyped(AdderMixed_1_2.io_i_3,Input),VarTyped(AdderMixed_1_2.io_i_4,Input),+),2)
Exception in thread "main" java.lang.AssertionError: assertion failed: checkWidth failed!
```

Figure 7: bit width error for AdderMixed(1)

This shows that our implementation is flexible to allow mixing static and dynamic designs and still maintain correctness.

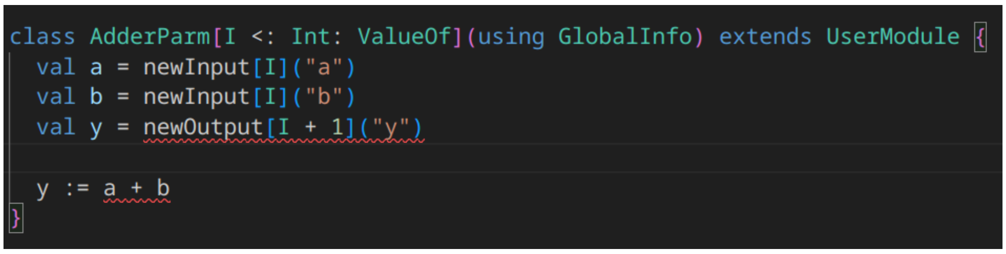
6.1.2 Example 2: parametric static module

It's possible to utilize the type system even further and parameterize the adder by numeric type parameter I while keeping static type checking:

```scala
class AdderParm[I <: Int: ValueOf](using GlobalInfo) extends UserModule {
    val a = newInput[I]("a")
    val b = newInput[I]("b")
    val y = newOutput[I]("y")

    y := a + b
  }
```

In line 1, $[I <: Int : ValueOf]$ shall be read and understand as $I <: Int$ and $I : ValueOf$. the ValueOf type class constraint make sure the value of I can be retrieved at compile time.

To show that this ensures static safety, If we change I to $I + 1$ for some input or output, there will also be errors:



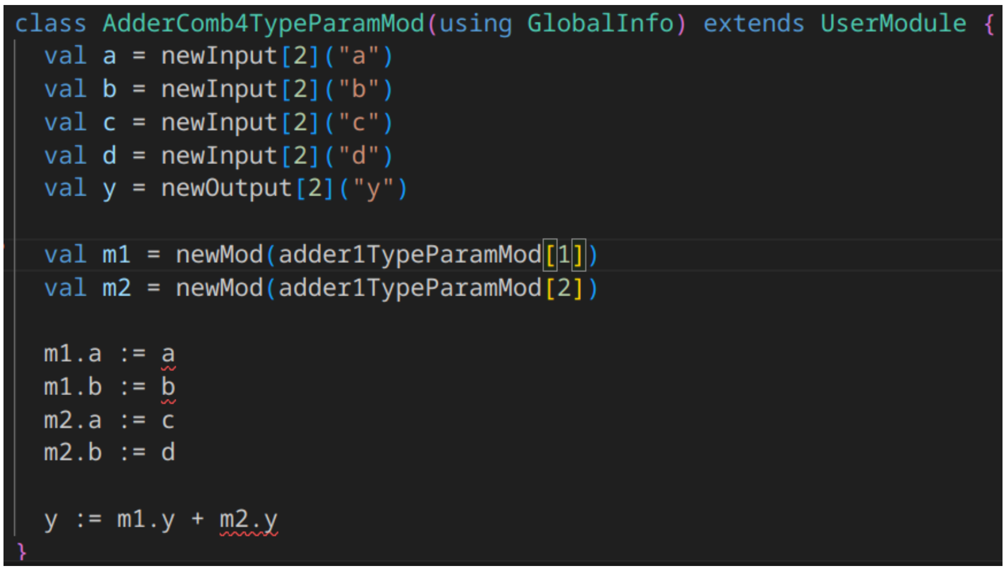
```scala
class AdderParm[I <: Int: ValueOf](using GlobalInfo) extends UserModule {
  val a = newInput[I]("a")
  val b = newInput[I]("b")
  val y = newOutput[I + 1]("y")

  y := a + b
}
```

Figure 8: static safety for parametric module

Later, this adder can be instantiated and called in other module to form more complicated circuits. Here is an example about an adder with 4 input. It instantiates two modules m1 and m2, and then wires it's own input and output with m1 and m2:

```scala
class Adder4(using GlobalInfo) extends UserModule {
    val a = newInput[2]("a")
    val b = newInput[2]("b")
    val c = newInput[2]("c")
    val d = newInput[2]("d")
    val y = newOutput[2]("y")

    val m1 = newMod(new AdderParm[2])
    val m2 = newMod(new AdderParm[2])

    m1.a := a
    m1.b := b
    m2.a := c
    m2.b := d

    y := m1.y + m2.y
  }
```

Here if you give an incorrect type parameter, for example, at line 8, the compiler or editor will give an instant type error which allows the programmer to fix it immediately rather than in the later stage:



```
class AdderComb4TypeParamMod(using GlobalInfo) extends UserModule {
  val a = newInput[2]("a")
  val b = newInput[2]("b")
  val c = newInput[2]("c")
  val d = newInput[2]("d")
  val y = newOutput[2]("y")

  val m1 = newMod(adder1TypeParamMod[1])
  val m2 = newMod(adder1TypeParamMod[2])

  m1.a := a
  m1.b := b
  m2.a := c
  m2.b := d

  y := m1.y + m2.y
}
```

Figure 9: bit width error for parametric module

In comparison, the adder implementation in Chisel doesn't give any errors because of implicit truncation as discussed before neither statically nor at runtime:

6.2 Case study : BubbleFifo

Here we introduce a more complex example called BubbleFifo[1], combining both dynamic and static check to show that our design is practical to use in the real world circuit design. Readers can refer to the Appendix A.1 for the original Chisel implementation. This is a modular design, the input and output is defined first, then FifoRegister is defined which is finally invoked to form the complete design.

A FIFO(First-In-First-Out) buffer is a queue like data structure where the first element entered is the first to exit. BubbleFifo is one of the hardware implementation of this data structure.

First the input and output for BubbleFifo is defined below :

```
class WriterIO(using ModLocalInfo)(size: Int :| Positive) {

    /** Input */
    val write = newIO[1](VarType.Input) // Bool is same as UInt<1>
    /** Output */
    val full = newIO[1](VarType.Output)

    /** Input */
```

```
 9     val din = newIODym(size, VarType.Input)
10     // val din = newIO(VarType.Input, Some(size))
11   }
12
13 class ReaderIO(using ModLocalInfo)(size: Int) {
14
15     /** Input */
16     val read = newIO[1](VarType.Input)
17     /** Output */
18     val empty = newIO[1](VarType.Output)
19     /** Output */
20     val dout = newIODym(size, VarType.Output)
21   }
```

In line 1, we use refinement types in $size : Int : |Positive$ to constrain the size to be positive, in contrast with the original Chisel implementation where the type annotation is just $size : Int$.

Then the input and output are called in FifoRegister, which is used later:

```
 1 class FifoRegister(using GlobalInfo)(size: Int :| Positive) extends
      UserModule {
 2     val enq = new WriterIO(size)
 3     val deq = new ReaderIO(size)
 4
 5     val (empty, full) = (newLit(0), newLit(1))
 6
 7     val stateReg = newRegInitDym(empty)
 8     val dataReg = newRegInitDym(newLit(0, Some(size)))
 9
10     If(stateReg === empty) {
11       IfElse(enq.write) {
12         stateReg := full
13         dataReg := enq.din
14       } {
15         If(stateReg === full) {
16           If(deq.read) {
17             stateReg := empty
18             /* in the book,it's dataReg := 0.U which has size error,but
                  firrtl and Chisel allows it */
19             dataReg := newLit(0, Some(size))
20           }
21         }
22 :   }
23     }
24
25     enq.full := (stateReg === full)
26     deq.empty := (stateReg === empty)
27     deq.dout := dataReg
28   }
```

It's exciting that dependent Chisel find out the **line 19** of above code violates our stricter bit width policy, where the assignment **dataReg := 0.U** will cause an error in our implementation where explicit size annotation in **line 20** is required. Like before, Chisel doesn't report errors or warnings about the bit width mismatch due to the implicit truncation feature.

This is not a bug in this particular situation, but rather a feature of Chisel that may save some typing. While it functions as intended in this context, other similar scenarios may actually result in bugs if they are used without proper caution.

Thus, although implicit width truncation won't necessarily cause errors in this design, it could potentially lead to very subtle bugs in other circuit design under different scenarios. That's why we always impose strict size check in circuit design to ensure correctness.

In the end, BubbleFifo invokes FifoRegister to form the final circuit:

```
class BubbleFifo(using GlobalInfo)(size: Int :| Positive, depth: Int)
      extends UserModule {
    val enq = new WriterIO(size)
    val deq = new ReaderIO(size)

    val buffers = Array.fill(depth) { newMod(new FifoRegister(size)) }

    val depList = 0 until depth - 1
    assert(depList.nonEmpty)
    depList foreach { i =>
      buffers(i + 1).enq.din := buffers(i).deq.dout
      buffers(i + 1).enq.write := ~buffers(i).deq.empty
      buffers(i).deq.read := ~buffers(i + 1).enq.full
    }

    // bulk conn : io.enq <> buffers(0).io.enq
    buffers(0).enq.din := enq.din
    enq.full := buffers(0).enq.full
    buffers(0).enq.write := enq.write

    // bulk conn :  io.deq <> buffers(depth - 1).io.deq
    deq.dout := buffers(depth - 1).deq.dout
    deq.empty := buffers(depth - 1).deq.empty
    buffers(depth - 1).deq.read := deq.read
  }
```

6.3 Wrap up and Reflection

Both objectives are achieved as illustrated in the examples that errors are either immediately shown in the editor or shown up during runtime execution, and statically typed design can be freely mixed with dynamic design. In the last example, it shows that our implementation is capable of designing complex circuits while maintaining bit width correctness.

By utilizing Scala 3 type system to enforce bit width check, developers can spot width mismatch instantly when the width is known statically. When the width is not known at compile time, the checks will be delayed to runtime and no correctness is compromised. Our implementation also allows those two approaches inter-operate in a seamless way.

However, if a full dependent type language is used then all width check can be performed at compile time since full dependent types blurs the line between runtime and compile time, then all computation can be done in compile time as well.

For the implementation side, it might be possible to remove the need for users to always add "(using GlobalInfo)" when defining modules. For better a architecture design for the compiler, It might be also possible to store bit width information in AST and then do type checking with this typed AST rather than the current ad-hoc implementation in section 5.1.

7 Conclusions

The main contribution of this thesis is to implement a new domain specific language to enforce compile time bit width check for circuit design. During the evaluation, we show that it's possible to statically check for bit width mismatch, and this feature could maintain compatibility between statically typed design and dynamic design. In Section 6.2 we also use refinement types to limit the scope of a value parameter to be positive.

The BubbleFifo example in Section 6.2 shows that dependent Chisel is capable of doing complex designs rather than just toy examples.

Reflecting on the lightweight dependent types, the static verification capabilities in our implementation are limited comparing to full dependent type languages like Idris. However, Scala 3 type system still offers significant improvements over traditional statically typed languages like Scala 2. Other new features such as dependent function types and the match types extension, might open new possibilities for other type-level checks, like detection of combinatorial loops.

In summary, dependent Chisel provides stricter and earlier bit width checks compared to both Chisel and SpinalHDL, potentially helping developers catch errors earlier and may ultimately lead to more reliable designs and lower the testing cost. The work is open sourced [32].

7.1 Future work

We have identified several areas for further research below.

7.1.1 More checks in design stage

Aside from trying out ideas in full dependent type languages like Idris and Agda, It would also be interesting to apply other features like match types and dependent function types in Scala 3 to port checks like combinatorial loop detection to compile time type check, as mentioned before.

An ambitious exploration is to bridge the gap between hardware design and verification, which is currently a separate workflow. For example, it's possible to express that a list is sorted in dependent types, it might also be possible to specify the behavior of the circuit and extract a concrete design from it, like compiling programs written in dependently typed languages to executable code.

7.1.2 More expressive language

It could also be a huge advancement if some high level synthesis features can be supported so developers can strike an optimal balance between productivity and hardware efficiency. In addition, Algebraic data types(ADT) is a graceful way to define data, and its support in a hardware language would be exciting.

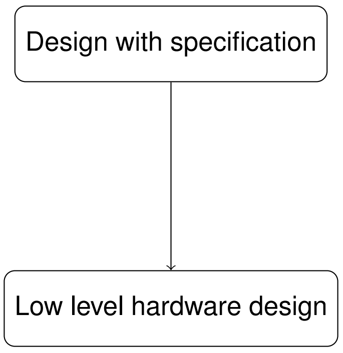


Figure 10: Extract implementation from specification

7.1.3 Usability

Currently, the error reporting is very rudimentary and not user friendly, and improvement on that side is helpful for further adaption.

A Appendix

A.1 The original BubbleFifo

This is the original Chisel implementation[1] for BubbleFifo:

```
package example

import chisel3._
import chisel3.util._

class WriterIO(size: Int) extends Bundle {
  val write = Input(Bool())
  val full = Output(Bool())
  val din = Input(UInt((size).W)) // size - 1
}

class ReaderIO(size: Int) extends Bundle {
  val read = Input(Bool())
  val empty = Output(Bool())
  val dout = Output(UInt(size.W))
}

class FifoRegister(size: Int) extends Module {
  val io = IO(new Bundle {
    val enq = new WriterIO(size)
    val deq = new ReaderIO(size)
  })

  val empty :: full :: Nil = Enum(2) // 0.uint,1.uint
//  val ee2 = Enum(200) //UInt<8>("hc7")
//  val (empty, full) = (0.U, 1.U)
  val stateReg = RegInit(empty)
  val dataReg = RegInit(0.U(size.W))

  when(stateReg === empty) {
    when(io.enq.write) {
      stateReg := full
      dataReg := io.enq.din
    }
  }.elsewhen(stateReg === full) {
    when(io.deq.read) {
      stateReg := empty
//      dataReg := 0.U // just to better see empty slots in the waveform
```

```
      dataReg := 1001.U // just to better see empty slots in the waveform
    }
  }.otherwise {
    // There should not be an otherwise state
  }

  io.enq.full := (stateReg === full)
  io.deq.empty := (stateReg === empty)
  io.deq.dout := dataReg
}

/** This is a bubble FIFO. */
class BubbleFifo(size: Int, depth: Int) extends Module {
  val io = IO(new Bundle {
    val enq = new WriterIO(size)
    val deq = new ReaderIO(size)
  })

  val buffers = Array.fill(depth) { Module(new FifoRegister(size)) }
  for (i <- 0 until depth - 1) {
    buffers(i + 1).io.enq.din := buffers(i).io.deq.dout
    buffers(i + 1).io.enq.write := ~buffers(i).io.deq.empty
    buffers(i).io.deq.read := ~buffers(i + 1).io.enq.full
  }

  io.enq <> buffers(0).io.enq
  io.deq <> buffers(depth - 1).io.deq
}
```

A.2 Convert monadic code to AST

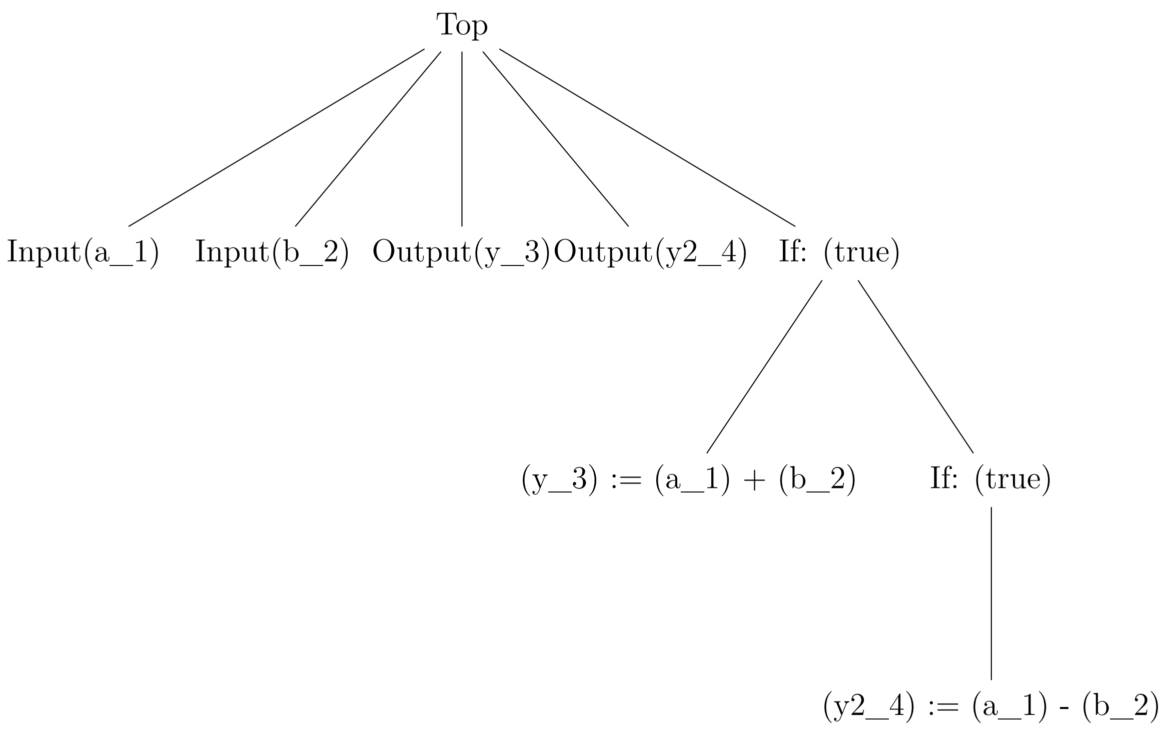
We have successfully implement an stateful compiler for free monads [36] that the contrived monadic code which is purely for illustration purposes below:

```
for {
      a <- newIn("a") // input
      b <- newIn("b") // input
      y <- newOut("y") // output
      y2 <- newOut("y2") // output
      _ <- if_(
        BoolConst(true),
        for {
          _ <- y := a + b
          _ <- if_(
            BoolConst(true),
            for {
              _ <- y2 := a - b
            } yield ()
          )
        } yield ()
      )
    } yield ()
```

can be converted to AST:

```
TreeNode(
  value = Top(),
  cld = ArrayBuffer(
    TreeNode(value = Decl(decl = "Input(a_1)"), cld = ArrayBuffer()),
    TreeNode(value = Decl(decl = "Input(b_2)"), cld = ArrayBuffer()),
    TreeNode(value = Decl(decl = "Output(y_3)"), cld = ArrayBuffer()),
    TreeNode(value = Decl(decl = "Output(y2_4)"), cld = ArrayBuffer()),
    TreeNode(
      value = If(cond = BoolConst(b = true)),
      cld = ArrayBuffer(
        TreeNode(
          value = Assign(assign = "Output(y_3) := Input(a_1) + Input(b_2)"),
          cld = ArrayBuffer()
        ),
        TreeNode(
          value = If(cond = BoolConst(b = true)),
          cld = ArrayBuffer(
            TreeNode(
              value = Assign(assign = "Output(y2_4) := Input(a_1) - Input(
                b_2)"),
              cld = ArrayBuffer()
            )
          )
        )
      )
    )
  )
)
```

The visualization of the AST code (Top is a placeholder root node) is:



The algorithm also utilize a stack based algorithm similar to the main implementation at Section 5.1.

A.3 Convert list of statements to AST

Below is the stack based algorithm to list of statements to AST:

```
type AST = TreeNode[NewInstStmt | FirStmt | Ctrl | VarDecls]

/** convert sequential commands to AST. multiple stmt is appended as
   multiple nodes
  */
def list2tree(cmdList: List[Cmds]): AST = {
  import scala.collection.mutable.Stack
  val parents: Stack[AST] = Stack(TreeNode(Ctrl.Top())) // default top
     level node

  cmdList.foreach { cmd =>
    cmd match {
      case Start(ctrl, uid) =>
        /* start of block: create new node and append as child of curr top
            parent node if exists.
       then push new node into parent stack as new top elem*/
        val newParNode: AST = TreeNode(ctrl) // new parent node
        // add this newParNode as child
```

```
          parents.top.cld += newParNode
          parents push newParNode
        case End(ctrl, uid) =>
          // end of block, pop out one parent
          parents.pop()
        // for other stmt,just append
        case stmt: (FirStmt | NewInstStmt | VarDecls) =>
          val newNd: AST = TreeNode(stmt)
          parents.top.cld += newNd
        case _ =>
      }
    }

    parents.pop()
  }
```

For example, the below module

```
class IfMod(using parent: GlobalInfo) extends UserModule {
    val a = newInput[16]("a")
    val b = newInput[16]("b")
    val y = newOutput[16]("y")
    val y2 = newOutput[16]("y2")

    y := a - b
    If(a === b) {
      y := a + b
      y2 := a - b

    }
  }
```

is first converted to the list of statements:

```
List(
  FirStmt(
    lhs = VarTyped(name = "IfMod_1_2.y_5", tp = Output),
    op = ":=",
    rhs = BinOp(
      a = VarTyped(name = "IfMod_1_2.a_3", tp = Input),
      b = VarTyped(name = "IfMod_1_2.b_4", tp = Input),
      nm = "-"
    ),
    prefix = ""
  ),
  Start(
    ctrl = If(
      cond = BinOp(
        a = VarTyped(name = "IfMod_1_2.a_3", tp = Input),
        b = VarTyped(name = "IfMod_1_2.b_4", tp = Input),
        nm = "=="
      )
    ),
    uid = 7
  ),
  FirStmt(
    lhs = VarTyped(name = "IfMod_1_2.y_5", tp = Output),
    op = ":=",
    rhs = BinOp(
      a = VarTyped(name = "IfMod_1_2.a_3", tp = Input),
      b = VarTyped(name = "IfMod_1_2.b_4", tp = Input),
      nm = "+"
    ),
    prefix = ""
  ),
  FirStmt(
    lhs = VarTyped(name = "IfMod_1_2.y2_6", tp = Output),
    op = ":=",
    rhs = BinOp(
      a = VarTyped(name = "IfMod_1_2.a_3", tp = Input),
      b = VarTyped(name = "IfMod_1_2.b_4", tp = Input),
      nm = "-"
    ),
    prefix = ""
  ),
  End(
    ctrl = If(
      cond = BinOp(
        a = VarTyped(name = "IfMod_1_2.a_3", tp = Input),
        b = VarTyped(name = "IfMod_1_2.b_4", tp = Input),
        nm = "=="
      )
    ),
    uid = 7
  )
)
```

Then, the algorithm *list2tree* as defined before is performed to convert it to AST below:

```
List(
  TreeNode(
    value = Top(),
    cld = ArrayBuffer(
      TreeNode(
        value = FirStmt(
          lhs = VarLit(name = "g_1"),
          op = ":=",
          rhs = BinOp(
            a = VarTyped(name = "io.a_3", tp = Input),
            b = VarTyped(name = "io.b_4", tp = Input),
            nm = "-"
          ),
          prefix = "node "
        ),
        cld = ArrayBuffer()
      ),
      TreeNode(
        value = FirStmt(
          lhs = VarTyped(name = "io.y_5", tp = Output),
          op = "<=",
          rhs = VarLit(name = "g_1"),
          prefix = ""
        ),
        cld = ArrayBuffer()
      ),
      TreeNode(
        value = FirStmt(
          lhs = VarLit(name = "g_2"),
          op = ":=",
          rhs = BinOp(
            a = VarTyped(name = "io.a_3", tp = Input),
            b = VarTyped(name = "io.b_4", tp = Input),
            nm = "=="
          ),
          prefix = "node "
        ),
        cld = ArrayBuffer()
      ),
      TreeNode(
        value = If(cond = VarLit(name = "g_2")),
        cld = ArrayBuffer(
          TreeNode(
            value = FirStmt(
              lhs = VarLit(name = "g_3"),
              op = ":=",
              rhs = BinOp(
                a = VarTyped(name = "io.a_3", tp = Input),
                b = VarTyped(name = "io.b_4", tp = Input),
                nm = "+"
              ),
              prefix = "node "
            ),
            cld = ArrayBuffer()
          ),
          TreeNode(
            value = FirStmt(
              lhs = VarTyped(name = "io.y_5", tp = Output),
              op = "<=",
              rhs = VarLit(name = "g_3"),
              prefix = ""
            ),
            cld = ArrayBuffer()
          ),
          TreeNode(
            value = FirStmt(
              lhs = VarLit(name = "g_4"),
              op = ":=",
              rhs = BinOp(
                a = VarTyped(name = "io.a_3", tp = Input),
                b = VarTyped(name = "io.b_4", tp = Input),
                nm = "-"
              ),
              prefix = "node "
            ),
            cld = ArrayBuffer()
          ),
          TreeNode(
            value = FirStmt(
              lhs = VarTyped(name = "io.y2_6", tp = Output),
              op = "<=",
              rhs = VarLit(name = "g_4"),
              prefix = ""
            ),
            cld = ArrayBuffer()
          )
        )
      )
    )
  )
```

```
)
```

A.4 Algorithm for bit width checking

```scala
object typeCheck {

  /** return if width check is ok,or the width of expr */
  private def getExprWidth(
      typeMap: mutable.Map[Expr[?] | Var[?], Int],
      expr: Expr[?]
  ): Int = {
    val tm = typeMap
    expr match {
      case BinOp(a, b, nm) =>
        val (i, j) = (getExprWidth(typeMap, a), getExprWidth(typeMap, b))
        val isWidthEqu = i == j
        assert(isWidthEqu, s"getExprWidth: Width mismatch $a $nm $b ")
        i
      case UniOp(a, nm) => tm(a)
      case Lit(i)           => i
      case LitDym(i, width) => width
      case x =>
        throwE(tm(x), "can't get type for " + x)
    }
  }

  def checkCmdWidth(
      typeMap: mutable.Map[Expr[?], Int],
      cmds: AtomicCmds
  ) = {
    cmds match {
      /* 1.add width field in FirStmt
        2. add width in lhs var and rhs expr
        3. use a map to store width of var and expr */
      case FirStmt(lhs, op, rhs, prefix) =>
        val lr = (getExprWidth(typeMap, lhs), getExprWidth(typeMap, rhs))
            match {
          // only check if both result are numbers
          case lrWidth @ (i, j) =>
            val isWidthEqu = i == j
            //val lhsGeqRhs = i >= j // firrtl allows width of lhs >= rhs in
                lhs:=rhs
            val isWidthOk = isWidthEqu // | lhsGeqRhs
            val msg =
              s"width mismatch in \n lhs: ${(lhs.getname, i.toString().toRed
                 ())}\n " +
                s"op: $op \n" +
                s" rhs: ${(rhs, j.toString().toRed())} "

            // assert(isWidthOk, msg)
            if (!isWidthOk) {
              println(msg)
            }
            isWidthOk
        }
        lr

      case x => true
    }
  }

}
```

References

[1] Schoeberl, Martin. (2019). Digital Design with Chisel. Kindle Direct Publishing. Available at: https://github.com/schoeberl/chisel-book

[2] https://www.chisel-lang.org/

[3] A. Izraelevitz et al., "Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations," 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), Irvine, CA, USA, 2017, pp. 209-216, doi: 10.1109/ICCAD.2017.8203780.

[4] https://github.com/DFiantHDL/DFiant/

[5] I. Sutherland, The tyranny of the clock, Comm. ACM, vol. 55, no. 10, pp. 3536, 2012.

[6] Armstrong, Alasdair, et al. "Formalisation of MiniSail in the Isabelle theorem prover." Proceedings of the Automated Reasoning Workshop. 2018.

[7] https://spinalhdl.github.io/

[8] https://www.cerc.utexas.edu/ jaa/verification/

[9] https://clash-lang.org/

[10] Gajski, Daniel D., et al. HighLevel Synthesis: Introduction to Chip and System Design. Springer Science Business Media, 2012.

[11] D.D. Gajski and R.H. Kuhn. Guest Editors Introduction: New VLSI Tools. IEEE Computer, 1983.

[12] Nikhil, Rishiyur. "Bluespec System Verilog: efficient, correct RTL from high level specifications." Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE'04.. IEEE, 2004.

[13] Panda, Preeti Ranjan. "SystemC: a modeling platform supporting multiple design abstractions." Proceedings of the 14th international symposium on Systems synthesis. 2001.

[14] https://spinalhdl.github.io/SpinalDoc-RTD/dev/SpinalHDL/Data%20types/AFix.html

[15] Huibiao Zhu, Jifeng He and J. Bowen, "From algebraic semantics to denotational semantics for Verilog," 11th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'06), Stanford, CA, USA, 2006, pp. 13 pp.-, doi: 10.1109/ICECCS.2006.1690363.

[16] M. Gordon, "The semantic challenge of Verilog HDL," Proceedings of Tenth Annual IEEE Symposium on Logic in Computer Science, San Deigo, CA, USA, 1995, pp. 136-145, doi: 10.1109/LICS.1995.523251.

[17] https://prev.rust-lang.org/en-US/faq.html

[18] https://doc.rust-lang.org/beta/embedded-book/static-guarantees/zero-cost-abstractions.html

[19] https://github.com/lampepfl/dotty-feature-requests/issues/354

[20] Nicolas Stucki, Jonathan Immanuel Brachthäuser, and Martin Odersky. 2021. Multi-stage programming with generative and analytical macros. In Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 2021). Association for Computing Machinery, New York, NY, USA, 110122. https://doi.org/10.1145/3486609.3487203

[21] Martin Berger, Laurence Tratt, Christian Urban (2017). Modelling Homogeneous Generative Meta-Programming. In 31st European Conference on Object-Oriented Programming (ECOOP 2017) (pp. 5:15:23). Schloss DagstuhlLeibniz-Zentrum fuer Informatik.

[22] Semantics with Applications: An Appetizer

[23] Type-Driven Development with Idris

[24] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, 2013. Available at: https://homotopytypetheory.org/book.

[25] Sail. http://www.cl.cam.ac.uk/ pes20/ sail/.

[26] https://scalameta.org/metals/

[27] Oleg Kiselyov and Hiromi Ishii. 2015. Freer monads, more extensible effects. In Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell (Haskell '15). Association for Computing Machinery, New York, NY, USA, 94105. https://doi.org/10.1145/2804302.2804319

[28] Eugenio Moggi. 1991. Notions of computation and monads. Inf. Comput. 93, 1 (July 1991), 5592. https://doi.org/10.1016/0890-5401(91)90052-4

[29] https://github.com/Iltotore/iron

[30] https://docs.scala-lang.org/scala3/reference/metaprogramming/compiletime-ops.html

[31] https://github.com/pocorall/scaloid/

[32] https://github.com/doofin/dependentChisel

[33] https://github.com/doofin/dependentChisel/blob/master/src/main/scala/dependentChisel/algo/seqCmd2tree.scala

[34] https://github.com/doofin/dependentChisel/blob/master/src/main/scala/dependentChisel/codegen/typeCheck.scala

[35] https://github.com/doofin/dependentChisel/blob/master/src/main/scala/dependentChisel/codegen/compiler.scala

[36] https://github.com/doofin/dependentChisel/blob/master/src/main/scala/dependentChisel/monadic/monadicTest.scala