Clash Documentation

Release 1.5.0

The Clash Developers

Jul 27, 2021

CONTENTS

Welcome to the Clash Language User Guide, the official documentation of the Clash Compiler. Clash is an open-source functional hardware description language (HDL) that borrows syntax and semantics from the Haskell programming language. To learn more, we suggest reading the *introduction to Clash* (page 3).

The table of contents below (and in the sidebar) allows easy access to different pages in the documentation. You can also use the search function in the top left corner.

Note: The Clash Compiler and Clash Language User Guide are open-source efforts developed by QBayLogic B.V. and other volunteers. The Clash Team always appreciates feedback and contributions to the project to help improve the development experience.

If you don't understand something, or think something is missing or incorrect in the documentation you can open an issue or pull request in the GitHub repository.

CHAPTER

ONE

GENERAL

1.1 Introduction to Clash

1.1.1 Functional Hardware

Clash is an open-source functional hardware description language (HDL) that closely mirrors the syntax and semantics of the Haskell programming language. It is used for creating hardware designs, typically for running on *field programmable gate arrays* (FPGAs) or *application-specific integrated circuits* (ASICs).

Clash is both a compiler, and a set of libraries for circuit design, that transform high level Haskell descriptions of synchronous, sequential logic into low-level VHDL, Verilog, or SystemVerilog. It provides a unique approach to design of sequential circuits, but with a high amount of abstraction power that blurs the line between strictly behavioral or structural synthesis approaches.

Clash aims to modernize the hardware development experience, making it easier to quickly and correctly develop complex circuit designs. This is achieved by making Clash:

Expressive Clash uses the Haskell type system to its full potential – including modern extensions and techniques – to being a high level of type safety and expressiveness to hardware design.

This expressive typing makes it easier to develop safe, maintainable hardware. Combinatorial and sequential logic is separated by type, and global safety invariants such as separating incompatible clock domains are enforced in the type system.

Intuitive Clash makes it easy to express circuit designs in an intuitive manner, allowing high level structural components to be easily connected in designs. Moreover, unlike most "high level synthesis" tools, this extends to precise control over register placement and pipelining.

Interactive Unlike traditional HDL tools, Clash has a fully interactive read-eval-print loop (REPL), allowing circuits to be interactively designed and tested.

Performant Clash reuses parts of the Glasgow Haskell Compiler to provide fast simulation of circuits for development and testing.

Efficient Clash uses a "whole program synthesis" approach in order to view the entire circuit at once, and optimizes this design before translating to a specific target. This allows meaningful optimizations to be performed on the entire design.

Extensible Additional primitives and black boxes can be added to Clash in the language of your choice, allowing you to use your own vendor or IP library within projects.

Clash allows seamless interoperability with libraries written in Haskell, including `mtl`, `lens` and `QuickCheck`. This makes it even easier to quickly prototype complex designs.

1.1.2 Intended Audience

Clash is ideal for developers from different backgrounds, although the main intended audiences are

Hardware Engineers You are a hardware engineer, used to using tools like VHDL and Verilog to implement circuit designs. Clash offers the familar mixed simulation / synthesis capabilities of these tools, while providing a langauge with powerful abstractions.

Haskell Programmers You are a Haskell programmer, looking to start developing hardware. Clash offers the ability to start prototyping and simulating designs in a familiar environment – lowering the learning curve significantly.

1.1.3 Maturity and Support

Clash is a continually evolving tool, having been actively developed since 2009. With the release of Clash 1.0 there has been an increased focus on maintaining API stability between releases, meaning circuit designs written in Clash should continue to work between minor releases. Today, the Clash Compiler is actively developed by QBayLogic B.V. and volunteers.

Several companies and enthusiasts are already using Clash to develop circuit designs, ranging from small designs on hobbyist boards to larger designs on modern FPGA and ASIC architectures.

While care is taken to thoroughly test the Clash compiler, some bugs may exist. We encourage users to file issues, or contribute pull requests on our GitHub repository.

1.1.4 Meta-information: Web Sites, Mailing Lists, etc.

Mailing list: for updates and questions join the mailing list clash-language+subscribe@googlegroups.com or read the forum

Slack: Invite yourself at fpchat-invite.herokuapp.com. To join #clash, click on "Channels" and search for "clash".

IRC: freenode#clash-lang

1.1.5 Clash Version Numbering Policy

Clash follows the Haskell PVP Specification for its version numbers, for all packages. The main libraries that make up the Clash compiler maintain the same version numbers, making it easy to identify which versions are compatible.

Note: Due to the Clash's tight integration with GHC, updates to the GHC version that Clash uses result in changes to the Clash version. As GHC's internals change frequently, even for minor bumps, it cannot be guaranteed that these changes will not result in Clash changes.

It is recommended (but not required) that downstream Clash packages and published Clash code also follow the PVP specification.

1.2 Release Notes

1.2.1 Clash 1.0.1

https://github.com/clash-lang/clash-compiler/releases/tag/v1.0.1

1.2.2 Clash 1.0.0

https://github.com/clash-lang/clash-compiler/releases/tag/v1.0.0

1.3 Frequently Asked Questions

1.3.1 Basic Questions

- **Q**: How do I install Clash?

  A: Check out the *Installing Clash* (page 9) page in the *Getting Started* section of the manual.

- **Q**: Is the name "Clash", "CLaSH", or "CλaSH"?

  A: It's **Clash**.

  In its research stages Clash was called "CλaSH", an acronym for the **C**AES **La**nguage for **S**ynchronous **H**ardware. CAES is a group of the Faculty of Electrical Engineering, Mathematics and Computer Science at the University of Twente. Clash was originally developed by Christaan Baaij and supervisor Jan Kuper. The stylization "CλaSH" is an homage to Haskell, whose official logo has long been the venerable Greek *lambda* character.

- **Q**: Is Clash a "high level synthesis" tool?

  A: While clash provides a high level language features, hardware descriptions written in Clash are not decoupled from clock-level timing. Clash does therefore not offer what is generally understood as "high level synthesis". Compared to the big three hardware description languages, *VHDL*, *Verilog*, and *SystemVerilog*, Clash arguably *is* high-level. It offers many of the powerful abstractions that modern software programming languages offer. In fact, it inherits many of the software's industry bleeding-edge features by virtue of basing its implemenation on Haskell.

1.3.2 Clash Support

- **Q**: Is Clash production ready?

  A: Clash is constantly evolving, and since the 1.0 release there is a focus on maintaining API backwards compatibility. Clash is used successfully in real-world scenarios, and QBayLogic Clash support can help with education and implementation of Clash projects.

- **Q**: Will Clash work with my EDA tools?

  A: In general, Clash should work well with Xilinx and Intel FPAGs and their EDA tools – as development typically focuses on these vendors. Clash has also been successfully used on Microsemi (formerly Actel) SmartFusion 2 and Lattice Semiconductor iCE40 FPGAs, and some basic IP for these exist.

  For most toolchains, the default primitives supplied by Clash should work with minimal effort. If not, it is possible to call your vendor's library manually, or use a tool like Yosys to do mapping. It is also possible to consult QBayLogic Clash support for more assistance.

- **Q**: Does Clash support Project IceStorm?

  A: The Verilog backend for Clash emits Verilog 2001, which is supported by Yosys. This means it can be placed and packed with *arachne-pnr* and *icestorm*. Additionally, Clash has some support for the Lattice Semiconductor iCE40 FPGA.

- **Q**: Can Clash be used for ASIC designs, as well as FPGA designs?

  Clash can be used for ASIC designs, however the RTL produced by Clash may not be immediately suitable as it is largely platform agnostic. While this is not a problem for FPGAs, it can make developing ASICs more complicated as many ASIC vendors have different proprietary tool flows, with limited information available about their workings.

  If you are using Clash to develop for ASIC, and need assitance with getting your toolchain to work, you can contact QBayLogic Clash support for assistance.

1.3.3 Clash and Haskell

- **Q**: Is Clash its own programming language, or is it "Haskell"?

  A: Clash is a programming language in its own right, complete with its own executable and standard library. Clash is also related to the Haskell programming language, and may be thought of as a dialect of Haskell for developing hardware. While the surface syntax and typing rules are the same, the semantics change as code progresses through the compilation pipeline.

  Do to the shared behavior in the early stages of the compiler, components from GHC (the most common Haskell compiler) are reused in the Clash compiler. This is how Clash achieves such high interoperability with existing Haskell projects.

- **Q**: Clash has better inference for type level natural numbers than GHC. How is this possible?

  A: Clash's enhanced type checking functionality is due to the use of GHC compiler plugins, which can be used in any Haskell project. To enable these plugins, pass the following compiler flags to GHC:

```
{-# OPTIONS_GHC -fplugin GHC.TypeLits.Normalise       #-}
{-# OPTIONS_GHC -fplugin GHC.TypeLits.Extra.Solver    #-}
{-# OPTIONS_GHC -fplugin GHC.TypeLits.KnownNat.Solver #-}
```

  These plugins come from the `ghc-typelits-natnormalise`, `ghc-typelits-extra`, and `ghc-typelits-knownnat` packages respectively, which are all available from Hackage and Stackage.

- **Q**: Do I need to know Haskell in order to use Clash?

  A: As Clash is deeply integrated with Haskell, it is recommended that users have some familiarity with Haskell, or functional programming in general. Clash uses some advanced features of Haskell, and real-world designs will often want to leverage the existing Haskell ecosystem.

  For developers who are particularly familiar with either Haskell or hardware design, Clash should be relatively intuitive to use. Additionally, obvious mistakes with designs will be identified and reported due to the strong type system identifying mistakes at compile-time.

1.3.4 Clash and other HDLs

- **Q**: Do I need to know existing RTL/HDL languages in order to use Clash?

  A: Clash currently outputs VHDL, Verilog, and SystemVerilog. While it's not necessary to understand these descriptions, you will need to some understanding of vendor tools to actually deploy it.

- **Q**: What's the difference between Clash and "Lava"?

  A: Lava dialects (including the modern variant Blarney) are all embedded domain specific languages (EDSLs) inside Haskell. On top of that they use a so-called *deep* embedding to be able to transform a circuit description into a netlist (to subsequently output that as a VHDL/Verilog file). Clash on the other hand uses "standard" compiler techniques to create a netlist from the Haskell abstract syntax tree (AST). This "standard" compiler technique enables the following features not available in (Haskell-based) EDSLs:

  1. Clash allows the use of normal Haskell operations such as (==) on both the meta-level (how the program is structured/generated), and the object-level (the functionality of the program).
  2. Clash allows the use of regular Haskell syntax to model the concept of 'choice' at the object-level (the functionality of the program): if-expressions, guards, case, etc.
  3. Clash allows programmers to use native Haskell pattern matching.

Basically, with Clash you can use regular Haskell to describe the behavior of the circuit, most importantly all of it's choice-constructs (case-expressions, guards, etc.). With an EDSL you are "limited" by the constructs of the DSL, making your circuit descriptions look less like regular Haskell functions.

- **Q**: What's the difference between Clash and Chisel/Spinal/Migen/Hardcaml?

  A: The biggest difference between these toolchains and Clash is that Clash exists as a Haskell derivative, with a full synthesizing compiler to RTL – while Chisel/Spinal/Migen/Hardcaml exists as an embedding of hardware semantics inside Scala/Scala/Python/OCaml. Aside from the "host language" differences, this means that Chisel/Spinal/Migen/Hardcaml are conceptually closer to something like *Lava/Blarney* than Clash. So within these languages you can only use the host language constructs to structure and compose the constructs of the EDSL, and you can't use host language constructs to describe the behavior of the circuit; i.e. you cannot use the host language's regular if-expression to model the concept of choice, but you have to use e.g. Chisel's *when*-function.

  Aside from the above, there is also a varying degree of *native* simulation and interactivity. In Clash you can evaluate/simulate any (sub-)component in the interactive interpreter for an immediate and localized design feedback loop. The only EDSLs that have a similar interactive interpreter for fast design feedback are the older variants of Lava. They used a so-called dual-embedding, where the EDSL primitives also contained a normal Haskell function which described their behavior, and so the composition of these primitives could be evaluated as a regular Haskell function.

  The other EDSLs all offer simulation, but there is a higher latency to get from a design to a simulation of a design, and they are not as interactive. Blarney emits Verilog, and you can then use a Verilog simulator to simulate the Blarney design. Spinal also emits Verilog, but it then uses Verilator to compile it to an object-file which is loaded back into Scala, allowing you to interact with your Spinal design from within Scala. Chisel is also not interpreted directly, instead, a Chisel description is "lowered" to FIRRTL where that FIRRTL description is then executed inside Scala by the FIRRTL interpreter. Migen works similarly to Chisel as far as the approach to simulation goes, although perhaps more direct: it directly interprets its own deep embedding data structure (its *IR*) to enable native simulation.

  All of this influences the style in which you write circuits and the creative process by which you come to a solution; the effects of this on the quality of results (QoR) and development time are, however, both hard to qualify and hard to quantify. That is, although all of these languages, both the EDSLs and Clash, enable full control over the QoR (i.e. you can get as many registers and as much logic as you intended), the way in which you get there can vary from problem domain to problem domain and person to person. If you have enough time, we encourage to try several of them and see which style is the most natural fit for you; if you're limited on time, we of course recommend that you just go with Clash ;-)

1.4 License

CHAPTER

TWO

GETTING STARTED

2.1 Installing Clash

Check out clash-lang.org/install to install the latest stable release of Clash, or to setup a Clash project.

2.1.1 Get Clash from source

Get the source code using Git and enter the cloned directory:

```
git clone git@github.com:clash-lang/clash-compiler.git

# Alternatively, if you haven't setup SSH keys with GitHub:
# git clone https://github.com/clash-lang/clash-compiler.git

cd clash-compiler
```

To check out a released version, use:

```
git checkout v1.2.3
```

To checkout a release *branch* use:

```
git checkout 1.2
```

Note that release branches might contain non-released patches.

Cabal

To use Cabal you need both Cabal and GHC installed on your system. For Linux and MacOS users we recommend using ghcup. Windows users are recommended to use the Haskell Platform.

To run *clash* use:

```
cabal v2-run --write-ghc-environment-files=always -- clash
```

If this fails, make sure you've got an up-to-date package index:

```
cabal update
```

Stack

Install Stack and run:

```
stack run -- clash
```

Nix

Or use Nix to get a shell with the `clash` and `clashi` binaries on your PATH:

```
nix-shell
```

2.2 Example: Multiply and Accumulate

2.2.1 Combinatorial MAC

With Clash installed, it is now possible to begin creating hardware designs. To give a brief overview of Clash, we will define a simple *multiply-and-accumulate* circuit. Make a new file called `MAC.hs`, and enter the following preamble:

```
module MAC where

import Clash.Prelude
import Clash.Explicit.Testbench
```

This declares the module and imports some useful modules from the Clash standard library. The standard library contains necessary functions and data types for writing circuit descriptions. As with Haskell, module identifiers in Clash must always start with a capital letter and correspond to the name of the file.

The logic of our circuit is expressed as a function which takes an accumulator and two extra inputs, and outputs the new value of the accumulator – which is the old value plus the product of the two other inputs.

```
mac :: (Num a) => a -> (a, a) -> a
mac acc (x, y) = acc + x * y
```

The type of the function is given after the `::`, and says that the type `a` is some numeric type (e.g. `Int`, `Signed 8`, `Double`), the first argument is a number, the second value is a pair of numbers, and the result is a number.

2.2.2 Synchronous MAC

By adding another output parameter to this function, with the previous value of the accumulator, we can define the function as a Mealy machine. This allows us to use our combinatorial definition of `mac` to create a synchronous circuit (which we call `macS`).

```
mac :: (Num a) => a -> (a, a) -> (a, a)
mac acc (x, y) = (acc + x * y, acc)

macS :: (HiddenClockResetEnable dom, Num a, NFDataX a) => Signal dom (a, a) ->␣
↪Signal dom a
macS = mealy mac 0
```

The input and output of `macS` are values of the `Signal` type. This type represents synchronous values (functions without signals are combinatorial). There is also an additional `dom` type, for synthesis domain, and a constraint `HiddenClockResetEnable` – which says the synthesis domain has a clock, reset and enable line. These are implicit, although they can be exposed using the `exposeClockResetEnable` funcion.

2.2.3 HDL Generation and Testing

To generate HDL from a synchronous circuit, a function needs to be marked as a `topEntity`. The simplest way to achieve this is to create a function with this name, as Clash will use this definition automatically (similar to how `main` is a special function in other languages).

```
topEntity
  :: Clock System
  -> Reset System
  -> Enable System
  -> Signal System (Int, Int)
  -> Signal System Int
topEntity = exposeClockResetEnable macS
```

It is now possible to generate HDL for this circuit description, by running either `clash --HDL` from the command line, or running `:HDL` in `clashi` (where `HDL` is either `vhdl`, `verilog` or `systemverilog`). This will generate the HDL in a subdirectory named after the HDL being output.

Warning: Any function used to generate HDL from must have a monomorphic type. This means there can be no type variables in the type signature (i.e. for the circuit defined so far you need to specify both `dom` and `a`.

We can test that this circuit works as expected by defining a test bench. This allows an input to be used and the actual output to be compared against an expected output.

```
testBench :: Signal System Bool
testBench = done
 where
  testInput    = stimuliGenerator clk rst ((1,1) :> (2,2) :> (3,3) :> (4,4))
  expectOutput = outputVerifier' clk rst (0 :> 1 :> 5 :> 14 :> 30 :> 46 :> 62)
  done         = expectOutput (topEntity clk rst en testInput)
  en           = enableGen
  clk          = tbSystemClockGen (fmap not done)
  rst          = systemResetGen
```

From `clashi` it is possible to sample this test bench, using the `sampleN` function, which takes in the number of samples to draw and the signal which generates samples.

```
>>> sampleN 8 testBench
[False, False, False, False, False, False, False, False]
```

CHAPTER

THREE

DEVELOPING HARDWARE WITH CLASH

3.1 Clash as a Language

As Clash reuses parts of the GHC compiler for its front-end, the syntax and semantics should be familiar to Haskell programmers. For people unfamiliar with Haskell, there are many resources to learn the language, such as

- Learn You a Haskell
- Real World Haskell
- The Haskell Wikibook

Clash does make some use of more advances features of GHC Haskell, which are exposed by GHC as language extensions. The extensions used by Clash are

- `BinaryLiterals`
- `ConstraintKinds`
- `DataKinds`
- `DeriveAnyClass`
- `DeriveGeneric`
- `DeriveLift`
- `DerivingStrategies`
- `ExplicitForAll`
- `ExplicitNamespaces`
- `FlexibleContexts`
- `FlexibleInstances`
- `KindSignatures`
- `MagicHash`
- `MonoLocalBinds`
- `NumericUnderscores`
- `NoImplicitPrelude`
- `NoMonomorphismRestriction`
- `NoStarIsType` (since GHC 8.6)
- `NoStrictData`
- `NoStrict`
- `QuasiQuotes`
- `ScopedTypeVariables`

- `TemplateHaskellQuotes`
- `TemplateHaskell`
- `TypeApplications`
- `TypeFamilies`
- `TypeInType` (until GHC 8.6)
- `TypeOperators`

Warning: Since GHC 8.6, `TypeInType` has been an alias for the `DataKinds` and `PolyKinds` language extensions and is not a default extension in Clash. If an existing design breaks on upgrade, try enabling `PolyKinds`.

Warning: Since GHC 8.6, the `StarIsType` extension is defined. This extension is explicitly turned off by Clash, meaning `Data.Kind.Type` must be used to refer to Haskell types.

Clash also enables some GHC plugins by default which improve the type inference for type level numbers. The plugins enabled by default are

- `ghc-typelits-extra`
- `ghc-typelits-knownnat`
- `ghc-typelits-natnormalise`

Users are free to control the language extensions and GHC options with the normal `OPTIONS_GHC` and `LANGUAGE` pragmas in source files. For more information, see the GHC User's Guide.

3.2 Clash Prelude

3.2.1 Basic Types

The Clash prelude includes many different numeric types, which are used to safely define other types / functions. These include, but may not be limited to

- Type level natural numbers (`Nat`), which allow numbers to be used in types. Conceptually, this is similar to *const generics* in *C++*.

  It is possible to have term level values which refer to a type level number. This is called `SNat n` (for *singleton natural number*). These are defined up to 1024 with the prefix "d" (e.g. `d256`).

- `Unsigned n` and `Signed n` numbers with an arbitrary width (given as a type level natural number). These allow fixed-width arithmetic to be used on arbitrary numbers.
- `Index n` provides natural numbers up to an arbitrary value (given as a type level natural number). These allow indexing into fixed width structures like `Vec n a`.

Another commonly used type is `BitVector n`. This provides a fixed size vector of `Bit` values which can be indexed, and used to perform *unsigned integer arithmetic*. Any type that can be marshalled to / from a `BitVector n` implements the `BitPack` class, which defines the conversion.

Note: It is also possible to derive instances of `BitPack` using `Generic`, by writing `deriving (Generic, BitPack)` in the type definition. This automatically determines how to do the conversion at compile-time.

More generally, there is a `Vec n a` type which allows collections of arbitrary values to be used. These vectors are tagged with their length, to prevent out of bounds access at compile-time.

Warning: The `Vec n a` type exports pattern synonyms for inserting at the left and right of a vector. The types of the `Cons` constructor and `(:>)` pattern are slightly different, and may behave differently in practice.

The `Cons` constructor has a more general type, allowing it to be used in some cases where the pattern cannot be used. However, this additional power comes at the cost of type inference. It is recommended that users use the `(:>)` pattern by default, and only use `Cons` when necessary.

3.2.2 Synthesis Domains

Synchronous circuits have a synthesis domain, which determines the behaviour of things which can affect signals in the domain. Domains consist of

- a name, which uniquely refers to the domain
- the clock period in ps
- the active edge of the clock
- whether resets are synchronous (edge-sensitive) or not
- whether the initial (power up) behaviour is defined
- whether resets are high or low polarity

The prelude provides some common domains, namely `XilinxSystem` and `IntelSystem` for the standard configurations of each vendor. There is also a generic domain, `System`, which can be used for vendor-agnostic purposes (i.e. writing a generic test bench). It is possible to define new synthesis domains for custom hardware using the `createDomain` function, which also defines the necessary instances for domains.

A value in a synchronous circuit is wrapped in the `Signal dom a` type, which specifies the synthesis domain and the type of value. Any function which needs access to a domain can use the constraints `HasDomain` (to find it's domain) or `KnownDomain` (to extract configuration).

The default API exposed by the prelude is implicit with regards to clocks, reset lines and enable lines – as these can be determined at compile time. However, if they are needed the `Clash.Explicit` module contains explicit versions of the API which expose these directly in function arguments. It is also possible to use functions like `exposeClockResetEnable` to turn an implicitly defined function to an explicitly defined function.

3.2.3 State Machines

The Clash prelude contains combinators for two classical finite state machines which can be used to define synchronous circuits. The first of these is `mealy`, which encodes a Mealy machine. This is a machine specified by

- A transfer function of type `state -> input -> (state, output)`
- An initial state
- An input signal which can change at each cycle

Note: The Mealy machine is similar to the State monad, which Haskell programmers may already be familar with. Practically speaking, the only difference is that this machine also has an input signal which is changed externally to the definition of the machine.

It is also possible to define a Moore machine using the `moore` function in the Clash prelude. This differs to the Mealy machine by providing output based on the previous state (as oppoesd to the newly calculated state), and is specified by

- A transfer function of type `state -> input -> state`
- An output function of type `state -> output`
- An initial state

- An input signal which can change at each cycle

Sometimes, there may be multiple inputs / outputs needed for a machine. As machines only input and output a single signal, there is a way to combine and separate multiple signals. The `Bundle` class specifies how to convert between some type which is a signal of a product, and some type which is a product of signals, e.g.

```
bundle   :: (Signal dom a, Signal dom b) -> Signal dom (a, b)
unbundle :: Signal dom (a, b) -> (Signal dom a, Signal dom b)
```

There are combinators which can automatically perform this bundling and unbundling for you as required, called `mealyB` and `mooreB`. The `Bundle` class is already defined for many types, including tuples (up to 62 elements), `Maybe a`, `Either a b` and `Vec n a`.

3.2.4 RAM and ROM

The Clash prelude provides the ability to work with synchronous and asynchronous ROM, asynchronous RAM and synchronous Block RAM. The simplest of these are ROM, which only allow indexing into a `Vec n a` of elements. ROM is defined using the functions in `Clash.Prelude.ROM`.

RAM is more complex, as it allows both reading and writing. The function to define a RAM takes in a signal for the address to read, and a signal for an optional address to update (bundled with the new value). At each cycle it outputs the value of the memory address read in the previous cycle. Asynchronous RAM is defined in `Clash.Prelude.RAM`.

An FPGA may include a block RAM, which is a larger memory structure and more suitable for some applications. Block RAM also has a synchronous read port, allowing memory access to be synchronized to a clock. Block RAM is used the same way as async RAM, allowing the two to be compared quickly. Block RAM is defined in `Clash.Prelude.BlockRam`.

3.2.5 Undefined Values

When working with hardware designs, there are times when undefined values may be encountered in simulation. Clash provides a custom exception type, `XException`, for cases when an undefined value is encountered. There are also many utility functions for working with exceptions, such as

- `errorX`, which throws an `XException`
- `isX` and `hasX`, which check for `XExceptions` when evaluating
- `maybeIsX` and `maybeHasX`, which discard inforamtion about exceptions

There are also implementations of typical classes in Haskell which have been changed to work with undefined values. Currently these are

- `ShowX`, which works like the `Show` class in Haskell. When an undefined value is encountered an "X" is printed. `Show` can still be used, but will throw an exception if an undefined value is encountered.
- `NFDataX`, which works like the `NFData` class in the `deepseq` library. This allows evaluating values to normal form in code when undefined may be present. `NFData` can still be used, but will bubble up exceptions if undefined is encountered.

3.3 Clash Compiler Flags

--vhdl Use the VHDL backend for code generation. This currently emits VHDL 1993 source which can be consumed by other tools.

--verilog Use the Verilog backend for code generation. This currently emits Verilog 2001 source which can be consumed by other tools.

--systemverilog Use the SystemVerilog backend for code generation. This currently emits SystemVerilog 2012 source which can be consumed by other tools.

-fclash-debug Set the debugging mode for the compiler, exposing additional output. The available options are

- `DebugNone` to show no debug messages
- `DebugSilent` to test invariants and error if any are violated. This is implicitly enabled by any debug flag
- `DebugFinal` to show expressions after they have been completely normalized
- `DebugCount` to count how often each transformation is applied
- `DebugName` to show the names of transformations as they are applied
- `DebugTry` to show names of tried and applied transformations
- `DebugApplied` to show sub-expressions after they are rewritten
- `DebugAll` to show all sub-expressions when a rewrite is attempted

Default: `DebugNone`

Note: This flag exists for backwards compatibility. It is now possible to set debugging flags individually with *-fclash-debug-invariants*, *-fclash-debug-info* and *-fclash-debug-count-transformations*.

-fclash-debug-invariants Check invariants while debugging and print warnings / errors which may be useful, such as alterting when unexpected changes occur or when a transformation introduces free variables / shadowing.

-fclash-debug-info Specify the information to show about individual transformations while debugging. From least to most information, these are

- `None` to show no information
- `FinalTerm` to show the final result of normalization
- `AppliedName` to show the names of applied transformations
- `AppliedTerm` to show the result of applied transformations
- `TryName` to show the names of attempted transforamtions, as well as the result of any transformations which are applied
- `TryTerm` to show the names and results of all transformations attempted whether they were applied or not

  Default: `None`

-fclash-debug-count-transformations Count the transformations that are applied and print a summary at the end of the normalization phase.

-fclash-debug-history[=FILENAME] Saves all applied rewrites into `FILENAME`, for later analysis with the clash-term tool. When no filename is given it defaults to `history.dat`.

-fclash-debug-transformations List the transformations that are to be debugged. This is given as a comma-separated list of transformations, e.g.

```
clash -fclash-debug-transformations inlineNonRep,topLet,
↪appProp
```

Default: []

-fclash-debug-transformations-from=N Only print debug output from applied transformation `N` and onwards.

```
clash -fclash-debug-transformations-from=21570
```

Default: 0

-fclash-debug-transformations-limit=N Only print debug output for `N` applied transformations.

```
clash -fclash-debug-transformations-limit=12
```

Default: MAX_INT

-fclash-hdldir Specify the directory that generated HDL is written into. For example

```
clash -fclash-hdldir build/hdl
```

will create a directory `build/hdl`

Default: Either `vhdl`, `verilog`, or `systemverilog` depending on the synthesis target.

-fclash-hdlsyn Specify the HDL synthesis tool which will be used. Available options are `Vivado`, `Quartus` and `Other`, but some synonyms for these exist (`Xilinx` and `ISE` are synonyms for `Vivado`, `Altera` and `Intel` are synyonyms for `Quartus`).

Default: `Other`

-fclash-no-cache Don't reuse previously generated output from Clash, instead generating HDL from a clean state. While this leads to longer builds, it can be useful in development.

> **Warning:** Previously this flag was called `-fclash-nocache`, however this is now deprecated.

Default: Cache generated HDL

-fclash-no-check-inaccessible-idirs Check that all include directories (containing primitives) exist when running Clash. If any directory does not exist, an error is thrown.

Default: Check directories

-fclash-clear Remove HDL directories before writing to them (if cache can't be used). By default, Clash will only write to non-empty directories if it can prove all files in it are generated by a previous run. This option applies to directories of the various top entities, i.e., the subdirectories made in the directory passed in with `-fclash-hdldir`.

Default: Clean before build

-fclash-no-prim-warn Disable warnings for primitives that are annotated with `warnAlways`. This means warnings from annotations like

```
{-# ANN f (warnAlways "This primitive is dangerous") #-}
```

will not be shown when compiling.

Default: Show warnings

-fclash-spec-limit Change the number of times a function can undergo specialization.

Default: 20

-fclash-inline-limit Change the number of times a function `f` can undergo inlining inside some other function `g`. This prevents the size of `g` growing dramatically.

Default: 20

-fclash-inline-function-limit Set the threshold for function size. Below this threshold functions are always inlined (if it is not recursive).

Default: 15

-fclash-inline-constant-limit Set the threshold for constant size. Below this threshold constants are always inlined. A value of 0 inlines all constants.

Default: 0

-fclash-evaluator-fuel-limit Set the threshold for unfolding potentially non-terminating bindings in the evaluator. A value of 0 only unfolds terminating bindings.

Default: 20

-fclash-intwidth Set the bit width for the `Int`/`Word`/`Integer` types. The only allowed values are 32 or 64.

Default: Machine word size (`WORD_SIZE_IN_BITS`)

-fclash-error-extra Print additional information with compiler errors if it as available. If there is extra information and this flag is not enabled, a message will be printed suggesting this flag.

Default: False

-fclash-float-support Enable support for floating point numbers. If this is disabled, Clash will not attempt to convert Float and Double values for hardware.

Default: False

-fclash-component-prefix Prefix the names of generated HDl components with a string. For example a component `foo` would be called `xcorp_foo` if run with

```
clash -fclash-component-prefix "xcorp"
```

Default: “”

-fclash-old-inline-strategy The new inlining strategy for Clash inlines all functions which are not marked with `NOINLINE` or a synthesize attribute. The old inlining strategy differed, attempting only to inline functions which were deemed “cheap”. The old inlining strategy may be quicker in practice for some circuits.

Default: False

-fclash-no-escaped-identifiers Disable extended identifiers, as used in some HDLs like VHDL to allow more flexibility with names. Clash will only generate basic identifiers if this is used.

Default: Escaped identifiers are allowed

-fclash-lower-case-basic-identifiers Clash will only generate lower case basic identifiers if this is used. This affects places where the various HDLs only allow basic identifiers to be used, most notably module and file names.

Default: Disabled

-fclash-compile-ultra Aggressively run the normalizer, potentially gaining much better runtime performance at the expense of compile time.

Default: False

-fclash-force-undefined{,0,1} Set the value to use when an undefined value is inserted into generated HDL. This flag can be suffixed with either 0 or 1 to force use of that bit, or left without a suffix to use a HDL-specific default (e.g. `x` in Verilog).

Default: Disabled

-fclash-aggressive-x-optimization Remove all undefined branches from case expressions, replacing them with another defined value in the expression. If only one branch is defined, the case expression is elided completely. If no branches are defined the entire expression is replaced with a call to `errorX`.

Implies: `-fclash-aggressive-x-optimization-blackboxes`

Default: False

-fclash-aggressive-x-optimization-blackboxes Allow blackboxes to detect undefined values and change their behavior accordingly. For example, if `register` is used in combination with an undefined reset value, it will leave out the reset logic entirely. This flag is enabled when using `-fclash-aggressive-x-optimization`.

Default: False

-fclash-edalize Generate metadata for use with Edalize. This generates edam.py files in all top entities with the configuration for building that entity. Users still need to edit this file to specify the EDA tool to use, and if necessary the device to target (for Quartus, Vivado etc.)

Default: False

-main-is When using one of `--vhdl`, `--verilog`, or `--systemverilog`, this flag refers to synthesis target. For example, running Clash with `clash My.Module -main-is top --vhdl` would synthesize `My.Module.top`.

CHAPTER

FOUR

HACKING ON CLASH

4.1 Clash/Haskell Style Guide

This is a short document describing the preferred coding style for this project. When something isn't covered by this guide you should stay consistent with the code in the other modules. The code style rules should be considered strong suggestions but shouldn't be dogmatically applied - if there's a good reason for breaking them *do it*. If you can't or don't want to apply a guideline or if a guideline is missing, consider:

- **How your style affects future changes.** Does changing part of it cause a lot of realignments? Is it easily extendable by copy-pasting lines?
- **Whether whitespace is effectively used.** Do new indent-blocks start 2 spaces deeper than the previous one? Is it easy to see which block is which?
- **How it scales.** Is the style applicable to small examples as well as large ones?

The guidelines formulated below try to balance the points above.

4.1.1 Formatting

Line Length

Try to keep below *80 characters* (soft), never exceed *90* (hard).

Indentation

Tabs are illegal. Use spaces for indenting. Indent your code blocks with *2 spaces*. Indent the `where` keyword two spaces to set it apart from the rest of the code and indent the definitions in a `where` clause 1 space. Some examples:

```
sayHello :: IO ()
sayHello = do
  name <- getLine
  putStrLn $ greeting name
 where
  greeting name = "Hello, " ++ name ++ "!"

filter
  :: (a -> Bool)
  -> [a]
  -> [a]
filter _ [] = []
filter p (x:xs)
  | p x       = x : filter p xs
  | otherwise = filter p xs
```

Blank Lines

One blank line between top-level definitions. No blank lines between type signatures and function definitions. Add one blank line between functions in a type class instance declaration if the function bodies are large. Use your judgement.

Whitespace

Surround binary operators with a single space on either side. Use your better judgement for the insertion of spaces around arithmetic operators but always be consistent about whitespace on either side of a binary operator. Don't insert a space after a lambda. Add a space after each comma in a tuple:

```
good = (a, b, c)
bad = (a,b,c)
```

Refuse the temptation to use the latter when almost hitting the line-length limit. Restructure your code or use multiline notation instead. An example of a multiline tuple declaration is:

```
goodMulti =
  ( a
  , b
  , c )

goodMulti2 =
  ( a
  , b
  , c
  )
```

Use nested tuples as such:

```
nested =
  ( ( a1
    , a2 )
  , b
  , c )
```

Similar to `goodMulti2`, you can put the trailing `)` on a new line. Use your judgement.

Data Declarations

Align the constructors in a data type definition. If a data type has multiple constructors, each constructor will get its own line. Example:

```
data Tree a
  = Branch !a !(Tree a) !(Tree a)
  | Leaf
  deriving (Eq, Show)
```

Data types deriving lots of instances may be written like:

```
data Tree a
  = Branch !a !(Tree a) !(Tree a)
  | Leaf
  deriving
    ( Eq, Show, Ord, Read, Functor, Generic, NFData
    , Undefined, BitPack, ShowX)
```

Data types with a single constructor may be written on a single line:

```
data Foo = Foo Int
```

Format records as follows:

```
data Person = Person
  { firstName :: !String
  -- ^ First name
  , lastName :: !String
  -- ^ Last name
  , age :: !Int
  -- ^ Age
  } deriving (Eq, Show)
```

List Declarations

Align the elements in the list. Example:

```
exceptions =
  [ InvalidStatusCode
  , MissingContentHeader
  , InternalServerError ]
```

You may put the closing bracket on a new line. Use your judgement.

```
exceptions =
  [ InvalidStatusCode
  , MissingContentHeader
  , InternalServerError
  ]
```

You may not skip the first newline.

```
-- WRONG!
directions = [ North
             , East
             , South
             , West
             ]
```

unless it fits on a single line:

```
directions = [North, East, South, West]
```

Vector Declarations

Small vectors may be written on a single line:

```
nrs = 1 :> 2 :> 3 :> 4 :> 5 :> Nil
```

Large vectors should be written like:

```
exceptions =
     North
  :> East
  :> South
  :> West
  :> Nil
```

Or:

```
exceptions =
     North :> East :> South
  :> West :> Middle :> Nil
```

Language pragmas

Place LANGUAGE pragmas right after a module's documentation. Do not align the `#-}`s. `Safe`, `Unsafe`, or in some way "special" language pragmas should follow the normal ones separated by a single blank line. Pragmas should be ordered alphabetically. Example:

```
{-|
  .. docs ..
-}

{-# LANGUAGE CPP #-}
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE QuasiQuotes #-}

{-# LANGUAGE Safe #-}
```

Pragmas

Put pragmas immediately following the function they apply to. Example:

```
id :: a -> a
id x = x
{-# NOINLINE id #-}
```

Hanging Lambdas

You may or may not indent the code following a "hanging" lambda. Use your judgement. Some examples:

```
bar :: IO ()
bar =
  forM_ [1, 2, 3] $ \n -> do
    putStrLn "Here comes a number!"
    print n

foo :: IO ()
foo =
  alloca 10 $ \a ->
  alloca 20 $ \b ->
  cFunction a b
```

Export Lists

Format export lists as follows:

```
module Data.Set
  (
    -- * The @Set@ type
    Set
  , empty
  , singleton
```

(continues on next page)

(continued from previous page)

```
    -- * Querying
  , member
  ) where
```

If-then-else clauses

Generally, guards and pattern matches should be preferred over if-then-else clauses. Short cases should usually be put on a single line.

When writing non-monadic code (i.e. when not using do) and guards and pattern matches can't be used, you can align if-then-else clauses like you would normal expressions:

```
foo =
  if cond0 then
    ...
  else
    ...
```

When used in monadic contexts, use:

```
foo =
  if cond0 then do
    ...
  else do
    ...
```

The same rule applies to nested do blocks:

```
foo = do
  instruction <- decodeInstruction
  skip <- load Memory.skip
  if skip == 0x0000 then do
    execute instruction
    addCycles $ instructionCycles instruction
  else do
    store Memory.skip 0x0000
    addCycles 1
```

Case expressions

The alternatives in a case expression can be indented using either of the two following styles:

```
foobar =
  case something of
    Just j  -> foo
    Nothing -> bar
```

or as

```
foobar =
  case something of
    Just j ->
      foo
    Nothing ->
      bar
```

In monadic contexts, use:

```
foobar =
  case something of
    Just j -> do
      foo
      bar
    Nothing -> do
      fizz
      buzz
```

Align the -> arrows when it helps readability, but keep in mind that any changes potentially trigger a lot of realignments. This increases your VCS's diff sizes and becomes tedious quickly.

Type signatures

Small type signatures can be put on a single line:

```
f :: a -> a -> b
```

Longer ones should be put on multiple lines:

```
toInt
  :: Int
  -- ^ Shift char by /n/
  -> Char
  -- ^ Char to convert to ASCII integer
  -> Int
```

Multiple constraints can be added with a "tuple":

```
toInt
  :: (Num a, Show a)
  => a
  -- ^ Shift char by /n/
  -> Char
  -- ^ Char to convert to ASCII integer
  -> Int
```

Many constraints need to be split accross multiple lines too:

```
toInt
  :: ( Num a
     , Show a
     , Foo a
     , Bar a
     , Fizz a
     )
  => a
  -- ^ Shift char by /n/
  -> Char
  -- ^ Char to convert to ASCII integer
  -> Int
```

`forall`'s dot must be aligned:

```
toInt
  :: forall a
   . (Num a , Show a)
  => a
  -- ^ Shift char by /n/
  -> Char
```

(continues on next page)

(continued from previous page)

```
  -- ^ Char to convert to ASCII integer
  -> Int
```

If you have many type variables, many constraints, and many arguments, your function would end up looking like:

```
doSomething
  :: forall
       clockDomain
       resetDomain
       resetKind
       domainGatedness
   . ( Undefined a
     , Ord b
     , NFData c
     , Functor f )
  => f a
  -> f b
  -> f c
```

4.1.2 Imports

Imports should be grouped in the following order:

0. `clash-prelude`†
1. standard library imports
2. related third party imports
3. local application/library specific imports

Put a blank line between each group of imports. Create subgroups per your own judgement. The imports in each group should be sorted alphabetically, by module name.

Always use explicit import lists or `qualified` imports for standard and third party libraries. This makes the code more robust against changes in these libraries. Exception: The Prelude.

† *When writing circuit designs. Does not apply when hacking on the compiler itself.*

4.1.3 Comments

Language

Use American English. Initiali**z**ation, synchroni**z**ation, ..

Punctuation

Write proper sentences; start with a capital letter and use proper punctuation.

Top-Level Definitions

Comment every top level function (particularly exported functions), and provide a type signature; use Haddock syntax in the comments. Comment every exported data type. Function example:

```
-- | Send a message on a socket. The socket must be in a connected
-- state. Returns the number of bytes sent. Applications are
-- responsible for ensuring that all data has been sent.
send
  :: Socket
  -- ^ Connected socket
  -> ByteString
  -- ^ Data to send
  -> IO Int
  -- ^ Bytes sent
```

For functions the documentation should give enough information apply the function without looking at the function's definition.

Record example:

```
-- | Bla bla bla.
data Person = Person
  { age  :: !Int
  -- ^ Age
  , name :: !String
  -- ^ First name
  }
```

For fields that require longer comments format them like so:

```
data Record = Record
  { field1 :: !Text
  -- ^ This is a very very very long comment that is split over
  -- multiple lines.

  , field2 :: !Int
  -- ^ This is a second very very very long comment that is split
  -- over multiple lines.
  }
```

End-of-Line Comments

Separate end-of-line comments from the code using 2 spaces. Align comments for data type definitions. Some examples:

```
data Parser =
  Parser
    !Int         -- Current position
    !ByteString  -- Remaining input

foo :: Int -> Int
foo n = salt * 32 + 9
  where
    salt = 453645243  -- Magic hash salt.
```

Links

Use in-line links economically. You are encouraged to add links for API names. It is not necessary to add links for all API names in a Haddock comment. We therefore recommend adding a link to an API name if:

- The user might actually want to click on it for more information (in your judgment), and
- Only for the first occurrence of each API name in the comment (don't bother repeating a link)

4.1.4 Naming

Use camel case (e.g. `functionName`) when naming functions and upper camel case (e.g. `DataType`) when naming data types.

For readability reasons, don't capitalize all letters when using an abbreviation. For example, write `HttpServer` instead of `HTTPServer`. Exception: Two letter abbreviations, e.g. `IO`.

Use American English. That is, `synchronizer`, not `synchroniser`.

Modules

Use singular when naming modules e.g. use `Data.Map` and `Data.ByteString.Internal` instead of `Data.Maps` and `Data.ByteString.Internals`.

4.1.5 Dealing with laziness

By default, use strict data types and lazy functions.

Data types

Constructor fields should be strict, unless there's an explicit reason to make them lazy. This avoids many common pitfalls caused by too much laziness and reduces the number of brain cycles the programmer has to spend thinking about evaluation order.

```
-- Good
data Point = Point
  { pointX :: !Double
  , pointY :: !Double
  }
```

```
-- Bad
data Point = Point
  { pointX :: Double
  , pointY :: Double
  }
```

Functions

Have function arguments be lazy unless you explicitly need them to be strict.

The most common case when you need strict function arguments is in recursion with an accumulator:

```
mysum :: [Int] -> Int
mysum = go 0
  where
    go !acc []     = acc
    go acc (x:xs) = go (acc + x) xs
```

4.1.6 Misc

Point-free style

Avoid over-using point-free style. For example, this is hard to read:

```
-- Bad:
f = (g .) . h
```

Warnings

Code should be compilable with `-Wall -Werror`. There should be no warnings.

4.2 The Clash Compiler

4.2.1 Prerequisites

Hacking on Clash requires more dependencies than simply running Clash. The test suite requires having a tool available to synthesize any backend being tested. This means you need

- ghdl installed to test *VHDL*
- iverilog installed to test *Verilog*
- ModelSim installed to test *System Verilog*
- SymbiYosys and Z3 installed to test *Verilog* and *System Verilog*

4.2.2 Subprojects

The Clash compiler consists of different cabal libraries, which together provide a complete compiler. Primarily, this consists of

`clash-ghc`

The front-end of the compiler, using parts of the GHC front-end. This provides the ability to load modules, translate GHC Core to Clash Core, and implements the `clash` and `clashi` executables.

A lot of the code in this library is separated by the version of GHC it works with. For example, `src-841` is specific to GHC 8.4.x.

`clash-lib`

The back-end of the compiler, exposed as a library. This is the largest library in the project, and includes the various ASTs (e.g. Core, Netlist), normalization, code generation, and primitives / black boxes.

`clash-prelude`

The standard library for Clash as a language. This includes anything that is used to develop hardware in Clash, such as Signals, Clocks and combinators for common forms of state machine.

The `clash-prelude` library also re-exports parts of the Haskell `base` library, allowing circuit designs to re-use common functions and definitions.

The repository also contains other libraries. These either provide additional functionality which is not required, or are not yet production-ready. These are

`clash-cores`

A collection of IP cores for use in Clash designs. Currently, this includes only Lattice Ice IO cores, and SPI (with slaves implemented with the Lattice SBIO found on Lattice FPGAs).

Note: This library is optional, and is not required to use Clash. In the future it may be extended with additional IP cores.

`clash-cosim`

Co-simulation for Clash, allowing Verilog to be run inline as though it were a normal Haskell function. This provides a QuasiQuoter for use in Haskell.

Warning: This library is very experimental, and is not guaranteed to work with the most recent development version of Clash.

`clash-term`

A development tool for analysing how the normalizer in `clash-lib` affects the core of a particular design. It allows the result of each different optimzer pass to be seen for debugging purposes.

CHAPTER FIVE

CHANGELOG FOR THE CLASH PROJECT

5.1 1.4.2 *May 18th 2021*

Fixed:

- Erroneous examples in `Clash.Annotation.TopEntity` documentation #646 and #654
- `unconcat` cannot be used as initial/reset value for a `register` #1756
- `showX` now doesn't crash if a spine of a `Vec` is undefined
- `~ISACTIVEENABLE` in blackboxes works again, and now acts on `Signal dom Bool` in addition to `Enable dom`. Since #1368, enable lines were always generated even if they were known to be always enabled. Fixes #1786.
- clash –show-options now shows -fclash-* options in GHC 9.0 #1787
- `makeRecursiveGroups` now correctly identifies mutual recursion between global binders (#1796).

5.2 1.4.1 *April 6th 2021*

Fixed:

- Broken VHDL primitive template for setSlice# #1715
- Unable to reduce nested type families #1721
- DEC transformation fails for functions applied to more than 62 arguments #1669
- Erroneous examples in BlockRam.File and ROM.File documentation #1608
- Blackboxes of `Clash.Sized.Vector` functions error on vectors containing `Clocks`, `Reset`, or `Enable` #1606
- `Clash.Signal.Delayed.delayI` cannot be reset, the `HiddenReset` constraint was unintentional. Asserting its reset has never worked. Removed the constraint #1739.
- Annotate attributes cannot use type families #1742

Changed:

- `Clash.Prelude.ROM.File.romFile` now takes an `Enum addr => addr` as address argument, making it actually useful. #407

5.3 1.4.0 *March 12th 2021*

Highlighted changes (repeated in other categories):

- Clash no longer disables the monomorphism restriction. See #1270, and mentioned issues, as to why. This can cause, among other things, certain eta-reduced descriptions of sequential circuits to no longer type-check. See #1349 for code hints on what kind of changes to make to your own code in case it no longer type-checks due to this change.
- Type arguments of `Clash.Sized.Vector.fold` swapped: before `forall a n . (a -> a -> a) -> Vec (n+1) a -> a`, after `forall n a . (a -> a -> a) -> Vec (n+1) a`. This makes it easier to use `fold` in a `1 <= n` context so you can "simply" do `fold @(n-1)`
- `Fixed` now obeys the laws for `Enum` as set out in the Haskell Report, and it is now consistent with the documentation for the `Enum` class on Hackage. As `Fixed` is also `Bounded`, the rule in the Report that `succ maxBound` and `pred minBound` should result in a runtime error is interpreted as meaning that `succ` and `pred` result in a runtime error whenever the result cannot be represented, not merely for `minBound` and `maxBound` alone.
- Primitives should now be stored in `*.primitives` files instead of `*.json`. While primitive files very much look like JSON files, they're not actually spec complaint as they use newlines in strings. This has recently been brought to our attention by Aeson fixing an oversight in their parser implementation. We've therefore decided to rename the extension to prevent confusion.

Fixed:

- Result of `Clash.Class.Exp.(^)` has enough bits in order to deal with `x^0`.
- Resizes to `Signed 0` (e.g., `resize @(Signed n) @(Signed 0)`) don't throw an error anymore
- `satMul` now correctly handles arguments of type `Index 2`
- `Clash.Explicit.Reset.resetSynchronizer` now synchronizes on synchronous domains too #1567.
- `Clash.Explicit.Reset.convertReset:` now converts synchronous domains too, if necessary #1567.
- `inlineWorkFree` now never inlines a topentity. It previously only respected this invariant in one of the two cases #1587.
- Clash now reduces recursive type families #1591
- Primitive template warning is now retained when a `PrimitiveGuard` annotation is present #1625
- `signum` and `RealFrac` for `Fixed` now give the correct results.
- Fixed a memory leak in register when used on asynchronous domains. Although the memory leak has always been there, it was only triggered on asserted resets. These periods are typically short, hence typically unnoticable.
- `createDomain` will not override user definitions of types, helping users who strive for complete documentation coverage [#1674] https://github.com/clash-lang/clash-compiler/issues/1674
- `fromSNat` is now properly constrained #1692
- As part of an internal overhaul on netlist identifier generation #1265:
  - Clash no longer produces "name conflicts" between basic and extended identifiers. I.e., `\x\` and `x` are now considered the same variable in VHDL (likewise for other HDLs). Although the VHDL spec considers them distinct variables, some HDL tools - like Quartus - don't.
  - Capitalization of Haskell names are now preserved in VHDL. Note that VHDL is a case insensitive languages, so there are measures in place to prevent Clash from generating both `Foo` and `fOO`. This used to be handled by promoting every capitalized identifier to an extended one and wasn't handled for basic ones.
  - Names generated for testbenches can no longer cause collisions with previously generated entities.

  - Names generated for components can no longer cause collisions with user specified top entity names.
  - For (System)Verilog, variables can no longer cause collisions with (to be) generated entity names.
  - HO blackboxes can no longer cause collisions with identifiers declared in their surrounding architecture block.

Changed:

- Treat enable lines specially in generated HDL #1171
- `Signed`, `Unsigned`, `SFixed`, and `UFixed` now correctly implement the `Enum` law specifying that the predecessor of `minBound` and the successor of `maxBound` should result in an error #1495.
- `Fixed` now obeys the laws for `Enum` as set out in the Haskell Report, and it is now consistent with the documentation for the `Enum` class on Hackage. As `Fixed` is also `Bounded`, the rule in the Report that `succ maxBound` and `pred minBound` should result in a runtime error is interpreted as meaning that `succ` and `pred` result in a runtime error whenever the result cannot be represented, not merely for `minBound` and `maxBound` alone.
- Type arguments of `Clash.Sized.Vector.fold` swapped: before `forall a n . (a -> a -> a) -> Vec (n+1) a -> a`, after `forall n a . (a -> a -> a) -> Vec (n+1) a`. This makes it easier to use `fold` in a `1 <= n` context so you can "simply" do `fold @(n-1)`
- Moved `Clash.Core.Evaluator` into `Clash.GHC` and provided generic interface in `Clash.Core.Evalautor.Types`. This removes all GHC specific code from the evaluator in clash-lib.
- Clash no longer disables the monomorphism restriction. See #1270, and mentioned issues, as to why. This can cause, among other things, certain eta-reduced descriptions of sequential circuits to no longer type-check. See #1349 for code hints on what kind of changes to make to your own code in case it no longer type-checks due to this change.
- Clash now generates SDC files for each topentity with clock inputs
- `deepErrorX` is now equal to `undefined#`, which means that instead of the whole BitVector being undefined, its individual bits are. This makes sure bit operations are possible on it. #1532
- From GHC 9.0.1 onwards the following types: `BiSignalOut`, `Index`, `Signed`, `Unsigned`, `File`, `Ref`, and `SimIO` are all encoded as `data` instead of `newtype` to work around an issue where the Clash compiler can no longer recognize primitives over these types. This means you can no longer use `Data.Coerce.coerce` to coerce between these types and their underlying representation.
- Signals on different domains used to be coercable because the domain had a type role "phantom". This has been changed to "nominal" to prevent accidental, unsafe coercions. #1640
- Size parameters on types in Clash.Sized.Internal.* are now nominal to prevent unsafe coercions. #1640
- `hzToPeriod` now takes a `Ratio Natural` rather than a `Double`. It rounds slightly differently, leading to more intuitive results and satisfying the requested change in #1253. Clash expresses clock rate as the clock period in picoseconds. If picosecond precision is required for your design, please use the exact method of specifying a clock period rather than a clock frequency.
- `periodToHz` now results in a `Ratio Natural`
- `createDomain` doesn't override existing definitions anymore, fixing #1674
- Manifest files are now stored as `clash-manifest.json`
- Manifest files now store hashes of the files Clash generated. This allows Clash to detect user changes on a next run, preventing accidental data loss.
- Primitives should now be stored in `*.primitives` files. While primitive files very much look like JSON files, they're not actually spec complaint as they use newlines in strings. This has recently been brought to our attention by Aeson fixing an oversight in their parser implementation. We've therefore decided to rename the extension to prevent confusion.
- Each binder marked with a `Synthesize` or `TestBench` pragma will be put in its own directory under their fully qualified Haskell name. For example, two binders `foo` and `bar` in module `A` will be synthesized in `A.foo` and `A.bar`.

- Clash will no longer generate vhdl, verilog, or systemverilog subdirectories when using `-fclash-hdldir`.
- `Data.Kind.Type` is now exported from `Clash.Prelude` #1700

Added:

- Support for GHC 9.0.1
- `Clash.Signal.sameDomain`: Allows user obtain evidence whether two domains are equal.
- `xToErrorCtx`: makes it easier to track the origin of `XException` where `pack` would hide them #1461
- Additional field with synthesis attributes added to `InstDecl` in `Clash.Netlist.Types` #1482
- `Data.Ix.Ix` instances for `Signed`, `Unsigned`, and `Index` #1481 #1631
- Added `nameHint` to allow explicitly naming terms, e.g. `Signal`s.
- Checked versions of `resize`, `truncateB`, and `fromIntegral`. Depending on the type `resize`, `truncateB`, and `fromIntegral` either yield an `XException` or silently perform wrap-around if its argument does not fit in the resulting type's bounds. The added functions check the bound condition and fail with an error call if the condition is violated. They do not affect HDL generation. #1491
- `HasBiSignalDefault`: constraint to Clash.Signal.BiSignal, `pullUpMode` gives access to the pull-up mode. #1498
- Match patterns to bitPattern #1545
- Non TH `fromList` and `unsafeFromList` for Vec. These functions allow Vectors to be created from a list without needing to use template haskell, which is not always desirable. The unsafe version of the function does not compare the length of the list to the desired length of the vector, either truncating or padding with undefined if the lengths differ.
- `Clash.Explicit.Reset.resetGlitchFilter`: filters glitchy reset signals. Useful when your reset signal is connected to sensitive actuators.
- Clash can now generate EDAM for using Edalize. This generates edam.py files in all top entities with the configuration for building that entity. Users still need to edit this file to specify the EDA tool to use, and if necessary the device to target (for Quartus, Vivado etc.). #1386
- `-fclash-aggressive-x-optimization-blackboxes`: when enabled primitives can detect undefined values and change their behavior accordingly. For example, if `register` is used in combination with an undefined reset value, it will leave out the reset logic entirely. Related issue: #1506.
- Automaton-based interface to simulation, to allow interleaving of cyle-by-cycle simulation and external effects #1261

New internal features:

- `constructProduct` and `deconstructProduct` in `Clash.Primitives.DSL`. Like `tuple` and `untuple`, but on arbitrary product types.
- Support for multi result primitives. Primitives can now assign their results to multiple variables. This can help to work around synthesis tools limits in some cases. See #1560.
- Added a rule for missing `Int` comparisons in `GHC.Classes` in the compile time evaluator. #1648
- Clash now creates a mapping from domain names to configurations in `LoadModules`. #1405
- The convenience functions in `Clash.Primitives.DSL` now take a list of HDLs, instead of just one.
- `Clash.Netlist.Id` overhauls the way identifiers are generated in the Netlist part of Clash.
- Added `defaultWithAction` to Clash-as-a-library API to work around/fix issues such as #1686
- Manifest files now list files and components in an reverse topological order. This means it can be used when calling EDA tooling without causing compilation issues.

Deprecated:

- `Clash.Prelude.DataFlow`: see #1490. In time, its functionality will be replaced by clash-protocols.

Removed:

- The deprecated function `freqCalc` has been removed.

5.4 1.2.5 *November 9th 2020*

Fixed:

- The normalizeType function now fully normalizes types which require calls to reduceTypeFamily #1469
- `flogBaseSNat`, `clogBaseSNat` and `logBaseSNat` primitives are now implemented correctly.Previously these primitives would be left unevaluated causing issues as demonstrated in #1479
- Specializing on functions with type family arguments no longer fails #1477
- `satSucc`, `satPred` correctly handle "small types" such as `Index 1`.
- `msb` no longer fails on values larger than 64 bits
- `undefined` can now be used as a reset value of `autoReg@Maybe` #1507
- Signal's `fmap` is now less strict, preventing infinite loops in very specific situations. See #1521
- Clash now uses correct function names in manifest and sdc files #1533
- Clash no longer produces erroneous HDL in very specific cases #1536
- Usage of `fold` inside other HO primitives (e.g., `map`) no longer fails #1524

Changed:

- Due to difficulties using `resetSynchronizer` we've decided to make this function always insert a synchronizer. See: #1528.

5.5 1.2.4 *July 28th 2020*

- Changed:
  - Relaxed upper bound versions of `aeson` and `dlist`, in preparation for the new Stack LTS.
  - Reverted changes to primitive definitions for 'zipWith', 'map', 'foldr', and 'init' introduced in 1.2.2. They have shown to cause problems in very specific circumstances.

5.6 1.2.3 *July 11th 2020*

- Changed:
  - Upgrade to nixos 20.03. Nix and snap users will now use packages present in 20.03.
- Added:
  - `instance Monoid a => Monoid (Vec n a)`
  - `instance Text.Printf(Index)`
  - `instance Text.Printf(Signed)`
  - `instance Text.Printf(Unsigned)`
- Fixed:
  - Clash renders incorrect VHDL when GHCs Worker/Wrapper transformation is enabled #1402
  - Minor faults in generated HDL when using annotations from `Clash.Annotations.SynthesisAttributes`

- Cabal installed through Snap (`clash.cabal`) can now access the internet to fetch pacakges. [#1411]https://github.com/clash-lang/clash-compiler/issues/1411
- Generated QSys file for `altpll` incompatible with Quartus CLI (did work in Quartus GUI)
- Clash no longer uses component names that clash with identifiers imported from:
  * IEEE.STD_LOGIC_1164.all
  * IEEE.NUMERIC_STD.all
  * IEEE.MATH_REAL.all
  * std.textio.all when generating VHDL. See https://github.com/clash-lang/clash-compiler/issues/1439.

5.7 1.2.2 *June 12th 2020*

- Changed:
  - The hardwired functions to unroll primitive definitions for 'zipWith', 'map', 'foldr', and 'init' have been changed to only unroll a single step, whereas they would previously unroll the whole definition in one step. This allows Clash to take advantage of the lazy nature of these functions, in turn speeding up compilation speeds significantly in some cases. Part of PR 1354.
- Added:
  - Support for GHC 8.10
  - Ability to load designs from precompiled modules (i.e., stored in a package database). See #1172
  - Support for '-main-is' when used with `--vhdl`, `--verilog`, or `--systemverilog`
  - A partial instance for `NFDataX (Signal domain a)`
- Fixed:
  - Clash's evaluator now inlines work free definitions, preventing situations where it would otherwise get stuck in an infinite loop
  - `caseCon` doesn't apply type-substitution correctly #1340
  - Clash generates illegal SystemVerilog slice #1313
  - Fix result type of head and tail Verilog blackboxes #1351
  - Certain recursive let-expressions in side a alternatives of a case-expression throw the Clash compiler into an infinite loop #1316
  - Fixes issue with one of Clash's transformations, `inlineCleanup`, introducing free variables #1337
  - Fails to propagate type information of existential type #1310
  - Certain case-expressions throw the Clash compiler into an infinite loop #1320
  - Added blackbox implementation for 'Clash.Sized.Vector.iterateI', hence making it usable as a register reset value #1240
  - `iterate` and `iterateI` can now be used in reset values #1240
  - Prim evaluation fails on undefined arguments #1297
  - Missing re-indexing in (Un)Signed fromSLV conversion #1292
  - VHDL: generate a type qualification inside ~TOBV, fixes #1360

5.8 1.2.1 *April 23rd 2020*

- Changed:
  - Treat `Signed 0`, `Unsigned 0`, `Index 1`, `BitVector 0` as unit. In effect this means that 'minBound' and 'maxBound' return 0, whereas previously they might crash #1183
  - Infix use of `deepseqX` is now right-associative
- Added:
  - Add 'natToInteger', 'natToNatural', and 'natToNum'. Similar to 'snatTo*', but works solely on a type argument instead of an SNat.
  - `Clash.Sized.Vector.unfoldr` and `Clash.Sized.Vector.unfoldrI` to construct vectors from a seed value
  - Added NFDataX instances for `Data.Monoid.{First,Last}`
- Fixed:
  - The Verilog backend can now deal with non-contiguous ranges in custom bit-representations.
  - Synthesizing BitPack instances for type with phantom parameter fails #1242
  - Synthesis of `fromBNat (toBNat d5)` failed due to `unsafeCoerce` coercing from `Any`
  - Memory leak in register primitives #1256
  - Illegal VHDL slice when projecting nested SOP type #1254
  - Vivado VHDL code path (`-fclash-hdlsyn Vivado`) generates illegal VHDL #1264

5.9 1.2.0 *March 5th 2020*

As promised when releasing 1.0, we've tried our best to keep the API stable. We think most designs will continue to compile with this new version, although special care needs to be taken when using:

- Use inline blackboxes. Instead of taking a single HDL, inline primitives now take multiple. For example, `InlinePrimitive VHDL ".."` must now be written as `InlinePrimitive [VHDL] ".."`.
- Use the `Enum` instance for `BitVector`, `Index`, `Signed`, or `Unsigned`, as they now respect their `maxBound`. See #1089.

On top of that, we've added a number of new features:

- `makeTopEntity`: Template Haskell function for generating TopEntity annotations. See the documentation on Haddock for more information.
- `Clash.Explicit.SimIO`: ((System)Verilog only) I/O actions that can be translated to HDL I/O. See the documentation on Haddock for more information.
- `Clash.Class.AutoReg`: A smart register that improves the chances of synthesis tools inferring clock-gated registers, when used. See the documentation on Haddock for more information.

The full list of changes follows. Happy hacking!

- New features (API):
  - `Clash.Class.Parity` type class replaces Prelude `odd` and `even` functions due to assumptions that don't hold for Clash specific numerical types, see #970.
  - `NFDataX.ensureSpine`, see #748
  - `makeTopEntity` Template Haskell function for generating TopEntity annotations intended to cover the majority of use cases. Generation failures should either result in an explicit error, or a valid annotation of an empty `PortProduct`. Any discrepancy between the *shape* of generated annotations and the *shape* of the Clash compiler is a bug. See #795. Known limitations:

    * Type application (excluding `Signals` and `:::`) is best effort:
    * Data types with type parameters will work if the generator can discover a single relevant constructor after attempting type application.
    * Arbitrary explicit clock/reset/enables are supported, but only a single `HiddenClockResetEnable` constraint is supported.
    * Data/type family support is best effort.
  - Added `Bundle ((f :*: g) a)` instance
  - Added `NFDataX CUShort` instance
  - Clash's internal type family solver now recognizes `AppendSymbol` and `CmpSymbol`
  - Added `Clash.Magic.suffixNameFromNat`: can be used in cases where `suffixName` is too slow
  - Added `Clash.Class.AutoReg`. Improves the chances of synthesis tools inferring clock-gated registers, when used. See #873.
  - `Clash.Magic.suffixNameP`, `Clash.Magic.suffixNameFromNatP`: enable prefixing of name suffixes
  - Added `Clash.Magic.noDeDup`: can be used to instruct Clash to /not/ share a function between multiple branches
  - A `BitPack a` constraint now implies a `KnownNat (BitSize a)` constraint, so you won't have to add it manually anymore. See #942.
  - `Clash.Explicit.SimIO`: ((System)Verilog only) I/O actions that can be translated to HDL I/O; useful for generated test benches.
  - Export `Clash.Explicit.Testbench.assertBitVector` #888
  - Add `Clash.Prelude.Testbench.assertBitVector` to achieve feature parity with `Clash.Explicit.Testbench`. #891
  - Add `Clash.XException.NFDataX.ensureSpine` #803
  - Add `Clash.Class.BitPack.bitCoerceMap` #798
  - Add `Clash.Magic.deDup`: instruct Clash to force sharing an operator between multiple branches of a case-expression
  - `InlinePrimitive` can now support multiple backends simultaneously #425
  - Add `Clash.XException.hwSeqX`: render declarations of an argument, but don't assign it to a result signal
  - Add `Clash.Signal.Bundle.TaggedEmptyTuple`: allows users to emulate the pre-1.0 behavior of "Bundle ()". See #1100
- New features (Compiler):
  - #961: Show `-fclash-*` Options in `clash --show-options`
- New internal features:
  - #918: Add X-Optimization to normalization passes (-fclash-aggressive-x-optimization)
  - #821: Add `DebugTry`: print name of all tried transformations, even if they didn't succeed
  - #856: Add `-fclash-debug-transformations`: only print debug info for specific transformations
  - #911: Add 'RenderVoid' option to blackboxes
  - #958: Prefix names of inlined functions
  - #947: Add "Clash.Core.TermLiteral"

  - #887: Show nicer error messages when failing in TH code
  - #884: Teach reduceTypeFamily about AppendSymbol and CmpSymbol
  - #784: Print whether `Id` is global or local in ppr output
  - #781: Use naming contexts in register names
  - #1061: Add 'usedArguments' to BlackBoxHaskell blackboxes
- Fixes issues:
  - #974: Fix indirect shadowing in `reduceNonRepPrim`
  - #964: SaturatingNum instance of `Index` now behaves correctly when the size of the index overflows an `Int`.
  - #810: Verilog backend now correctly specifies type of `BitVector 1`
  - #811: Improve module load behavior in clashi
  - #439: Template Haskell splices and TopEntity annotations can now be used in clashi
  - #662: Clash will now constant specialize partially constant constructs
  - #700: Check work content of expression in cast before warning users. Should eliminate a lot of (superfluous) warnings about "specializing on non work-free cast"s.
  - #837: Blackboxes will now report clearer error messages if they're given unexpected arguments.
  - #869: PLL is no longer duplicated in Blinker.hs example
  - #749: Clash's dependencies now all work with GHC 8.8, allowing `clash-{prelude,lib,ghc}` to be compiled from Hackage soon.
  - #871: RTree Bundle instance is now properly lazy
  - #895: VHDL type error when generating `Maybe (Vec 2 (Signed 8), Index 1)`
  - #880: Custom bit representations can now be used on product types too
  - #976: Prevent shadowing in Clash's core evaluator
  - #1007: Can't translate domain tagType.Errors.IfStuck…
  - #967: Naming registers disconnects their output
  - #990: Internal shadowing bug results in incorrect HDL
  - #945: Rewrite rules for Vec Applicative Functor
  - #919: Clash generating invalid Verilog after Vec operations #919
  - #996: Ambiguous clock when using `ClearOnReset` and `resetGen` together
  - #701: Unexpected behaviour with the `Synthesize` annotation
  - #694: Custom bit representation error only with VHDL
  - #347: topEntity synthesis fails due to insufficient type-level normalisation
  - #626: Missing Clash.Explicit.Prelude definitions
  - #960: Blackbox Error Caused by Simple map
  - #1012: Case-let doesn't look through ticks
  - #430: Issue warning when not compiled with `executable-dynamic: True`
  - #374: Clash.Sized.Fixed: fromInteger and fromRational don't saturate correctly
  - #836: Generate warning when `toInteger` blackbox drops MSBs
  - #1019: Clash breaks on constants defined in terms of `GHC.Natural.gcdNatural`
  - #1025: `inlineCleanup`will not produce empty letrecs anymore

- #1030: `bindConstantVar` will bind (workfree) constructs
- #1034: Error (10137): object "pllLock" on lhs must have a variable data type
- #1046: Don't confuse term/type namespaces in 'lookupIdSubst'
- #1041: Nested product types incorrectly decomposed into ports
- #1058: Prevent substitution warning when using type equalities in top entities
- #1033: Fix issue where Clash breaks when using Clock/Reset/Enable in product types in combination with Synthesize annotations
- #1075: Removed superfluous constraints on 'maybeX' and 'maybeIsX'
- #1085: Suggest exporting topentities if they can't be found in a module
- #1065: Report polymorphic topEntities as errors
- #1089: Respect maxBound in Enum instances for BitVector,Index,Signed,Unsigned

- Fixes without issue reports:
  - Fix bug in `rnfX` defined for `Down` (baef30e)
  - Render numbers inside gensym (bc76f0f)
  - Report blackbox name when encountering an error in 'setSym' (#858)
  - Fix blackbox issues causing Clash to generate invalid HDL (#865)
  - Treat types with a zero-width custom bit representation like other zero-width constructs (#874)
  - TH code for auto deriving bit representations now produces nicer error messages (7190793)
  - Adds '–enable-shared-executables' for nix builds; this should make Clash run *much* faster (#894)
  - Custom bit representations can now mark fields as zero-width without crashing the compiler (#898)
  - Throw an error if there's data left to parse after successfully parsing a valid JSON construct (#904)
  - `Data.gfoldl` is now manually implemented, in turn fixing issues with `gshow` (#933)
  - Fix a number of issues with blackbox implementations (#934)
  - Don't inline registers with non-constant clock and reset (#998)
  - Inline let-binders called [dsN | N <- [1..]] (#992)
  - ClockGens use their name at the Haskell level #827
  - Render numbers inside gensym #809
  - Don't overwrite existing binders when specializing #790
  - Deshadow in 'caseCase' #1067
  - Deshadow in 'caseLet' and 'nonRepANF' #1071
- Deprecations & removals:
  - Removed support for GHC 8.2 (#842)
  - Removed support for older cabal versions, only Cabal >=2.2 supported (#851)
  - Reset and Enable constructors are now only exported from Clash.Signal.Internal
  - #986 Remove -fclash-allow-zero-width flag

5.10 1.0.0 *September 3rd 2019*

- 10x - 50x faster compile times
- New features:
  - API changes: check the migration guide at the end of `Clash.Tutorial`
  - All memory elements now have an (implicit) enable line; "Gated" clocks have been removed as the clock wasn't actually gated, but implemented as an enable line.
  - Circuit domains are now configurable in:
    - (old) The clock period
    - (new) Clock edge on which memory elements latch their inputs (rising edge or falling edge)
    - (new) Whether the reset port of a memory element is level sensitive asynchronous reset) or edge sensitive (synchronous reset)
    - (new) Whether the reset port of a memory element is active-high or active-low (negated reset)
    - (new) Whether memory element power on in a configurable/defined state (common on FPGAs) or in an undefined state (ASICs)
    - See the blog post on this new feature
  - Data types can now be given custom bit-representations: http://hackage.haskell.org/package/clash-prelude/docs/Clash-Annotations-BitRepresentation.html
  - Annotate expressions with attributes that persist in the generated HDL, e.g. synthesis directives: http://hackage.haskell.org/package/clash-prelude/docs/Clash-Annotations-SynthesisAttributes.html
  - Control (System)Verilog module instance, and VHDL entity instantiation names in generated code: http://hackage.haskell.org/package/clash-prelude/docs/Clash-Magic.html
  - Much improved infrastructure for handling of unknown values: defined spine, but unknown leafs: http://hackage.haskell.org/package/clash-prelude/docs/Clash-XException.html#t:NFDataX
  - Experimental: Multiple hidden clocks. Can be enabled by compiling `clash-prelude` with `-fmultiple-hidden`
  - Experimental: Limited GADT support (pattern matching on vectors, or custom GADTs as longs as their usage can be statically removed; no support of recursive GADTs)
  - Experimental: Use regular Haskell functions to generate HDL black boxes for primitives (in an addition to existing string templates for HDL black boxes) See for example: http://hackage.haskell.org/package/clash-lib/docs/Clash-Primitives-Intel-ClockGen.html
- Fixes issues:
  - #316
  - #319
  - #323
  - #324
  - #329
  - #331
  - #332
  - #335
  - #348
  - #349
  - #350

- #351
- #352
- #353
- #358
- #359
- #363
- #364
- #365
- #371
- #372
- #373
- #378
- #380
- #381
- #382
- #383
- #387
- #393
- #396
- #398
- #399
- #401
- #403
- #407
- #412
- #413
- #420
- #422
- #423
- #424
- #438
- #450
- #452
- #455
- #460
- #461
- #463
- #468
- #475

- #476
- #500
- #507
- #512
- #516
- #517
- #526
- #556
- #560
- #566
- #567
- #569
- #573
- #575
- #581
- #582
- #586
- #588
- #591
- #596
- #601
- #607
- #629
- #637
- #644
- #647
- #661
- #668
- #677
- #678
- #682
- #691
- #703
- #713
- #715
- #727
- #730
- #736
- #738

5.11 0.99.3 *July 28th 2018*

- Fixes bugs:
  - Evaluator recognizes `Bit` literals #329
  - Use existential type-variables in context of GADT pattern match
  - Do not create zero-bit temporary variables in generated HDL
  - Use correct arguments in nested primitives #323
  - Zero-constructor data type needs 0 bits #238
  - Create empty component when result needs 0 bits
  - Evaluator performs BigNat arithmetic
- Features:
  - Bundle and BitPack instances up to and including 62-tuples
  - Handle undefined writes to RAM properly
  - Handle undefined clock enables properly

5.12 0.99.1 *May 12th 2018*

- Allow `~NAME[N]` tag inside `~GENSYM[X]`
- Support HDL record selector generation #313
- `InlinePrimitive` support: specify HDL primitives inline with Haskell code
- Support for `ghc-typelits-natnormalise-0.6.1`
- `Lift` instances for `TopEntity` and `PortName`
- `InlinePrimitive` support: specify HDL primitives inline with Haskell code

5.13 0.99 *March 31st 2018*

- New features:
  - Major API overhaul: check the migration guide at the end of `Clash.Tutorial`
  - New features:
    - Explicit clock and reset arguments
    - Rename `CLaSH` to `Clash`
    - Implicit/`Hidden` clock and reset arguments using a combination of `reflection` and `ImplicitParams`.
    - Large overhaul of `TopEntity` annotations
    - PLL and other clock sources can now be instantiated using regular functions: `Clash.Intel.ClockGen` and `Clash.Xilinx.ClockGen`.
    - DDR registers:
      - Generic/ASIC: `Clash.Explicit.DDR`
      - Intel: `Clash.Intel.DDR`
      - Xilinx: `Clash.Intel.Xilinx`

   - `Bit` is now a `newtype` instead of a `type` synonym and will be mapped to a HDL scalar instead of an array of one (e.g `std_logic` instead of `std_logic_vector(0 downto 0)`)
   - Hierarchies with multiple synthesisable boundaries by allowing more than one function in scope to have a `Synthesize` annotation.
     * Local caching of functions with a `Synthesize` annotation
   - `Bit` type is mapped to a HDL scalar type (e.g. `std_logic` in VHDL)
   - Improved name preservation
   - Zero-bit values are filtered out of the generated HDL
   - Improved compile-time computation
- Many bug fixes

5.14 Older versions

Check out:

- https://github.com/clash-lang/clash-compiler/blob/3649a2962415ea8ca2d6f7f5e673b4c14de26b4f/clash-prelude/CHANGELOG.md
- https://github.com/clash-lang/clash-compiler/blob/3649a2962415ea8ca2d6f7f5e673b4c14de26b4f/clash-lib/CHANGELOG.md
- https://github.com/clash-lang/clash-compiler/blob/3649a2962415ea8ca2d6f7f5e673b4c14de26b4f/clash-ghc/CHANGELOG.md

CHAPTER

SIX

REFERENCES

- Appel, R.N. and Folmer, H.H. (2016) Analysis, optimization, and design of a SLAM solution for an implementation on reconfigurable hardware (FPGA) using CλaSH. MSc thesis, University of Twente, Enschede, The Netherlands, December 2016.
- Vossen, J.J. (2016) Offloading Haskell functions onto an FPGA. MSc thesis, University of Twente, Enschede, The Netherlands, December 2016.
- Verheij, J.G.J. (2016) Co-simulation between CλaSH and traditional HDLs. MSc thesis, University of Twente, Enschede, The Netherlands, August 2016.
- Raa, I. te (2015) Recursive functional hardware descriptions using CλaSH. MSc thesis, University of Twente, Enschede, The Netherlands, November 2015.
- Wester, R. (2015) A transformation-based approach to hardware design using higher-order functions. PhD thesis, University of Twente, Enschede, The Netherlands, July 2015.
- Bakker, M. (2015) Numerical mathematics on FPGAs using CλaSH. BSc thesis, University of Twente, Enschede, The Netherlands, July 2015.
- Dam, M.R. (2015) Auditory processing using CλaSH. MSc thesis, University of Twente, Enschede, The Netherlands, May 2015.
- Harmsen, R. (2015) Specifying the WaveCore in CλaSH. MSc thesis, University of Twente, Enschede, The Netherlands, March 2015.
- Baaij, C.P.R. (2015) Digital Circuits in CλaSH: Functional Specifications and Type-Directed Synthesis. PhD thesis, University of Twente, Enschede, The Netherlands, January 2015.
- Wester, R. and Kuper, J. (2014) Design space exploration of a particle filter using higher-order functions. In: *Reconfigurable Computing: Architectures, Tools, and Applications*. Lecture Notes in Computer Science 8405. Springer Verlag, London, pp. 219-226. ISSN 0302-9743 ISBN 978-3-319-05959-4.
- Bos, J.C.H. (2014) Synthesizable Specification of a VLIW Processor in the Functional Hardware Description Language CλaSH. MSc thesis, University of Twente, Enschede, The Netherlands, September 2014.
- Niedermeier, A. (2014) A Fine-Grained Parallel Dataflow-Inspired Architecture for Streaming Applications. PhD thesis, University of Twente, Enschede, The Netherlands, August 2014.
- Kuper, J. and Wester, R. (2014) N Queens on an FPGA: Mathematics, Programming, or Both?. In: *Communicating Processes Architectures 2014*, 24-27 August 2014, Oxford, UK. Open Channel Publishing. ISBN 978-0-9565409-8-0.
- Bronkhorst, T.A.W. (2014) Hardware design of a cooperative adaptive cruise control system using a functional programming language. MSc thesis, University of Twente, Enschede, The Netherlands, August 2014.
- Jin, X. (2014) Implementation of the MUSIC Algorithm in CλaSH. MSc thesis, University of Twente, Enschede, The Netherlands, June 2014.
- Nee, F. van (2014) To a new hardware design methodology: A case study of the cochlea model. MSc thesis, University of Twente, Enschede, The Netherlands, March 2014.
- Baaij, C.P.R. and Kuper, J. (2014) Using Rewriting to Synthesize Functional Languages to Digital Circuits. In: *Jay McCarthy, editor, Trends in Functional Programming (TFP)*, Provo, UT, USA, May 14-16, 2013. Volume

8322 of Lecture Notes in Computer Science (LNCS). pages 17–33. Springer-Verlag. ISBN 978-3-642-45340-3.

- Wester, R. and Baaij, C.P.R. and Kuper, J. (2012) A two step hardware design method using CλaSH. In: *Proceedings of the 22nd International Conference on Field Programmable Logic and Applications (FPL)*, Aug 29-31, 2012, Oslo, Norway. pages 181-188. IEEE Computer Society. ISBN 978-1-4673-2257-7.
- Wester, R. and Sarakiotis, D. and Kooistra, E. and J. Kuper. (2012) Specifications of APERTIF Polyphase Filter Bank in CλaSH. In: *Communicating Process Architectures (CPA)*, pages 53-64, United Kingdom, August 2012. Open Channel Publishing. ISBN 978-0-9565409-5-9.
- Gerards, M.E.T. and Baaij, C.P.R. and Kuper, J. and Kooijman, M. (2011) Higher-Order Abstraction in Hardware Descriptions with CλaSH. In: *Proceedings of the 14th Conference on Digital System Design (DSD)*, Oulu, Finland. pages 495-502, 31 Aug - 2 September, 2011. IEEE Computer Society. ISBN 978-0-7695-4494-6.
- Niedermeier, A. and Wester, R. and Rovers, K.C. and Baaij, C.P.R. and Kuper, J. and Smit, G.J.M. (2010) Designing a dataflow processor using CλaSH. In: *28th Norchip Conference*, 15-16 November 2010, Tampere, Finland. 69. IEEE Circuits and Systems Society. ISBN 978-1-4244-8971-8.
- Kuper, J. and Baaij, C.P.R. and Kooijman, M. and Gerards, M.E.T. (2010) Exercises in architecture specification using CλaSH. In: *Proceedings of Forum on Specification and Design Languages (FDL)*, 2010, Southampton, England, Sept 13-16. pages 178-183. Electronic Chips & Systems design Initiative (ECSI). ISSN 1636-9874.
- Baaij, C.P.R. and Kooijman, M. and Kuper, J. and Boeijink, W.A. and Gerards, M.E.T. (2010) CλaSH: Structural Descriptions of Synchronous Hardware using Haskell. In: *Proceedings of the 13th Conference on Digital System Design (DSD)*, Lille, France, Sept 1-3, 2010. pages 714-721. IEEE Computer Society. ISBN 978-0-7695-4171-6.
- Smit, G.J.M. and Kuper, J. and Baaij, C.P.R (2010) A mathematical approach towards hardware design. In: *Dagstuhl Seminar on Dynamically Reconfigurable Architectures*, 11-16 July 2010, Dagstuhl, Germany.
- Baaij, C.P.R. (2009) CλasH : from Haskell to hardware. MSc thesis, University of Twente, Enschede, The Netherlands, December 2009.
- Kooijman, M. (2009) Haskell as a higher order structural hardware description language. MSc thesis, University of Twente, Enschede, The Netherlands, December 2009.