Refinement Types for Hardware

Robin Webbers
Vrije Universiteit Amsterdam

Klaus v. Gleissenthall
Vrije Universiteit Amsterdam

Abstract

Ensuring that hardware behaves correctly according to its specification is difficult. Existing approaches either require considerable user expertise and effort, or fail to provide strong guarantees in the form of proofs. In this paper, we propose to adapt refinement typing—a technique that has shown considerable success in software—to the hardware setting. To show feasibility of our approach, we use our technique to prove functional correctness for a simple load-store core with respect to a specification of an ISA as a formal interface.

ACM Reference Format:
Robin Webbers and Klaus v. Gleissenthall. 2022. Refinement Types for Hardware. In *Proceedings of* . ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Modern accelerators are complex, highly concurrent and filled to the brim with clever performance optimizations. While this makes them fast, it also makes it notoriously tricky to get them right.

Unfortunately, getting it wrong can be uniquely damaging in hardware. Unlike software, hardware, once deployed, cannot easily be patched when new bugs are found. Moreover, since hardware is at the bottom of the stack, potential bugs might propagate upwards causing unexpected or faulty behaviour in other parts of the system.

As such, we need to make it easy to provide strong correctness guarantees at compile-time. Recently, there have been some promising efforts in this direction. Verification languages such as Kami [4] and Cava [1] embed hardware description languages (HDLs) within theorem provers such as Coq. This allows them to give expressive guarantees using machine checked proofs. Unfortunately, theorem proving via dependent typing requires both considerable expertise, and substantial manual labour when spelling out proofs. Bounded model checking (BMC) based techniques [2], on the other hand, require little user effort or expertise. They automatically check whether the system preserves certain user-specified properties. Unfortunately, BMC only proves correctness for finite runs of a certain depth, thus falling short of a full proof.

, ,
2022. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

In this work, we want to provide strong guarantees in the form of proofs while striking a balance between expressiveness and automation. For this, we propose to adapt refinement types [5], which have been successful in software, to hardware verification. In this article, we use Clash [3] and Liquid Haskell [6] to write hardware descriptions and their refinements.

2 Refinement Types in Software

Refinements are predicates over base-types that allow users to restrict the base-type to a subdomain. For example, the `Even` type restricts `Int` to even numbers, only. For some value `v`, we can capture this restriction through the predicate `(v `mod` 2) = 0`. This gives us the following type.

```
type Even = {v:Int | (v `mod` 2) = 0}
```

This type can then be placed on function inputs and outputs. When used on an input, the type acts as a precondition. This means, we can use the predicate as an assumption when showing correctness of the function body. Conversely, any potential caller has to prove that the argument is indeed even. When placed on a function output, the predicate acts as post-condition. That is, we create a proof obligation requiring us to show that the output should indeed be even. Any caller may then use the fact that the output is even in its own proof.

Predicates are not restricted to single values, but can also express *relations between* values. For example in the function `max`, shown below, we specify that the function output must be larger than both its operands.

```
max :: Ord a ⇒ x:a → y:a
    → {v:a | x ≤ v ∧ y ≤ v}
max x y = if x ≤ y then y else x
```

3 Refinement Types in Hardware

In a functional HDL, such as Clash, we describe combinational logic via pure functions. As such, any combinational description of hardware can be refined, just like in software.

Using refinements, we want to show that the hardware behaves according to a specification. We can supply such a specification via the RISC-V formal interface—an interface that describes semantics and side effect for each instruction. This interface is usually used in combination with a bounded model checker that then validates it against the circuit.

Here, we utilize refinement typing for a similar purpose; the idea is to encapsulate this specification within the refinement language. From there, one can then show that the core adheres to this specification via refinements. We have

used this method to specify and verify a small load-store processor[1]. The specification contains detailed descriptions of state changes based on instruction execution. Examples of state changes we were able to capture are correctness of program counter updates, register read/writes and address calculation.

3.1 Specification

A specification can be viewed as the description of a state machine that transitions in a single step. Since the processor may take several steps to execute the same instruction, a proof then needs to produce invariants over intermediate values that imply the final correctness predicates.

To illustrate this, we specify a toy interface. After that, we will show how an implementation of this interface can be shown to adhere to the interface.

We specify a system with two registers `x` and `y`, where both are incremented by one at each state transition:

```
data Spec = Spec
  { x  :: Int
  , y  :: Int
  , x' :: {v:Int | v = x + 1}
  , y' :: {v:Int | v = y + 1}
  }
```

A core now has to satisfy the specification whenever it retires a state transition.

3.2 Checking Correctness

Now, consider an implementation where we wish to be conservative in circuit size. We therefore choose to have a *single* increment circuit and thus need to increment `x` and `y` at different cycles. We maintain an internal state `Stage` to know what value to increment. Synchronization with the interface is done at the `y` increment.

```
data Stage = IncX | IncY
type State = (Stage, Int, Int)

impl :: State → () → (State, Maybe Spec)
impl (IncX, x,  y) _ = ((IncY, x+1, y), None)
impl (IncY, x', y) _ = ((IncX, x',  y), spec)
  where
    spec = Just $ Spec x y x' y'
    y'   = y + 1
```

When we are in the `IncY` stage, it can be inferred from the function body that `y' = y + 1`. However, we cannot guarantee that `x' = x + 1` as this happened in the previous cycle. We solve this by placing a pre-condition on the input `State` of `impl`[2]:

```
type RefState = {(stage, x', _):State
                | stage = IncY ⇒ x' = x + 1}
```

Since this pre-condition also has to be provided by `impl` in a previous cycle, it has to be a post-condition on the state as well. In fact, *refinements on the state can only be invariants.* As such, we arrive at the following annotation:

```
impl :: RefState → () → (RefState, Maybe Spec)
```

With this invariant on the state, the refinement type system can show validity of our system.

3.3 Relational Properties

With our interface, we can capture most of the same checks as the RISC-V formal interface. One property that we cannot express yet however is instruction ordering; we cannot express that a core may retire instructions out-of-order as long as causality is preserved. This is because we have to relate the *infinite* sequence of retired instructions to make sure that no pair of two instructions breaks causality. With the interface above, we can relate only a *finite* number of previous states.

Pairwise list refinements provide a way to specify a binary relation between elements of a list. We relate all elements x_i and x_j such that their indices are constrained by $0 \leq i < j \leq n$, where n is the length of the list. With this, we can for example specify that the elements of a list are in increasing order:

```
type IncList a = [a]<{\xi xj → xi < xj}>
```

We leverage pairwise list refinements to specify infinite relations over a list. The list is then used as an output specification for the sequential circuit.

Consider, for example, a counter circuit that counts infinitely upwards, producing the sequence `[0, 1, 2, 3, ...]`. It has a single register to keep track of the current number and produces this as output at each cycle. It takes no input. The description is shown below[3].

```
countStep acc _ = (acc + 1, acc)

countSeq :: [()] -> IncList Int
countSeq = mealy countStep 0
```

The idea now is to use a similar specification for instruction ordering. We attach a sequence number to memory reads such that a retired instruction has a larger sequence number than all previous numbers. To allow for out-of-order instruction retiring, we can relax the constraint if there are no read-write conflicts. Notice how for any pair that has conflicting sequence numbers, we should prove that they retain causality. As such, we have to relate a theoretically infinite sequence.

[1] We could not use the RISC-V ISA due to small incompatibilities between LiquidHaskell and Clash. From a theoretical point of view though, refining a RISC-V core does not impose any additional complications.

[2] Notice in this implementation how we do not have `x` in scope when we commit our specification. We currently solve this by keeping a ghost-variable in the state (not present here to reduce clutter). We are looking into a way to separate the specification state, such that the previous state is maintained separately and is only available for refinements.

[3] The type inference here is possible due to complex bounded refinements on `mealy`, however; this is out of scope for the paper.

References

[1] [n. d.]. Cava https://github.com/project-oak/silveroak.

[2] [n. d.]. RISC Formal https://github.com/SymbioticEDA/riscv-formal.

[3] [n. d.]. https://clash-lang.org/.

[4] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. 2017. Kami: A Platform for High-Level Parametric Hardware Specification and Its Modular Verification. In *International Conference on Functional Programming (ICFP)*. ACM SIGPLAN. http://plv.csail.mit.edu/kami/papers/icfp17.pdf

[5] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid Types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 159–169. https://doi.org/10.1145/1375581.1375602

[6] Niki Vazou, Eric L. Seidel, and Ranjit Jhala. 2014. LiquidHaskell: Experience with Refinement Types in the Real World. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell* (Gothenburg, Sweden) *(Haskell '14)*. Association for Computing Machinery, New York, NY, USA, 39–51. https://doi.org/10.1145/2633357.2633366