Metaprogramming with Combinators

Mahshid Shahmohammadian
Department of Computer Science
Drexel University
Philadelphia, USA
ms4323@drexel.edu

Geoffrey Mainland
Department of Computer Science
Drexel University
Philadelphia, USA
mainland@drexel.edu

Abstract

There are a wide array of methods for writing code generators. We advocate for a point in the design space, which we call *metaprogramming with combinators*, where programmers use (and write) combinator libraries that directly manipulate object language terms. The key language feature that makes this style of programming palatable is quasiquotation. Our approach leverages quasiquotation and other host language features to provide what is essentially a rich, well-typed macro language. Unlike other approaches, metaprogramming with combinators allows full control over generated code, thereby also providing full control over performance and resource usage. This control does not require sacrificing the ability to write high-level abstractions. We demonstrate metaprogramming with combinators through several code generators written in Haskell that produce VHDL targeted to Xilinx FPGAs.

CCS Concepts: • **Software and its engineering → Domain specific languages**; **Source code generation**; • **Hardware → Hardware description languages and compilation**.

Keywords: metaprogramming, code generation, hardware, combinators, VHDL, Haskell

ACM Reference Format:
Mahshid Shahmohammadian and Geoffrey Mainland. 2021. Metaprogramming with Combinators. In *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '21), October 17–18, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3486609.3487198

GPCE '21, October 17–18, 2021, Chicago, IL, USA

ACM ISBN 978-1-4503-9112-2/21/10…$15.00
https://doi.org/10.1145/3486609.3487198

1 Introduction

A primary motivation for writing code generators is the desire to avoid boilerplate—repetitive code that is straightforward but tedious to write. Boilerplate becomes necessary when a language does not provide features that permit abstraction over the pattern of computation embodied by the boilerplate. With any new abstraction there comes a tension between expressivity and performance—an abstraction may allow the programmer to avoid repetitive code, but at what cost? This tension is particularly acute in the hardware domain, where the two primary languages, VHDL and Verilog, do not even allow easy abstraction over types. Furthermore, in hardware, local inefficiencies can have far ranging implications by forcing a reduction in the (global!) clock rate. Because it is a particularly challenging domain, we choose hardware generation to demonstrate our approach to providing new abstraction facilities for a language without sacrificing performance, which we call *metaprogramming with combinators* (MWC).

The essence of our approach is to directly embed the language being generated—the *object language*—into the language that is used for code generation—the *metalanguage*. In this paper, the object language is VHDL and the metalanguage is Haskell. "Directly" embedding the object language means that programmers literally write VHDL concrete syntax mixed with Haskell. This is made possible by Haskell's support for quasiquotation [20]. Our primary points of comparison are Kansas Lava [12], which exemplifies the standard technique for deeply embedding a language in Haskell, and CλaSH [4], which offers a path for compiling a subset of Haskell directly to hardware. Both Kansas Lava and CλaSH can interpret terms as either Haskell programs or as hardware generators, allowing programmers to test circuits using standard Haskell tools, like QuickCheck [8]. A code generator with embedded VHDL terms cannot be interpreted as a Haskell program—it can only be used to generate VHDL—but we show how to recover Haskell interpretability for a subset of programs (Sections 3 and 4). This allows many MWC programs to be executed either as Haskell programs or as code generators.

Metaprogramming with combinators combines full control over generated code (when needed) with the full powers of abstraction of a functional language (Haskell). Like deeply embedded languages (Kansas Lava) or languages that offer (partial) translation of a functional language to another

domain (CλaSH), MWC can express powerful new programming constructs using libraries instead of requiring language (and compiler) modifications (Section 4). Unlike these other approaches, the performance penalty is minimal (Section 2). By carefully separating out the pure, functional portion of hardware circuits, a subset of MWC programs can be interpreted as Haskell programs and tested with standard Haskell tools like QuickCheck (Section 3). Without sacrificing the simplicity and elegance of functional code, generated VHDL can even be competitive with vendor-supplied IP (Section 6). Concretely, this paper makes the following contributions:

- A Haskell quasiquoter for VHDL, which allows VHDL concrete syntax to be embedded in Haskell programs.
- A library for expressing combinational (purely functional) circuits such that they can be interpreted directly as Haskell programs or used to generate VHDL.
- A high-level combinator library for building hardware pipelines and generating testbenches.
- Implementations of several low-level IP components (convolutional encoder, divider, CORDIC), built using these two libraries, with performance comparable to Xilinx IP.

All source code and IP described in this paper are open source.[1]

2 Background and Motivating Example

Hardware is a particularly challenging domain for code generation because of its sensitivity to local performance characteristics and pervasive parallelism. In this paper, we assume fully synchronous circuits where all operations are driven by a single clock, which is typical for FPGAs. Circuits are composed of *combinational logic*, which can be expressed as a pure function of its inputs, and (stateful) registers, which can be read and updated every clock cycle. Clock rate is limited by the time it takes the combinational logic in a circuit to reach a stable output. The time needed to reach stability is influenced by factors including the number of gates needed to express combinational logic as well as the signal delay between gates—longer paths and larger gate counts both require more time for outputs to stabilize. Because there is a single global clock, the clock rate is limited by the time it takes the *slowest* combinational path to reach stability. This means that adding a single slow step to a multi-step computation can slow down *every* step in the computation by imposing a lower clock rate. In hardware, poor local performance has global implications that would not be present in a typical software system.

Synchronous circuits are massively parallel—on each clock cycle, every register in a circuit has the opportunity to be read, updated, and written. To avoid global synchronization, computations are often organized in pipelines where each stage uses local flow control to respond to backpressure. In this model, a multi-step computation can be organized as a single pipeline stage, which must complete all computational steps before accepting a new item to process. Alternatively, the steps can be split across multiple pipeline stages, which allows for multiple items to be processed in parallel. By controlling the number of stages over which a computation occurs, the circuit designer can trade resource usage (circuit size) for data processing latency. Flow control protocols are boilerplate that is easy to get subtly wrong, so they are an ideal candidate for abstraction. We describe our implementation of a pipeline abstraction in Section 4.

Our initial motivating example is a circuit that is useful only to demonstrate the challenges inherent in generating VHDL: a multi-step incrementer. This circuit takes a number as input and adds a constant n by repeatedly adding 1 in each of n steps, where each step requires a full clock cycle to complete. Two variants of the incrementer are useful in our investigation: a serial variant and a fully parallel variant. The serial variant executes all n steps in a single pipeline stage, so it has a latency of n clock cycles and can only process one item at a time, giving it a throughput of $1/n$ items per clock cycle. The parallel variant splits all n steps across n pipeline stages, so although it also has a latency of n clock cycles, it can process n items at a time for a steady-state throughput of 1 item per clock cycle. The parallel variant offers increased throughput at the expense of extra resource utilization since the hardware necessary to perform each step must be duplicated n times.

The boilerplate necessary for implementing a pipeline stage is something over which we want control so that we can implement a specific protocol, and it is something we don't want to have to write ourselves since it is mechanical but also error-prone. We can eliminate this boilerplate, but at what cost in terms of speed (clock rate) and (FPGA) resource utilization? Although the computation itself is simple, the multi-step incrementer performs a minimal but non-zero amount of work at each step, which allows us to investigate the overhead each environment imposes on pipeline construction in an attempt to answer this question. We compare implementations of the serial and (fully) parallel incrementer using three different code generation tools: Kansas Lava [12], CλaSH [4], and our library for metaprogramming with combinators (MWC).

Listing 1 shows the definitions of both the serial and parallel 16-step incrementers using our library. The type `UQ 16 0` is the type of unsigned fixed-point numbers with a 16-bit integral component and a 0-bit fractional component, i.e., unsigned 16-bit integers. Our library provides general support for computing with both unsigned (the `UQ` type) and signed (the `Q` type) fixed-point types in Haskell, where the type is indexed by type-level naturals specifying the number of integer and fractional bits. The `VExp` type is a lightweight type for representing VHDL expressions that is indexed by the Haskell type of the VHDL term it represents (see Section 3).

[1] https://github.com/mainland/vhdl-combinators-examples.

```
sinc :: Pipeline p
     => p (VExp (UQ 16 0)) (VExp (UQ 16 0))
sinc = siter id (\x _ -> x + 1) id 16

pinc :: Pipeline p
     => p (VExp (UQ 16 0)) (VExp (UQ 16 0))
pinc = piter id (\x _ -> x + 1) id 16
```

Listing 1. 16-step serial and parallel incrementers that operate on unsigned, 16-bit integers written using the metaprogramming with combinators technique.

Table 1. Resource requirements (LUTs and flip-flops) and maximum frequency of serial and parallel 16-step incrementer for 16 bit unsigned numbers.

Approach	LUT	FF	f_{MAX} (MHz)
MWC			
Serial	25	24	770.
Parallel	17	272	842
Kansas Lava			
Serial	44	39	712
Parallel	99	319	605
Kansas Lava (RTL)			
Serial	22	24	715
Parallel	55	278	485
CλaSH			
Serial	42	23	516
Parallel	64	303	785

The type class **Pipeline** represents a pipeline, so a value of type **Pipeline** p => p a b is a pipeline that consumes values of type a and produces values of type b (see Section 4). The siter and piter combinators iterate a function for a statically known (at compile time) number of steps and allow both a single pre- and post-processing step, both of which are unused in this example. The iterated function receives two arguments: the item to process and the index of the current step. The full pipeline interface is shown in Listing 6 and further described in Section 4. Combinators equivalent to siter and piter were also written in Kansas Lava and CλaSH.

Table 1 shows the resource utilization and maximum estimated frequency of four implementations of the 16-stage, unsigned 16-bit incrementer. Our methodology for measuring these quantities is fully described in Section 6. We attempted to implement combinators in each environment that are equivalent to siter and piter as efficiently as possible while still writing idiomatic code. For example, CλaSH allows VHDL primitives to be defined by the programmer, but we used the subset of Haskell that CλaSH can synthesize to VHDL to design serial and parallel pipeline combinators. We also wrote two Kansas Lava implementations. The first uses the high-level library functions provided by Kansas Lava. The second implementation, labeled "RTL" (for register transfer language) in Table 1, uses the low-level, stateful, monadic DSL that is provided in an extra module in the Kansas Lava library. This DSL corresponds closely to VHDL, so the RTL implementations are not much different from what one would write directly in VHDL.

The metaprogramming with combinators (MWC) library has the lowest resource requirements and highest estimated clock rate for both the serial and parallel versions of the circuit. We make several observations about our implementation efforts.

Writing high-level serial and parallel combinators was relatively painless in all environments. Most of our effort across all four implementations was directed towards coercing the environment to produce roughly the VHDL we knew would result in a reasonably performing pipeline. This was easy when using the MWC library because we could literally write the VHDL that we wanted. While this may seem to unfairly bias the example in favor of our library, a circuit designer generally knows what VHDL is needed. The challenge is not to figure out how to write low-level code efficiently, but to figure out how to *avoid* constantly re-writing small variations on known patterns because the language available (VHDL) doesn't provide the necessary abstraction mechanisms to eliminate boilerplate. High-level languages "obviously" provide the facilities needed to avoid boilerplate, and MWC "obviously" allows full control over generated code because the programmer can directly write the desired VHDL as part of the code generator. Thus, the real question is whether MWC can simultaneously provide both low-level control of generated code and high-level abstractions.

Kansas Lava provided more control over the generated VHDL, but we had to drop down to the monadic RTL DSL and essentially write VHDL anyway. Still, this approach didn't match the performance we attained using the MWC approach. The correspondence between the Kansas Lava source and generated VHDL was not always obvious, even when using the RTL DSL. In contrast, the VHDL generated by CλaSH was more concise, and its mapping from the Haskell source was clearer since it corresponds closely to GHC's core language. However, it was more difficult to control the generated VHDL in CλaSH—we could find no CλaSH analogue to Kansas Lava's RTL language. Because CλaSH primitives are template strings, there is no good way to generate them other that writing them directly by hand. Since they are stored in separate JSON-like files and are not part of the Haskell source of a program, they cannot be computed—they must be written by hand.

CλaSH also imposes a somewhat rigid structure on all generated VHDL code—each generated VHDL component has clock, clock enable, and reset signals (with fixed names). Although this structure is standard for VHDL code, it is not always what one wants, and we found that we could slightly

Table 2. Resource requirements and maximum frequency of serial and parallel 16-step incrementer for unsigned numbers with 8 integral bits and 8 fractional bits.

Approach	LUT	FF	f_{MAX} (MHz)
MWC			
Serial	27	24	635
Parallel	21	174	786
Kansas Lava			
Serial	47	74	589
Parallel	354	314	430
Kansas Lava (RTL)			
Serial	44	63	552
Parallel	323	544	386
CλaSH			
Serial	37	24	543
Parallel	80	288	446

improve resource utilization by using our own, alternative reset mechanism. Because this subversion of CλaSH's built-in reset mechanism is non-idiomatic and would prevent our circuit from being composed with other CλaSH circuits, we report the statistics from the idiomatic version of our CλaSH implementation. Local changes to Haskell source that would not normally change performance when compiled with GHC did cause performance changes in generated VHDL. For example, the step function to our iteration combinators takes both the value to step and the current step index, but even though the step index was not used in the incrementer example, the existence of a second, unused argument caused an increase in resource usage. The GHC optimizer did not eliminate this overhead.

Table 2 shows the resource usage and estimated maximum frequency of the incrementer circuits instantiated at a different type—the type of 16-bit unsigned numbers with 8 integral bits and 8 fractional bits. The disparity between MWC and other approaches is even greater in this case. We did nothing special to tune our implementation, but only used the standard VHDL libraries for fixed-point numbers when mapping the Haskell `Q` and `UQ` types to VHDL.

The incrementer example is a very small circuit. How does the overhead of CλaSH and Kansas Lava scale with circuit size? We are concerned with two types of "overhead": area (circuit size) and clock frequency. It is unclear how area overhead scales with circuit size in CλaSH because VHDL is generated from GHC's core language. Some Haskell constructs may result in GHC Core that translates to small VHDL circuits, and other Haskell constructs may not—part of the problem is that the programmer cannot predict how Haskell will map to VHDL. Kansas Lava programmers face a similar problem. In both Kansas Lava and CλaSH, there is some (approximately) fixed overhead *per pipeline stage*.

```
append [vunit|
  entity $id:entity is
    port (clk : in std_logic;
          rst : in std_logic;
          in_ready : out std_logic;
          in_valid : in std_logic;
          $idecls:in_idecls;
          out_ready : in std_logic;
          out_valid : out std_logic;
          $idecls:out_idecls);
  end;|]
```

Listing 2. A quasiquoted VHDL entity declaration. The syntax `[vunit|...|]` denotes a quasiquoted VHDL unit. The entity's identifier and additional input and outputs signals are antiquoted.

Since larger pipelined circuits consist of more pipeline stages rather than larger stages, this fixed overhead would grow with the size of a pipelined circuit.

Area overhead is important, but the real catch is clock frequency. Since a circuit can only run as fast as its critical path (the slowest computation than must complete in a single clock cycle), making the critical path slower has a global effect on speed. The incrementer example shows that even when the computation on the critical path is minimal (a single addition!), there is a substantial clock frequency penalty for Kansas Lava and CλaSH. That penalty is paid by any circuit, making any circuit, small or large, run more slowly. The incrementer example is arguably a best case for Kansas Lava and CλaSH because it does so little.

Metaprogramming with combinators allows fine control over the performance of generated code. This is not surprising since by design it allows direct control over generated code. We were nonetheless surprised by the amount of overhead Kansas Lava and CλaSH imposed on this simple circuit. In the following two sections, we show that gaining the ability to tightly control generated code does not require giving up many of the advantages of other approaches based on functional languages.

3 Embedding VHDL Terms in Haskell

The metaprogramming with combinators approach is built on quasiquotation [20], which allows concrete syntax for the language in which code is generated to be mixed with Haskell. This paper uses our implementation of a quasiquoter for VHDL. Quasiquoters exist for many other languages, e.g., C and Javascript. The MWC library also includes a code generation monad that provides scoping constructs, the ability to gensym names, and other operations typically needed during code generation.

Listing 2 shows a fragment of the pipeline code generator described in Section 4 that quasiquotes a VHDL entity declaration. The `append` function is aggregating generated code in

```
data VExp a where
  VConst :: ToExp a => a -> VExp a
  VExp   :: VHDL.Exp -> VExp a
```

Listing 3. A simplified version of the mixed shallow/deep VHDL term representation used by the metaprogramming with combinators library.

the code generation monad. The syntax `[vunit|...|]` specifies that the `vunit` quasiquoter, which parses VHDL unit declarations, should be used to parse the bracketed code that occurs in place of the ellipses. Within the quasiquoted code, expressions prefixed with the dollar symbol are *antiquotes*. For example, `$id:entity` is an antiquoted identifier—the value of the Haskell variable `entity` will be spliced into the generated code and used as the name of the declared VHDL entity. The antiquote `$idecls:in_idecls` will splice in the list of interface declarations bound to the Haskell variable `in_idecls`. In the quasiquoters provided by our VHDL quasiquotation library, concrete VHDL syntax is parsed to Haskell terms built from the data constructors of Haskell data types that represent VHDL abstract syntax. Quasiquotation is syntactic sugar—it allows programmers to use concrete syntax instead of deeply nested data constructor applications in both Haskell expressions and Haskell patterns. Quasiquotation and antiquotation provide a flexible framework for writing code templates using concrete syntax. We give a more complete description of how quasiquotation is used to write a pipeline combinator in Section 5 after first explaining the pipeline abstraction itself in Section 4.

If we represent circuits as VHDL terms, how can we hope to interpret and run them as Haskell programs? We can recover the ability to interpret a subset of circuits as Haskell programs by using a mixed shallow/deep term representation. Our library uses a **`VExp`** data type, indexed by the (Haskell) type of the VHDL term it represents, whose simplified definition is shown in Listing 3. The values of a type that is a member of the **`ToExp`** type class can be represented as VHDL terms. The **`VConst`** data constructor of the **`VExp`** type represents a shallowly-embedded term—a Haskell value that can be converted to a VHDL representation at code generation time. In contrast, the **`VExp`** data constructor represents an arbitrary, deeply-embedded VHDL term. Allowing both deeply- and shallowly-embedded representations in VHDL terms gives us the ability to partially evaluate expressions involving only compile-time constants "for free." In essence, we are using a small amount of abstract interpretation to move computation from generated code to the code generator. Kiselyov and Taha [17] use a similar representation to generate efficient FFT implementations.

If we maintain the invariant that functions whose arguments are all shallowly-embedded values evaluate to shallowly-embedded values, we can guarantee that computations involving only statically known values will be fully evaluated

```
class LiftEq f where
  (.==.) :: Eq a => f a -> f a -> f Bool
  (./=.) :: Eq a => f a -> f a -> f Bool
```

Listing 4. Lifted version of the **`Eq`** type class.

at code generation time. This invariant provides an additional benefit: it allows functions that operate on values of type **`VExp`** `a` to be interpreted as Haskell functions, not just as code generators, by injecting shallowly embedded values into the **`VExp`** data type before calling a function and projecting the result out of the **`VExp`** data type. If we always used a deep embedding, we would lose both of these advantages.

Separating shallow and deep operations is key to both partial evaluation and interpreting circuits as Haskell programs. We extend the classic technique for embedding a language in Haskell and providing instances of standard type classes, like **`Num`**, for embedded terms [10] by separating shallow and deep operations in different type classes. We also use the standard technique of providing lifted versions of many operations as separate type classes. For example, Listing 4 shows the lifted version of the **`Eq`** type class, which is needed because the type of equality is `(==) :: Eq a => a -> a -> Bool`, but in an embedded language the return type must be `f Bool`, where `f` is the type constructor for terms in the embedded language. The metaprogramming with combinators library provides lifted versions of many standard Haskell type classes, indexed by the type constructor of terms in the embedded language, as well as instances for the **`VExp`** data type. The **`VExp`** instances of the lifted classes handle partial evaluation when possible, and otherwise they delegate computation to separate classes representing the corresponding operations on deeply embedded values. These type classes are indexed not by the type constructor of terms in the embedded language, but by the *type index* of these terms. For example, there is a **`LiftBits VExp`** instance representing bit operations on VHDL terms of type **`VExp`** `a`, but there is a **`DeepBits Int`** instance representing bit operations on deeply embedded VHDL terms of type **`VExp Int`**. Stratifying operations in this way allows type-specific operations on deeply-embedded terms to be specified orthogonally from operations on values that contain constants.

Listing 5 shows the specification of a convolutional encoder pipeline using the metaprogramming with combinators library. The following points are notable:

- Sized types are used throughout. For example, the first argument to `encodeP` (line 2) is a vector of `r` unsigned $k+1$-bit numbers—the generator polynomials for the convolution code. The type statically states that this is a rate $1/r$ code with constraint length $k+1$. The type **`SLV`** `n` is a vector of n bits, which corresponds to the VHDL **`std_logic_vector`**.
- The generator polynomials are statically known at code generation time. However, the current encoder

```
1  encodeP :: forall r k p. (KnownNat r, KnownNat k, Pipeline p)
2          => Vec r (Unsigned (k+1)) -- ^ Generator polynomials
3          -> p (VExp Bit) (VExp (SLV r))
4  encodeP gs = moore step out (lift (zeroBits :: SLV (k+1)))
5    where
6      step :: VExp (SLV (k+1)) -> VExp Bit -> VExp (SLV (k+1))
7      step s i = V.tail s `V.snoc` i
8
9      out :: VExp (SLV (k+1)) -> VExp (SLV r)
10     out s = V.liftVec $ Vec.map (\g -> parity g s) gs
11
12 -- | Compute parity of bits using generator polynomial @g@.
13 parity :: forall k . KnownNat k
14        => Unsigned (k+1)   -- ^ Generator polynomial
15        -> VExp (SLV (k+1)) -- ^ Current state
16        -> VExp Bit
17 parity g state =
18     xorreduce $ msum [extract i | i <- [n-1,n-2..0]]
19   where
20     n :: Int
21     n = finiteBitSize (undefined :: Unsigned (k+1))
22
23     -- Reduce list of bits using xor
24     xorreduce :: [VExp Bit] -> VExp Bit
25     xorreduce = foldr1 xor'
26
27     -- Extract bit i of state if corresponding bit
28     -- is set in g
29     extract :: MonadPlus m => Int -> m (VExp Bit)
30     extract i = if testBit g i
31                 then pure (V.index state (lift i))
32                 else mzero
```

Listing 5. Specification of a rate $1/r$ convolutional encoder pipeline with constraint length $k + 1$ from r generator polynomials of $k + 1$ bits each using the metaprogramming with combinators approach.

state is a staged value—it can potentially be represented using a deeply-embedded VHDL expression—because it has the type `VExp (SLV (k+1))`.

- The programmer does need to track which values are potentially deeply embedded. For example, the xor reduction in line 25 must use the lifted version of the standard Haskell operation `xor`, which is a member of the Haskell type class **`Bits`**. The lifted version is named `xor'` and is part of the **`LiftBits`** class defined in the metaprogramming with combinators library. Haskell values must be explicitly lifted into the **`VExp`** data type, as in lines 4 and 31.
- The staged implementation of the encoder otherwise looks much like an unstaged version would. Staging the computation so that values that must have VHDL runtime representations have type **`VExp`** `a` gives us the ability to use this specification to generate VHDL code.
- We explain the **`Pipeline`** type class and `moore` function for building stateful pipelines in the following section.

Staging the encoder *does not* prevent us from running it as a pure Haskell function, as explained in the following section. Instead of directly writing VHDL, staged operations are abstracted using the **`VExp`** data type and lifted versions of standard Haskell type classes for expressing common operations. These abstractions maintain the invariant that computations with lifted Haskell constants produce lifted Haskell constants. This allows functions written using these abstractions to be run as pure Haskell functions in addition to being used as VHDL code generators. Writing the combinational portions of circuits in this style—without explicitly writing VHDL—is a great fit because combinational logic is expressible in the language of pure functions. For stateful circuits, we need a different approach.

4 Pipeline Combinators

For combinational logic, we regained the ability to interpret code-generating functions as pure Haskell functions by abstracting VHDL terms using the **`VExp`** data type and lifted type classes for common operations. We can do the same for stateful hardware circuits by introducing new abstractions. The stateful abstraction we introduce in this section is embodied by the **`Pipeline`** type class, whose definition is given in Listing 6. Although it has a narrow interface, the **`Pipeline`** type class provides enough functionality to write many useful stateful circuits. Moore and Mealy machines are very general forms of circuits, and both can be built using the **`Pipeline`** interface.

Although the metaprogramming with combinators library includes the definition of the **`Pipeline`** type class and two instances, one that generates VHDL and one that yields pure Haskell functions, this is the kind of abstraction that we expect programmers who use the metaprogramming with combinators approach to be able to build themselves. The VHDL-generating instance of the **`Pipeline`** type class is less than 1000 lines of code, which includes additional functionality we don't describe here. Building this type of abstraction provides the following advantages:

- The same source code expression of a stateful computation can be used to either generate code or as a pure Haskell function, which can be tested with tools like QuickCheck.
- The programmer still has full control over generated VHDL. The VHDL-generating instance of the **`Pipeline`** type class uses quasiquotation to produce exactly the VHDL code the programmer wants.
- Different **`Pipeline`** instances can be used to produce implementations that use different handshaking protocols without requiring any source code changes to the circuit. Our implementation uses a standard valid/ready protocol, but adding implementations of other protocols, like AXI4-Lite, would be straightforward.

```
class Category p => Pipeline p where
  -- | Sequencing pipelines
  seq :: p a b -> p b c -> p a c

  -- | "Arrow" pipeline
  arr :: (Pack a, Pack b) => (a -> b) -> p a b

  -- | Serial iteration pipeline
  siter :: (Pack a, Pack b, Pack c)
        => (a -> b)             -- ^ Pre-process
        -> (b -> VExp Int -> b) -- ^ Function to iterate
        -> (b -> c)             -- ^ Post-process
        -> Int                  -- ^ Number of iterations
        -> p a c

  -- | Parallel iteration pipeline
  piter :: (Pack a, Pack b, Pack c)
        => (a -> b)             -- ^ Pre-process
        -> (b -> VExp Int -> b) -- ^ Function to iterate
        -> (b -> c)             -- ^ Post-process
        -> Int                  -- ^ Number of iterations
        -> p a c

  -- | Moore machine pipeline
  moore :: (Pack s, Pack i, Pack o)
        => (s -> i -> s) -- ^ State transfer function
        -> (s -> o)      -- ^ Output function
        -> s             -- ^ Initial state
        -> p i o

  -- | Mealy machine pipeline
  mealy :: (Pack s, Pack i, Pack o)
        => (s -> i -> (s, o)) -- ^ State transfer function
        -> s                  -- ^ Initial state
        -> p i o
```

Listing 6. The pipeline interface.

- The programmer using the pipeline abstraction can focus on just the combinational logic needed to express a computation and does not have to worry about the details of the handshaking protocol being used.

The **Pipeline** interface shown in Listing 6 uses two features specific to staged code generation. The first is the **Pack** type class, which provides an interface for representing values of a given type using one or more VHDL expressions. A similar type class is used by Kansas Lava. The second feature specific to staged code generation is the use of the **VExp Int** type for the second argument to the iteration function in `siter` and `piter`. Use of this type ensures that the step index can be a staged value, i.e., the step function must be able to use a staged (symbolic) VHDL expression as the step index.

Pipelines can be represented as pure Haskell functions. A **Pipeline** instance using this representation is shown in Listing 7. The function `reifyP` in the listing allows a pipeline that operates on staged values to be converted to a pure Haskell function on *unstaged* values. Lifting a value of type `a` to a value of type **VExp** `a` is always possible, but projecting

```
newtype HaskPipeline a b = P { runP :: [a] -> [b] }

instance Category HaskPipeline where
  id = P id

  f . g = P (runP f . runP g)

instance Pipeline HaskPipeline where
  f `seq` g = P (runP g . runP f)

  arr f = P $ map f

  siter f g h n = P $ map (h . go 0 . f)
    where
      go i x | i == n    = x
             | otherwise = go (i+1) (g x (lift i))

  ...

reifyP :: ToExp a
       => HaskPipeline (VExp a) (VExp b)
       -> [a] -> [b]
```

Listing 7. An instance of the **Pipeline** type class where pipelines are represented as pure Haskell functions.

a value of type `a` from a value of type **VExp** `a` is only possible for shallow terms, so this function is necessarily partial. However, if the combinational code used to build the pipeline obeys the key invariant stated in Section 3—operations that involve only shallowly-embedded (constant) values produce shallowly-embedded values—then `reifyP` will return a total function. The MWC library maintains this invariant for all operations on values of type **VExp** `a`, so combinational code and code written to use the abstract **Pipeline** interface can always be interpreted as either a code generator or as a pure Haskell function, as in Kansas Lava and CλaSH.

A similar interface could (almost) be implemented in either Kansas Lava or CλaSH. However, neither of those environments provides the kind of fine control over generated VHDL needed to minimize overhead, as we demonstrated in Section 2. We were unable to implement this exact interface in Kansas Lava because Kansas Lava does not provide an explicit representation for embedded terms, like our **VExp** type. This prevented us from enforcing the requirement that the second argument to the step function passed to `siter` and `piter` must be able to take a staged value rather than a constant value as the step index. When generating VHDL for the serial version of the iteration combinator, the step index is a VHDL loop variable, so we must be able to represent it using a deep embedding.

Although there is no guarantee that the Haskell interpretation of a pipeline and the code generating interpretation have the same semantics in our approach, neither Kansas Lava nor CλaSH make this guarantee either. Our pipeline supports

```
1  piter :: forall a b c m . (Pack a, Pack b, Pack c, MonadCg m)
2        => Id                     -- ^ Name of VHDL entity
3        -> (a -> b)               -- ^ Pre-processing function
4        -> (b -> VExp Int -> m b) -- ^ Function to iterate
5        -> (b -> c)               -- ^ Post-processing function
6        -> [String]               -- ^ (Optional) input names
7        -> [String]               -- ^ (Optional) state names
8        -> [String]               -- ^ (Optional) output names
9        -> Int                    -- ^ Number of iterations
10       -> m (Pipeline a c)
11 piter entity f g h in_names snames out_names n = do
12   (_ :: a, in_vars)    <- genPack in_names
13   (_ :: b, state_vars) <- genPack snames
14   (_ :: c, out_vars)   <- genPack out_names
   :
23   let in_idecls  = [[videcl|$id:(in_ v) : in $ty:tau|]
24                        | (v, tau) <- in_vars]
25       out_idecls = [[videcl|$id:(out_ v) : out $ty:tau|]
26                        | (v, tau) <- out_vars]
27
28   append [vunit|
29     entity $id:entity is
30       port (clk : in std_logic;
31             rst : in std_logic;
32             in_ready : out std_logic;
33             in_valid : in std_logic;
34             $idecls:in_idecls;
35             out_ready : in std_logic;
36             out_valid : out std_logic;
37             $idecls:out_idecls);
38     end;|]
39
40   withArchitecture "behavioral" (toName entity noLoc) $ do
   :
51     withProcess ["clk"] $ do
52       onRisingEdge $ do
53         if [vexp|rst = '1'|]
54           then do
55             append [vstm|valid <= (others => '0');|]
56             append [vstm|ready <= (others => '1');|]
57           else do
   :
65             forS "i" [vrange|1 to $(n-1)|] $ \i -> do
66               when [vexp|arrname ready($i)|] $ do
67                 x <- renamePack
68                        (\v -> [vexp|arrname $id:v($i-1)|])
69                        state_vars
70                 g x (VExp i) >>=
71                   sigassignPack
72                     (\v -> [vname|arrname $id:v($i)|])
73                     state_vars
74                 append [vstm|valid($i) <=
75                                 arrname valid($i-1);|]
   :
90     append [vcstm|out_valid <= valid(valid'high);|]
91
92   return Pipeline { ... }
```

Listing 8. Implementation of the `piter` combinator for the instance of the **`Pipeline`** class that generates VHDL.

testbench generation, which allows the same pipeline to be tested either as a pure Haskell function using QuickCheck, or as a compiled VHDL implementation. In both cases, random inputs can be generated using QuickCheck **`Arbitrary`** instances, but when testing the VHDL implementation, these values are serialized so that they can be read by the generated VHDL testbench. Pipelines can be developed purely in Haskell, and the same source program can then be used to generate VHDL that is automatically tested. We used this approach—develop in Haskell and test generated VHDL only when Haskell development is complete—to write implementations of several IP blocks in Section 6.

5 Implementing a Pipeline Combinator

Listing 8 shows the implementation of the `piter` combinator for the VHDL-generating instance of the **`Pipeline`** abstraction, which is written using our VHDL quasiquoter. We have removed code that does not illustrate features beyond those already shown in the included code. The entire implementation is fewer than 100 lines, including the lines that we have removed for this presentation.

The first thing to note is that the code is not just a giant VHDL code template—it looks like standard, structured Haskell with some quasiquotations thrown in. Our VHDL code generation library includes a large set of code-generating combinators that allow the programmer to write Haskell syntax instead of quoted VHDL. The mapping between the Haskell combinators and the VHDL they generate is direct and easy for the programmer to reason about; the combinators are syntactic sugar. These combinators rely on a code generation monad whose operations are embodied by the **`MonadCg`** type class. This monad provides features like name generation (gensym) and the collection of generated code. We will walk through the implementation of `piter`, pointing out interesting features and combinators as we go.

Lines 12–22 generate VHDL bindings for the input, output, and internal state of the component. The **`Pack`** type class allows easy conversion between a (possibly nested) structured value and a flattened sequence of VHDL terms. The former is useful when writing Haskell, and the latter is useful when generating VHDL. For example, generating VHDL to assign a tuple to a variable is simple when the value and binder can both be unpacked into their constituent parts.

Pipelines can consume and produce structured values, but the VHDL generated from the pipeline always flattens these structured values. Lines 23–38 declare the VHDL entity for the pipeline and its input and output signals. This entity declaration is the same shown in Listing 2. The `append` combinator in line 28 appends the generated entity declaration to the rest of the code collected by the code generation monad.

The next few lines show the use of scoped code generation combinators. The `withArchitecture` and `withProcess` combinators generate a declaration for a VHDL architecture and

process, respectively. The final argument to these two combinators is a monadic action, which itself generates code. This code is collected by the combinator and becomes the body of the architecture/process. This style of scoped, monadic combinator enables modularity, and we find it vastly easier to use than direct quasiquotation.

We make use of GHC's rebindable syntax to overload **`if`** expressions in lines 53–75. This allows the programmer to use standard Haskell **`if`** syntax to generate code for a VHDL **`if`** statement. Instead of a value of type **`Bool`**, the conditional is a VHDL expression—`vexp` is a quasiquoter for VHDL expressions. The branches of the **`if`** expression are both monadic actions that generate code. The overloaded **`if`** uses the conditional and the code generated by the branches to construct the corresponding VHDL statement. Similarly, in line 66, the standard `when` combinator is substituted with an overloaded version that can take a VHDL conditional.

Quasiquotations are not hygienic. However, hygiene can be recovered by using combinators. Line 65 builds a **`for`** statement using the `forS` combinator, which generates a fresh binder to use as the loop variable. The first argument to `forS` is a suggested name meant to improve the readability of the generated code. The implementation of `forS` uses the code generation monad's gensym functionality to create a fresh name. Nothing prevents the programmer from introducing a name clash with the monad's gensym, but doing so requires intent. In practice, hygiene can be maintained through consistent use of gensym and combinators like `forS`.

Code-generation combinators provide a thin veneer on top of quasiquotation that reduces the impedance mismatch between the quasiquoted language and Haskell. Because they are such a shallow abstraction, the programmer knows exactly what code will be produced. There is no downside: combinators make code generators easier to write without introducing layers of abstraction that make it difficult to reason about what code will be generated. When an appropriate combinator cannot be written, the programmer can always drop down to raw quasiquotation, which provides absolute control over generated code.

6 Evaluation

We are concerned with evaluating both the code generated by programs using the metaprogramming with combinators approach and the ease with which this approach can be applied. We first evaluate the resource usage and maximum clock speed of three commonly used IP blocks written using our approach: a convolutional encoder, a divider, and CORDIC. Equivalent IP blocks were generated using the Xilinx IP Core tools. We describe each of the three in turn after outlining our measurement methodology. We then provide a short evaluation of the programmer effort required to implement the pipeline abstraction used in the incrementer example (Section 2) using CλaSH, Kansas Lava, and the MWC approach.

6.1 Performance of Generated Code

Methodology. All results reported in this paper were collected using Vivado 2018.3. Circuits were synthesized out-of-context targeting the Virtex-7 FPGA used on the Xilinx VC707 evaluation board, and the Explore directive was used to optimize placement and routing. Measurements were performed in batch mode to facilitate reproducibility, and the repository holding our implementations includes tools for reproducing the metrics we report. Maximum frequency (f_{MAX}) was calculated based on worst negative slack estimates using the standard formula $f_{\text{MAX}} = 1/(t - t_{\text{WNS}})$, where t is the circuit clock rate and t_{WNS} is Vivado's estimate of the circuit's worst negative slack. In addition to f_{MAX}, we report the number of lookup tables (LUT) and flip-flops (FF), i.e., registers, needed to implement each circuit.

Convolutional encoder. The full convolutional encoder is shown in Listing 5. It operates on a window of $k + 1$ bits, where $k \geq 0$. The parameter $k + 1$, which must be positive, is the convolutional code's *constraint length*. The convolutional code is also parametrized by r generator polynomials, each of $k + 1$ bits. For each input bit, the generator polynomials are convolved with the most recent $k + 1$ bits, and the results of the r convolutions are output. Convolution is performed in $\mathbb{F}_2$, i.e., using xor. This produces a rate $1/r$ code, since each input bit requires r bits to be transmitted. Our IP generator supports an arbitrary constraint length and arbitrary generator polynomials, but it does not support some features that the Xilinx IP core generator supports, like puncturing. These features would be easy to add. The Xilinx generator, on the other hand, only supports rates up to $1/7$. The convolutional encoder we use for comparison has constraint size $k + 1 = 7$ and rate $1/2$, with generator polynomials $(121, 91)$.

Divider. Division is the most complex and costly basic arithmetic operation in hardware circuits. For comparison, we generated a fully parallel pipeline that performs non-restoring division on 16-bit unsigned numbers using both our library and the Xilinx IP core tools. Our implementation uses the `piter` combinator from the **`Pipeline`** type class, and we could have generated a serial implementation, which would have lower throughput but use fewer resources, by changing only the combinator used from `piter` to `siter`.

CORDIC. We implemented a generalized version of the CORDIC algorithm [1, 16, 18, 19, 24, 25] that can compute trigonometric functions, hyperbolic functions, exponentiation, and logarithms of an arbitrary base. This circuit also uses the **`Pipeline`** iteration combinators. As with the divider, throughput and resource usage can be traded off by switching between the `siter` and `piter` combinators. We show metrics for an instance of CORDIC that simultaneously calculates the sine and cosine of a 32 bit quantity.

Table 3. Performance results of IP cores generated using the metaprogramming with combinators approach and equivalent Xilinx IP.

IP Core	LUT	FF	f_{MAX} (MHz)
Convolution Encoder			
Xilinx IP	31	26	683
MWC	11	14	892
Divider			
Xilinx IP	377	943	666
MWC	353	312	675
CORDIC			
Xilinx IP	3663	3681	516
MWC	2601	3108	609

Table 3 shows the resources required and maximum speed of the three IP blocks we use for comparison. The performance of the circuits we generated is comparable to, and usually superior to, the performance of the equivalent Xilinx blocks. Because the Xilinx tools generate (encrypted) RTL instead of VHDL, we can only speculate as to why our implementations perform better. One factor in our favor is that we can precisely tailor the representations used. For example, the Xilinx convolutional encoder always takes input bits as an 8-bit signal and produces output bits packed into an 8 bit quantity. The extra wasted wires require more hardware, whereas we produce signals that are only as wide as needed. If we wanted to use our generated convolutional encoder as a drop-in replacement for a Xilinx encoder, we could write pipeline stages that act as adapters to widen the inputs and outputs of our encoder.

Another possible source of inefficiency in the Xilinx blocks is their general support for the AXI4-Lite protocol. Although we configured the Xilinx generators to only produce the signals required for the valid/ready-style subset of AXI4-Lite, we expect that eliminating some AXI4-Lite signals in the IP core specification does not completely eliminate the associated circuitry in the generated IP block. We could generate AXI4-Lite versions of our circuits by writing an appropriate `Pipeline` type class instance—no changes to our IP implementations would be necessary. This is another advantage of using the pipeline abstraction to organize our implementations.

6.2 Programmer Effort

Table 4 shows the lines of code needed to implement the pipeline abstraction used in the incrementer example from Section 2. We tailored the CλaSH and Kansas Lava pipeline combinators to the incrementer problem, whereas the MWC combinators were taken from the general pipeline implementation described in Section 4. These results show that the MWC approach requires about twice as much code as either CλaSH or Kansas Lava. Anecdotally, this is consistent with our general experience. It is also expected, since the MWC approach generates VHDL directly, whereas both CλaSH and Kansas Lava provide additional layers of abstraction that insulate the programmer from dealing directly with VHDL. VHDL tends to be verbose, which inflates the number of lines of code needed for a quasiquotation-based code generator. Using code generation combinators helps to mitigate this cost—our pipeline implementation would have been much larger (and rather unwieldy) if we had not used code generation combinators as shown in Section 5. One goal of the metaprogramming with combinators approach is to isolate code generation complexity through reusable code generation combinators and abstractions like `Pipeline`, moving it to reusable libraries where its cost can be amortized across what is hopefully many uses of the library.

Table 4. Lines of code required to implement and use the pipeline abstraction for the incrementer example using CλaSH, Kansas Lava, and MWC. The Kansas Lava implementation uses the RTL DSL.

Approach	Implementation Effort (LOC)
CλaSH	63
Kansas Lava	73
MWC	141

7 Related Work

The connection between functional languages and hardware has a long history that can be traced back to μFP [22], a functional language used to describe both the behavior and the layout of circuits. Control over circuit layout was inspired by Henderson's elegant formulation of functional geometry [14, 15]. A more recent example of a language for controlling circuit layout is Wired [2].

Embedded domain-specific languages. The prototypical Haskell embedded domain-specific language for hardware is Lava [6]. Kansas Lava [12] is a more recent incarnation that leverages modern language features, like type-level naturals. The distinction between shallow and deep embeddings was originally clarified by Boulton et al. [7] in the context of a hardware EDSL embedded in HOL. Svenningsson and Axelsson [23] note the advantages of a mixed deep and shallow embedding, but their approach is relatively heavyweight and is trying to solve a more general problem—we only want to ensure that we can partially evaluate as many terms as possible. Our approach to mixing deep and shallow embeddings is similar to that of Kiselyov and Taha [17].

Chisel [5] is an EDSL embedded in Scala that is used to design RISC-V processor cores. Circuits are specified as directed graphs where each node is an operator that receives zero or more inputs and produces one output. Expressions

are converted to a circuit tree in which the wires are named at the leaves and operators are named at internal nodes. The root of the tree represents the value of the expression.

Lightweight Modular Staging (LMS) [21] adds staging support to Scala programs, allowing programmers to explicitly mark which portions of a program are staged through type annotations. The staged and unstaged portions of a program are both written in Scala, which is one of the many advantages of LMS. LMS is a system for homogeneous metaprogramming, where the code generator and the staged language are the same. In contrast, MWC explicitly supports *heterogeneous* metaprogramming, where the code generator and the staged language are *different*. If the target language is similar to Scala, LMS is a great fit. However, if there is not a close correspondence between the target language and Scala, the programmer may have difficulty reasoning about (and controlling) how the staged code they write maps to generated code. For fine control over generated VHDL, MWC seems to be a better fit. The LMS approach has been used to generate hardware [11], but in that case the DSL in question expresses programs using familiar functional operations like zips, maps, and folds, which are an excellent fit for a language like Scala.

High-level synthesis. CλaSH [3, 4] translates a subset of Haskell to hardware. It is not able to translate recursive functions and has no support for side effects. There is no clear separation between compile-time and runtime execution. CλaSH, like Vivado's HLS, which compiles C (with many annotations) to hardware, forces the programmer to reason about how source code is translated to hardware. This mapping is not always apparent—or possible—and if the translation is not what the programmer wants or expects, there is not always a way to "fix" the translation without resorting to writing VHDL or Verilog. CλaSH allows the user to specify new VHDL, Verilog, or System Verilog primitives, but use of such primitives prevents CλaSH programs from being interpreted as pure Haskell functions. Primitives are described by special source files using a JSON-like format.

Handshaking. Gill and Neuenschwander [13] build a set of types and combinators to make the construction of hardware easier. The handshaking protocol between components is represented using a **`Patch`** type. The laws applicable to patches are defined so that large and complex dataflows can be managed and connected to the interfaces of the design. The **`Enabled`** protocol is responsible for valid control signal of the data stream. To complete the handshaking between two components, an **`Ack`** (acknowledgement signal) is also introduced, and this scheme works with the Wishbone handshaking protocol. A **`Patch`** is a circuit or stream processor between a pair of protocols. Patches can be bridges between protocols, or they can be computational. These patches can be chained together in the system. Combinators are developed for lifting functions into the patch domain and executing and composing patches. However, only combinational (pure) functions can be lifted to the **`Patch`** type.

Compositional pipelines. Edwards et al. [9] discuss the composition of data-dependent actors for constructing dataflow networks and their implementation in hardware. They demonstrate an implementation of nondeterministic merge and show how to break long combinational paths and loops with the help of data network buffers and backpressure. Our **`Pipeline`** abstraction offers similar functionality.

8 Conclusions

Our work began with the observation that if we want full control over the performance of generated code, we need full control of the generated code itself. GHC's support for quasiquotation seemed to provide a reasonable path to writing abstractions that weren't a horror show of (untyped) string templates or (typed) massively nested abstract syntax data constructors. We did not expect that this approach to metaprogramming could allow functions to be interpreted either directly as Haskell or as code generators, a wonderful feature of classic Haskell domain-specific language embeddings like Kansas Lava.

The power of the metaprogramming with combinators approach is built on two fundamental insights. First, **provide an embedded language for the purely functional subset of the language that does not require the programmer to directly write code in the language being generated**. In the hardware domain, this subset is combinational logic, which is a very good match for a purely functional language. We represent combinational logic using the **`VExp`** type, and the correspondence between combinational logic written using **`VExp`** and the generated VHDL is very close, so a programmer almost always knows what VHDL will be produced. Our experience is that it is easier to write combinational logic in Haskell because it is less verbose and more readily admits common functional patterns of computation.

Second, **design abstractions that separate stateful and purely functional computation by offering combinators for patterns of stateful computation that are parametrized by pure functions**. This is exactly what the **`Pipeline`** abstraction in Section 4 does. Separating computation in this way allows stateful computations to be interpreted either as pure Haskell functions or as hardware generators. Rather than providing a canned set of such abstractions as a library, our expectation is that programmers will define new such abstractions themselves. Quasiquotation makes this prospect palatable, although this requires a quasiquoter for the language one wants to generate.

The metaprogramming with combinators approach provides fine control over generated code without sacrificing the ability to build sophisticated new abstractions or the

ability to run a code generator as a pure Haskell function. Although we have demonstrated our approach in the hardware domain, we expect that these same lessons can be used to build code generators for many other domains.

Acknowledgments

We are grateful to the anonymous reviewers for their many helpful suggestions. This work is supported by the National Science Foundation under Grant No. CCF-1717088.

References

[1] Ray Andraka. 1998. A Survey of CORDIC Algorithms for FPGA Based Computers. In *Proceedings of the ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*. ACM, Monterey, CA, 191–200. https://doi.org/10.1145/275107.275139

[2] Emil Axelsson, Koen Claessen, and Mary Sheeran. 2005. Wired: Wire-Aware Circuit Design. In *Correct Hardware Design and Verification Methods (CHARME '05)*, Dominique Borrione and Wolfgang Paul (Eds.). Number 3725 in Lecture Notes in Computer Science. Springer, Saarbrücken, Germany, 5–19. https://doi.org/10.1007/11560548_4

[3] Christiaan Baaij. 2009. *CλasH: From Haskell to Hardware*. Master's thesis. Universiteit Twente. http://essay.utwente.nl/59482/

[4] Christiaan Baaij. 2015. *Digital Circuits in CλaSH: Functional Specifications and Type-Directed Synthesis*. Ph. D. Dissertation. Universiteit Twente. https://doi.org/10.3990/1.9789036538039

[5] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: Constructing Hardware in a Scala Embedded Language. In *Proceedings of the 49th ACM/EDAC/IEEE Design Automation Conference*. San Francisco, CA, 1212–1221. https://doi.org/10.1145/2228360.2228584

[6] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. 1998. Lava: Hardware Design in Haskell. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*. Baltimore, MD, 174–184. https://doi.org/10.1145/289423.289440

[7] Richard Boulton, Andrew Gordon, Mike Gordon, John Harrison, John Herbert, and John Van Tassel. 1993. Experience with Embedding Hardware Description Languages in HOL. In *Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience (IFIP Transactions A: Computer Science and Technology, Vol. A-10)*. 129–156.

[8] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. ACM, Montreal, Canada, 268–279. https://doi.org/10.1145/351240.351266

[9] Stephen A. Edwards, Richard Townsend, Martha Barker, and Martha A. Kim. 2019. Compositional Dataflow Circuits. *ACM Transactions on Embedded Computing Systems* 18, 1 (Jan. 2019), 5:1–5:27. https://doi.org/10.1145/3274280

[10] Conal Elliott, Sigbjørn Finne, and Oege de Moor. 2003. Compiling Embedded Languages. *Journal of Functional Programming* 13, 3 (2003), 455–481. https://doi.org/10.1017/S0956796802004574

[11] Nithin George, HyoukJoong Lee, David Novo, Tiark Rompf, Kevin J. Brown, Arvind K. Sujeeth, Martin Odersky, Kunle Olukotun, and Paolo Ienne. 2014. Hardware System Synthesis from Domain-Specific Languages. In *24th International Conference on Field Programmable Logic and Applications (FPL '14)*. 1–8. https://doi.org/10.1109/FPL.2014.6927454

[12] Andy Gill, Tristan Bull, Garrin Kimmell, Erik Perrins, Ed Komp, and Brett Werling. 2009. Introducing Kansas Lava. In *Implementation and Application of Functional Languages*. Springer, Berlin, Heidelberg, 18–35. https://doi.org/10.1007/978-3-642-16478-1_2

[13] Andy Gill and Bowe Neuenschwander. 2012. Handshaking in Kansas Lava Using Patch Logic. In *Practical Aspects of Declarative Languages (PADL '12)*. Springer, Berlin, Heidelberg, 212–226. https://doi.org/10.1007/978-3-642-27694-1_16

[14] Peter Henderson. 1982. Functional Geometry. In *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming*. ACM, Pittsburgh, Pennsylvania, USA, 179–187. https://doi.org/10.1145/800068.802148

[15] Peter Henderson. 2002. Functional Geometry. *Higher-Order and Symbolic Computation* 15, 4 (Dec. 2002), 349–365. https://doi.org/10.1023/A:1022986521797

[16] X. Hu, R.G. Harber, and S.C. Bass. 1991. Expanding the Range of Convergence of the CORDIC Algorithm. *IEEE Trans. Comput.* 40, 1 (Jan. 1991), 13–21. https://doi.org/10.1109/12.67316

[17] Oleg Kiselyov and Walid Taha. 2004. Relating FFTW and Split-Radix. In *Proceedings of the First International Conference on Embedded Software and Systems (ICESS '04)*. Lecture Notes in Computer Science, Vol. 3605. Springer, Hangzhou, China, 488–493. https://doi.org/10.1007/11535409_71

[18] Yuanyong Luo, Yuxuan Wang, Yajun Ha, Zhongfeng Wang, Siyuan Chen, and Hongbing Pan. 2019. Corrections to "Generalized Hyperbolic CORDIC and Its Logarithmic and Exponential Computation with Arbitrary Fixed Base". *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27, 9 (Sept. 2019), 2222–2222. https://doi.org/10.1109/TVLSI.2019.2932174

[19] Yuanyong Luo, Yuxuan Wang, Yajun Ha, Zhongfeng Wang, Siyuan Chen, and Hongbing Pan. 2019. Generalized Hyperbolic CORDIC and Its Logarithmic and Exponential Computation with Arbitrary Fixed Base. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27, 9 (Sept. 2019), 2156–2169. https://doi.org/10.1109/TVLSI.2019.2919557

[20] Geoffrey Mainland. 2007. Why It's Nice to Be Quoted: Quasiquoting for Haskell. In *Proceedings of the ACM SIGPLAN 2007 Haskell Workshop (Haskell '07)*. Freiburg, Germany, 73–82. https://doi.org/10.1145/1291201.1291211

[21] Tiark Rompf and Martin Odersky. 2012. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. *Commun. ACM* 55, 6 (June 2012), 121–130. https://doi.org/10.1145/2184319.2184345

[22] Mary Sheeran. 1984. muFP, a Language for VLSI Design. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*. Austin, TX, 104–112. https://doi.org/10.1145/800055.802026

[23] Josef Svenningsson and Emil Axelsson. 2012. Combining Deep and Shallow Embedding for EDSL. In *Proceedings of the 2012 Conference on Trends in Functional Programming*, Vol. 7829. Springer, St. Andrews, UK, 21–36. https://doi.org/10.1007/978-3-642-40447-4_2

[24] J. E. Volder. 1959. The CORDIC Trigonometric Computing Technique. *IRE Transactions on Electronic Computers* EC-8, 3 (Sept. 1959), 330–334. https://doi.org/10.1109/TEC.1959.5222693

[25] J. S. Walther. 1971. A Unified Algorithm for Elementary Functions. In *Proceedings of the 1971 Spring Joint Computer Conference (AFIPS '71 (Spring))*. AFIPS Press, Atlantic City, NJ, 379–385. https://doi.org/10.1145/1478786.1478840