Hardware Software Co-design in Haskell

Markus Aronsson
Mary Sheeran
Chalmers University of Technology
Sweden

Abstract

We present a library in Haskell for programming Field Programmable Gate Arrays (FPGAs), including hardware software co-design. Code for software (in C) and hardware (in VHDL) is generated from a single program, along with the code to support communication between hardware and software. We present type-based techniques for the simultaneous implementation of more than one embedded domain specific language (EDSL). We build upon a generic representation of imperative programs that is loosely coupled to instruction and expression types, allowing the individual parts to be developed and improved separately. Code generation is implemented as a series of translations between progressively smaller, typed EDSLs, safeguarding against errors that arise in untyped translations. Initial case studies show promising performance.

CCS Concepts • **Software and its engineering** → **Functional languages**; **Domain specific languages**; • **Hardware** → *Reconfigurable logic and FPGAs*;

Keywords hardware software co-design, domain specific language

ACM Reference format:
Markus Aronsson and Mary Sheeran. 2017. Hardware Software Co-design in Haskell. In *Proceedings of 10th ACM SIGPLAN International Haskell Symposium, Oxford, UK, September 7-8, 2017 (Haskell'17),* 12 pages.
DOI: 10.1145/3122955.3122970

1 Introduction

Field Programmable Gate Arrays (FPGAs) represent an interesting point on the spectrum between general purpose processors and application specific integrated circuits (ASICs). An ASIC implements a fixed function, while an FPGA can be reprogrammed again and again, giving varying functionality over time. Such a programmable circuit typically consists of a large array of configurable logic blocks, connected by programmable interconnects. However, modern FPGAs also contain various discrete components, such as blocks of RAM, digital signal processing slices, and processor cores. Indeed, a modern FPGA can be viewed as a prototypical *heterogeneous system* that mixes components (such as processors and logic blocks) that need to be programmed in different ways.

FPGAs can combine the benefits of highly parallel, high performance hardware-based systems with the programmability of software, and at an attractive price point. In both embedded systems and high performance computing, FPGAs are increasingly combined with accelerators such as graphics processing units, and with multicores. And soon, there will be FPGAs, GPUs and CPUs all on one chip. The rise of FPGAs, while inexorable, has been slowed by the fact that they are difficult to program. The logic blocks are usually programmed in a hardware description language (HDL) such as VHDL or Verilog, while the embedded cores are typically programmed in a low level dialect of C. This mixture of hardware and software development is often called *co-design* (Teich 2012). The developer must specify both hardware and software parts, and how they communicate; ideally she would like to experiment with various choices of what to put in hardware and what in software. Current tools provide little support for such design exploration.

This paper presents first steps towards a system in which the entire design process, including exploration to decide where the boundary between hardware and software should be, is done in Haskell. We aim to give the user a reasonable degree of control over the generated code, both in hardware and software, by generating C and VHDL and making use of a standard Register Transfer Level (rather than high level) synthesis tool to produce the final hardware implementation. We build upon earlier work on code-generating embedded domain specific languages (EDSLs) (Axelsson 2016), and consider how to work with more than one EDSL at a time, in this case one for hardware and one for software. This paper provides methods that can be used by others who wish to combine EDSLs.

Our longer term aim is to use similar methods to program heterogeneous systems containing multicores and more than one type of accelerator (for example FPGAs and GPUs). We regard this work as a first step towards that larger goal and this is why we felt the need to develop a general approach to combining EDSLs, rather than a one-off combination of two particular EDSLs.

The paper makes the following contributions:

- We present a library for programming FPGAs in Haskell that supports hardware software co-design, generating both VHDL and C, including necessary connections between hardware and software for the Xilinx Zynq system-on-chip.
- We demonstrate the use of the library on a cryptographic example, and benchmark software and mixed hardware-software implementations. Repartitioning is far easier than it would be if the parts were written directly in C and VHDL.
- We present type-based techniques for the implementation of more than one EDSL. We build upon a generic representation of imperative programs that is loosely coupled to instruction and expression types, allowing the individual parts to be developed and improved separately.
- We implement code generation by a series of translations between progressively smaller, typed EDSLs, safeguarding against errors that arise in untyped translations.

Haskell'17, Oxford, UK
© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM. 978-1-4503-5182-9/17/09...$15.00
DOI: 10.1145/3122955.3122970

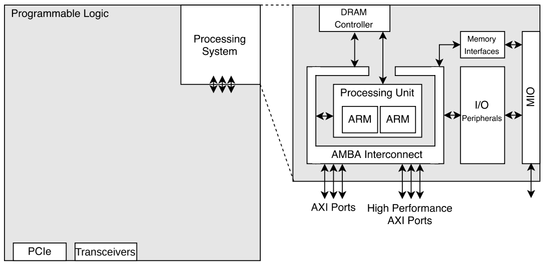


Figure 1. Overview of a Zynq system-on-chip (SoC).

2 A Co-design Language

Figure 1 shows the main parts of the Zynq system-on-chip. For our purposes here, the important parts are the ARM cores (of which we will use one, programmed in C), the programmable logic (which we will program by synthesis from VHDL) and the AXI ports that connect the two. So, starting with a single Haskell program, we have three tasks: generate C for the software parts, generate VHDL for the hardware parts, and generate a combination of C and VHDL that controls the handshaking and communication across the AXI ports to allow the software and hardware parts to communicate. We will return to this last task in section 4.

C and VHDL are both imperative. To represent them, we build on recent work on EDSLs as deep embeddings of monads (Axelsson 2016; Bracker and Gill 2014; Hickey et al. 2014; Svenningsson and Svensson 2013). The general idea is that one can view an imperative program as a sequence of instructions to be executed on some machine, which looks similar to programs written in a stateful monad. In fact, a stateful program composed with monadic operations can be directly translated into statements in an imperative language. Our hardware and software languages are both based on such a deep embedding of imperative programs:

```
data Prog ins exp a
instance Monad (Prog ins exp)
```

Having `Prog` parameterized on the instruction type, `ins`, and the expression type, `exp`, lets us define the software and hardware monads as programs, parameterized on their respective types:

```
type Software = Prog SIns SExp
type Hardware = Prog HIns HExp
```

As an example, a program for reading an integer from a stack and then putting it back again can be written in do-notation as:

```
reput :: (Monad m, Stack m) ⇒ m ()
reput = do x ← get; put x
```

`get` and `put` are brought into scope by the `Stack` constraint on the monad `m`, which ensures that any instantiation of `m` will support the two aforementioned stack operations. In the case of the stack example, software and hardware both support its operations and it can thus be interpreted in both monads.

A language embedding based on monads gives us a representation of statements in an imperative program, but most meaningful programs also include a notion of pure expressions. The expression type is, however, not readily available in the type signature of a program like `reput`, as it only refers to the monad `m`. We therefore use a type family `Exp` to retrieve the expression type of a program, which lets us combine instruction and expression constraints:

```
add :: (Monad m, Stack m, Num (Exp m)) ⇒ m ()
add = do x ← get; y ← get; put (x + y)
```

While `Software` and `Hardware` both support the previous stack operations, we can also point out groups of operations which are only supported by some interpretations but not by others, thus forming a hierarchy of classes. Monads form the base of the hierarchy, but operations intended for either software or hardware branches also require that their monad is an extension of their respective `Software` and `Hardware` monads. We call these monads `SoftwareM` and `HardwareM`, and they provide the language specific operations from their respective monads. For example, using the `SoftwareM` constraint, we can write a stack program that prints its top element to standard output:

```
print :: (SoftwareM m, Stack m, Printf m) ⇒ m ()
print = do x ← get; printf "%d" x
```

`print` is, according to its constraints, an entirely software based program; VHDL lacks an instruction that corresponds to the `printf` function found in C.

The interesting aspect of programming for the Zynq is that we can mix hardware and software fragments to form a whole program. As an example, we create small hardware program for the multiplication of two unsigned bytes:

```
multiply :: HExp Word8 → HExp Word8 → HExp Word8
multiply a b = a * b
```

To make the multiplier accessible from the software side, we turn it into a component called `multComp`:

```
multComp :: Component (Word8 → Word8 → Word8)
multComp = component $
  inp $ λa → inp $ λb → out $ multiply a b
```

where `inp` and `out` capture the type of the multiplier, which we normally cannot inspect in Haskell, and turn the type into a signature of the multiplier (Axelsson and Persson 2015). This signature is then encapsulated by `component` to yield a hardware program that can be invoked from software through an AXI channel or from hardware. Then, after synthesizing and placing the component on the FPGA, we get its physical address and use memory-mapped I/O to access the component from software:

```
soft :: SExp Word8 → SExp Word8 → Software ()
soft a b = do mult ← mmap "0x38C00000" multComp
              rout ← call mult (a .: b .: nil)
              printf "%d" rout
```

Here, the `mmap` function implements the necessary memory-mapping for creating a pointer to the multiplier at address “0x38C0000”, from the normally virtual memory space of programs running in software on the Zynq (which commonly has Linux installed). The pointer is used by `call`, which writes its arguments `a` and `b` to the addresses of the multiplier arguments, and then listens for the result. While software interacts with a component through pointers, writes to these pointers spawn corresponding actions on the AXI channels. The AXI controller intercepts these actions and forwards them to the addressed hardware component, which results in the component, possibly, responding over AXI.

Having components and their arguments be typed does ensure some type safety when interacting with a component, assuming the correct address has been given. The manual input of addresses comes as a consequence of having to go through the design tools and synthesize a component before we can get its physical address.

The following section presents a larger example, a software implementation of a standard function from cryptography.

2.1 Introducing the PBKDF2 example

In most applications of public-key cryptography, user security is ultimately dependent on secret texts or passwords. These secrets are often not directly applicable as keys to conventional cryptosystems, and require some processing before they can be used in cryptographic operations. Moreover, as passwords are often chosen by users, they come from a relatively small space and special care is required in that processing to defend against search attacks.

A common approach to password-based cryptography, as described by Morris and Thompson (1979), is to combine a password with a salt to produce a key. The salt can be viewed as an index into a large set of keys derived from a password, and thus needs not be kept secret. While the salt does not rule out search based attacks, it does limit an opponent to searching through passwords separately for each salt. Another common approach to password-based cryptography is to construct the keys from relatively compute intensive techniques, thereby increasing the cost of a search-based attack. One such technique for hash-based key derivations is to include an iteration count, indicating how many times the hash function should be iteratively applied to produce a key.

Salt and iteration count of a pseudorandom function form the basis of PBKDF2 (Moriarty 2017), a common key derivation technique for web-based applications. PBKDF2 applies a pseudorandom function, such as the SHA1 hash-based message authentication code (HMAC-SHA1), to an input password along with a salt. Internally, the key derivation is based around the iterative hashing of blocks of bytes that each represent a part of the derived key, where a block's length is the length in octets of the pseudorandom function's output. The derived key is then obtained from the concatenation of these blocks:

$$Key = \text{take } len \text{ of } (T_1||T_2||\ldots||T_l) \tag{1}$$

where $||$ is concatentation, l denotes the number of blocks required to represent the len bytes of the derived key, and "take len of" extracts the first len bytes of a byte string. Individual blocks of T are defined as:

$$T_i = F(P, S, c, i) \tag{2}$$

Here, P is the password, encoded as a byte string, S is the salt, also encoded as a byte string, and c the iteration count as a positive integer. The function F is defined as the exclusive-or sum of the first c iterates of the underlying pseudorandom function (HMAC-SHA1) applied to the password P and the concatenation of the salt S and the block index i:

$$F(P, S, c, i) = U_1 \oplus U_2 \oplus \ldots \oplus U_c \tag{3}$$

with individual blocks of `U` defined as:

$$\begin{aligned} U_1 &= \text{HMAC-SHA1}(P, S||INT(i)) \\ U_i &= \text{HMAC-SHA1}(P, U_{i-1}) \end{aligned} \tag{4}$$

where $INT(i)$ is a four-byte encoding of the integer i.

To implement PBKDF2, we use some helper functions to model its three main components: T, F, and U. Due to the combinatorial nature of these components, they can be implemented quite nicely using the standard `foldl` and `map` functions in Haskell. In the co-design library, however, we make use of the methods of resource-aware functional programming developed for the Feldspar EDSL. Memory management is done through explicit, monadic operations, giving the user control over a program's memory usage. This entails the implementation of new combinators such as:

```
foreach :: (Comput m, Typ m Int, Typ m Word8)
  ⇒ (Exp m Int → m (Arr Word8)) → Exp m Int
  → m (Arr Word8)
iterate :: (Comput m, Typ m Int, Typ m Word8)
  ⇒ (Arr Word8 → m (Arr Word8)) → Exp m Int
  → Arr Word8 → m (Arr Word8)
```

where the `Comput` constraint asserts that any interpretation of `m` will support the computation operations needed to implement the functions and `Typ m a` is a collection of constraints that ensures `a` is well typed and supports the necessary expressions:

```
type Typ m a = (Num (Exp m a), ...)
```

The first combinator, `foreach`, applies its function to each index number from zero up to the given limit and returns the concatenation of the arrays. The second combinator, `iterate`, returns an array given by iterative applications of its function to an initial array. In the setting of PBKDF2, we can use `foreach` to create each block of T and `iterate` to create and join together the blocks of U. Now, PBKDF2 can be expressed as:

```
pbkdf2 :: (Comput m, Typ m Int, Typ m Word8)
       ⇒ Arr Word8 → Arr Word8 → Exp m Int
       → Exp m Int → m (Arr Word8)
pbkdf2 pswd salt c l = take l <$> ts where
  u0 i = int4b i >>= concat salt >>= hmac pswd
  uN a = hmac pswd a >>= zipWith xor a
  f  i = u0 i >>= iterate uN (c-1)
  ts   = foreach f (1 + l `div` 20)
```

`zipWith`, `take` and `concat` are like the standard functions from Haskell, but they handle arrays instead of lists; except for `take`, which reuses memory, the result type of these functions is monadic as they use new memory when creating arrays.

The underlying HMAC-SHA1 function is given by `hmac` and `sha1`, which are implemented in a similar manner as `pbkdf2` and given the following type signatures:

```
hmac :: (Comput m, Typ m Int, Typ m Word8)
     ⇒ Arr Word8 → Arr Word8 → m (Arr Word8)
sha1 :: (Comput m, Typ m Int, Typ m Word8)
     ⇒ Arr Word8 → m (Arr Word8)
```

Interpreting and compiling `pbkdf2` as a software program for an iteration count of 4096 and 256 blocks—the wpa2 encryption standard—yields the following C code (with variable declarations and the inlined `hmac` and `sha1` functions elided to save space):

```
for (v3 = 0; v3 ≤ 256 / 20 - 1; v3++) {
  a4[0 + 0] = (uint8_t) ((uint32_t) v3 >> 24);
  a4[1 + 0] = (uint8_t) ((uint32_t) v3 >> 16);
  a4[2 + 0] = (uint8_t) ((uint32_t) v3 >> 8);
  a4[3 + 0] = (uint8_t) (uint32_t) v3;
  memcpy(a5, a1, 16 * sizeof(uint8_t));
  memcpy(a5 + 16, a4, 4 * sizeof(uint8_t));
  a6 = { hmac over a5 and password. }
  for (v7 = 1; v7 ≤ 4096 - 1; v7++) {
    a8 = { hmac over a6 and password. }
    for (v9 = 0; v9 ≤ 20; v9++) {
      a6[v9 + 0] = a6[v9 + 0] ^ a8[v9 + 0];
  }}
  memcpy(a2 + (v3 * 20), a6, 20 * sizeof(uint8_t));
}
```

While the software implementation of pbkdf2 can be obtained by instantiation, offloading either sha1 or hmac to hardware requires that we modify the program to make use of components. As an example, we offload sha1 to hardware. sha1 is only constrained to computational programs, and can be interpreted as a hardware program. The first step towards offloading sha1 is straightforward: wrap it in a component.

```
sha1C :: Component ([Word8] → [Word8])
sha1C = component $ inpArr 64 $ λmessage →
  outArr 20 $ sha1 message
```

Here, inpArr and outArr are similar to the signature functions inp and out shown in section 2, but they handle arrays instead, represented by lists in the signature. Note also the length argument for the input and output arrays, as any array sent over AXI must have its length known at compile time to enable construction of an equal length array on the receiving end. As we are using passwords, which are normally quite short, a length of 64 is enough.

The second step to offloading sha1 is to edit the hmac function, which currently calls the old sha1 and is defined as:

```
hmac message salt =
  do opad ← mapA (0x5c `xor`) salt
     ipad ← mapA (0x36 `xor`) salt
     fst  ← sha1 =<< concatA ipad message
     sha1 =<< concatA opad fst
```

where mapA and concatA behave like their similarly named Haskell functions, but have been adapted to work with arrays instead.

From inspecting hmac, we see that sha1 is used as a regular combinator, for hashing the message twice with two different salts. As the calls to hardware components behave like regular function calls, we can simply replace each occurrence of sha1 in hmac with a function that calls the hardware component instead. For that purpose, we define callSha1,

```
callSha1 :: Arr Word8 → Software (Arr Word8)
callSha1 arr = do c ← mmap "0x38C00000" sha1C
                  call c (arr .: nil)
```

The need for an explicit hex address, as with the previous soft example, comes from the use of an external design tool to synthesize the component (and we plan to automate this process). The last two lines of hmac are updated:

```
  fst ← callSha1 =<< concatA ipad message
  callSha1 =<< concatA opad fst
```

These two changes produce a PBKDF2 design that makes use of both software and hardware.

Interpreting and compiling the new pbkdf2 as a software program, with sha1 offloaded to hardware, yields roughly the same C code as before (minus the sha1 code). The biggest change comes from the extra pointers used for sending and receiving data from the sha1 component:

```
f_map(0x83C00000, (void**) &output_sha1,
      &offset_sha1);
output_sha1 = output_sha1 + offset_sha1;
int * input0 = output_sha1 + 21;
int * output0 = output_sha1;
```

where f_map handles the memory mapping for connecting a pointer to the physical address of the component. input0 and output0 are then connected to the beginning of the component's input and output addresses, respectively. The remaining C code is then free to use them as regular variables, where writing and reading are done synchronously.

3 Implementation

In the following section, we present the techniques, types and abstractions used to implement a co-design library based on monads. While the focus is on implementing a hardware software co-design language, the ideas presented apply generally to the construction of multiple EDSLs that can be mixed. The four steps are:

1. *Implement a deeply embedded, imperative core language.* Its purpose is not to act as a convenient user interface, but rather to accurately describe the two target languages and offer an efficient platform for evaluation and code generation. As a consequence, it is kept as simple as possible.
2. *Give monad instances for the embedded languages.* A monad instance gives us access to useful combinator libraries as well as syntactic support to seamlessly mix computations from the embedded core languages with those in Haskell.
3. *Build higher-level extensions of the core languages.* The purpose of extending the core languages is to get back the user-friendly abstractions that were forsaken by the simpler core languages. Translate each extended language into its core language, to avoid costs associated with the abstraction.
4. *Implement user-friendly interfaces as a hierarchy of shallow embeddings on top of the extended core languages.* Each interface is given as a separate type-class and provides overloaded operations on that type. Structuring the type-classes in a hierarchy gives guarantees about the presence, and absence, of constructs in an embedded program.

3.1 Imperative programs

Inspired by the work of Svenningsson and Svensson (2013) and by the Operational Monad (Apfelmus 2017), we implement the first step outlined above, and declare our monadic representation of imperative programs. Like our inspiration, we capture monadic computations with an algebraic data type, but we also take into account the fact that different languages support different operations as well as expressions. We call the representation Prog:

```
data Prog ins exp a where
  Return :: a → Prog ins exp a
  Instr  :: ins (Prog ins exp) exp a
         → Prog ins exp a
  (:>>=) :: Prog ins exp a → (a → Prog ins exp b)
         → Prog ins exp b
```

In addition to having an extra parameter, our representation of programs has instructions as higher-order functors, parameterized on the program they are part of and its expression type. This facilitates an extensible definition of instructions, as shown in section 3.2.

As one would expect from a deep embedding of monads, `Prog` includes constructs to represent the standard monadic *bind* and *return* operations: `Return a` lifts a value `a` without introducing any effects; `m :>>= f` executes `m` and applies `f` to its result, producing a new computation as a result. The monadic instance declaration for `Prog` thus follows naturally[1]:

```
instance Monad (Prog ins exp) where
  return = Return
  (>>=)  = (:>>=)
```

The monadic instance for programs corresponds to the second step outlined above, as it provides syntactic support for sequencing instructions in either language.

The gist of the idea behind separating the instructions of `Prog` from its monadic constructs is straightforward: an instruction's effect will only depend on its interaction with other instructions, permitting their sequencing to be handled separately. The task of implementing a language based on `Prog` is then the same as writing an interpreter for the language's instructions. In particular, we can define a generic interpreter for programs that maps them to their intended meaning:

```
interpret :: Monad m
  ⇒ (∀a . ins (Prog ins exp) exp a → m a)
  → Prog ins exp a → m a
interpret f (Instr i)  = f i
interpret f (Return a) = return a
interpret f (m :>>= k) =
  interpret f m >>= interpret f . k
```

`interpret` lifts a monadic interpretation of the primitive instructions, which may be of varying types, to a monadic interpretation of the whole program. By using different types for the monad `m`, one can implement different "back ends" for programs. For example, interpretation in Haskell's `IO` monad gives a way to *run programs*, while interpretation in a code generation monad can be used to make a *compiler to another language*.

The ideas behind `Prog` and `interpret` are perhaps best understood by looking at a concrete example. We define `Comp`, a collection of purely computational instructions:

```
data Comp prg exp a where
  NewRef :: Type a ⇒ exp a → Comp prg exp (Ref a)
  GetRef :: Type a ⇒ Ref a → Comp prg exp (Val a)
  SetRef :: Type a ⇒ Ref a → exp a
         → Comp prg exp ()
  For :: (Type a, Integral a) ⇒ exp a → exp a
      → (Val a → prg ()) → Comp prg exp ()
```

`NewRef`, `GetRef` and `SetRef` handle the creation and management of mutable references. `For` represents a for-loop over a specified range, with a parameterised function indicating the program to run at each iteration. The `Val` and `Ref` types play a particular rôle. `Val` is used whenever a pure value is produced by one of the instructions and represents a value in *any* expression; returning a value in `exp` directly would be problematic for translating between instructions with different expression types. For similar reasons, any reference returned by an instruction is represented by `Ref`.

[1]The observant reader may suspect that `Prog` does not obey the monad laws, since it allows us to observe the nesting of binds in a program, and indeed this is the case. In fact, we only interpret `Prog` in ways that are ignorant of the nesting of bind, but there is nothing to guarantee this.

Different representations of `Val` and `Ref` are needed in different interpretations. For instance, when an instruction involving values of type `Val a` is evaluated, a concrete value of type `a` is needed, while compilation needs a symbolic representation of the value. To accommodate both representations, we use a sum type:

```
data Val a = ValE a          | ValC String
data Ref a = RefE (IORef a) | RefC String
```

Using sum types for values and references is a simple solution to having these types assume different roles during interpretation[2], but those interpretations will sometimes have to do incomplete pattern matching as a result. For example, we have to assume that values are constructed with `ValE` during evaluation, as values of `ValC` cannot be evaluated. Luckily, by making `Val` and `Ref` abstract types, we can hide any such concerns from users.

The predicate type `Type` restricts the set of types that can be used in `Comp` instructions. For simplicity, we have assumed that both software and hardware support the types of `Type`, and that those types can be represented using the following class:

```
class    (Eq a, Ord a, Show a) ⇒ Type a
instance (Eq a, Ord a, Show a) ⇒ Type a
```

We are now ready to define the evaluation of programs with computational instructions as an interpretation in Haskell's `IO` monad:

```
run :: EvalExp exp ⇒ Prog Comp exp a → IO a
run = interpret runComp

runComp :: EvalExp exp
  ⇒ Comp (Prog Comp exp) exp a → IO a
runComp (NewRef a) = RefE <$> newIORef (evalE a)
runComp (GetRef (RefE r)) = ValE <$> readIORef r
runComp (SetRef (RefE r) a) =
  writeIORef r (evalE a)
runComp (For l u body) =
  mapM_ (run . body . ValE) [evalE l..evalE u]
```

Most of the heavy lifting is done by the `runComp` function, which defines the interpretation of each instruction. The `IO` references themselves, and their associated functions, are provided by the `IORef` library. Note that only `RefE` and `ValE` are used on both sides of the equation during evaluation; `RefC` and `ValC` are not supported in the standard interpretation.

Because running a program in the `IO` monad also involves running any expressions its instructions may contain, we have limited `run` to instructions that contain runnable expressions. This is captured by the `EvalExp` constraint that provides the function `evalE` for evaluating closed expressions. `EvalExp` is defined as follows:

```
class EvalExp exp where
  evalE :: exp a → a
```

Code generation can be defined using `interpret` as well, given a function `compToC` which translates individual instructions into C code but using a monad for code generation instead of `IO`:

```
compileC :: CompileCExp exp
         ⇒ Prog Comp exp a → C a
compileC = interpret compToC
```

[2]Having `Val` and `Ref` as sum types is sufficient for the example languages we implement in this paper; another approach could see them implemented as type families.

As with evaluation, the compiler itself is straightforward and it is the mapping from instructions to C code that does most of the work. Also, just as evaluation required its expressions to be runnable, compilation now requires that we constrain its instructions with compileC to expressions that can be compiled as well. Code generation through interpret is not limited to just C code, however. Compilation to hardware descriptions in VHDL only requires us to supply a VHDL code generation monad and a mapping from instructions to VHDL fragments:

```
compileVHDL :: CompileVHDLExp exp
            ⇒ Prog Comp exp a → VHDL a
compileVHDL = interpret compToVHDL
```

A quick summary of progress is in order. We have shown a method of embedding imperative programs using the general Prog type. The monadic sequencing of instructions in the embedding has a close correspondence to statements in an imperative language and, because it is parameterized on the kind of instructions it accepts, we can use it for both the software and hardware monads. So far, however, it might not look like we have come far in the pursuit of a unified hardware software co-design language—as programs have been restricted to purely computational instructions. Such computational instructions are part of both C and VHDL; we only need to add the language-specific instructions.

3.2 Software and Hardware

Introducing the Prog type has brought real progress. The problem has been reduced from defining a whole co-design language to defining its instruction sets and expression languages. For instructions, the computational ones we have seen so far are part of both the software and hardware languages, and should be reused in the definition of both languages. Fortunately, the problem of extending Comp with data types from either software or hardware is already solved in Data Types à la Carte (DTC) (Swierstra 2008). DTC introduces a type composition operator (:+:) that can be seen as a higher-kinded version of Haskell's Either type, and lets us define the instruction set of a language as a sum of smaller data types—like Comp.

We will use a tweaked version of (:+:) that has been adapted to work with our higher-order functors, such as the instruction type, and demonstrate its use by introducing three new, language-specific instruction sets:

```
data Printf prg exp a where
  Printf :: String → [Arg exp]
         → Printf prg exp a

data Process prg exp a where
  Process :: [Identifier] → prg ()
          → Process prg exp ()

data Signal prg exp a where
  NewSig :: Type a ⇒ exp a
         → Signal prg exp (Sig a)
  GetSig :: Type a ⇒ Sig a
         → Signal prg exp (exp a)
  SetSig :: Type a ⇒ Sig a → exp a
         → Signal prg exp ()
```

Printf adds an instruction that models the printf function from C and thus takes a string, which may contain formatting symbols, and a list of values to substitute said symbols with. Process and Signal introduce instructions for modeling processes and signals from VHDL. It takes a list of identifiers on which the process will trigger, and the program to run once it does trigger. Signal introduces instructions for managing signals, which are similar in behavior to variables, but introduce a small delay when writing to them—a signal's new value will not be immediately available when writing to them, and will retain the old value for one time step.

With the help of (:+:), the new instructions can be combined with those in Comp to form the instruction sets for the software and hardware monads.

```
type SIns = Comp :+: Printf
type HIns = Comp :+: Process :+: Signal
```

Unfortunately, introducing (:+:) also means that constructing instructions becomes more complicated:

```
newSig :: Type a ⇒ exp a → Prog HIns exp (Sig a)
newSig a = Instr (Inj_R (Inj_R (NewSig a)))
```

The problem is that instructions are now tagged with injections, and the ordering and number of these injections will change as the instruction set gets larger. Fortunately, DTC provides a solution to this problem as well: the (:<:) class provides an inj function that automatically handles injection of single instructions based on their instruction set type. The class defines a form of subsumption where its instances perform a linear search at the type level to find the right nesting of injections. With this class, we can define a function that automatically injects instructions into programs:

```
inject :: (i :<: ins) ⇒ i (Prog ins exp) exp a
       → Prog ins exp a
inject = Instr . inj
```

We can now define the smart constructor for signals as follows:

```
newSig :: (Type a, Signal :<: ins) ⇒ exp a
       → Prog ins exp (Sig a)
newSig = inject . NewSig
```

While instructions that only use Sig or Ref can be directly injected into programs, a bit more care has to be taken for those that use Val since we want to keep Val hidden from users, and instead use an abstract type like exp to hold values. We note that exp need only support lifting of pure values and variables in order to model the features of Val. A type class implements these operations:

```
class FreeExp exp where
  varE :: Type a ⇒ String → exp a
  litE :: Type a ⇒ a      → exp a
```

Using FreeExp, we can define a general function for converting Val to an abstract type of exp that works independently of which interpretation it is used in:

```
express :: (FreeExp exp, Type a) ⇒ Val a → exp a
express (ValC s) = varE s
express (ValE a) = litE a
```

Now we can define any constructors that employ Val internally, using express as needed:

```
getRef :: (Type a, FreeExp exp, Comp :<: ins)
       ⇒ Ref a → Prog ins exp (exp a)
getRef r = express <$> inject (GetRef r)
```

Note that smart constructors for computational instructions have become more general in their type than the previous constructors were. For example, the type of `getRef` says that we can read references in programs with *any* instruction set, as long as it contains `Comp`, that is, any instruction set of the form `(... :+: Comp :+: ...)`. In other words, `getRef` is a valid function in *any language* that supports computational instructions.

3.3 Extensible Interpretation

DTC enables an extensible definition of software and hardware instructions, but compilation and evaluation are still locked to the closed domain of `Comp`. To extend interpretation as well, we need to abstract over the expression type and factor out the translation of instructions to a user-provided function. We do this by defining a new type class:

```
class Interp ins prg exp m where
  interp :: ins prg exp a → m a
```

`interp` describes a translation from an instruction of type `ins`, parameterized by its program type `prg` and expression types `exp`, into a monadic expression of type `m`. And, by providing an instance of `Interp` for `(:+:)`, we can further separate the translation into that of single of instructions. We define the new interpreter as:

```
interpret₂
  :: (Interp ins (Prog ins exp) exp m, Monad m)
  ⇒ Prog ins exp a → m a
interpret₂ = interpret interp
```

In order to get back support for evaluation or compilation of `Comp` under the new interpretation scheme, we simply add an instance for it with `Interp` and the desired monad. The instances themselves are similar to their earlier, corresponding interpretation functions, and we can even reuse their definitions for the new instances—assuming that we rewrite the recursive interpretation of programs in for-loops to use `interpret₂` instead of `run`:

```
instance ( Interp ins  (Prog ins exp) exp IO
         , EvalExp exp)
        ⇒ Interp Comp (Prog ins exp) exp IO
  where interp (For l u body) = mapM_
    (interpret₂ . body . ValE)
    [evalE l..evalE u]
```

The other cases of `interp` are defined in the same way as they were in the earlier `runComp` function.

3.4 Type classes

Some instructions are only meaningful in certain language interpretations. The co-design library is therefore structured with type classes. For example, a hardware program can deal with signals and processes, whereas a software program has no notion of such concepts at all. We can point out groups of instructions that are supported by some interpretations but not others, forming a hierarchy of classes. The base class is the standard `Monad` class from Haskell, ensuring that any language can handle the necessary information plumbing needed for sequencing of instructions in a program. Subclasses of `Monad` include `References`, a class that provides support for mutable references:

```
class Monad m ⇒ References m where
  newRef :: Type a ⇒ Exp m a → m (Ref a)
  getRef :: Type a ⇒ Ref a → m (Exp m a)
  setRef :: Type a ⇒ Ref a → Exp m a → m ()
```

Reference instructions, like most instructions, refer to the expression language associated with the monad `m`. We therefore use a type family `Exp` to retrieve the expression type of a program. `Exp` is defined as follows:

```
type family Exp m where Exp (Prog ins exp) = exp
```

Also, we should note that classes like `References` bear a resemblance to how programs are represented in a finally tag-less setting; a class interface gives part of the language's syntax, although an instruction's semantics are given by the appropriate `Interp` instances rather than instances of a class like `References`.

With the help of the earlier `inject` function for lifting expressions into program stubs, we can provide a default `References` instance for programs whose instructions contain `Comp`.

```
instance (Comp :<: ins, FreeExp exp)
    ⇒ References (Prog ins exp) where
  newRef v   = inject (NewRef v)
  getRef r   = express <$> inject (GetRef r)
  setRef r v = inject (SetRef r v)
```

Constraints like `References` form the fourth step outlined in section 3, and give a guarantee about the presence, or even absence, of certain instructions in a program. This, in turn, means that we can determine from a program's type whether it can be implemented in software or hardware, or both. A group of common instructions, such as the references and basic control structures found in both software and hardware, can be defined as a collection of constraints:

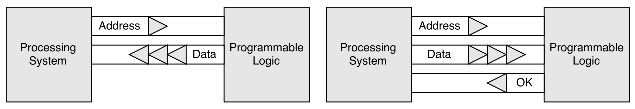
```
type Comput m =
  (Monad m, References m, Arrays m, Control m)
```

Programs that can only be interpreted in software will have the `SoftwareM m` constraint, while programs that can only be interpreted as hardware will have the `HardwareM m` constraint.

```
class Monad m ⇒ SoftwareM m where
  liftS :: Prog SIns (Exp m) a → m a

class Monad m ⇒ HardwareM m where
  liftH :: Prog HIns (Exp m) a → m a
```

Sub-classes of `SoftwareM` and `HardwareM` include classes that model the earlier `printf` instruction, and the process and signal instructions, respectively.

```
class SoftwareM m ⇒ Printf m where
  printf :: String → [Arg (Exp m)] → m ()

class HardwareM m ⇒ Process m where
  process :: [Identifier] → m () → m ()

class HardwareM m ⇒ Signal m where
  newSig :: Type a ⇒ Exp m a → m (Sig a)
  getSig :: Type a ⇒ Sig a → m (Exp m a)
  setSig :: Type a ⇒ Sig a → Exp m a → m ()
```



(a) Reading data. **(b)** Writing data.

Figure 2. Transactions.

A program will typically be constrained in the type to indicate what interpretations are allowed to run the program. The following program can be run by any language that supports computational operations:

```
modifyRef :: (Comput m, Type a)
  ⇒ (Exp m a → Exp m a) → Ref a → m ()
modifyRef f r = getRef r >>= setRef r . f
```

The architecture of the class system makes it easy for the user to add new classes of operations to the hierarchy, and, in combination with DTC, new interpretations that give semantics to them. As such, we have addressed the fourth point in the outline of our method at the beginning of this section. Software and hardware languages have, however, been kept separate from each other. In most co-design settings, there is interleaved execution of software and hardware programs, which therefore need to communicate with each other. The following section introduces the necessary communication channels.

4 Co-Design

The Zynq SoC couples a processing system of two ARM cores with a programmable logic array. The processing system and the programmable logic are connected by AXI4 compliant interfaces.

Full AXI4 offers a range of interconnects that include variable data and address bus widths, high bandwidth burst and cached transfers, and various other transaction features. The full specification provides great flexibility to users, and we plan to implement it eventually. Here, we make use of a simpler interconnect, AXI4-lite, which is also provided by the Zynq. This subset of the full specification forgoes the more advanced burst and streaming transactions for a simpler communication model of writing and reading single pieces of data, one at a time. Five channels make up the whole AXI4-lite specification: the read and write address channels, the read and write data channels, and the write acknowledge channel. We introduce these channels from the processor's perspective.

The read address channel allows the processor to indicate that it wishes to start reading data from a hardware component, and carries the address and handshaking information necessary to initiate the transaction. The hardware component then answers through the read data channel, which carries the data values that are sent during the transaction, along with the necessary handshaking. It is these two channels that together implement the protocol for a read data transaction. In the opposite direction, from processor to hardware, it is the write address channel that initiates a write transaction, with data flowing on the write data channel. A write response channel lets the component acknowledge a successful write or an error. Read and write transactions are visualized in Figure 2.

AXI4-lite channels are represented in a hardware description language as signals, driven by processes that implement the associated handshaking. Other than processes and signals, we also need to wrap these AXI4-lite channels in a hardware component, like the one that we saw very briefly in section 2. Components are ordinary hardware programs, but encapsulated and given a signature which describes how software programs or other hardware blocks should connect their pointers or wires to ports on the component. We define a new hardware instruction for components:

```
data Architecture prg exp a where
  NewComp :: Signature a
    → Architecture prg exp (Component a)
  PortMap :: HArgs a → Component a
    → Architecture prg exp ()
```

The `NewComp` constructor deals with the creation of hardware components from programs, while `PortMap` is used to instantiate components, using the `portmap` function in VHDL to connect components. The behavior of `portmap` is similar to function application on the software side; we name the component to be instantiated and its arguments, but we also give the target for its output. While function application in Haskell is simply another sequential operation, `portmap` is a concurrent statement and is more akin to plugging a hardware component into a socket on a board.

Normally, one cannot inspect the types of a component, and thus neither its incoming nor outgoing signals, from within a program in Haskell. To ensure that any application of the component on the software side is type correct, a component's type signature is encapsulated by the `Signature` type (Axelsson and Persson 2015). Such signatures enable the creation of a safe mapping of each argument to its signal in the hardware component, and one can think of them as adding a top-level lambda abstraction to the hardware language. On the software side, we also make use of the `Args` type, as a heterogenous list of arguments to match the signature of a hardware component when calling.

We add components to the set of hardware instructions:

```
type HIns = ... :+: Architecture
```

and note that the earlier compiler no longer supports the resulting hardware programs, as it cannot interpret components yet. But, as both constructs from the component instructions are based on existing VHDL constructs, their interpretation is a straightforward translation.

Now, to implement the AXI4-lite protocol we simply need to realize the *ready* and *valid* principle that it's based on; *ready* indicates that a hardware component is ready to accept a transfer of data or an address, and *valid* signals that any data or addresses sent by the processor are valid and can be sampled. In fact, reading and writing over AXI both follow this pattern, and can both be implemented using a process with a few conditional statements and logical expressions. Other than handshaking, we also need processes for managing the routing of data to and from AXI4-lite wires to a hardware component. These extra processes can be implemented in very much the same way as the previous ones: they are clocked processes that listen on their relevant signals and, given that the correct handshaking signals have been set, load the given address or store the received data in the addressed register. It is these registers that are connected to a hardware component using the `portmap` function, and this allows data received through the AXI4-lite signals to be forwarded to the component.

While implementing the AXI4-lite protocol was quite a low-level ordeal, in that it mostly consists of reading and writing to bit flags, the result is a hardware function that automatically connects a given hardware component to an AXI4-lite interconnect by inspecting its signature (we omitted parts of its signature to save space as the AXI4-lite protocol consists of over twenty signals):

```
axi4lite :: HardwareM m ⇒ Component a
  → m (Component (Signal Word32 → ...))
```

Having connected a hardware component to an AXI4-lite interconnect, the next step to get it onto the programmable logic is to compile it. The Haskell description of the component generates VHDL, which is input to a synthesis tool that produces the bit stream that causes the programmable logic array to be configured to implement the required behavior. In addition, the synthesizer gives the processor access to the inputs and outputs of the component through memory mapped I/O. As a consequence of running Linux on the processors, such memory mapping is handled by the mmap function, which creates a new mapping in the virtual address space of the calling processor. We introduce a software instruction to model memory mapping:

```
data MMap exp a where
  MMap :: String → Component a
       → MMap prg exp (Addr a)
  Call :: Addr a → SArgs (Argument a)
       → MMap prg exp (Result a)
```

MMap takes a hardware component along with its physical address on the programmable logic, gives the component an AXI4-lite wrapper using the earlier axi4lite function, and produces a memory reference in software that can be accessed using Call. Call is similar to portmap from hardware, but constructs its argument list using software expressions, and its result is the result of the signature.

We extend the previous set of software instructions to include memory mapped I/O:

```
type SIns = ... :+: MMap
```

A program that supports memory mapped I/O can freely communicate with a hardware component placed on the FPGA.

5 Expressions

Programs and their instructions represent the statements in an imperative language, but an expression language is also needed:

```
data CExp a where
  Var :: Type a ⇒ String → CExp a
  Lit :: Type a ⇒ a → CExp a
  Add :: (Num a, Type a)
    ⇒ CExp a → CExp a → CExp a
```

CExp is a type of numerical expressions with variables, integer literals and addition. By using a generalized algebraic datatype we make sure that only well-typed expressions can be constructed—barring the fact that variables can have any type.

The expression type of CExp provides little to no abstractions; its operations all have a relatively small semantic step to their corresponding operations in the target domains. Interpretation of CExp is consequently a straightforward translation of its constructs into matching expressions of the target domains. For example,

```
instance EvalExp CExp where
  evalE (Lit a)   = a
  evalE (Add a b) = evalE a + evalE b
```

It is due to these attributes that CExp forms the core software language in our library (which constitutes the first step in section 3):

```
type SoftwareC = Prog SIns CExp
```

One of the hallmarks of functional programming is the presence of powerful abstractions that hide many of the details of mundane operations, such as the sharing of values through let-bindings. Even though CExp can be translated into efficient code, from a user's perspective, it is really too simple to be useful. One way to introduce some user-friendly abstractions is to make an extension of CExp where such constructs are present. We implement the third step outlined in section 3, extending the software language, and define EExp:

```
data EExp a where ...
  Let :: Type a
      ⇒ EExp a → (EExp a → EExp b) → EExp b
```

In addition to the numerical constructs already found in CExp, EExp introduces let-binding as a new construct—using higher-order abstract syntax as a convenient technique for introducing variable binders.

```
type Software = Prog SIns EExp
```

Now, we can write a pure function that shares the result of another pure function f across an addition

```
*> let ex = λ x → Let (f x) (λy → y `Add` y)
*> :t ex
ex :: (Type a, Num a) ⇒ EExp a → EExp a
```

which resembles an expression we might have written in ordinary Haskell, barring the use of constructors to build the expression rather than the overloaded operators from Num. This discrepancy can however be remedied by declaring a Num instance for EExp. For Haskell classes or functions that cannot be instantiated by expressions, we instead provide a custom class that is intended to override its Haskell version. For example, let-bindings are built-in Haskell functions, and cannot be easily overridden. We therefore provide a type class which models let-bindings:

```
class Let exp where
  share :: (Type a, Type b) ⇒ exp a
        → (exp a → exp b) → exp b
```

In contrast to any construct of CExp, the let-binding found in EExp is a higher-level abstraction that does not have a corresponding construct in the expressions of either the software or hardware languages. As a consequence, EExp is semantically further away from its target domains and its interpretation is no longer a straightforward translation. However, rather than providing a new interpretation for EExp, which is tedious and error-prone task, we define a translation to programs of the simpler CExp type, which already supports evaluation and compilation. The interesting case occurs during the translation of Let. Since let-bindings cannot be represented in CExp, we instead have to realize them through computational constructs. We provide one possible translation, translating a let-binding into reference statements:

```
elaborateSoft :: EExp a → SoftwareC (CExp a)
elaborateSoft (Let v body) = do
  r ← newRef =<< elaborateSoft v
  x ← inject (GetRef r)
  elaborateSoft $ body $ express x
```

The argument expression of Let is evaluated first, and the resulting value is then bound to a variable and fed into the function. Note the explicit use of the GetRef construct instead of its constructor function. This is because the constructor is instantiated for the core language and therefore expects a CExp expression, but the function we wish to pass the variable into expects an EExp expression.

At this point, we have managed to define both the core languages and the extended languages for software and hardware. For the core languages, we have a way of evaluating and generating source code from programs. On the other hand, for the extended languages we have the ability to write high-level programs. What is missing before we can evaluate or compile the high-level programs is a function that takes either language's elaboration function and lifts it to an interpretation of entire programs. That is, and we generalize the problem a bit here, we need a way to elaborate the expressions of one program type to a program over another expression type. Fortunately, programs are regular monads like any other, and we can actually implement this interpretation using interpret₂:

```
elaborateS :: Software a → SoftwareC a
elaborateS = interpret₂
```

We can now define a compiler and an evaluator for software programs by piecing together the elaboration of expressions and the corresponding interpreter for core programs.

```
runS :: Software a → IO a
runS = runSoft . elaborateS

compileS :: Software a → String
compileS = compileSoft . elaborateS
```

To extend the running and compilation technique to hardware programs, we only need to define a similar elaboration process, but for hardware expressions instead.

The above approach to expressions has several advantages: the translation is typed, which rules out many potential errors, and is easier to write than a complete translation into source code; the low-level expression type is reusable, and can be shared as an elaboration target between multiple high-level expression types. Each stage opens up for different optimization possibilities—higher-level expressions reveal semantic information that's not readily available in the lower-level languages. Separating the two language versions also, in turn, separates the compiler from the user interface, allowing the two to be improved independently of each other.

6 Evaluation

We aim for high performance code in both hardware and software (that is VHDL and C). We have not yet done extensive benchmarking, but the generated code typically performs well in both C and VHDL, being comparable to handwritten implementations. Compared to optimized code, like the C implementation of SHA1 from OpenSSL (The OpenSSL Project 2003), our naive implementation is slower by a factor of two for smaller messages. As the library programmer has access to C and VHDL at a relatively low level, it would, however, be feasible for our users to implement similar optimizations. Still, both our C and VHDL implementations show a definite advantage over their corresponding Haskell implementations, as show in Figure 3. The three implementations of SHA1 were run on a Zynq SOC (FPGA with an ARM A9 core), where the C and Haskell versions ran on the ARM core and the VHDL component on the FPGA, but reading and writing values over an AXI channel to a software test bench. The test bench consisted of iteratively applied the three different hashes a number of times to an initial password. The hardware and software programs perform equally well, up until a thousand iterations where the hardware program performs slightly better. The reference Haskell implementation performs quite poorly in comparison.

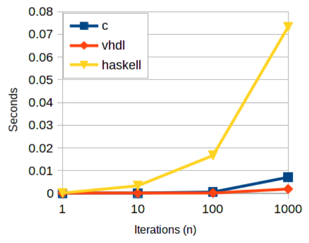


Figure 3. SHA1

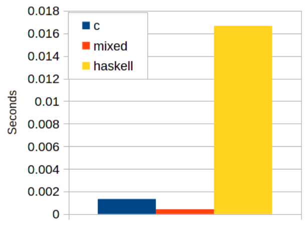


Figure 4. PBKDF2

We have also evaluated how well the library handles the bigger PBKDF2 example from section 2 in two variants, one that runs entirely in software (C) and another that offloads SHA1 to the FPGA and runs its remaining HMAC and PBKDF2 in software (Mixed). The performance of the two designs is compared in figure 4, along with a comparison to the reference implementation in Haskell (Compiled with GHC 8.0.2 with the -O2 flag). These initial results show promising performance.

7 Related Work

There has been much work on FPGA design in functional programming languages. Some approaches concentrate on describing hardware *structure* (Baaij et al. 2010; Bachrach et al. 2012; Bjesse et al. 1998; Claessen et al. 2003; Gill et al. 2010), while others describe *semantics* and synthesise hardware from more general functional programs (Ghica et al. 2011; Zhai et al. 2015), see the survey by Gammie (Gammie 2013). SPIRAL (Puschel et al. 2005) demonstrates the advantages of using a Domain Specific Language equipped with algebraic laws in the generation of either hardware or software. However, none of these approaches, to our knowledge, considers the problem of how to develop the software and the hardware of a system implemented on FPGA *together*, although we note that a Master's thesis describes a first attempt at a foreign function interface for CλaSH to enable offloading of tasks from software to hardware (Vossen 2016).

Bluespec's BSV is translated to synthesisable Register Transfer Level (RTL) code (Nikhil 2004). With its facilities for expressing concurrency and its libraries for generating finite state machines, pipelined architectures and much else, it provides an attractive alternative for FPGA design using a higher level language. Still, as far as we know, it concentrates on the hardware side, linking to standard synthesis tools.

George et al. (2013) explore the use of Lightweight Modular Staging (LMS) to ease the construction of a domain-specific High Level Synthesis (HLS) system. They argue that this eases the reuse of modules between different HLS flows, and makes it easier to link to existing tools such as C-to-RTL compilers and core generators. Though the language-specific challenges for LMS are different from ours, the two approaches are comparable in terms of capability. The way in which code generation of our programs is built upon the idea of monadic reification is also reminiscent of Sunroof (Bracker and Gill 2014), a DSL for generating JavaScript.

Outside the domain of HLS, examples of DSLs with similar ambitions to ours include the Cryptol DSL for cryptography (Browning and Weaver 2010) and Microsoft's Accelerator (Tarditi et al. 2005) for programming GPUs and various other platforms. In both DSLs, loops are automatically unrolled during code generation. Our library thus provides a step up in expressiveness, as its keeps loops in its generated code. On the other hand, Cryptol and Accelerator provide language constructs that are well suited for expressing dataflow algorithms, at a level of abstraction appropriate for hardware—although the latest versions of Cryptol no longer support hardware generation.

Early work by Mycroft and Sharp, based on the resource aware functional programming language SAFL, explored partitioning of SAFL code into software and hardware parts, with the software parts running on generated soft processors also on the FPGA (Mycroft and Sharp 2001). IBM's Liquid Metal aimed to use a single programming language, lime, based on Java, to program FPGAs, GPUs and multicores (Auerbach et al. 2012). However, this ambitious project seems to have been cancelled, and the final publicly released version supports only FPGAs and CPUS (and not GPUs). It does not support driving vendor synthesis tools or execution on a real FPGA board.

The closest work to ours that we have found is Dave's work on Bluespec Codesign Language (BCL), an extension of Bluespec that is a unified language model (based on guarded atomic actions) for hardware-software codesign (Dave 2011). A notion of computational domain resembling an extra type abstraction is introduced to express partitioning into hardware and software. Synchronizers, which are often variants of a FIFO interface, implement the communication between hardware and software parts. The thesis concentrates on formal reasoning about partitions and refinements, and so there is no description or evaluation of examples actually running on FPGA. A later MIT thesis made further progress towards implementing BCL by embedding it in Haskell (King 2013) and an impressive study of various partitionings of a set of benchmarks is performed.

Our approach lacks the notion of rewrite rule that BCL borrows from Bluespec and it will be interesting to see if we can do without this expressive feature. One option that we should consider is using Bluespec rather than VHDL as the hardware description language. On the software side, we benefit from building upon a recent set of tools for EDSL implementation in Haskell (Axelsson 2016).

Our current implementation demands that the user program at a relatively low level of abstraction (for both software and hardware). In order to make better use of the embedding in Haskell, we would need to add layers that provide Lava-like combinators to express common regular circuit patterns (Bjesse et al. 1998), behavioural constructs like those implemented in York Lava (Naylor et al. 2009) and a synchronous programming layer that makes use of our earlier work on streams in embedded languages (Aronsson et al. 2015). Hopefully, raising the level of abstraction will allow us to protect the user from having to think about the details (such as explicit memory mapped addresses) of the communication between hardware and software.

We would also like to take advantage of having a single description of the entire system by providing the means to *test* the resulting hardware and software, for example by building on work on BlueCheck (Naylor and Moore 2015), which is a version of QuickCheck for BSV. We feel that it would be an advantage to have a single language for expressing properties of both software and hardware, but we need to perform experiments to verify this. We would also like to provide facilities for formally verifying the hardware using SAT-based techniques, as in Chalmers Lava. These are our intended next steps.

Teich (2012) provides an excellent survey on hardware software codesign in general, as well as some predictions for the future. In particular, according to Teich: "too little effort is spent in this important area of joint coverification of hardware and software". We would like to apply property-based testing here.

8 Discussion

We have presented a hardware software co-design language embedded in Haskell. The library introduces `Prog` as an extensible representation of imperative programs, and we showed the basic methods for designing a hardware and a software language with it. Together, the two imperative languages form the basis of the co-design library. The library allows a single program description to be interpreted in many different ways, so that evaluation in Haskell's IO monad or compilation to hardware and software are supported. Furthermore, new instructions and interpretations can be added with relatively little disturbance to the existing system, allowing the library to be used in the development of new languages. Even if software and hardware end up as the only embedded languages, we feel that the idea of establishing an interface towards the interpreter and classes for compilation and evaluation is a good one; keeping them separated lets us treat the languages and their interpreters separately. To be able to provide these features, we have relied on Haskell's type system, including monads and type classes, and higher-order functions for capturing patterns. For example, the `interpret` function shown earlier is a higher-order function that takes a function for interpreting single instructions, and gives us an interpretation for entire programs. So, with the help of a type class, users wishing to define a new interpretation only have to give an instance that describes how to interpret their instructions. They get the rest for free.

We have shown how to generate C and VHDL from the imperative software and hardware programs using an extensible compiler. We could have chosen to target a high level language, such as OpenCL or SystemC, to describe both software and hardware. In many applications, this is an attractive approach, as it permits software developers to do hardware design without extensive training. However, for the kinds of high performance acceleration tasks that we aim to support, this does not seem to work well (see recent experiments with OpenCL on Zynq by Svensson (Svensson 2017)). Also, recent work by (Zohouri et al. 2016) showed that it was difficult to get good performance on FPGA, compared to GPUs for instance, for HPC kernels using OpenCL, even with specialized, FPGA-oriented optimization of the OpenCL code.

We have tried to keep the translation between design and source code transparent, for example with explicit memory management, avoiding high level synthesis or compilation steps that are difficult to control or predict. We believe in leaving matters firmly in the hands of the programmer. It would be significantly harder to analyze the performance or memory profile of programs were we to rely on high level synthesis. Later, we expect to introduce autotuning, but again giving control of the search for good solutions to the programmer, in the style of Vollmer et al. (2015).

To remain focused on reusable techniques, we have left out or simplified important details of the full implementation of the co-design language. The real system[3] makes use of a sophisticated software stack supporting the development of EDSLs in Haskell, with a rich set of expressions and types. In particular, we use Axelsson's `imperative-edsl`[4] library for C generation, `hardware-edsl`[5] for VHDL generation, and a standalone package for imperative programs called `operational-edsl`[6], also developed with Axelsson.

The final system will support the full AXI protocol, to permit streaming computations and to gain access to the high performance AXI ports. We are, therefore, in the process of implementing it, making use of the library's own VHDL implementation. We plan to introduce streaming not as another primitive instruction in the language, but rather as a layer on top of the current library, where it will extend the current software and hardware programs with support for expressing synchronous data-flow. Another possible extension would be to provide access to the remaining components of the Zynq, such as its DSP slices or other processing cores, without having to rely on the synthesis tool. Such extensions would require us to introduce new layers with syntactical support for more a fine-grained execution control than simply hardware or software. Its would also be interesting to explore the opposite direction and introduce an actor language for describing control flow at higher-level in the style of Bluespec (Nikhil 2004).

Writing the co-design library has been an educational exercise in the engineering of software and hardware programs. To make the library more usable, we need to explore additional interpretations. For example, we would like to explore parallelism for software programs and to introduce verification of programs as an additional interpretation. Nevertheless, the library is an interesting practical application of Haskell to the area of hardware software co-design. As was evident from the implementation of the PBKDF2 cryptographic function, the library allows different hardware software partitions to be easily explored. It is this ease of design exploration that will likely prove to be the key advantage of this approach to co-design, especially when we explore greater uses of meta-programming. Our experience from Lava indicates that this will be a fruitful area for further research.

Acknowledgements

The authors acknowledge the major contributions to this work made by Emil Axelsson and Anders Persson.

References

H. Apfelmus. 2017. The Operational Monad Tutorial (Blog Post). http://apfelmus.nfshost.com/articles/operational-monad.html. (2017).

Markus Aronsson, Emil Axelsson, and Mary Sheeran. 2015. Stream Processing for Embedded Domain Specific Languages. In *Revised Selected Papers from 26th Int. Symp. on Implementation and Application of Functional Languages (IFL)*. ACM.

Joshua Auerbach et al. 2012. A Compiler and Runtime for Heterogeneous Computing. In *Proc. 49th Design Automation Conference (DAC)*. ACM.

E. Axelsson. 2016. Compilation as a Typed EDSL-to-EDSL Transformation (Blog post). http://fun-discoveries.blogspot.se/2016/03/. (2016).

E. Axelsson and A. Persson. 2015. Programmable Signatures. In *Trends in Functional Programming, Revised Selected Papers*. Springer.

Christiaan Baaij, Matthijs Kooijman, Jan Kuper, Arjan Boeijink, and Marco Gerards. 2010. CλaSH: structural descriptions of synchronous hardware using Haskell. In *Proc. 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools (DSD)*. IEEE.

Jonathan Bachrach et al. 2012. Chisel: Constructing Hardware in a Scala Embedded Language. In *Proc. 49th Design Automation Conference (DAC)*. ACM.

P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. 1998. Lava: Hardware Design in Haskell. In *Proc. Third Int. Conf. on Functional Programming (ICFP)*. ACM.

Jan Bracker and Andy Gill. 2014. Sunroof: A Monadic DSL for Generating JavaScript. In *Proc. 16th Int. Symp. on Practical Aspects of Declarative Languages*. Springer International Publishing, 65–80.

Sally Browning and Philip Weaver. 2010. Designing tunable, verifiable cryptographic hardware using Cryptol. In *Design and Verification of Microprocessor Systems for High-Assurance Applications*. Springer, 89–143.

K. Claessen, M. Sheeran, and S. Singh. 2003. Using Lava to design and verify recursive and periodic sorters. *IJSTTT* 4, 3 (2003).

Nirav Dave. 2011. *A Unified Model for Hardware/Software Codesign.* Ph.D. Dissertation. EECS Dept., MIT.

Peter Gammie. 2013. Synchronous Digital Circuits As Functional Programs. *ACM Comput. Surv.* 46, 2, Article 21 (Nov. 2013), 27 pages.

N. George, D. Novo, T. Rompf, M. Odersky, and P. Ienne. 2013. Making domain-specific hardware synthesis tools cost-efficient. In *2013 International Conference on Field-Programmable Technology (FPT)*. 120–127.

Dan R. Ghica, Alex Smith, and Satnam Singh. 2011. Geometry of Synthesis IV: Compiling Affine Recursion into Static Hardware. In *Proc. 16th Int. Conf. on Functional Programming (ICFP)*. ACM.

Andy Gill et al. 2010. Introducing Kansas Lava. In *Proc. 21st Int. Conf. on Implementation and Application of Functional Languages (IFL)*. Springer-Verlag.

Patrick C. Hickey, Lee Pike, Trevor Elliott, James Bielman, and John Launchbury. 2014. Building Embedded Systems with Embedded DSLs (Experience Report). In *Proc. 19th Int. Conf. on Functional Programming (ICFP)*. ACM.

Myron King. 2013. *A Methodology for Hardware-Software Codesign.* Ph.D. Dissertation. EECS Dept., MIT.

K. Moriarty. 2017. Password-Based Cryptography Specification Version 2.1. (2017). https://tools.ietf.org/html/rfc2898

Robert Morris and Ken Thompson. 1979. Password Security: A Case History. *Commun. ACM* 22, 11 (Nov. 1979), 594–597.

Alan Mycroft and Richard Sharp. 2001. *Hardware/Software Co-design Using Functional Languages.* Springer-Verlag, 236–251.

M. Naylor and S. Moore. 2015. A generic synthesisable test bench. In *ACM/IEEE Int. Conf. on Formal Methods and Models for Codesign (MEMOCODE)*. 128–137.

Matthew Naylor, Colin Runciman, and Jason Reich. 2009. The Reduceron home page, fetched May 2017. (2009). https://www.cs.york.ac.uk/fp/reduceron/

Rishiyur Nikhil. 2004. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *ACM/IEEE Int. Conf. on Formal Methods and Models for Co-Design (MEMOCODE)*. IEEE, 69–70.

M. Puschel et al. 2005. SPIRAL: Code Generation for DSP Transforms. In *Proc. IEEE*, Vol. 93. Issue 2.

Josef David Svenningsson and Bo Joel Svensson. 2013. Simple and Compositional Reification of Monadic Embedded Languages. *Proc. 18th Int. Conf. on Functional Programming (ICFP)* (2013).

Bo Joel Svensson. 2017. OpenCL Reduction on the ZYNQ. (2017). http://svenssonjoel.github.io/writing/zynqreduce.pdf

Wouter Swierstra. 2008. Data Types à La Carte. *J. Funct. Program.* 18, 4 (July 2008).

David Tarditi, Sidd Puri, and Jose Oglesby. 2005. Accelerator: simplified programming of graphics processing units for general-purpose uses via data-parallelism. *Rapport Technique, Microsoft Research* (2005).

J Teich. 2012. Hardware Software Codesign: the Past, the Present, and Predicting the Future. *Proc. of the IEEE* 100 (2012).

The OpenSSL Project. 2003. OpenSSL: The Open Source toolkit for SSL/TLS. (April 2003). www.openssl.org.

Michael Vollmer, Bo Joel Svensson, Eric Holk, and Ryan R. Newton. 2015. Meta-programming and Auto-tuning in the Search for High Performance GPU Code. In *Proc. 4th Int. Workshop on Functional High-Performance Computing (FHPC)*. ACM.

J.J. Vossen. 2016. *Offloading Haskell functions onto an FPGA.* Master's thesis. Univ. Twente.

Kuangya Zhai, Richard Townsend, Lianne Lairmore, Martha A. Kim, and Stephen A. Edwards. 2015. Hardware Synthesis from a Recursive Functional Language. In *Proc. 10th Int. Conf. on Hardware/Software Codesign and System Synthesis (CODES)*. IEEE.

Hamid Reza Zohouri, Naoya Maruyama, Satoshi Matsuoka, and Aaron Smith. 2016. Evaluating and Optimizing OpenCL Kernels for High Performance Computing with FPGAs. In *Proc. SuperComputing Conference*. IEEE.

[3] https://github.com/markus-git/co-feldspar
[4] https://github.com/emilaxelsson/imperative-edsl
[5] https://github.com/markus-git/hardware-edsl
[6] https://github.com/emilaxelsson/operational-edsl