acmqueue

Domain-specific Languages and Code Synthesis Using Haskell

Looking at embedded DSLs

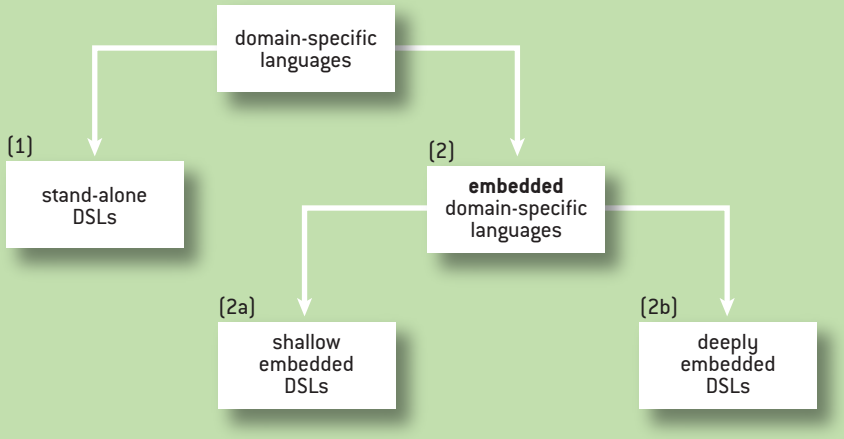
Andy Gill, University of Kansas

There are many ways to give instructions to a computer: an electrical engineer might write a MATLAB program; a database administrator might write an SQL script; a hardware engineer might write in Verilog; and an accountant might write a spreadsheet with embedded formulas. Aside from the difference in language used in each of these examples, there is an important difference in *form* and *idiom*. Each uses a language customized to the job at hand, and each builds computational requests in a form both familiar and productive for programmers (although accountants may not think of themselves as programmers). In short, each of these examples uses a DSL (domain-specific language).

A DSL is a special-purpose language, designed to encapsulate possible computations in a specific domain. In the examples of MATLAB, SQL, Verilog, and spreadsheets, the domains would be scientific modeling, database queries and updates, hardware circuits, and financial computations, respectively. Considering SQL specifically, there is nothing it does that could not be done in Java or C, or any other general-purpose programming language. SQL simply bundles the actions needed to interact with a database into a usable and productive package, and the language becomes the interface to communicate requests to the database engine.

There are two fundamental types of DSLs. The first is a first-class language, shown in figure 1(1), with its own compiler or interpreter, and it is often used in its own ecosystem. All the examples mentioned so far fall into this category. The primary difference between the SQL DSL and, for

FIGURE 1 **Types of Domain-Specific Languages**



example, Java is one of scope and focus, although sometimes DSLs grow to be as large as general-purpose languages.

The other class of DSL is a language embedded in a host language, as shown in figure 1(2). Such languages can have the look and feel of being their own language, but they leverage the host language's existing ecosystem and initial semantics. This article is concerned with this second class of DSLs.

HASKELL PRIMER

An EDSL (embedded DSL) is a language inside a language. Haskell,[17] the premier pure functional programming language, is a great host for EDSLs because of flexible overloading, a powerful type system, and lazy semantics. This section provides a terse introduction to Haskell, sufficient to make this article self-contained. It is an extended version of the Haskell primer given by this author in 2011 at the International Conference on Engineering of Reconfigurable Systems and Algorithms.[10]

Haskell is all about *types*. Types in Haskell, like those in other languages, are constraining summaries of structural values. For example, in Haskell `Bool` is the type of the values `True` and `False`; `Int` is the type of machine-sized words; `Double` is the type of double-precision IEEE floating-point values; and this list goes on in the same manner as in C, C++, Java, and other traditional languages. All these type names in Haskell start with an uppercase letter.

On top of these basic types, Haskell has two syntactic forms for expressing compound types. First, pairs, triples, and larger structures can be written using tuple-syntax, comma-separated types inside parentheses. Thus, `(Int,Bool)` is a structure with both an `Int` and a `Bool` component. Second, lists have a syntactic shortcut, using square brackets. Thus, `[Int]` is a list of `Int`.

Haskell also has other container types. A container that *may* contain one `Int` has the type `Maybe Int`, which is read *Maybe of Int*. These container names also start with uppercase letters.

Types can be nested to any depth. For example, you can have a `[(Maybe (Int,Bool))]`, read as *list of Maybe of (Int and Bool)*.

Polymorphic values are expressed using lowercase letters and play a similar role to `void*` pointers in C and polymorphic arguments in the Java generics facility. These polymorphic values can have constraints expressed over them, using the Haskell equivalent of an object hierarchy.

Finally, a function is written using an arrow from argument type to result type. Thus, in Haskell, a function that takes a list and returns a list is written as: `[a] -> [a]`.

Here is an example of a Haskell function:

```
sort :: (Ord a) => [a] -> [a]
sort []     = []
sort (x:xs) = sort before ++ [x] ++ sort after
  where
        before = filter (<= x) xs
        after  = filter (> x) xs
```

This function sorts a list using a variant of quicksort in which the pivot is the first element of the list:

• The first line is the type for `sort`. This is `∀a`, such that `a` can be ordered (admits comparisons like `<=`); the function takes and return a list of such `a`'s.
• The second line says that an empty list is already sorted.
• The remaining lines state that a non-empty list can be sorted by taking the first and rest of the list (called `x` and `xs`, respectively), sorting the values before this pivot and after this pivot, and concatenating these intermediate values together.
• Finally, intermediate values can be named using the `where` syntax; in this case the values of `before` and `after`.

Haskell is a concise and direct language. Structures in Haskell are denoted using types, constructed and deconstructed, but never updated. For example, the `Maybe` type can be defined using two constructors, `Nothing` and `Just`:

```
data Maybe where
  Nothing ::      Maybe a
  Just    :: a -> Maybe a
```

`Nothing` is a `Maybe` of anything; `Just`, with an argument, is a `Maybe` with the type of the argument. These constructors can be used to construct and deconstruct structures, but there is never any updating; all structures in Haskell are immutable.

It is possible to give specific types extra powers, such as equality and comparison, using the class-based overloading system. The `Maybe` type, for example, can be given the ability to test for equality, using an instance:

```
instance Eq a => Eq (Maybe a) where
  Just a  == Just b  = a == b
  Nothing == Nothing = True
  _       == _       = False
```

This states that for any type that can be tested for equality, you can also check `Maybe` for the same type. You take the `Maybe` apart, using pattern matching on `Just`, to check the internal value.

In Haskell, side effects such as writing to the screen or reading the keyboard are described using a `do`-notation:

```
main :: IO ()
main = do
  putStrLn “Hello”
  xs <- getLine
  print xs
```

In this example a *value* called `main` uses the `do`-notation to describe an interaction with a user. Actually, the `do`-notation captures this as a structure called a *monad*; purity is not compromised. More detailed information is available on how the `do`-notation and monads can provide an effectful interface inside a pure language such as Haskell.[18] For the purposes of this article, `do`-notation is

a way of providing syntax and structure that looks like interaction. There are many tutorials on Haskell; the Haskell Web site (http://haskell.org) is a good starting point for further reading.

EMBEDDED DSLS

An EDSL is a library in a host language that has the look, feel, and semantics of its own language, customized to a specific problem domain. By reusing the facilities and tools of the host language, an EDSL considerably lowers the cost of both developing and maintaining a DSL. Benefiting from Haskell's concise syntax, the Haskell community—and the functional programming community in general—has taken the ideas of EDSLs and developed a large number of DSLs that provide higher-level interfaces and abstractions for well-understood systems. What follows are two examples of EDSLs: one for automatically generating test cases for software testing; and a second for specifying hardware-circuit behaviors.

EXAMPLE EDSL: QUICKCHECK PROPERTIES

Consider the challenge of writing test cases—or more specifically, writing the *properties* that test cases need to satisfy:

```
-- The reverse of a reverse'd list is itself
prop_reverse_twice (xs :: [Int]) = reverse (reverse xs) == xs
```

In this example, `prop_reverse_twice` is a regular Haskell function that takes a list of Int and returns a Boolean, based on the validity of what is being proposed—specifically, that two reverses cancel each other out. Here is the neat part: `prop_reverse_twice` is *also* a domain-specific statement and as such can be considered a sublanguage inside Haskell. This style of using functions (in this case, functions with names prefixed with `prop_`, taking a number of typed arguments, and returning a conditional) is a small language. The property written in Haskell is also an EDSL for properties, called QuickCheck.[4] This EDSL can be run using a function also called `quickCheck`:

```
Prelude Test.QuickCheck> quickCheck prop_reverse_twice
+++ OK, passed 100 tests.
```

By running `quickCheck` with this explicit and specific property, the EDSL executes inside Haskell. The `quickCheck` function generates 100 test cases for the property and executes them on the fly. If they all hold, then the system prints a message reflecting this. The test cases are generated using the type class system—QuickCheck gives specific types the power of test-case generation—and the `quickCheck` function uses this to generate random tests.

As an example of an incorrect property, consider this property for `reverse`.

```
prop_reverse xs ys = reverse xs ++ reverse ys == reverse (xs ++ ys)
```

This states that the reverse of two distinct lists is the same as the reverse of both lists appended together, but this property is false.

```
Prelude Test.QuickCheck> quickCheck prop_reverse
Falsifiable, after 5 tests:
[0]
[2,-2]
```

It turns out that this sort of mini-language is really useful in practice. Despite the simplicity of how Haskell is being used, the QuickCheck EDSL provides a way of thinking about and directly expressing properties. It has additional functionality, including the ability to generate random function arguments, to control the distribution of the random test cases, and to state preconditions to a property. From this DSL, many other implementations of these ideas have been constructed. There is even a Swedish company, QuviQ, that sells a QuickCheck for the concurrent programming language Erlang.

EXAMPLE EDSL: KANSAS LAVA

To take another example, consider describing hardware. Hardware description languages and functional languages have long enjoyed a fruitful partnership. Lava is the name given to a class of Haskell DSLs that implement a function-based version of the hardware description language Ruby.[12,13] Not to be confused with the modern programming language of the same name, Ruby was based on relations, which was in turn inspired by the seminal work in μFP.[21]

Kansas Lava[11] is a Haskell-hosted DSL that follows the Lava line of research. It is a language for expressing gate-level circuits. Haskell abstractions allow the programmer to work at a slightly higher level of abstraction, where the model is one of recursive components communicating via synchronized streams. Kansas Lava has been used for the generation of high-performance circuits for telemetry decoders, though the model used is general.

As an example of Kansas Lava, consider:

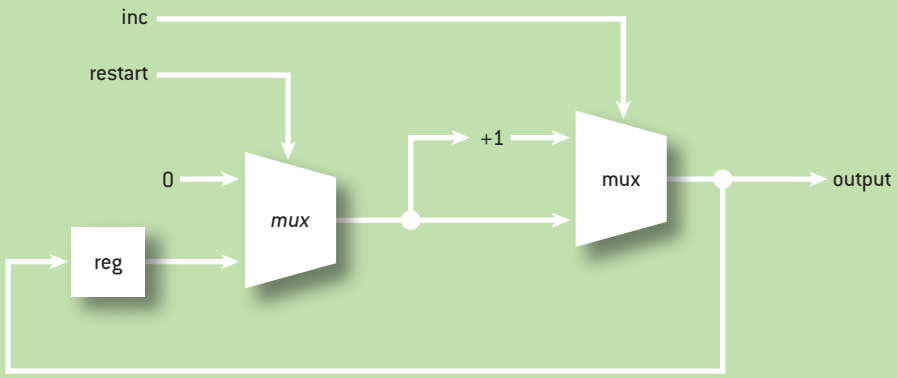
```
counter :: (Rep a, Num a) => Signal Bool -> Signal Bool -> Signal a
counter restart inc = loop
   where reg = register 0 loop
         reg' = mux2 restart (0,reg)
         loop = mux2 inc (reg' + 1, reg')
```

This circuit connects two multiplexers (`mux2`), an adder, and a `register` to give a circuit that counts the number of clocked pulses on a signal `inc`. The circuit takes two clocked signals and returns a clocked signal that explicitly operates using the same clock, because they share the same type. The use of arithmetic is understated, but simply uses (via overloading) the standard syntax for addition; the `Num` constraint allows this. Figure 2 illustrates the circuit intended for this description.

You can simulate sequential circuits with the same directness as the combinational functions invoked.

```
GHCi> toSeq (cycle [True,False,False])
True : False : False : True : False : False : True : False : False : ...
GHCi> counter low (toSeq (cycle [True,False,False]))
1 : 1 : 1 : 2 : 2 : 2 : 3 : 3 : 3 : ...
```

FIGURE 2 **Schematic Kansas Lava Parity Counter**



As well as basic signal types, you can build circuits that operate on Haskell functions directly, provided the domain of the function is finite. The `Rep` capability is used to signify that you can enumerate all possible representable values in a type, giving the `funMap` function:

```
funMap :: (Rep a, Rep b) => (a -> Maybe b) -> Signal a -> Signal b
```

The generated circuit is implemented using a ROM, and you can generate control logic directly in terms of Haskell functions and data structures. As an example, consider a small ROM that stores the square of a value:

```
squareROM :: (Num a, Rep a) => Signal a -> Signal a
squareROM = funMap (\ x -> return (x * x))
```

In this way, direct Haskell functions can be lifted into the `Signal` world. Notice how the `squareROM` function is not specific about size but is completely generic, requiring only the type of the argument stream to be representable as a number.

The clock-squaring ROM can now be used at specific types. For example, at eight-bit you can generate the following:

```
GHCi> squareROM (toSeq [0,1..] :: Signal Word8)
0 : 1 : 4 : 9 : 16 : 25 : 36 : 49 : 64 : 81 : 100 : 121 : 144 : 169 : 196 : 225 : 0 :
...
```

This level of circuit specification has been used to great effect in many Lava and Lava-like languages. One notable instance is Hawk,[15] a Lava-like EDSL that was used to specify the entire micro-architecture of the Pentium Pro, including the super-scaler design, and register bypass capabilities.

Now, if DSLs are so powerful as an idiom for library design, then why have they not taken over? As a means for expressing things that can be *simulated*, EDSLs are an invaluable design pattern; however, not everything is a simulation. What if you wanted to use an EDSL to express something that you want to run somewhere else, not inside the Haskell system? Can Lava be used to generate circuits to be run on FPGAs (field-programmable gate arrays)? Can EDSLs be used to generate code for embedded processors or GPUs? Such an ability—to synthesize external solutions—would be extremely useful. The EDSL idiom can be extended to do so, with significant caveats. The remainder of this article is about how to capture and offshore work from inside an EDSL; what this capability can be used for; and what the limitations are.

DEEPLY EMBEDDED DOMAIN-SPECIFIC LANGUAGES

EDSLs are simply a way of thinking about a library of provided functions, often called *combinators*, because they combine their arguments into terms inside the DSL. In the previous Lava example, the `register` combinator takes an initial value and an incoming stream of values and provides the new stream, delayed by a single cycle, with the initial value occupying the initial cycle. Critically, `register` is compositional; it combines smaller parts of the DSL to make larger solutions. If a DSL follows this composability carefully by design, an important alternative implementation is possible.

The most common flavor of EDSL is one that uses so-called shallow embedding, as seen in figure 1(2a), where values are computed directly. The result of a computation in a shallow EDSL is a value. All the examples so far are shallow. There is another class of EDSLs, however: specifically, those that use a deep embedding to build an abstract syntax tree, as shown in figure 1(2b). The result of a computation inside a deeply embedded DSL (deep EDSL) is a structure, not a value, and this structure can be used to compute a value or be cross-compiled before being evaluated.[7] Such deep EDSLs follow the composability mantra pedantically, by design and mandate.

Historically, EDSLs have been shallow—simply a way of structuring an API for a library. Deep EDSLs, however, have the ability to *stage* code—that is, executing a program can generate another program, much like the well-known yacc DSL, but at the cost of significantly restricting what forms of the DSL can generate valid output. There are a growing number of deep EDSLs, along with a body of research around their form and limitations. The unifying theme is that deep EDSLs can be pragmatic, productive, and useful.

This section investigates the basic structure of a deep EDSL compared with a shallow EDSL, and looks at three pragmatic tricks for improving the usefulness of deep EDSLs.

BUILDING A DEEP EDSL

A deeply embedded DSL exposes its own composition and structure. Rather than using functions operating directly on values (a shallow DSL), a deep DSL builds a structure, then allows some secondary agent to provide interpretation of this structure. To make this idea concrete, consider a DSL for arithmetic, with addition, subtraction, multiplication, and constants. For a shallow embedding, running this DSL is trivial; you just use the built-in arithmetic. A deep embedding is where things get interesting. Consider a data type for our arithmetic:

```
data Expr where
 Lit :: Integer -> Expr
 Add :: Expr -> Expr -> Expr
 Sub :: Expr -> Expr -> Expr
 Mul :: Expr -> Expr -> Expr
 deriving Eq
```

Now overload the arithmetic to use this E data type; in Haskell, `Num` is the overloading for integer arithmetic:

```
instance Num Expr where
  fromInteger n = Lit n
  e1 + e2 = Add e1 e2
  e1 - e2 = Sub e1 e2
  e1 * e2 = Mul e1 e2
```

By building expressions of type E, you can observe the structure of the computation:

```
GHCi> 1 + 2 * 3 :: Expr
Add (Lit 1) (Mul (Lit 2) (Lit 3))
```

This is profound, and it is the key idea that makes deep embeddings work. You can write an expression and extract a tree of *what* to do, not a direct result. With deep embeddings, it is common also to write a `run` function that computes the result of a captured computation:

```
run :: Expr -> Integer
run (Lit n)
= n
run (Add a b) = run a + run b
run (Sub a b) = run a - run b
run (Mul a b) = run a * run b
```

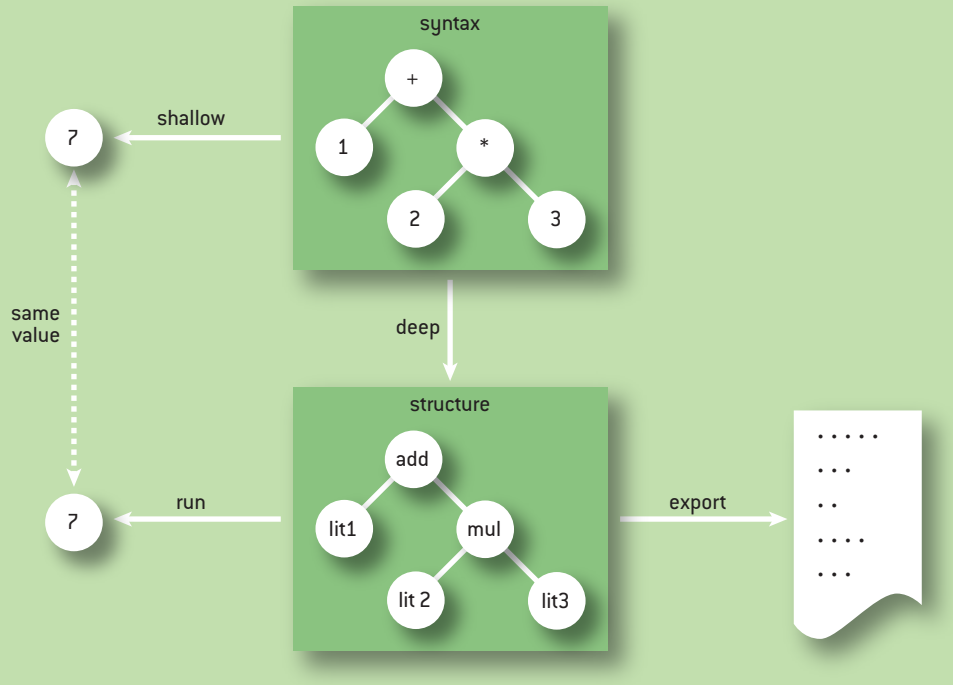
Figure 3 illustrates the differences between shallow and deep DSLs, and how a deep embedding combined with a specific `run` function gives the same result. For a deep embedded DSL, the `run` function restores the capability of the shallow embedding, but another function takes the embedded structure and uses it in some creative way.

To make deep DSLs practical, there are two additional tricks in the DSL folklore that are almost always used. The first trick allows the capture of functions, via dummy arguments. The second trick can observe loops, via some form of observable sharing.

HOW TO EXTRACT A DEEP EMBEDDING FROM A FUNCTION

Expressing function calls in terms of constructors and building expression trees is useful, but by itself is a gimmick. With careful construction, however, you can also capture function definitions,

FIGURE 3 **Shallow and Deep Embedding of Arithmetic**



as well as other syntactical structures, directly from a deep embedding. It is at this point that the idea of capturing code, then using the captured code to execute code on a different target, becomes possible. Consider a simple function to add one to its argument:

```
f :: Expr -> Expr
f x = x + 1
```

Here is a function that acts over the new type `Expr` and returns a new `Expr`. How can you capture this function? The trick is to invent a unique `Expr` and pass it as a (dummy) argument to `f`:

```
data Expr where
  Lit :: Integer -> Expr
  Add :: Expr -> Expr -> Expr
  Sub :: Expr -> Expr -> Expr
  Mul :: Expr -> Expr -> Expr
  Var :: String -> Expr -- new constructor
```

You can now run the function directly and see the result in your deep embedding, or pass in the `Var` argument and see the actual function:

```
-- Just running the function
GHCi> f 4
Add (Lit 4) (Lit 1)
-- reifying the function, using our unique Var.
GHCi> f (Var "x")
Add (Var "x") (Lit 1) -- reified version of the function
```

This is remarkable! You've run a function with a dummy argument (called the *prototypical argument*) and extracted the body of the function.

This idea scales to multi-argument functions. Consider `g`:

```
g :: Expr -> Expr -> Expr
g x y = x * x + y + 2
```

Two prototypical arguments to `g` will capture the function:

```
GHCi> g (Var "x") (Var "y")
Add (Add (Mul (Var "x") (Var "x")) (Var "y")) (Lit 2)
```

There are many places this design pattern can be used. One example is the specification of surface textures as functions; it is possible to export these into code executable on GPUs, simultaneously lifting the abstractions used to write textures and speeding up how fast an implementation of the same operations would run. There is nothing that is specific about Haskell or even functional languages here. Indeed, the same ideas have been used in Java for a VHDL (VHSIC Hardware Description Language) generator.[2] Haskell, with its powerful abstractions, allows deep DSLs almost to feel like a straightforward shallow embedding.

HOW TO SPOT A LOOP

Lava programs are written as equations of recursive bindings. An attempt to build a deep embedding of Lava directly will lead to an infinite cycle of structures. To illustrate the challenge, let's build a deep embedding of Lava, see where it goes wrong, and fix it using a technique called observable sharing.

First the Lava language needs a structure. We define the functions used before but give them a deep embedding, called `Signal`:

```
data Signal where
  Register :: a -> Signal a                       -> Signal a
  Mux2     :: Signal Bool -> (Signal a,Signal a)  -> Signal a
  Lit      :: a                                   -> Signal a
  Add      :: Signal a -> Signal a                -> Signal a
  Var      :: String                               -> Signal a -- the Var trick
```

```
instance Num a => Num (Signal a) where
  a + b = Add a b

mux2 :: Signal Bool -> (Signal a,Signal a) -> Signal a
mux2 c (a,b) = Mux2 c (a,b)

register :: a -> Signal a -> Signal a
register d s = Register d s
```

Now, when attempting to extract `counter`, things go horribly wrong:

```
GHCi> counter (Var “restart”) (Var “inc”)
Mux2 (Var “inc”) (Add (Mux2 (Var “restart”) (Lit 0,Register 0 (Mux2 (Var “inc”) ...
```

The output tree is infinite. What has happened is the recursive definitions are unrolling during attempts to reify the function, or more specifically, the body of `counter` is looping. At this point, the EDSL community was stymied. There were efforts to use monadic structure, where the loop was expressing using `do`-notation,[8] making the loop an observable effect. There was an unsafe extension to observe a limited form of sharing by circumventing part of the purity of Haskell, called observable sharing.[5] There was also an extension of the Haskell I/O mechanism that allowed loops to be observed indirectly, called I/O-based observable sharing.[9] The net effect of all three mechanisms is that the observed tree is rendered as a graph with named edges.

At this point Haskell rescues us from some complexity. Advanced type-system mechanisms, such as higher-kinded arguments, allow a structure to be either a tree or a graph, depending on type-level instantiation. Omitting the details here, the reified function is a tree with sharing, then translated into a graph with explicit sharing. The final result for this example entity `counter` is:

```
GHCi> reify (counter (Var “restart”) (Var “inc”))
[(0,MUX2 1 (2,3)),
 (1,VAR “inc”),
 (2,ADD 3 4),
 (3,MUX2 5 (6,7)),
 (4,LIT 1),
 (5,VAR “restart”),
 (6,LIT 0),
 (7,REGISTER 0 0)]
```

In this output, each uppercase constructor corresponds to its deep-embedding constructor. A quick inspection shows that the circuit has been captured, as shown in figure 2. From this netlist-style structure, generating VHDL is straightforward. For the example of four-bit numbers, the VHDL is provided in figure 4.

These two tricks (prototypical argument and I/O-based observable sharing) are the technical fundamentals of Kansas Lava. On top of this base, and with help from the Haskell type system, an

FIGURE 4

VHDL Generated by Kansas Lava for Counter

```
entity counter is
  port(rst : in std_logic;
       clk : in std_logic;
       clk_en : in std_logic;
       restart : in std_logic;
       inc : in std_logic;
       output : out std_logic_vector(3 downto 0));
end entity counter;
architecture str of counter is
  signal sig_2_o0 : std_logic_vector(3 downto 0);
  ...
begin
  sig_2_o0 <= sig_5_o0 when (inc = '1')  else sig_6_o0;
  sig_5_o0 <= std_logic_vector(...);
  sig_6_o0 <= "0000" when (restart = '1') else sig_10_o0;
  sig_10_o0_next <= sig_2_o0;
  proc14 : process(rst,clk) is
  begin
    if rst = '1' then
      sig_10_o0 <= "0000";
    elsif rising_edge(clk) then
      if (clk_en = '1') then
        sig_10_o0 <= sig_10_o0_next;
  ....
end architecture;
```

entire ecosystem for circuit generation has been developed. The DSL idiom allows programmers to use high-level abstraction in Haskell and generate efficient circuits. Not all is rosy, however; writing a Lava program is not the same as writing a Haskell program because of the limitations of deep embeddings.

A DEEP EMBEDDING IS ONLY HALF A PROGRAM

The basis of a deep EDSL is one of constructiveness. Functional programming is about constructing *and deconstructing* values. Because of this, a deep embedding cannot reify any pattern matching—or even direct usage of if-then-else—and other control flow. Kansas Lava sidestepped this—for example, by using a `mux2` constructor, which encodes choice. How much further can the idiom be pushed if you need to be deconstructive? The result is surprising. Let's start with the three capabilities:

- Basic expressions can be captured by constructing a tree that is an analog to your syntax.
- Functions can be captured using a fake unique argument.
- Local bindings can be observed using some form of observable sharing.

With these three comes an automatic fourth capability:

• The host language provides a built-in macro capability to the embedded language. Any part of Haskell (including control flow and pattern matching) can be used to *generate* the embedded language.

There are also extensions to the basic techniques. The principal ones are:

• Internal function calls can be captured as notes on a graph, rather than directly inlined.[14] This helps compilation of large programs, giving a basic separate compilation capability.

• The `do` statement can be reified by normalization.[16,19,22] This result, called *monadic reification*, is surprising. There are strong technical reasons to believe monadic reification should be impossible; however, the normalization refactors the constraints that, by themselves, would be impossible to solve and matches them up, one-on-one, with a matching witness, allowing the whole `do`-notation to be solved and reified. Monadic reification is a recent discovery but has already been used in several deep DSLs, including Feldspar[1] and Sunroof.[3]

• Control flow is problematic and cannot be used directly, but there is a generalization of Haskell Boolean that does allow deep-embedding capture.[6] Using this library, a DSL with control flow can be constructed, but it needs to be explicit code, at the DSL level, using constructors. The `mux2` function used previously is a simplification of this idea. The usage is clumsy but workable, and we should be able to do better.

Where does this leave deep DSLs? They are clearly a useful design pattern for the language implementer, but they come with costs and limitations. How can we therefore push the state of the art and allow more of the Haskell language to be reified? There are two primary shortcomings. One we have discussed already: control flow and pattern matching remain a thorn in deep DSLs.

Parametric polymorphism, one of the strengths of a functional program, is the other issue for deep DSLs. A specific structure is needed to represent what has been captured, and arbitrary polymorphism interferes with this. Current systems sidestep this issue by always instantiating at a specific type, but this is expensive because the size of the captured program can expand exponentially. Polymorphism was the technical reason that it was thought that monadic reification was not possible, but in that case it was sidestepped by normalization; this technique does not generalize to all polymorphism.

A deep DSL is a value-level way of extracting an expression, but there are other ways. Quasi-quoting is a mechanism for extracting expressions, but at the syntactic level. Haskell comes with an extensive template system called Template Haskell[20], which is often used for DSLs. There is a sense of unease with such solutions; however, in much the same way the C preprocessor is used even though it is not considered elegant. The principal issue is that the syntax of Haskell is huge, consisting of around 100 syntactical terms. An expression-based solution, such as a deep embedding, can avoid the need to rewrite front translations. Quasi-quoting has one important advantage: specifically, it can cope with control flow and deconstruction of values. Perhaps the future of deep DSLs is some hybrid between expression generation and quasi-quoting, combining the best of both systems.

ACKNOWLEDGEMENTS

This paper is based upon work supported by the National Science Foundation under Grant No. CCF-1117569, and was originally presented as a master class under the Scottish Informatics & Computer Science Alliance Visiting Fellow program, in November 2013. The Kansas Lava examples and description were adapted from an earlier article about Lava written by the author.[10]

REFERENCES

1. Axelsson, E., Claessen, K., Sheeran, M., Svenningsson, J., Engdal, D., Persson, A. 2011. The design and implementation of Feldspar: an embedded language for digital signal processing. In *Proceedings of the 22nd International Conference on Implementation and Application of Functional Languages*. Springer-Verlag: 121-136.
2. Bellows, P., Hutchings, B. 1998. JHDL—an HDL for reconfigurable systems. Annual IEEE Symposium on Field-programmable Custom Computing Machines.
3. Bracker, J., Gill, A. 2014. Sunroof: a monadic DSL for generating JavaScript. *Practical Aspects of Declarative Languages*. Matthew Flatt and Hai-Feng Guo, eds. Volume 8324, *Lecture Notes in Computer Science*: 65-80. Springer International Publishing.
4. Claessen, K., Hughes, J. 2000. Quickcheck: A lightweight tool for random testing of Haskell programs. In *ACM Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming:* 268-279.
5. Claessen, K., Sands, D. 1999. Observable sharing for functional circuit description. In *Proceedings of 5th Asian Computer Science Conference, Lecture Notes in Computer Science*. Springer Verlag.
6. Elliott, C. Boolean package; hackage.haskell.org/package/Boolean.
7. Elliott, C., Finne, S., de Moor, O. 2003. Compiling embedded languages. *Journal of Functional Programming* 13(2).
8. Erkök, L., Launchbury, J. 2000. Recursive monadic bindings. In *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming*: 174-185.
9. Gill, A. 2009. Type-safe observable sharing in Haskell. In *Proceedings of the 2nd ACM SIGPLAN Haskell Symposium*: 117-128.
10. Gill, A. 2011. Declarative FPGA circuit synthesis using Kansas Lava. The International Conference on Engineering of Reconfigurable Systems and Algorithms.
11. Gill, A., Bull, T., Farmer, A., Kimmell, G., Komp, E. 2013. Types and associated type families for hardware simulation and synthesis: the internals and externals of Kansas Lava. *Higher-Order and Symbolic Computation*: 1-20.
12. Hutton, G. 1993. The Ruby interpreter. Research Report 72, Chalmers University of Technology.
13. Jones, G., Sheeran, M. 1990. Circuit design in Ruby. *Formal Methods for VLSI Design*. Jorgen Staunstrup, ed. Elsevier Science Publications.
14. Mainland, G., Morrisett, G. 2010. Nikola: embedding compiled GPU functions in Haskell. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Haskell*: 67-78.
15. Matthews, J., Cook, B., Launchbury, J. 1998. Microprocessor specification in Hawk. In *Proceedings of the International Conference on Computer Languages*: 90-101.
16. Persson, A., Axelsson, E., Svenningsson, J. 2012. Generic monadic constructs for embedded languages. In *Implementation and Application of Functional Languages*: 85-99. Springer.
17. Peyton Jones, S. L., ed. 2003. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge, England: Cambridge University Press.
18. Peyton Jones, S. L., Wadler, P. 1993. Imperative functional programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*: 71-84.
19. Sculthorpe, N., Bracker, J., Giorgidze, G., Gill, A. 2013. The constrained-monad problem. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*: 287-298.
20. Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In Manuel M. T.

Chakravarty, ed., ACM SIGPLAN Haskell Workshop 02, pages 1–16. ACM Press, Oct 2002.
21. Sheeran, M. 1984. *μ*FP, a language for VLSI design. In *Proceedings of the ACM Symposium on LISP and Functional Programming*: 104-112.
22. Svenningsson, J., Svensson, B. J. 2013. Simple and compositional reification of monadic embedded languages. In *Proceedings of the International Conference on Functional Programming*: 299-304.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

Andy Gill (andygill@ku.edu) is an Assistant Professor in the Department of Electrical Engineering and Computer Science at the University of Kansas. His research interests include optimization, language design, debugging, and dependability. The long-term goal of his research is to offer engineers and practitioners the opportunity to write clear and high-level executable specifications that can realistically be compiled into efficient implementations.