Abstract—The functional programming community has developed a number of powerful abstractions for dealing with diverse programming models in a modular way. Beginning with a core of pure, side effect free computation, modular monadic semantics (MMS) allows designers to construct domain-specific languages by adding layers of semantic features, such as mutable state and I/O, in an à la carte fashion. In the realm of interpreter and compiler construction, the benefits of this approach are manifold and well explored. This paper advocates bringing the tools of MMS to bear on hardware design and verification. In particular, we shall discuss a prototype compiler called ReWire which translates high-level MMS hardware specifications into working circuits on FPGAs. This enables designers to tackle the complexity of hardware design in a modular way, without compromising efficiency.

I. INTRODUCTION

Modularity is a pervasive concern in computer science, and hardware design is no exception. When evaluating a design language, it is therefore reasonable to ask: does the language support the kind of modular design strategies that will make us more productive? High assurance presents another concern: do the abstractions provided by the language help ensure that our designs conform to the intended semantics?

In one sense, mainstream hardware design languages (HDL) such as VHDL and Verilog do support modular design. For example, a CPU design in VHDL might be broken down into one module for the ALU, one module for the register file, one module for the fetch/decode/execute logic, and so on. Fig. 1a gives a typical example of this kind of structural decomposition (the PicoBlaze [1] 8-bit soft microcontroller from Xilinx). This paradigm—let us call it structural modularity—serves designers well when the semantics of the device (e.g., the instruction set of a processor) is fixed. When a hardware designer wishes to explore semantically novel ideas, however, structural modularity may be of limited utility. Suppose, for example, that we wish to extend the PicoBlaze design of Fig. 1a with hardware-level support for separation among security domains [2], enabling the safe interleaving of processes handling both classified and unclassified data. With this kind of cross-cutting concern, structural modularity gives us little traction: simply put, there is no one place in the block diagram of Fig. 1a to insert a “separation module”.

In this paper, we advocate a design process that is driven not by structural modularity, but by semantic modularity. In particular, we adopt a technique known as monadic semantics (MMS). With MMS, designers can construct domain-specific languages (DSL) for particular flavors of effectful computation (state, I/O, etc.). The set of effects supported by such a DSL is limited to just those specified by the designer. Thus we avoid dealing with the full semantic complexity of (say) a C-style imperative language.

The remainder of this section gives an overview of monads and modular monadic semantics. This overview is necessarily quite brief. Readers requiring more background are invited to consult the references in Section IV. The paper then proceeds in Section II to explore a case study in the form of a small CPU specification. Synthesis of this spec via a prototype compiler called ReWire, which generates VHDL from MMS specifications written in a subset of Haskell [3], is discussed in Section III. Section IV presents related work, and Section V concludes.

Monads: For reasons of space we must omit the formal definition of a monad here; further reading is recommended in Section IV. Broadly speaking, a monad is a way of embedding computations with side effects into an otherwise pure (side effect-free) language. A monad type and its attendant operators define, in essence, a domain-specific language representing a certain notion of computation with effects. If \( M \) is a monad, we say that \( M t \) is the type of \( M \)-computations producing a result of type \( t \). The exact nature of the allowed side effects depends on the monad. Any given monad must support, at a minimum, an operation for injecting pure values into the monad (called return in Haskell), and an operator that allows monadic computations to be chained together sequentially (written \( >>= \) in Haskell and pronounced bind). The exact behavior of these operations varies from monad to monad: for example, if \( M \) is the monad of mutable state, its bind operator will handle the passing of state from one computation to the next, alongside the return value of the previous computation. Any useful monad will also support a number of primitive operations specific to that particular monad: e.g., the mutable state monad \( St \) will have an operator get of type \( St s \) where \( s \) is the type of the state. Another useful monad, called the reactive resumption monad, represents computations that may communicate with the outside world via synchronous I/O channels during the course of a computation.

Effects à la Carte: Many applications require us to mix different kinds of effects. For example, a processor like PicoBlaze will need synchronous communication and mutable state. A monad that embodies multiple notions of effect can be constructed using monad transformers. In essence, a
monad transformer is a means of extending an existing monad with more kinds of effects. For example, the state monad transformer \( \text{StT} \) extends a “base” monad with stateful features. Thus a monad \( \text{Re} \) with a mixture of stateful and reactive computation may be defined according to the type declaration:

\[
\text{type } \text{Re} = \text{ReT} \text{ Out In } (\text{StT} \text{ Sto Id})
\]

Beginning with the “trivial” monad \( \text{Id} \) of side effect free computation, we extend that monad with stateful computation (the state monad transformer \( \text{StT} \)), and further extend \( \text{that} \) monad with reactivity (the reactive resumption monad transformer \( \text{ReT} \)). The types \( \text{Out, In, and Sto} \), whose definitions are not shown, are respectively the types of the output signals, the input signals, and the mutable state.

**PicoBlaze in ReWire:** The very same layering of reactive and stateful computation can be used to define the monad of computation for a processor like PicoBlaze. Fig. 1b defines a few types pertaining to the PicoBlaze. The PicoBlaze controller features 16 single byte registers, a 64 byte internal RAM, a 31x10-bit ring buffer serving as a stack, support for up to 256 input/output devices, and an ALU with carry and zero flags. The register file type, \( \text{RegFile} \), is declared in ReWire as a Table \( W4 \ W8 \) (i.e., a mapping from \( W4 \) to \( W8 \)). Generally, for any \( n \), the type \( Wn \) is a built-in type in ReWire which stands for \( n\)-bit word.” There are five flags in PicoBlaze, which are, respectively, the zero (\( Z \)), carry (\( C \)), zero-save (\( Zsave \)), carry-save (\( Csave \)), and interrupt enable (\( IE \)) flags. The type declaration for the flag registers is \( \text{FlagFile} \) in Fig. 1b. The scratchpad RAM is represented as a table, Table \( W6 \ W8 \). If we bundle these types together into a single record type called \( \text{PicoState} \), the monad \( \text{PicoBlaze} \) can be declared as follows.

\[
\text{type } \text{PicoBlaze} = \text{ReT} \text{ Outputs Inputs } (\text{StT} \text{ PicoState I})
\]

The PicoBlaze monad defines a new domain-specific language that allows us to write the program describing the PicoBlaze processor. Rather than delving into the details, it is simpler to understand how a familiar idea is represented with it. Below, the fetch-decode-execute loop for PicoBlaze (called “\( fde \)”) is written in ReWire:

```rewire
fde :: PicoBlaze ()
fde = getPicoState >>= \ s ->>
let i = inputs s
    instr = instruction_in s
    in getFlagIE >>= \ ie ->>
```

The \( fde \) computation first gets the current state of the processor with \( \text{getPicoState} \) and assigns it to \( s \). The inputs on the input ports are bound to \( i \), and \( \text{inst} \) is bound to the instruction word. The current value of the interrupt enable flag is read and assigned to \( ie \). If the reset signal has been set (i.e., \( \text{reset_in} i == 1 \)), then the processor transitions to the \( \text{reset_event} \) state (not shown). Otherwise, if the interrupt enable flag is set and an interrupt has occurred, then the processor makes the transition to \( \text{interrupt_event} \) (not shown). Otherwise, the processor decodes and executes the instruction. Finally, \( fde \) starts its loop again.

The full PicoBlaze specification is too large to include here. Therefore the remainder of this paper addresses a case study in the form of an even simpler 8-bit CPU.

### II. CASE STUDY: A SIMPLE CPU IN ReWire

This case study concerns the description in ReWire of a simple CPU including its instruction set and fetch-decode-execute cycle. The remainder of this section describes the ReWire program for the CPU in detail. The full ReWire code for this example is available online [4]. The CPU has an 8-bit address space and its register set contains two 8-bit general purpose registers, \( r1 \) and \( r2 \), and an 8-bit program counter, \( pc \). It has two input ports, the first of which is a single-bit reset line to (re)start the CPU. The second port is a connection to a 16-bit data bus. Likewise, the CPU has two output ports connecting it to a single-bit line controlling an LED as well as to an 8-bit address line, which is assumed to be connected to an external memory.

The heart of the ReWire program for the CPU is an interpreter for its instruction set—this interpreter is precisely the execute phase of the fetch-decode-execute cycle.

The abstract syntax for the instruction set is:

```rewire
data Instr = Branch0 W8 -- branch if \( i == 0 \)
             | LoadR1 W8 -- load \( r1 \) from offset
             | LoadR2 W8 -- load \( r2 \) from offset
             | Add -- \( r1 := r1 + r2 \)
             | SetLED -- \( led := led \( r1 \) \)
             | Invalid -- invalid instruction
```

The full ReWire program for the CPU is

```rewire
if reset_in i == 1
then reset_event
else if ie == 1 && interrupt_in i == 1
then interrupt_event >> fde
else decode instr >> fde
```

The full ReWire program for the CPU is

```rewire
if reset_in i == 1
then reset_event
else if ie == 1 && interrupt_in i == 1
then interrupt_event >> fde
else decode instr >> fde
```
The instructions include instructions to branch, load the r1 and r2 registers, and a simple add. There is also an instruction to assign the least significant bit of the r1 register to the LED output port—assigning 1 (resp., 0) to the port turns on (resp., off) the LED. Finally, there is an Invalid instruction because not all 16 bit words correspond to valid instructions. The effect of executing Invalid will be to return to the reset state. Obviously this is a very small instruction set, but expanding it to be more fully featured requires only extending the type Instr and the exec function of Fig. 2.

The following are types underlying the architecture:

```
type InpSig = (Bit, W16) -- external reset line
  W16 -- input word from bus

type OutSig = (Bit, Addr) -- output address to bus

type Addr = W8

type RegF = (W8,W8,W8) -- (r1,r2,pc)
```

InpSig is the type of the input ports and OutSig is the type of the output ports. Addresses (Addr) are 8-bits wide and the register file (RegF) contains three registers as described above.

The CPU monad is a composition of three monad transformers. The first is a reactive component (re), encapsulating the I/O of signals from the external environment. The second and third components add the raw material for manipulating the register file (StT RegF) and the current signals (StT CPUState).

```
type Re = React OutSig InpSig

type U = StT RegF Re

type CPU = StT CPUState U

type CPUState = OutSig
```

Instructions on the CPU are defined in terms of the following operations. The operation, put rf, sets the current register file to register file rf and get reads the current register file. The operations putOutSig and getOutSig are similar, affecting the output signals. These are defined in terms of the state monad transformer operations u (update state) and g (get state).

```
put :: RegF -> CPU ()
get :: CPU RegF
putOutSig :: OutSig -> CPU ()
getOutSig :: CPU OutSig
```

With put and get, we define commands for setting and reading the individual RegF components; those for r1 and r2 are similar to putPC and getPC, so we omit them.

```
putPC :: W16 -> CPU ()
putPC pc = get >>= (r1,r2,_) -> put (r1,r2,pc)
getPC :: CPU W16
getPC = get >>= (r1,r2,pc) -> return pc
```

With putOutSig and getOutSig, we define commands for setting and reading the output signals. The LED is cleared and set via putLED. Similar commands for reading from and writing to the address bus are omitted:

```
putLED :: Bit -> CPU ()
putLED led = getOutSig >>= (a) -> putOutSig (led,a)
```

The decode phase of the fetch-decode-execute cycle is simply a function that splits a W16 into two bytes, an opcode byte and an operand byte (used only for branches and loads). Due to the small size of the instruction set, there are a number of wasted bits in this encoding.

```
declare :: W16 -> Instr
decode w = case split16 w of
  (0x80,byt) -> Branch0 byt
  (0x81,byt) -> LoadR1 byt
  (0x82,byt) -> LoadR2 byt
  (0x83,byt) -> Add
  (0x84,byt) -> SetLED
  (_,_) -> Invalid
```

The CPU monad includes the current input signals, and the sigCPU function simply returns the current values of those signals.

```
start :: CPU ()
start = putPC 0 >> fetch = getPC >> \pc->
  putAddr pc >>
  putOutSig (0,0)>>
  sigCPU >>=
  \(r,w)->
  resetIf r
  (exec(decode w))
```

The function resetIf is a helper function that returns the CPU to the reset state if the test is false (0). Finally, the execution phase for the CPU is found in Fig. 2.

### III. Synthesis

The prototype ReWire compiler produces VHDL implementations from high-level specifications via a three-step process. The first step, partial evaluation [5], unwinds any occurrences of unsynthesizable features such as recursive data structures and higher-order functions. If partial evaluation successfully eliminates these unsynthesizable constructs, the next step is generation of a state machine in a simple intermediate language called SMIL (for State Machine Intermediate Language). A few minor optimizations such as constant propagation and common subexpression elimination can then be performed on the resulting SMIL specification. Post-optimization, the SMIL is translated into a typical one-process VHDL implementation of the resulting state machine.

The VHDL generated by the compiler for the simple CPU described here has been synthesized for a Xilinx Spartan-3E XC3S500E FPGA, speed grade -4, using Xilinx's XST synthesis tools. The maximum clock rate is 133.515 MHz, and the device usage is characterized by the following table.

<table>
<thead>
<tr>
<th>Component</th>
<th>Used</th>
<th>Available</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slices</td>
<td>115</td>
<td>4656</td>
<td>2.47%</td>
</tr>
<tr>
<td>Slice Flip Flops</td>
<td>48</td>
<td>9312</td>
<td>0.52%</td>
</tr>
<tr>
<td>4-Input LUTs</td>
<td>213</td>
<td>9312</td>
<td>2.29%</td>
</tr>
</tbody>
</table>

### IV. Related Work

 Readers requiring an introduction to monads and modular monadic semantics may wish to consult the work of Moggi [6], Liang, Hudak, and Jones [7].

From the point of view of programming language taxonomy, ReWire falls into the category of functional languages, with a strong emphasis on monadic programming. The most well known system for hardware synthesis within the functional programming model is Lava [8], [9]. Lava is a domain-specific language for hardware specification embedded in...
Haskell. Primitives in Lava essentially specify circuit behavior at the level of signals. ReWire, by contrast, compiles a subset of Haskell itself to hardware circuits, and relies on a somewhat more abstract set of primitives (namely reactive resumptions). C\textsc{ash} [10] is a compiler for a subset of Haskell to VHDL. Like ReWire, C\textsc{ash} uses Haskell itself as a source language rather than embedding a domain-specific language in Haskell. C\textsc{ash} requires some limits be placed on kinds of algebraic data types used as well as the basic operating types. ForSyDe is a platform to compile models of hardware written in Haskell to circuitry [11]. ForSyDe operates similarly to Lava in that it uses Haskell as a host for an embedded domain-specific language, with circuits represented by types provided by the ForSyDe platform.

V. CONCLUSIONS AND FUTURE WORK

Edwards [12] has commented on the difficulty of compiling from a C-like language to hardware. This has led him to pursue Haskell as a source [13]. The case study in this paper suggests that synthesizing sequential circuits from purely functional languages is indeed feasible. Furthermore, the use of higher-order functional abstractions, such as monads, greatly speeds the construction of complex circuits, and makes their specifications much more extensible. The compiler presented here is only an early prototype, and much further optimization of the state machines generated by ReWire should be possible. In particular, partial evaluation may result in an unnecessary propagation of control states which could be eliminated. The layout of temporary data may also be a target for optimization.

Recent research has demonstrated the value of monadic semantics to the formal specification and verification of x86- and ARM-based systems [14], [15]. The besetting challenge for all such formal methods research is that the systems under consideration are formalized post facto to their construction and consequently the formal methods scientist must engage in a painstaking reconstruction of the system semantics from informal and sometimes incomplete natural language documents. With ReWire, the text of the design is verified as-is and the compiler transforms that same design into hardware, thereby unifying the languages of specification, design and implementation. This may alleviate the necessity of reconstructing system semantics as the design and formal specification are one and the same.

ReWire, being a strongly typed, functional language, may provide a vector for adapting a range of language-based security techniques [16] to hardware. Recent work by several of the authors applies effect systems to fault isolation for kernels written in a resumption monad-based language akin to ReWire [17]. As part of an ongoing project investigating security in the setting of many-core computers, the authors are extending and adapting this fault isolation effect system to ReWire to enforce security properties.

ACKNOWLEDGEMENTS

The authors would like to thank Chris Hathhorn for his very helpful feedback on an earlier draft of this paper. Adum Procter is supported by U.S. Department of Education GAANN grant no. P200A100053.

REFERENCES