Model-driven Engineering from Modular Monadic Semantics: Implementation Techniques Targeting Hardware and Software*

William L. Harrison[1], Adam M. Procter[1], Jason Agron[2], Garrin Kimmell[3], and Gerard Allwein[4]

[1] Department of CS, University of Missouri, Columbia, Missouri, USA
[2] Department of CS & CE, University of Arkansas, Fayetteville, Arkansas, USA
[3] Department of EECS, University of Kansas, Lawrence, Kansas, USA
[4] US Naval Research Laboratory, Code 5543, Washington, DC, USA

Abstract. Recent research has shown how the formal modeling of concurrent systems can benefit from monadic structuring. With this approach, a formal system model is really a program in a domain specific language defined by a monad for shared-state concurrency. Can these models be compiled into efficient implementations? This paper addresses this question and presents an overview of techniques for compiling monadic concurrency models directly into reasonably efficient software and hardware implementations. The implementation techniques described in this article form the basis of a semantics-directed approach to model-driven engineering.

1 Introduction

System software is notoriously difficult to reason about—formally or informally—and this, in turn, greatly complicates the construction of high assurance systems. This difficulty stems from the conceptual "distance" between the abstract models of systems and concrete system implementations. Formal system models are expressed in terms of high-level abstractions while system implementations reflect the low-level details of hardware, machine languages and C. One recent trend in systems engineering—model-driven engineering (MDE) [40]—attempts to overcome this distance by synthesizing implementations directly from system specifications. The MDE methodology is attractive for high assurance applications because the process proceeds from a domain specific modeling language that, when specified with a suitable formal semantics, will support verification.

The starting point for this work is recent research applying modular monadic semantics to the design and verification of trustworthy concurrent systems [15, 14, 12, 13]. There are a number of natural "next" questions concerning the set of design and verification principles developed in the aforementioned publications.

* This research was supported by NSF CAREER Award 00017806; US Naval Res. Lab. Contract 1302-08-015S; DARPA/AFRL Contract FA8650-07-C-7733; the Gilliom Cyber Security Gift Fund; Cadstone, LLC; and the ITTC Tech. Transfer Fund.

Can system implementations be generated from monad-based specifications and, if so, how is this best accomplished? Are critical system properties preserved by implementation techniques? Can acceptable performance across many dimensions (including speed, size, power consumption, etc.) be achieved? This paper addresses the first question and leaves the others to future work.

This paper considers an instance of MDE for which the models are based in the modular monadic semantics (MMS) of concurrent languages [12, 34]. Monads have a dual nature, being both algebraic structures with properties and a kind of domain-specific language (DSL) supporting programming. The view of monads as DSLs is just the standard view within the functional programming community expressed in a non-standard way: monads encapsulate a specific flavor of computation and provide language constructs (i.e., monadically-typed operators) in which to build computations.

The contributions of this paper are: (1) An exploration of the design requirements for a monadic DSL for describing small concurrent systems with shared state (e.g., embedded controllers). This language is called *Cheap Threads* after an article of the same name [18]. (2) Implementation techniques for Cheap Threads (CT) programs targeting both a fixed instruction set and hardware circuits. The first technique is a traditional compiler for a non-traditional language. The second technique translates a CT program into VHDL code from which circuitry may be synthesized and loaded into an FPGA. A DSL for specifying state machines called *FSMLang* [3] serves as an intermediate language. (3) A significant case study demonstrating the application of these techniques to the automatic generation of software-defined radios [23] from CT-style specifications.

It is sometimes said of domain specific languages that what is left out of them is more important than what is left in. By restricting its expressiveness, a DSL design can better capture the idioms of its targeted domain. This is just as true in our case where the limitations imposed are intended to aid in the ultimate goal of verifying high assurance properties. Both with respect to its semantics and implementation strategies, what is left out of CT is crucial to the success of this approach. Because the semantics of CT is fundamentally simpler than that of Haskell, its implementation can be made far simpler as well. Although it dispenses with the constructs that complicate Haskell 98's semantics, CT retains the necessary expressiveness—but no more expressiveness than is necessary.

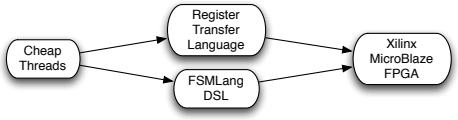
The decision to create a standalone CT language (as opposed to an embedding within a language such as Haskell) was made for the sake of semantic simplicity. Haskell 98 [36], for example, has a surprisingly complicated semantics [17, 21, 11] due to certain of its features (e.g., the `seq` operator, expressive pattern-matching, polymorphic recursion and type classes), while its extension, GHC Haskell, has no standard semantics at all. Furthermore, excluding features such as general recursion results in much more predictable time and space behavior. CT possesses a built-in semantics as each CT language construct corresponds to an algebraic operator of a fixed monad.

Syntactically, CT is simply a sublanguage of Haskell, extended with a simple declaration form for specifying monads. Any CT program is also a Haskell pro-

gram that can be executed with standard implementations like GHC or Hugs. CT contains only the small core of Haskell 98 necessary for defining computations within these monads (function and data declarations, etc.). In particular, CT dispenses with first-class functions, curried functions and partial application, recursive data structures, type classes, polymorphism, the IO monad, and much of the complexity of Haskell 98's pattern matching. General recursion is eschewed in favor of an explicit fixed point operator which operates only on tail-recursive functions.

Another reason to define CT as a standalone language is that, as it will be independent of Haskell implementations, the ultimate results of this research can be re-used far more readily in a variety of settings. There is no high assurance run-time system for Haskell, so relying on the GHC run-time, for example, just "kicks the high assurance can down the road." For embedded systems and many varieties of system software (e.g., garbage collectors, network controllers, device drivers, web servers, etc.), the size of the object code produced would have to be small to be practical, thus eliminating all current Haskell implementations. The presence of garbage collection in Haskell run-time systems is completely unacceptable for some software systems because of timing issues (e.g., flight control software). It is fortunate, therefore, that CT does not require the full power of the Haskell RTS with its attendant unpredictable time and space behavior.

When considered individually, software compilation and hardware synthesis represent signal achievements in Computer Science. While there has been some success with mixed target compilation—i.e., the generation of either hardware or software implementations from a single source—the record is mixed. The challenge derives in trying to compile the same specification into targets—i.e., hardware and software—that encapsulate vastly different notions of computation. The use of monads in the present work allows us to explicitly tailor the source notion of computation so that it may be implemented efficiently in either hardware or software. The implementation techniques considered here are portrayed in the inset figure. The Cheap Threads language is defined in Section 3. The top route (described in Section 4) shows what is essentially a traditional compiler for a non-traditional language producing RISC MicroBlaze machine language. The lower path (described in Section 5) first generates an intermediate form in *FSMLang*, which is a DSL specifying for abstract state machines. FSMLang provides a clean target for synthesis at a lower level of abstraction than CT and can be readily translated into a netlist for the Xilinx FPGA. The rest of this section presents related work and Section 2 presents an overview of monadic semantics and defunctionalization. Section 6 presents a case study in the application of Cheap Threads to the specification and implementation of software defined radios. Section 7 presents conclusions and future work.



Related Work. Recent research concerns the construction and verification of formal models of separation kernels using concepts from language semantics [15,

14, 12]. These models may be viewed as a domain-specific language (DSL) for separation kernels and can easily be embedded in the higher-order functional programming language Haskell 98 to support rapid prototyping and testing.

The "by layer" approach to implementing monadic programs—i.e., compiling by monad transformer—was first explored by Filinski [9] who demonstrated how the most commonly used layers (i.e., monad transformers) may be translated into a λ-calculus with first-class continuations; the resulting program could be then further compiled in the manner of Appel [5]. Filinski's approach would work here as it handles all of the monad transformers used in CT. Li and Zdancewic [26] show how a monadic language of threads may be implemented via the GHC compiler; their implementation is efficient with respect to execution speed. Their work is not intended to provide efficiency with respect to code size as is ours. Liang and Hudak [27] embed non-proper morphisms of a source monadic expression in Standard ML, thereby providing a means of implementing monadic programs. We opted for a more standard approach to compilation than these as doing so would give more control over the target code produced and, furthermore, seemed more susceptible to verification.

Recent research applies DSLs to bridge the distance between the abstract models of systems and concrete system implementations. DSLs targeting the design, construction, and/or verification of application-specific schedulers are CATAPULTS [39], BOSSA [24], and Hume [10]. DSLs have also been successfully applied to the construction of sensor networks and network processors [25, 7]. The research reported here has similar goals to this work, although we also seek to address high assurance as well; this motivated our use of monadic semantics as a basis for system modeling and implementation.

Semantics-directed compilation (SDC) [35] is a form of programming language compilation which processes a semantic representation of the source program to synthesize the target program. Because one starts from a formal specification of the input program, semantics-directed compilers are more easily proved correct than traditionally structured compilers, and this is the principal motivation for structuring compilers in this manner. This research uses a classic program transformation called *defunctionalization* as a basis for SDC. Defunctionalization is a powerful program transformation discovered originally in the early 1970s by Reynolds [38] that has found renewed interest in the work of Danvy et al. [8, 1, 2]. Hutton and Wright [20] defunctionalize an interpreter for a small language with exceptions into an abstract machine implementation. Earlier, they described a verified compiler for the same language [19].

Monads have been used as a modular structuring technique for DSLs [42] and language compilers [16]. The research reported here differs from these in that monads are taken as a source language to be implemented rather than as a tool for organizing the implementations of language processors.

Lava [6] is a domain-specific language for describing hardware circuitry, embedded in Haskell. Similarly, HAWK [29] is a Haskell-based DSL for the modeling and verification of microprocessor architectures. Both of these DSLs utilize the embedding within Haskell to allow the modeling of hardware signals as Haskell

lazy lists, providing a simple simulation capability. Moreover, Lava and Hawk allow a programmer to define the *structural* implementation of circuits, using the Haskell host language to capture common structural patterns as recursive combinators. This stands in contrast to the work presented in this paper, where we use a monadic language to describe and compile *behavioral* models of systems, in keeping with our goal of enabling verification of high-level system properties.

SAFL [32, 41] is a functional language designed for efficient compilation to hardware circuits. SAFL provides a first-order language, restricted to allow static allocation of resources, in which programs are described behaviorally and then compiled to Verilog netlists for simulation and synthesis. An extension, called SAFL+, adds mutable references and first-class channels for concurrency. CT provides similar capabilities [23], especially in regards to hardware compilation. However, language extensions facilitating stateful and concurrent effects are constructed using the composition of monad transformers to add these orthogonal notions of computation.

2 Background

We assume of necessity that the reader possesses some familiarity with monads and their uses in functional programming and the denotational semantics of languages with effects. This section includes additional background information about monadic semantics intended to serve as a quick review. Readers requiring more should consult the references for further background [31, 28].

Monads. A monad is a triple $\langle M, \eta, \star \rangle$ consisting of a type constructor M and two operations η (unit) and $\star$ (bind) with the following types: $\eta\ :\ a \to M\ a$ and $(\star)\ :\ M\ a \to (a \to M\ b) \to M\ b$. These operations must obey the well-known *monad laws* [31]. The η operator is the monadic analogue of the identity function, injecting a value into the monad. The $\star$ operator is a form of sequential application. The "null bind" operator, $\gg\ :\ M\ a \to M\ b \to M\ b$, is defined as: $x \gg k\ =\ x \star \lambda\_.k$. The binding (i.e., "$\lambda\_$") acts as a dummy variable, ignoring the value produced by x.

Recent research [12, 18] has demonstrated how concurrent behaviors (including process synchronization, asynchronous message passing, preemption, forking, etc.) may be described formally and succinctly using monadic semantics. These kernels are constructed using *resumption* monads [34]. *Resumptions* are a denotational model for concurrency discovered by Plotkin [37] that were later formulated as monads by Moggi [31]. Intuitively, a resumption model views a program as a (potentially infinite) sequence of *atoms*, $[a_0, a_1, \ldots]$, where each a_i may be thought of as an atomic machine instruction. Concurrent execution of multiple programs may then be modeled as an interleaving of each of these program threads (or as the set of all such interleavings).

Monad transformers allow us to easily combine and extend monads. There are various formulations of monad transformers; we follow that given in Liang et al. [28]. Below we give several equivalent definitions for both a "layered" state monad, K, and a resumption monad, R. The first definition of monad K

is in terms of state monad transformers, $StateT\ Reg_i$, and the identity monad, $I\ a = a$. The state types, Reg_i, can be taken, for the sake of this article, to represent integer registers. The resumption monad, R, is defined first with the resumption monad transformer $ResT$, and then in Haskell. The definitions of the state monad and resumption transformers can be found elsewhere [28, 12].

$$\begin{array}{ll} K & = StateT\ Reg_1\ (\cdots (StateT\ Reg_n\ I)\cdots) \\ K\ A & \cong Reg_1 \to \cdots \to Reg_n \to (A \times Reg_1 \times \cdots \times Reg_n) \\ R & = ResT\ K \\ \textbf{data}\ R\ a = & Done\ a \mid Pause\ (K\ (R\ a)) \end{array}$$

These two monads define the following language:

$$get_i : K\ Reg_i \qquad put_i : Reg_i \to K\ () \qquad step : K\ a \to R\ a$$

The operation, get_i, reads the current contents of the ith register and $(put_i\ v)$ stores v in the ith register. The operation, $step\ \varphi$, makes an atomic action out of the K-computation φ. Monadic operations are sometimes referred to as *non-proper* morphisms.

Resumption based concurrency is best explained by an example. We define a *thread* to be a (possibly infinite) sequence of "atomic operations." Think of an atomic operation as a single machine instruction and a thread as a stream of such instructions characterizing program execution. Consider first that we have two simple threads $a = [a_0; a_1]$ and $b = [b_0]$. According to the "concurrency as interleaving" model, concurrent execution of threads a and b means the set of all their possible interleavings: $\{[a_0; a_1; b_0], [a_0; b_0; a_1], [b_0; a_0; a_1]\}$.

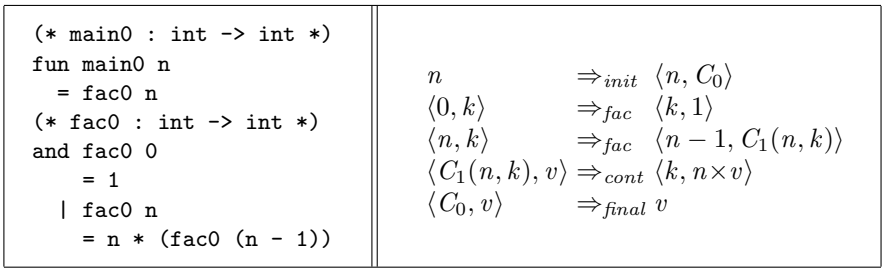
The $ResT$ monad transformer introduces lazy constructors *Pause* and *Done* that play the rôle of the lazy cons operator in the stream example above. If the atomic operations of a and b are computations of type $K\ ()$, then the computations of type $R\ ()$ are the set of possible interleavings:

$$\begin{array}{l} Pause\,(a_0 \gg \eta(Pause\,(a_1 \gg \eta(Pause\,(b_0 \gg \eta(Done\,()))))))) \\ Pause\,(a_0 \gg \eta(Pause\,(b_0 \gg \eta(Pause\,(a_1 \gg \eta(Done\,()))))))) \\ Pause\,(b_0 \gg \eta(Pause\,(a_0 \gg \eta(Pause\,(a_1 \gg \eta(Done\,())))))))) \end{array}$$

In CT, these threads would be constructed without reference to *Pause* and *Done* using *step*: $(step\ a_0 \gg\!=_R step\ a_1 \gg\!=_R step\ b_0)$; this thread is equal to the first one above.

Just as streams are built with a lazy "cons" operation $(h : t)$, the resumption-monadic version uses an analogous formulation: $Pause\,(h \gg \eta t)$. The laziness of *Pause* allows infinite *computations* to be constructed in R just as the laziness of cons in $(h : t)$ allows infinite *streams* to be constructed.

A refinement to the concurrency model provided by $ResT$, called *reactive* resumptions [12], supports a request-and-response form of concurrent interaction, while retaining the same notion of interleaving concurrency. Reactive concurrency monads also define a *signal* construct for sending signals between threads. We do not consider the *signal* construct in this article, although it is used in the case study in Section 6. Its implementation is similar to a procedure call.

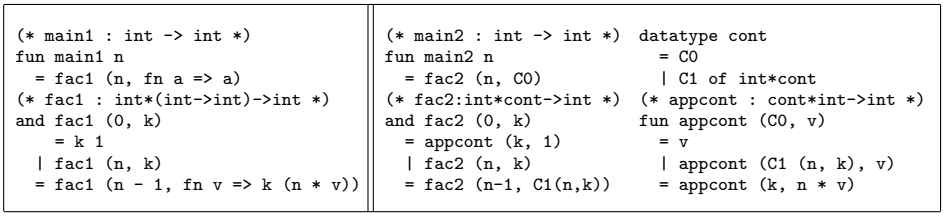


```
(* main0 : int -> int *)
fun main0 n
  = fac0 n
(* fac0 : int -> int *)
and fac0 0
    = 1
  | fac0 n
    = n * (fac0 (n - 1))
```

$$
\begin{array}{lll}
n & \Rightarrow_{init} & \langle n, C_0 \rangle \\
\langle 0, k \rangle & \Rightarrow_{fac} & \langle k, 1 \rangle \\
\langle n, k \rangle & \Rightarrow_{fac} & \langle n-1, C_1(n,k) \rangle \\
\langle C_1(n,k), v \rangle & \Rightarrow_{cont} & \langle k, n \times v \rangle \\
\langle C_0, v \rangle & \Rightarrow_{final} & v
\end{array}
$$

Fig. 1. *Defunctionalizing Produces an Abstract State Machine.* The factorial function, `fac0` (left), is transformed via defunctionalization into the state machine (right). Example originally appears in Danvy [2].

Defunctionalization. Defunctionalization is a program transformation (due originally to Reynolds [38] and rejuvenated of late by Danvy et al. [1, 2]) that transforms an arbitrary higher order functional program into an equivalent abstract state machine. Defunctionalization may also be performed on monadic programs as well [2]. This section reviews defunctionalization.



```
(* main1 : int -> int *)
fun main1 n
  = fac1 (n, fn a => a)
(* fac1 : int*(int->int)->int *)
and fac1 (0, k)
    = k 1
  | fac1 (n, k)
  = fac1 (n - 1, fn v => k (n * v))
```

```
(* main2 : int -> int *)
fun main2 n
  = fac2 (n, C0)
(* fac2:int*cont->int *)
and fac2 (0, k)
  = appcont (k, 1)
  | fac2 (n, k)
  = fac2 (n-1, C1(n,k))
```

```
datatype cont
  = C0
  | C1 of int*cont
(* appcont : cont*int->int *)
fun appcont (C0, v)
  = v
  | appcont (C1 (n, k), v)
  = appcont (k, n * v)
```

Fig. 2. *Defunctionalization process for the factorial function.* The function, `fac0` (see Fig. 1, left), is transformed into an equivalent state machine by CPS transformation (left), closure conversion (right), and defunctionalization (see Fig. 1, right).

Defunctionalizing the factorial function (Fig. 1, left) produces an equivalent state machine (Fig. 1, right). In this machine, there are three types of configurations on which the rules act; integers are initial/final configurations, pairs of type `int*cont` and pairs of type `cont*int`. The translation first performs the continuation-passing style (CPS) transformation (Fig. 2, left) to expose control flow and then performs closure conversion (Fig. 2, right) to represent the machine states as concrete, first-order data. The function `fac2` resulting from closure conversion is then reformatted into the familiar arrow style for rewrite rules (Fig. 1, right).

Defunctionalization for monadic programs proceeds along precisely the same lines as before once the definitions for the monadic operators (i.e., η, $\star$, and non-proper morphisms) are unfolded [2]. CT programs are simpler to defunc-

tionalize than general higher-order functions. Because the control flow within a CT program is already made manifest by its monadic structure, the CPS transformation is unnecessary. We show below in Section 5 how CT programs may be defunctionalized.

```
monad K = StateT(Int) G
monad R = ResT K

actionA  :: K ()
actionA  = get_G ⋆_K λ g. put_G (g + 1)

actionB  :: K ()
actionB  = get_G ⋆_K λ g. put_G (g − 1)

chan     :: Int → Int → R ()
chan     = fix (λ κ. λ a. λ b.
                step (put_G a >>_K actionA >>_K get_G) ⋆_R λ newa.
                step (put_G b >>_K actionB >>_K get_G) ⋆_R λ newb.
                  κ newa newb)

main     :: R ()
main     = chan 0 0
```

Fig. 3. Example Cheap Threads program

3 Defining the Cheap Threads Language

The CT language is a proper subset of Haskell 98, extended with a special declaration form for specifying monads. It shares a concrete syntax with Haskell—in fact, any CT program may be tested in a standard Haskell environment such as GHC or Hugs, provided that Haskell definitions are supplied for the monads declared in that program. The implementation of tuples and algebraic data types, while straightforward, is not discussed in this paper in order to simplify the presentation. These features are not needed for the case study.

Figure 4 gives a grammar for CT. A program consists of one or more declarations, which may be a type signature, a function declaration, a data declaration, or a monad declaration. All function declarations must be preceded by an explicit type signature. The distinguished symbol *main* serves as the main entry point to the program and must have type $R\ ()$. An example program is given in Figure 3. The example defines two atomic state operations *actionA* and *actionB*, which respectively increment and decrement the global register G. The function

chan interleaves an infinite sequence of *actionA* operations with an infinite sequence of *actionB*. Between atomic operations, *chan* performs a context switch by saving the value of G in process A, and restoring the value of G in process B (or vice versa). As a result, the processes A and B do not affect each other's execution, even though they both make use of the same global register.

```
program   ::= decl*
decl      ::= tysig
            | fundecl
            | datadecl
            | monaddecl
fundecl   ::= ident ident* = expr
datadecl  ::=
    data dtype = condecl {| condecl}*
condecl   ::= constr basetype*
monaddecl ::= monad K = layer {+ layer}*
            | monad R = ResT K
layer     ::= StateT (basetype) ident
tysig     ::= ident :: type
basetype  ::= Int
            | Bool
            | ()
            | (basetype{,basetype}+)
            | dtype
type      ::= (type)
            | basetype
            | type → type
            | m basetype
pat       ::= _
            | ident
            | constr ident*
            | (ident{,ident}+)
```

```
expr   ::= (expr)
         | expr expr
         | expr binop expr
         | integer_literal
         | boolean_literal
         | -expr
         | if expr then expr else expr
         | case expr of
             {pat → expr}*
         | (expr{,expr}+)
         | ()
         | expr ⋆m expr
         | expr ⋆m lambda
         | expr >>=m expr
         | ηm expr
         | fix expr
         | fix lambda
         | get_ident
         | put_ident expr
         | step expr
lambda ::= (lambda)
         | λident.expr
         | λident.lambda
m      ::= K | R
```

Fig. 4. Grammar for the Cheap Threads language.

While CT's concrete syntax is borrowed from Haskell, it is fundamentally a much simpler language. We note a few important distinctions at the outset. **Declaration Form for Monads.** A special *monad* declaration form is used to construct state and resumption monads. The types and morphisms associated with these declarations are built in to the language—they are not represented in terms of the source language. We require that a program define exactly one state monad named K, and one resumption monad named R defined in terms of K. **Recursion Tightly Controlled, No Mutual Recursion.** Recursive functions may be defined only by explicit use of the fixed point operator *fix*, which only accepts tail-recursive functions. Any function not defined in terms of *fix* is total. Recursion without *fix* is not possible, because the body of a function may not refer to that function's name, nor to any named function that does not precede it in the source text. Algebraic data types also are not allowed to be recursive. **Simplified Type System.** The type system dispenses entirely with polymorphism and type classes. **No Higher-Order Functions.** The *only* place where the type system allows a higher-order function to be used is as the operand to *fix*. Lambda expressions are only allowed to occur on the right-hand

side of a monadic "bind" operator, or as the operand of *fix*. **Simplified Pattern Matching.** Pattern matching is limited to deconstructing algebraic data types and tuples, and may occur only in the context of a *case* expression. Patterns may not be nested.

Monad Declarations. CT provides built-in support for constructing monads from the standard state monad transformer *StateT* [28], as well as the resumption monad transformer *ResT* [34, 12] for concurrency. Monads are specified by a special declaration form, similar to (but less general than) that provided by MonadLab [22]. The example in Figure 5 (top) defines two monads K and R. Monad K is built from three applications of the state monad transformer, reflecting a notion of state comprised of two *Int*-typed registers and one *Bool*-typed flag. Note that *StateT* components must have unique names. Monad R applies the resumption transformer to K, enabling us to express interleavings of K-computations.

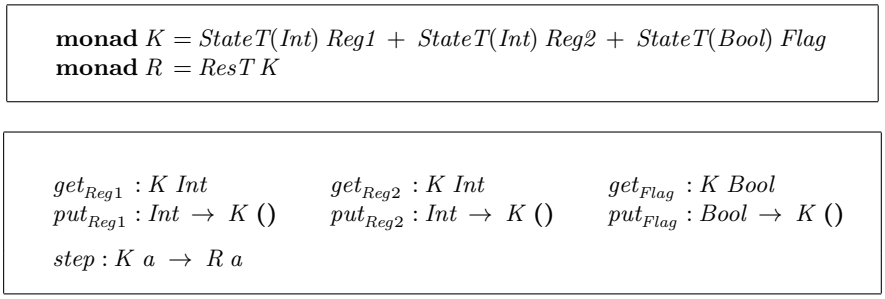


Fig. 5. Example monad declarations in Cheap Threads (top). Non-proper morphisms produced by the declarations (bottom). N.b., *step* can be typed at any base type a; CT is not polymorphic.

These declarations produce "bind" and "unit" operations for the K and R monads, along with the non-proper morphisms for state and resumptions given in Figure 5 (bottom). It is important to note that these operations are primitive within the CT language; unlike their Haskell counterparts, they are *not* implemented in terms of newtypes, higher-order functions, etc., but instead are handled directly by the compiler.

Restricting Recursion. An important design consideration of CT is that it should be implemented in a straightforward manner with loop code. To that end, one important distinction between CT and Haskell is that recursion in CT is strictly limited to tail recursive functions whose codomain is in the resumption monad. Recursive declarations must include explicit applications of the fixed point operator *fix*. This operator takes a function whose type is of the form

$$(\tau_1 \to \tau_2 \to \cdots \to \tau_n \to R\ t) \to (\tau_1 \to \tau_2 \to \cdots \to \tau_n \to R\ t)$$

where τ_1, τ_2, $\cdots$, τ_n, t are base types (i.e. non-functional and non-monadic), and iterates it. A static check, separate from the type system, enforces the requirement that the iterated function be tail recursive. Algebraic data types also are not allowed to be recursive.

Type System. CT is a simply-typed language with primitive types *Int*, *Bool*, and $()$; tuples and non-recursive algebraic data types; function types; and monadic types. Support for higher-order functions is limited. A simply-typed system was chosen over a polymorphic one because it makes it easier to restrict type expressiveness. Because the typing rules are mostly standard, we will discuss only the unusual cases.

Due to the fact that functions are not true first-class values, partial application is not allowed, and the use of higher-order functions is limited to the built-in *fix* operator. These restrictions are expressed by the rule for application:

$$\frac{\Gamma \vdash e_1, e_2, \cdots, e_n : \tau_1, \tau_2, \cdots, \tau_n \quad \Gamma \vdash f : \tau_1 \to \tau_2 \to \cdots \to \tau_n \to t}{\Gamma \vdash f\ e_1\ e_2\ \cdots\ e_n : t}\ (\tau_1, \cdots, \tau_n, t \text{ do not contain } \to)$$

Note that while this rule does not stop the programmer from *defining* higher-order functions, it does preclude the *application* of higher-order functions. We make this distinction rather than excise higher-order functions altogether because *fix* actually does operate on higher-order functions of a certain form.

The monadic "bind" and "unit" operators for each supported monad are built in to the language. Rather than supply a single set of operators overloaded over all monads, the operators are subscripted to make explicit the monad in which they operate. In each of the following rules, m may stand for a monad (K or R in Figure 5 (top)), and τ, τ' must be base types.

$$\frac{\Gamma \vdash \varphi : m\,\tau \quad \Gamma \vdash f : \tau \to m\,\tau'}{\Gamma \vdash \varphi \star_m f : m\,\tau'} \qquad \frac{\Gamma \vdash \varphi : m\,\tau \quad \Gamma \vdash \varphi' : m\,\tau'}{\Gamma \vdash \varphi \gg_m \varphi' : m\,\tau'}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \eta_m\, e : m\,\tau}$$

State and resumption monad operations are also built in, and have analogous types to their Haskell counterparts [28, 34]. If the state monad K has a component $\mathit{StateT}(\tau)\ \mathit{Elem}$, it is assumed that the tags (e.g., *Elem*) are unique. The state operations $\mathit{get}_{\mathit{Elem}}$ and $\mathit{put}_{\mathit{Elem}}$ are typed as follows:

$$\frac{}{\Gamma \vdash \mathit{get}_{\mathit{Elem}} : K\ \tau} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathit{put}_{\mathit{Elem}}\, e : K\ ()}$$

The *step* morphism of the R monad is typed as follows:

$$\frac{\Gamma \vdash \varphi : K\,\tau}{\Gamma \vdash \mathit{step}\,\varphi : R\,\tau}$$

Finally, the resumption monad R has an associated fixed point operator *fix*:

$$\frac{\Gamma \vdash f : (\tau_1 \to \cdots \to \tau_n \to R\,t) \to (\tau_1 \to \cdots \to \tau_n \to R\,t)}{\Gamma \vdash \mathit{fix}\ f\ :\ \tau_1 \to \cdots \to \tau_n \to R\,t}$$

where τ_1, τ_2, $\cdots$, τ_n, t are base types. Use of *fix* is subject to the further restriction (enforced by a static check outside the type system) that f is tail recursive. Notice that the argument to *fix* is a higher-order function. As we mentioned above, this is the *only* place in CT where user-defined higher-order functions may be used.

4 Compiling Cheap Threads

In this section we discuss the compilation of CT to intermediate code. The compiler targets the intermediate language described in Table 1, which we will refer to as the RTL. The RTL is a simple register transfer language. Two types of registers exist: general-purpose virtual registers (`r`n), and "named" registers (`r`s where s is a string beginning with a non-numeric character) which are used to hold global state (i.e. state components of K).

Instruction	Meaning
l`:`	Label.
r `:=` n	Stores the constant value n in register r.
$r1$ `:=` $r2$	Copies the value in register $r2$ into register $r1$.
$r1$ `:=` $r2$ `+` $r3$	Adds the value stored in $r2$ to the value stored in $r3$ and stores the result in $r1$.
$r1$ `:=` $r2$ `-` $r3$	Subtracts the value stored in $r3$ from the value stored in $r2$ and stores the result in $r1$.
`BZero` r l	Jumps to label l if the value stored in r is zero.
`Jump` l	Jumps to label l.

Table 1. The intermediate language targeted by the compiler.

It is a simple matter to generate instructions for a RISC-like architecture from our intermediate language. We have implemented an instruction selection phase for MicroBlaze, a 32-bit RISC soft processor designed to be run on FPGA fabric. Instruction selection is entirely straightforward, so we omit the details.

4.1 Translation from Cheap Threads

This section describes the compilation of CT to the RTL. The compiler is implemented in Haskell, using the parser from the `haskell-src` library. Compilation proceeds in three passes: type checking (see Section 3), inlining, and code generation.

Inlining. In the inlining pass, all calls to non-recursive functions are inlined, starting with *main* at the root of the call tree. The output from the inliner is essentially one "giant term" in which the only function applications that occur are those of recursive functions, and the only variable references that occur are to λ-bound variables, which may appear only in a function passed to *fix* or on the right-hand side of $\star_m$.

Code Generation. After inlining, we generate code for the resulting term in a syntax-directed fashion. The code generation function *codegen* returns a sequence of RTL commands implementing the source expression, and the register in which those commands store the result, if any. Let *CtExpr* be the data type corresponding to *expr* in Figure 4, and *RtlCom* be the data type representing RTL commands. Then the type of *codegen* is $\mathit{CtExpr} \to \mathit{CM}\ ([\mathit{RtlCom}], \mathit{Register})$, where *CM* is a monad providing mechanisms for generating fresh registers and labels, and binding variables to registers.

Notational Convention. For the sake of brevity, we will describe the translation rules according to the convention that $\ulcorner e \urcorner$ is the code generated by translating the expression e, and anywhere after an occurrence of $\ulcorner e \urcorner$, we may refer to the register in which e's result is stored as r_e. That is, $\ulcorner e \urcorner$ is the list of *RtlCom*s returned by *codegen* e, and r_e is the *Register* returned by *codegen* e. We use Haskell-style comments (beginning with a long dash — and extending to the end of a source line), and a comment at the end of the translation rule indicates which register is the result register.

Compiling Pure Expressions. We first consider the compilation of pure, that is non-monadic, expressions. If the expression to be compiled is of type *Int*, *Bool*, or `()`, the result is simply stored in a freshly-generated register. For example, we compile addition as follows:

$$\ulcorner e_1 + e_2 \urcorner = \ulcorner e_1 \urcorner\ ;\ \ulcorner e_2 \urcorner\ ;\ r\ \texttt{:=}\ r_{e_1}\ \texttt{+}\ r_{e_2}$$
— Where r is fresh. Result register is r.

Values of type `()` have no result register.

Compiling Monadic Expressions. The $\star_m$ and $\texttt{>>}_m$ operators are compiled much like function composition. Assume without loss of generality that terms on the right-hand side of $\star_m$ are always λ-expressions. Then:

$\ulcorner \varphi \star_m \lambda x.\chi \urcorner = \ulcorner \varphi \urcorner\ ;\ \ulcorner \chi[x \mapsto r_\varphi] \urcorner$ — Result register is r_χ.
$\ulcorner \varphi \texttt{>>}_m \varphi' \urcorner = \ulcorner \varphi \urcorner\ ;\ \ulcorner \varphi' \urcorner$ — Result register is $r_{\varphi'}$.

where $\ulcorner \chi[x \mapsto r_\varphi] \urcorner$ denotes that χ is compiled in the current environment, extended by binding variable x to register r_φ.

The "unit" operator η_m serves to inject pure computations into a monad. The resumption monad operator *step* serves a similar function, lifting K computations into R. For *step* we generate a label to delineate "*step*-ed" state operations; for the purposes of this paper, these labels are purely informational.

$\ulcorner \eta_m\ e \urcorner = \ulcorner e \urcorner$ — Result register is r_e.

$\ulcorner step\,\varphi \urcorner = l : \ulcorner \varphi \urcorner$ — Where label l is fresh. Result register is r_φ.

Finally, the state monad morphisms get_{Elem} and put_{Elem} simply read or modify the associated register `r`*Elem*.

$\ulcorner get_{Elem} \urcorner = r$ `:=` `r`*Elem* — Where r is fresh. Result register is r.
$\ulcorner put_{Elem}\ e \urcorner = \ulcorner e \urcorner$; `r`*Elem* `:=` r_e — No result register.

Compiling *fix*. As we mentioned previously, functions produced by the *fix* operator must be tail recursive. This means that we can compile applications of *fix* to simple loop code. Let f be any function satisfying this condition, with parameters named $\kappa, v_1, v_2, \cdots, v_n$. Call the body of this function b.

$$\begin{aligned}\ulcorner (fix f)\ e_1\ e_2\ \cdots\ e_n \urcorner = {} & \ulcorner e_1 \urcorner ; \ulcorner e_2 \urcorner ; \cdots \ulcorner e_n \urcorner \\ & l : \\ & \quad \ulcorner b \urcorner \quad \text{— in a special environment—see below} \\ & \quad \text{— Where label } l \text{ is fresh. Result register is } r_b.\end{aligned}$$

Expression b is compiled in an environment where $v_1, v_2, \cdots, v_n$ are bound to registers $r_{e_1}, r_{e_2}, \cdots, r_{e_n}$, and in which application of κ is compiled by the following special rule:

$$\begin{aligned}\ulcorner \kappa\ e'_1\ \cdots\ e'_n \urcorner = {} & \ulcorner e'_1 \urcorner ; \ulcorner e'_2 \urcorner ; \cdots ; \ulcorner e'_n \urcorner \\ & r_{e_1} := r_{e'_1}\ ;\ r_{e_2} := r_{e'_2}\ ;\ \cdots\ ;\ r_{e_n} := r_{e'_n} \\ & \texttt{Jump}\ l \\ & \quad \text{— No result register.}\end{aligned}$$

Note that the translation schema given above is slightly simplified in that it produces a separate loop for each (outermost) application of a recursive function, resulting in code duplication. This duplication can easily be avoided by memoizing the label at which $\ulcorner b \urcorner$ is stored.

Example. Figure 6 gives the code generated for the example program in Figure 3. Each monadic operation compiles to only a handful of RTL instructions. The resulting code contains only 17 instructions, which one may reasonably expect, at an extremely conservative estimate, to translate to at most a few hundred machine language instructions on any given architecture.

By comparison, ghc-6.10.1 compiling the same code produces a 650-kilobyte binary when targeting x86 on Windows with optimization enabled (disabling optimization produces a slightly larger binary). We believe that most of this code (432 kilobytes) is accounted for by the Haskell runtime system, and much of the remainder is code for various prelude and library functions. Of course, we do not claim that this comparison provides evidence that the CT compiler is "better" than GHC—obviously, GHC compiles a far more expressive source language, and therefore an increase in code size is inevitable. But the comparison does highlight a major advantage of a directly-compiled DSL over one embedded in Haskell: the code produced by Cheap Threads is several orders of magnitude smaller, making it more suitable for use in embedded systems.

```
     -- Init G-save for A and B          12:
     r1 := 0                                  -- Restore G for process B
     r2 := 0                                  rG := r2
mainloop:                                     -- Execute actionB
l1:                                           r7 := rG
     -- Restore G for process A               r8 := 1
     rG := r1                                 r9 := r7 - r8
     -- Execute actionA                       rG := r9
     r3 := rG                                 r10 := rG
     r4 := 1                                  -- Save G vals for next iteration
     r5 := r3 + r4                            r1 := r6
     rG := r5                                 r2 := r10
     r6 := rG                                 -- Loop
                                              Jump mainloop
```

Fig. 6. RTL code for the program in Figure 3

5 Synthesizing Circuits from Cheap Threads Programs

Producing a circuit from a Cheap Threads program proceeds in two steps. First, the source program is defunctionalized, producing an abstract state machine that is readily formatted in the syntax of the FSMLang DSL. The FSMLang compiler is used to produce VHDL code from which a hardware implementation on an FPGA may be synthesized. Section 5.1 defines the defunctionalization transformation for CT. Section 5.2 describes the design, syntax and implementation of FSMLang.

5.1 Defunctionalizing Cheap Threads.

This section formulates the defunctionalization transformation for CT. The resulting state machine, $\langle \mathit{States}, \mathit{Rules} \rangle$, consists of a set of states, *States*, and a set of transformation rules, *Rules*, of type $\mathit{States} \to \mathit{States}$. Defunctionalization takes place "by layer" in that terms typed in K are defunctionalized separately from those typed in R.

Defunctionalizing Layered State Monads. The states and rules of the target machine arise directly from the definitions of K and the source term being transformed, respectively. Let us assume that K is a state monad of the form, $K = \mathit{StateT}\ \mathit{Reg}_1\ (\cdots\ (\mathit{StateT}\ \mathit{Reg}_n\ I)\cdots)$. The elements of *States* are tuples determined by (1) the λ-bound variables within the program, (2) the states Reg_i within monad K, and an additional component for the value returned by the computation. Variables bound by λ are assumed without loss of generality to be unique and any *fix* bound variables (e.g., the "κ" in "$\mathit{fix}(\lambda\kappa.\cdots)$") are not considered λ-bound. For the language defined in Section 3, the type of return values will always be one of *Int*, *Bool*, or $()$; let type $\mathit{Val} = \mathit{Int} + \mathit{Bool} + ()$.

Taking this into consideration, the elements of *States* of the target machine have type $\mathit{Reg}_1 \times \cdots \times \mathit{Reg}_n \times \mathit{Var}_1 \times \cdots \times \mathit{Var}_m \times \mathit{Val}$. For each λ-bound variable or state in K, there is a corresponding component within the *States* type. Define c as the total number of components as: $c = m + n$. We define the update and read transitions, upd_x and read_{x_i}, as:

$$(x_1, \cdots, x, \cdots, x_c, v) \mapsto_{upd_x} (x_1, \cdots, v, \cdots, x_c, v)$$
$$(x_1, \ldots, x_i, \ldots, x_c, v) \mapsto_{read_{x_i}} (x_1, \ldots, x_i, \ldots, x_c, x_i)$$

The upd_x transition sets the x "slot" to the value component while the $read_{x_i}$ transition sets the value component to the current contents of x_i.

Each K term gives rise to one rule only and is derived in a syntax-directed manner. The get, put and unit operations are straightforward. Below, assume that $1 \leq i \leq n$ (i.e., put_i and get_i are defined only on components corresponding to K states) and let $s = (x_1, \ldots, x_c, v)$ be the input state in:

$$\begin{aligned} K\lceil put_i\ e\rceil &= s \mapsto (x_1, \ldots, eval\ e\ s, \ldots, x_c, ()) \\ K\lceil get_i\rceil &= read_{x_i} \\ K\lceil \eta_K\ e\rceil &= s \mapsto (x_1, \ldots, x_c, eval\ e\ s) \end{aligned}$$

The unit computation, $\eta_K\ e$, is defunctionalized to a transition that only updates the return value state component to the value of e in the input state s, $eval\ e\ s$. The definition of *eval* is not given as it is an unremarkable expression interpreter. Note that, by construction, expression e will be of base type and will only refer to the components corresponding to λ-bound variables.

The bind operations for K, $\star_K$ and $>>_K$, are defined in terms of function composition. The transitions are total endofunctions on system configurations and, therefore, possess the standard notion of function composition.

$$\begin{aligned} K\lceil \varphi >>_K \gamma\rceil &= K\lceil\gamma\rceil \circ K\lceil\varphi\rceil \\ K\lceil \varphi \star_K \lambda x.\gamma\rceil &= K\lceil\gamma\rceil \circ upd_x \circ K\lceil\varphi\rceil \end{aligned}$$

Defunctionalizing R. This section first describes the defunctionalization of the resumption layer at a high level. Then equations formulating $R\lceil -\rceil$ are given next. Finally, the results of defunctionalizing the running example are then presented. Defunctionalizing an R computation produces a state machine whose state type includes one more component than does K's:

$$PC \times Reg_1 \times \cdots \times Reg_n \times Var_1 \times \cdots \times Var_m \times Val \quad \text{where} \quad PC = Int$$

This additional component may be thought of as a "program counter" represented as an Int. The resulting state machine also includes multiple transitions.

High-level overview. Whereas layered state adds functionality to CT languages in the form of *put* and *get* operations, the resumption layer adds control mechanisms in the form of *step* and *fix*. Roughly speaking, an R-computation is a sequence of K-computations chained together by $\star_R$ or $>>_R$; using only $>>_R$ for presentation purposes, an R-computation looks like:

$$(step\ \varphi_1)\ >>_R \cdots >>_R\ (step\ \varphi_j)$$

Defunctionalizing each $(step\ \varphi_i)$ is performed by applying $K\lceil -\rceil$ to each φ_i and thereby producing a single corresponding rule, $l_i \mapsto r_i$. Defunctionalizing $>>_R$ in

the above makes the control flow explicit by attaching a "program counter" (starting without loss of generality from 1) to states; this produces the set of rules:

$$\{(1, l_1) \mapsto (2, r_1), \ldots, (j{-}1, l_{j-1}) \mapsto (j, r_{j-1})\}$$

Consider now a fixed computation, $\mathit{fix}\ (\lambda\kappa.\lambda\bar{x}.\gamma)$. As it is tail recursive, occurrences of a recursive call, $\kappa\ \bar{e}\ :\ R\ \mathit{Val}$, will be accomplished by a rule that (1) updates the state variables corresponding to λ-bound variables $\bar{x}$ to the arguments $\bar{e}$ and (2) changing the program counter component to point to the "beginning" of γ.

Detailed formulation. We present the equations defining $R\lceil - \rceil$ in a Haskell-like notation, suppressing certain representation details for the sake of clarity. These equations are written in terms of a layered monad, M, containing an integer state for generating labels and an environment mapping recursively bound variables to their formal parameters. The specification of monad M is given in the MonadLab DSL [22], which allows monads to be specified simply in terms of monad transformers and hides certain technical details (e.g., order of application and lifting of operations through transformers):

$$\begin{array}{ll}\textbf{monad}\ M & = \mathit{EnvT}(\mathit{Bindings})\ \mathit{Env}\ +\ \mathit{StateT}(\mathit{Int}) \\ \textbf{type}\ \mathit{Bindings} & = \mathit{Var}\ \rightarrow\ [\mathit{Var}] \\ \textbf{type}\ \mathit{Var} & = \mathit{String}\end{array}$$

MonadLab generates Haskell code defining the first four functions below; the last operator is defined as: $\mathit{counter}\ =\ \mathit{get}\ \texttt{>>=}\ \lambda i.\ \mathit{put}\ (i+1)\ \texttt{>>}\ \textbf{return}\ i$.

$$\begin{array}{lll}\mathit{rdEnv} & :: M\ \mathit{Bindings} & \text{— read current bindings} \\ \mathit{inEnv} & :: \mathit{Bindings} \rightarrow M\ a \rightarrow M\ a & \text{— resets current bindings} \\ \mathit{get} & :: M\ \mathit{Int} & \\ \mathit{put} & :: \mathit{Int}\ \rightarrow\ M\ () & \\ \mathit{counter} & :: M\ \mathit{Int} & \text{— gensym-like label generator}\end{array}$$

The defunctionalization, $R\lceil e \rceil$, is a computation of the transitions corresponding to e. Defunctionalizing a stepped K-computation first defunctionalizes its argument (φ), producing a transition, $l \mapsto r$. This K-transition is converted into an R-transition by adjoining program counter components to both sides; $(i, (x_1, \ldots, x_c, v))$ is identified with $(i, x_1, \ldots, x_c, v)$. The unit computation, $(\eta_R\ e)$, is translated analogously to *step*:

$$\begin{array}{ll}R\lceil - \rceil & :: \mathit{CTExpr}\ \rightarrow\ M\ [\mathit{Rule}] \\ R\lceil \mathit{step}\ \varphi \rceil & = \mathit{counter}\ \texttt{>>=}\ \lambda i.\ \textbf{return}\ [(i, l) \mapsto (i{+}1, r)] \\ & \quad \textbf{where}\ (l \mapsto r) = K\lceil \varphi \rceil \\ R\lceil \eta_R\ e \rceil & = \mathit{counter}\ \texttt{>>=}\ \lambda i.\ \textbf{return}\ [(i, l) \mapsto (i{+}1, r)] \\ & \quad \textbf{where}\ (l \mapsto r) = K\lceil \eta_K\ e \rceil\end{array}$$

To defunctionalize a recursive expression, one first gets the next fresh label (i) and reads the current bindings (β). In an expanded environment (β') that binds

the recursive variable (κ) to its formal parameters ($v_1, \ldots, v_m$), the body (e) is defunctionalized to a list of rules (ρ). Assuming there is a unique label associated with κ, called L_κ, the transition list ρ is augmented with a transition, $\mathit{mkstart}\ \kappa\ i$, that serves as the "beginning of the loop"; the augmented list is returned:

$$\begin{aligned} R\lceil \mathit{fix}(\lambda\kappa.\lambda v_1.\cdots\lambda v_l.e)\rceil = {} & \mathit{get}\ \texttt{>>=}\ \lambda i. \\ & \mathit{rdEnv}\ \texttt{>>=}\ \lambda\beta. \\ & (\mathit{inEnv}\ \beta'\ R\lceil e\rceil)\ \texttt{>>=}\ \lambda\rho. \\ & \quad \mathbf{return}\ (\mathit{mkstart}\ \kappa\ i\ :\ \rho) \end{aligned}$$

where

$$\begin{aligned} \beta' &= \beta\{\kappa{:=}[v_1, \ldots, v_l]\} \\ \mathit{mkstart}\ \kappa\ i &= (L_\kappa, x_1, \cdots, x_c, v) \mapsto (i, x_1, \cdots, x_c, v) \end{aligned}$$

A recursive call is translated to a transition that takes an input state and moves to a state with label L_κ (defined above) and, in effect, this transition jumps to the head of the "loop". For the sake of simplifying the presentation, the definition below only gives the case where κ has one argument (i.e., κ has type $\tau \to R\tau'$); the full definition is analogous. This recursive call occurs within the body of an expression of form, $\mathit{fix}\ (\lambda\kappa.\lambda x.\ \mathit{body})$. In the following, assume that x is the inner λ-bound variable represented in the state configuration:

$$\begin{aligned} R\lceil \kappa e\rceil = {} & \mathit{counter}\ \texttt{>>=}\ \lambda i. \\ & \quad \mathbf{return}\ [(i, x_1, \ldots, x, \ldots, x_c, v)\ \mapsto\ (L_\kappa, x_1, \ldots, \mathit{eval}\ e\ s, \ldots, x_c, v)] \\ & \mathbf{where}\ s = (x_1, \ldots, x, \ldots, x_c, v) \end{aligned}$$

This presentation is simplified also in that the details of looking up x in the bindings for κ are suppressed. Defining $R\lceil -\rceil$ for the bind operations:

$$\begin{aligned} R\lceil \gamma\ \texttt{>>}_R\ \chi\rceil &= R\lceil\gamma\rceil\ \texttt{>>=}\ \lambda\rho_1.R\lceil\chi\rceil\ \texttt{>>=}\ \lambda\rho_2.\ \mathbf{return}\ (\rho_1 +\!\!+ \rho_2) \\ R\lceil \gamma\ \star_R\ \lambda v.\chi\rceil &= R\lceil\gamma\rceil\ \texttt{>>=}\ \lambda[\rho_1].\ R\lceil\chi\rceil\ \texttt{>>=}\ \lambda\rho_2.\ \mathbf{return}\ (f\ v\ \rho_1\ :\ \rho_2) \\ & \mathbf{where} \\ & \quad f\ ::\ \mathit{Var} \to \mathit{Rule} \to \mathit{Rule} \\ & \quad f\ v\ ((i,s){\mapsto}(i',s')) = ((i,s){\mapsto}(i',\mathit{upd}\ v\ s')) \end{aligned}$$

Example. Returning to the running example presented in Fig. 3; the relevant portion of the channel code is:

$$\begin{aligned} & \mathit{chan} :: \mathit{Reg}_1 \to \mathit{Reg}_2 \to R\ () \\ & \mathit{chan} = \ \mathit{fix}\ \ (\lambda k.\ \lambda a.\ \lambda b. \\ & \qquad\qquad \mathit{step}\ \ (\mathit{put}_G\ a\ \ \texttt{>>}_K\ \ \mathit{actionA}\ \ \texttt{>>}_K\ \ \mathit{get}_G)\ \ \star_R\ \ \lambda \mathit{newa}. \\ & \qquad\qquad \mathit{step}\ \ (\mathit{put}_G\ b\ \ \texttt{>>}_K\ \ \mathit{actionB}\ \ \texttt{>>}_K\ \ \mathit{get}_G)\ \ \star_R\ \ \lambda \mathit{newb}. \\ & \qquad\qquad\quad k\ \ \mathit{newa}\ \ \mathit{newb}\ \ ) \end{aligned}$$

Multiple declarations can be easily accommodated, but rather than elaborate on such details, assume that the two actions stand for particular transitions:

$$\begin{aligned} & (a, b, \mathit{newa}, \mathit{newb}, \mathit{reg}, \mathit{val}) \mapsto (a, b, \mathit{newa}, \mathit{newb}, \mathit{reg}{+}1, ()) \quad -\!\!- \ \mathit{actionA} \\ & (a, b, \mathit{newa}, \mathit{newb}, \mathit{reg}, \mathit{val}) \mapsto (a, b, \mathit{newa}, \mathit{newb}, \mathit{reg}{-}1, ()) \quad -\!\!- \ \mathit{actionB} \end{aligned}$$

The transitions of the state machine produced by defunctionalization are then:

```
initial_state = k
(k,a,b,newa,newb,r,v) -> (1,a,b,newa,newb,r,v)
(1,a,b,newa,newb,r,v) -> (2,a,b,a+1,newb,a+1,a+1)
(2,a,b,newa,newb,r,v) -> (3,a,b,newa,b-1,b-1,b-1)
(3,a,b,newa,newb,r,v) -> (k,newa,newb,newa,newb,r,v)
```

These transitions may be easily reformatted in FSMLang syntax:

```
initial_state = state_k

state_k -> state_1 where
{ }

state_1 -> state_2 where
{  newa' <= a+1;
   r'    <= a+1;
   v'    <= a+1; }

state_2 -> state_3 where
{  newb' <= b-1;
   r'    <= b-1;
   v'    <= b-1; }

state_3 -> state_k where
{  a' <= newa;
   b' <= newb; }
```

This constitutes the `TRANS` section of an FSMLang implementation. The full version includes headers defining a number of initial conditions (e.g., values and sizes of registers, etc.). FSMLang syntax is defined below.

5.2 Overview of FSMLang

FSMLang is a domain-specific language (DSL) for describing finite-state machines (FSMs) [3]. FSMLang targets the configurable logic, embedded memories, and soft-core processors that can be found in modern platform Xilinx FPGAs. FSMLang eliminates the need for a programmer to manually control sensitivity lists, state enumerations, FSM reset behavior, and FSM default output behavior. FSMLang descriptions are much smaller, and less cluttered, than equivalent code written in an HDL. Additionally, the FSMLang compiler is re-targetable – while we focus here on producing hardware, it is also capable of producing FSM implementations in software.

The structure and syntax of an FSMLang program is shown in Figure 7. The `TRANS` section defines the transitions of the state machine. The defunctionalized Cheap Threads program is formatted directly into FSMLang and insert into the `TRANS` section of a template with the other sections defined.

6 Application: Huffman Encoding

Cheap Threads serves as a basis for generating efficient software and hardware implementations from a monadic model. The ability to generate both hardware and software components of a complete system is of major benefit when constructing embedded systems because it allows performance-critical elements to

```
-- Internal state signal names
CS: <current_state_signal_name>;
NS: <next_state_signal_name>;

-- Compile-time variables
GENERICS:
  (<genName>, <type>, <static_value>;)*

-- Definitions of input/output ports
PORTS:
  (<portName>, <in|out>, <type>;)*
CONNECTIONS:
  (<outputPortName>   <=  <rhs>;)*

-- Definitions of memories
MEMS:
 (<mName>,
 <dataWidth>, <addrWidth> [,EXTERNAL];)*
```

```
-- Definitions of FIFO channels
CHANNELS:
  (<channelName>, <dataWidth>;)*

-- Internal FSM signals
SIGS:
  (<sigName>, <type>;)*

-- Definition of logic/transitions
INITIAL:
  <stateName>;
TRANS:
(
<curr_st> [|<bool_guard>] -> <next_st>
    [where { (<lhs>  <= <rhs>;)* } ]
)*

-- Native VHDL Definitions
VHDL: <un-parsed VHDL code>
```

Fig. 7. *FSMLang Program Structure and Syntax.*

be compiled into hardware, capitalizing on the implicit parallelism that that fabric provides, while at the same time allowing less performance-critical components to standard microprocessors.

The Computer Systems Design Laboratory at the University of Kansas has used monad compilation for the implementation of software-defined radios. In contrast to a traditional radio, which generally consists of a series of analog and digital hardware components to implement a specific type of radio (called a waveform in the nomenclature of the domain), a software-defined radio (SDR) uses the flexibility of the (traditionally software) platform to allow the radio to be reconfigured to support a variety of different waveforms, as application requirements demand [30].

A typical software defined radio will include a variety of components performing digital signal processing such as modulation, spreading, error correction, compression, and encryption. As an example component, Figure 8 shows the definition of a simple Huffman decoder component. Huffman compression is a simple form of data compression which encodes fixed-sized data into stream of variable-sized symbols, and decoding performs the inverse operation. For example, in the decoder definition, the component converts a stream of bits into a stream of integers. The number of bits needed to represent an integer on the input stream varies depending on the frequency that a particular integer occurs in the original (uncompressed) text. This allows integers that occur relatively frequently to be encoded with fewer bits, which in turn reduces the number of bits that must be transmitted.

The decoder uses a tree to represent the encoding of integers. The path from the root of the tree to a leaf identifies the encoding of the integer stored at that leaf. Cheap Threads does not allow recursive data types, so a tree is represented using a state monad, where each value in the state is a *Node*. A *Node* can either be *Emit*, which indicates an integer value, or *Branch*, which indicates an additional

```
data Node  = Emit Int | Branch Int Int
data BMsg  = BRead | BWrite Bit | BVal Bit
data IMsg  = IRead | IWrite Int | IVal Int
data Bit   = Low | High

decoder pos tree input output =
    get tree pos ⋆R λval.
    case val of
     Emit v      →  signal output (IWrite v) >>R decoder 0 tree input output
     Branch l r  →  signal input BRead ⋆R λi.
                    case i of
                    BVal bit →  case bit of
                                  Low   →  decoder l tree input output
                                  High  →  decoder r tree input output
```

Fig. 8. *Huffman Decoder*. In this example, the *fix* is implicit and the R monad includes another operator, *signal*. See text for further description.

bit is needed to encode the values in the sub-trees. The two fields for the *Branch* constructor represent the addresses of the left and right child nodes in the state. To generate the encoding for a node, follow the path from the root of the tree (at address 0). At each *Branch*, if the path enters the left child, then generate a 0 (*Low*) bit, and if the path enters the right child, generate a 1 (*High*) bit.

For example, consider the Huffman tree representing the encoding for arbitrary integers a, b, and c. Given an encoding with a as "0", b as "10", and c as "11", the associated tree written using a recursive tree structure is *Branch (Emit a) (Branch (Emit b) (Emit c))*. The non-recursive store representation for this tree using the *Node* structure, mapping integer addresses to *Node* values, is $0 \mapsto$ *Node 1 2*, $1 \mapsto$ *Emit a*, $2 \mapsto$ *Node 3 4*, $3 \mapsto$ *Emit b*, $4 \mapsto$ *Emit c*.

The decoder component in Figure 8 takes advantage of an additional monadic construct, called *signal*, which allows isolated concurrent computations to communicate using a message passing scheme. The semantics of *signal* have been described in detail elsewhere [12], as has the compilation of the construct to VHDL and C [23]. In simple terms, *signal* takes request and passes it to an external entity which interprets the request and generates a response. The computation that generates the request is blocked until a response is generated. As such, the construct models an alternative form of concurrency based on message passing, rather than shared state.

In the implementation of the Huffman decoder, the *signal* construct is used to model the consumption of values from an input stream, one at a time, and to generate values on an output stream. This behavior replaces the common use of lazy lists in languages such as Haskell to model infinite streams, defining the stream transformation notion of computation in the monadic model rather than in the host language.

The decoder function receives the encoded stream one bit at a time. The *pos* argument to the function is used to track the current position of the decoder as an address in the state, identified by the *tree* parameter. If that address contains an *Emit* value, then the decoder sends that value, using the *signal* construct, on the output stream. Alternatively, if the current position contains a *Branch* node, then the decoder will read a value from the input stream, again using the *signal* construct. The decoder then makes a tail call, using the value of the input bit to determine the value of the *pos* parameter.

Having defined the Huffman decoder, the component can be compiled to either a hardware or a software implementation and then integrated into the overall desired radio waveform. Moreover, the monadic representation of the component provides a sound basis for constructing an assurance argument for the correctness of the component, a key capability in the software defined radio domain. Compiling the monadic model directly to an executable implementation eliminates the gap between the model used for verification and the resulting implementation.

7 Conclusions

This article presents a foundation for the model-driven engineering of concurrent systems based in semantics-directed compilation. The main vehicle for this approach is the Cheap Threads domain-specific language which encapsulates shared-state concurrency with direct support for state and resumption monads. Cheap Threads may be compiled in a straightforward manner to either a fixed instruction set or to hardware. The defunctionalization program transformation, furthermore, seems to be well-suited to the generation of hardware implementations of Cheap Threads programs.

Defunctionalization is a well-known technique that remained little used until recently. According to Danvy and Nielsen [8], "compared to closure conversion and to combinator conversion, defunctionalization has been used very little." We believe that hardware synthesis from a declarative language is an attractive and practical use-case for defunctionalization. The growing prevalence of FPGA and reconfigurable computing technologies means that expressive source languages for mixed hardware/software systems will only become more important, and this article provides evidence that the state machines produced by defunctionalization are a natural fit for hardware synthesis.

A substantial case study of the compilation of monadic programs to hardware and software has been presented as well, in the form of a Huffman decoder for a software-defined radio. Other substantial examples may be found in Kimmell [23]. These case studies indicate that the approach to MDE described here seems to "scale up" and other applications (esp., monadic separation kernels) are currently under development. Monadic modeling provides a formal basis for verifying high assurance properties of the targeted artifacts. It is expected that the semantics-directed basis will yield benefits in this regard, although formal verification has been left for future work.

Future Work. Hardware/software co-design presents system designers with a continuum: at one end, a system may be implemented entirely in software, and at the other end, it may be implemented entirely in hardware. The present work has explored the extrema of this continuum. An important question for future research is: can we compile some parts of a Cheap Threads program to hardware, and other parts to software, in an efficient manner? One challenge here is in devising an efficient interface for communication between hardware and software logic. Previous work on the Hybridthreads project [4] at the University of Kansas and the University of Arkansas has explored this problem, using C and the POSIX threads API for mixed-target synthesis.

A particularly interesting follow-on revisits a classic paper by Wand [43] which describes a method for deriving abstract machine instruction sets from source language semantics. Modern FPGA technology may enable us to apply similar techniques to synthesize soft processors that directly implement such a derived instruction set. The defunctionalization-based techniques may well shed new light on this classic research.

The High Assurance Security Kernel (HASK) Lab at the University of Missouri is exploring the use of the techniques described here to implement formally verifiable monadic security kernels [15, 14] in hardware and software. The Cheap Threads language, when extended with support for system calls, protected memory, and asynchronous exceptions [33, 13], will provide a separation kernel modeling language with a clear semantics that supports formal verification.

References

1. M. S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. A functional correspondence between evaluators and abstract machines. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP03)*, pages 8–19, New York, NY, USA, 2003. ACM Press.
2. M. S. Ager, O. Danvy, and J. Midtgaard. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. *Theoretical Computer Science*, 342(1):149–172, 2005. Extended version available as BRICS technical report RS-4-28.
3. J. Agron. Domain-specific language for HW/SW co-design for FPGAs. In *Proceedings of the IFIP Working Conference on Domain Specific Languages (DSL09) (to appear)*, July 2009.
4. D. Andrews, W. Peck, J. Agron, K. Preston, E. Komp, M. Finley, and R. Sass. hthreads: a hardware/software co-designed multithreaded RTOS kernel. *Emerging Technologies and Factory Automation, 2005. ETFA 2005. 10th IEEE Conference on*, 2:8 pp.–338, Sept. 2005.
5. A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.
6. P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware design in Haskell. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming (ICFP98)*, pages 174–184, New York, NY, USA, 1998. ACM Press.
7. J. Dai, B. Huang, L. Li, and L. Harrison. Automatically partitioning packet processing applications for pipelined architectures. *SIGPLAN Notices*, 40(6):237–248, 2005.

8. O. Danvy and L. R. Nielsen. Defunctionalization at work. In *Proceedings of the 3rd ACM International Conference on Principles and Practice of Declarative Programming (PPDP01)*, pages 162–174, 2001.
9. A. Filinski. Representing layered monads. In *Proceedings of the 26st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL99)*, pages 175–188. ACM Press, 1999.
10. K. Hammond and G. Michaelson. Hume: A domain-specific language for real-time embedded systems. In *Proceedings of the 2nd International Conf. on Generative Programming and Component Engineering (GPCE03)*, pages 37–56, 2003.
11. W. Harrison. A simple semantics for polymorphic recursion. In *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems (APLAS05)*, pages 37–51, 2005.
12. W. Harrison. The essence of multitasking. In *Proceedings of the 11th International Conference on Algebraic Methodology and Software Technology (AMAST06)*, pages 158–172, July 2006.
13. W. Harrison, G. Allwein, A. Gill, and A. Procter. Asynchronous exceptions as an effect. In *Proceedings of the 9th International Conference on the Mathematics of Program Construction (MPC08)*, volume 5133 of *LNCS*, pages 153–176, 2008.
14. W. Harrison and J. Hook. Achieving information flow security through precise control of effects. In *Proceedings of the 18th IEEE Computer Security Foundations Workshop (CSFW05)*, pages 16–30, Aix-en-Provence, France, June 2005.
15. W. Harrison and J. Hook. Achieving information flow security through monadic control of effects. Invited submission to: *Journal of Computer Security*, 2008. 51 pages. In press. Extends [14].
16. W. Harrison and S. Kamin. Metacomputation-based compiler architecture. In *5th International Conference on the Mathematics of Program Construction (MPC00)*, volume 1837 of *LNCS*, pages 213–229, 2000.
17. W. Harrison and R. Kieburtz. The logic of demand in Haskell. *Journal of Functional Programming*, 15(5):837–891, 2005.
18. W. Harrison and A. Procter. Cheap (but functional) threads. 44 pages. Submitted for publication to *Higher-Order and Symbolic Computation*; extends [12].
19. G. Hutton and J. Wright. Compiling exceptions correctly. In *Proceedings of the 7th International Conference on the Mathematics of Program Construction (MPC04)*, volume 3125 of *Lecture Notes in Computer Science*, Stirling, Scotland, July 2004.
20. G. Hutton and J. Wright. Calculating an exceptional machine. In Hans-Wolfgang Loidl, editor, *Trends in Functional Programming*, volume 5. February 2006.
21. P. Johann and J. Voigtländer. Free theorems in the presence of *seq*. In *Proceedings of the 31st ACM SIGPLAN Symposum on Principles of Programming Languages*, pages 99–110, January 2004.
22. P. Kariotis, A. Procter, and W. Harrison. Making monads first-class with Template Haskell. In *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell (Haskell08)*, pages 99–110, 2008.
23. G. Kimmell. *System Synthesis from a Monadic Functional Language*. PhD thesis, University of Kansas, 2008.
24. J. Lawall, G. Muller, and H. Duchesne. Language design for implementing process scheduling hierarchies (invited paper). In *Proceedings of the ACM SIGPLAN 2004 Symposium on Partial Evaluation and Program Manipulation (PEPM04)*, pages 80–91, August 2004.
25. P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. TinyOS: An operating system for wireless sensor networks. In *Ambient Intelligence*. Springer-Verlag, 2005.

26. P. Li and S. Zdancewic. Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI06)*, pages 189–199, 2007.
27. S. Liang and P. Hudak. Modular denotational semantics for compiler construction. In *Proceedings of the 6th European Symposium on Programming (ESOP96)*, volume 1058 of *Lecture Notes in Computer Science*, pages 219–234, 1996.
28. S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL95)*, pages 333–343. ACM Press, 1995.
29. J. Matthews, B. Cook, and J. Launchbury. Microprocessor Specification in Hawk. In *Proceedings of the IEEE Computer Society 1998 International Conference on Computer Languages (ICCL98)*, pages 90–101, 1998.
30. G. J. Minden, J.B. Evans, L. Searl, D. DePardo, V.R. Petty, R. Rajbanshi, T. Newman, Q. Chen, F. Weidling, J. Guffey, D. Datla, B. Barker, M. Peck, B. Cordill, A. M. Wyglinski, and A. Agah. KUAR: A Flexible Software-Defined Radio Development Platform. In *2nd IEEE Symposium on New Frontiers in Dynamic Spectrum Access Networks (DySPAN)*, Dublin, Ireland, April 2007.
31. E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Dept. of Computer Science, Edinburgh Univ., 1990.
32. A. Mycroft and R. Sharp. A statically allocated parallel functional language. In *Proceedings of the 27th International Colloquium on Automata, Languages and Programming (ICALP00)*, pages 37–48. Springer-Verlag, 2000.
33. J. Palsberg and D. Ma. A typed interrupt calculus. In *Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT02)*, pages 291–310, London, UK, 2002. Springer-Verlag.
34. N. Papaspyrou. A resumption monad transformer and its applications in the semantics of Concurrency. In *Proceedings of the 3rd Panhellenic Logic Symposium*, 2001. An expanded technical report is available from the author by request.
35. L. Paulson. A semantics-directed compiler generator. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL82)*, pages 224–233, 1982.
36. S. Peyton Jones, editor. *Haskell 98 Language and Libraries, Revised Report*. Cambridge Univ. Press, April 2003.
37. G. Plotkin. A powerdomain construction. *SIAM Journal of Computing*, 5(3), 1976.
38. J. Reynolds. Definitional interpreters for higher order programming languages. *ACM Conference Proceedings*, pages 717–740, 1972.
39. M. Roper and R. Olsson. Developing embedded multi-threaded applications with CATAPULTS, a domain-specific language for generating thread schedulers. In *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES05)*, pages 295–303, 2005.
40. D. C. Schmidt. Model-driven engineering. *IEEE Computer*, 39(2), February 2006.
41. R. Sharp and A. Mycroft. The FLaSh compiler: efficient circuits from functional specifications. Technical Report tr.2000.3, AT&T Research, 2000.
42. T. Sheard, Z. Benaissa, and E. Pasalic. DSL implementation using staging and monads. In *Proceedings of the 2nd Conference on Domain-Specific Languages*, pages 81–94, Berkeley, CA, October 3–5 1999. USENIX Association.
43. M. Wand. Semantics-directed machine architecture. In *Proceedings of the 9th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL82)*, pages 234–241, New York, NY, USA, 1982. ACM Press.