System Synthesis from a Monadic Functional Language

By

Garrin Kimmell

Submitted to the graduate degree program in Electrical Engineering and Computer Science and the Graduate Faculty of the University of Kansas School of Engineering in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

The Dissertation Committee for Garrin Kimmell certifies
that this is the approved version of the following dissertation:

System Synthesis from a Monadic Functional Language

Committee:

____
Perry Alexander, Chairperson

____
Andy Gill

____
William Harrison

____
Craig Huneke

____
Gary Minden

Date approved: _______________________________

Abstract

SYSTEM SYNTHESIS FROM A MONADIC FUNCTIONAL LANGUAGE

Garrin Kimmell

The University of Kansas.

Advisor: Perry Alexander

December 2008.

Embedded systems typically combine a mixture of heterogeneous components, some that are software executing on general purpose CPUs, some that are off-the-shelf hardware components, and some that are application specific circuitry. A major challenge when designing and implementing such systems is the dissimilar models of computation exhibited by hardware and software targets. To successfully navigate this challenge, components must be implemented in a way that does not unnecessarily bias the implementation towards either computational model, allowing the components to be retargeted as application requirements change.

This dissertation presents an approach to this problem using a functional programming language extended with monadic imperative and concurrency effects. We argue that these language features allow components to be implemented and compiled to either hardware or software targets. To demonstrate this claim, we detail the design of such a language, Oread. Moreover, we describe the compilation of Oread to both hardware, via VHDL, and software, via C. Using these compilation techniques, we describe the development of a digital processing component in Oread and the integration of that component into a larger system.

Acknowledgments

My appreciation goes to my advisor, Dr. Perry Alexander, who has served me well by providing both the guidance to see this dissertation to completion and the freedom to pursue many paths along the way. His commitment to his students, both in the classroom and in the research lab, serve as model which I will use throughout my life.

The members of my committee, Andy Gill, Bill Harrison, Craig Huneke, and Gary Minden, along with Perry Alexander, deserve thanks for their valuable suggestions for this disseration. Many thanks go to Mr. Ed Komp, who is always eager to listen to any ideas I may have, but always questioning those ideas.

The systems level design group have been a pleasure to work with and have continually made going to work every day something to look forward to. The list of members both current and past has grown considerably in the time I have been working in the lab, but all deserve my gratitude: Cindy, Brandon, Murthy, Kalpesh, Justin, Jesse, Mark, Nick, Phil, and Megan. Special thanks go to Nick Frisby, Wesley Peck, and Jason Agron.

My path through graduate school would have been much shorter if I had not been surrounded by people who made the time so enjoyable: Andy, Halle, Rustin, Howard, Emily, Janet and Mirco, Manoj, and the many that I have played kickball and softball with.

I thank the KU EECS department and ITTC for their financial support during the writing of this dissertation. Thanks to Dr. Peter Ashenden and the University of Adelaide, Australia, provided the opportunity to visit and study.

Finally, I wish to thank my family, including my parents, my brother Rustin, and my wife Morgan, who provided love and support without which this dissertation would not be possible.

for mosho

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Embedded systems typically combine a mixture of heterogeneous components, some that are software executing on general-purpose CPUs, some that are off-the-self hardware components, and some that are application-specific circuitry. The decomposition of the system into individual components and the mapping of those components into an implementation target, or fabric, is motivated by a variety of both functional requirements and performance requirements. The challenge of systems engineering is to integrate the multitude of different requirements into a complete system that respects those requirements and to preserve those requirements as they are refined into executable implementations. As with most tasks, the role of abstraction is central to successfully navigating this challenge: choose a requirements model that is too abstract, and the model loses its predictive power; choose a model that is too concrete and lose the ability to adapt the model to changing requirements or new problems.

This dissertation focuses on the aspect of system engineering related to the refinement of high-level functional requirements to executable implementations. This problem goes beyond simply compiling a program to software or alternatively, synthesize to hardware. When considered individually, both software compilation and hardware synthesis represent amazing successes in the computer science research literature. When taken together – in single-source systems which can be refined to both hardware or software systems – the record is positive, but certainly more mixed. The challenges of mixed-target compilation[1] is rooted in the fundamentally different models of computation exhibited by different targets. Attempts to take an implementation designed with one model in mind and map it into another model tend to introduce inefficiencies and unnecessary implementation rigidity. To be successful, it is crucial to design from the beginning with the intention of targeting differing fabrics.

1.1 Hardware and Software Computational Models

The root challenge in designing and integrating heterogeneous – mixed hardware and software – systems is the the fundamental difference in the computational models used to construct those systems. Distilled to its core, this difference is that software systems are implemented as a series of sequential steps over a relatively small fixed set of high-level computational blocks, while hardware systems are implemented by arranging an adaptable set of concurrently-operating, low-level components.

A CPU generally contains a single ALU and control unit.[2] The ALU provides a variety of high-level arithmetic and logic data processing functions, such as adders, multipliers. There may be dedicated units for different data representations, so that integer and floating-point arithmetic operations are handled by different components. Nevertheless, these components operate on fixed-size data representations, usually the native word size of the processor for integral data and the IEEE standard data size for floating point operations. The control unit provides the mechanism for coordinating

[1]We use the terms compilation and synthesis interchangeably throughout the thesis. In situations where we wish to distinguish between the two, we will make the distinction explicit.

[2]This characterization is less apt for modern super-scalar architectures which may have multiple functional units. These architectures came about as an attempt to avoid some of the limitations inherent in the von Neumann computational model. However, the discussion below remains valid.

the operation of data processing units as a series of state transformations on data stored locally in registers (for fast access) and in global memory (for longer-term storage).

The intent of the von Neumann architecture exhibited by general purpose CPUs is to provide a common abstraction for implementing systems. In many ways, this intent has been realized to great success, as evidenced by the relative ease in which software can be ported between different instruction set architectures which subscribe to this computational model. This is especially true when the software is written in a imperative language such as C that abstracts many details of the processor and obviates the need for the programmer to operate at the level of the ISA.

The architecture of a standard CPU simplifies the construction of software systems and eases the migration of systems between different CPU families, yet this architecture has significant drawbacks in embedded applications. The ISA presents a one-size-fits-all model for constructing systems. Consider an application consisting of a large number of independent addition operations. Because of the limited number of adders in the CPU, these operations must be serialized. Whether the application ever uses (for example) an integer multiplication operation or any floating-point arithmetic is irrelevant; the transistors needed to implement those unused operations in the CPU are dedicated to that task and cannot be re-purposed. If the application manipulates data of a non-standard size, the data must be adapted to the CPU, resulting in inefficiencies both in the computation needed to perform the data transformation as well in the silicon overhead for the component being larger than necessary.

The above inefficiencies are present when performing the same operations on collections of data (SIMD) the limitations of a general-purpose CPU become even more pronounced when performing *different* operations upon data (MIMD). The most common way to implement this sort of computation on a CPU uses threads. Each thread has its own control and data state, and different threads may be executing logically different programs. The CPU time-multiplexes the various threads by executing each thread for a small amount of time, then saving the thread's state at a safe point and restore a different thread's saved state.

Communication between the threads is performed via manipulations on a shared global store. Manipulating the global state is tricky, as it requires every externally-visible manipulation to be atomic. Common thread libraries, such as POSIX threads (IEE 2004) offer a variety of concurrency-control primitives for insuring that inter-thread communication is safe. However, proper use of those primitives is notoriously error-prone.

In many ways, the computational model used for constructing hardware implementations of components is the inverse of the software model. Rather than a fixed set of high-level functional components, a hardware fabric provides many orders of magnitude (perhaps millions) more low-level *gates*, each of which are capable of performing a single 2-input boolean operation. Modern technology has eclipsed this characterization to a degree. For example, gates in modern FPGA technology are implemented using look-up tables (LUTs) which generally implement 4- or 6-input boolean circuits. Nevertheless, these advances do not nullify the following analysis of the hardware model of computation.

In the software model, control is expressed either implicitly in the sequencing of instructions, or explicitly as branching and jump instructions that direct the operation of the CPU control unit. In the hardware computational model, the distinction between control is less well separated. Registers are sequential elements of state that store values at regular intervals dictated by a global clock. As with the components of a CPU, the circuitry used to implement operations is present in the fabric whether the instantaneous control requires the output of those operations or not. Control is manipulated by selecting, using a multiplexer circuit, the outputs of various combinational circuits for input to a sequential register. Modern synthesis tools can minimize the number of idle combinational circuits by automatically introducing additional control logic, allowing the same circuit to be reused for mutually exclusive control paths.

This gate-level computational model precludes several of the drawbacks of the software model. For example, there is no need to artificially sequentialize data-independent operations due to a limited fixed number of computational resources – if an application demands *n* adders, then the circuit can include exactly *n* adders.[3] Equally important, the fine control provided by the low-level of abstraction of the hardware model allows unneeded resources to be omitted in the implementation. The circuit implementing an application requiring no floating point operations will include no floating point components.

The gate-level computational model provides maximum design control for implementing components, yet the level of abstraction is so low as that the design task becomes intractable as the application scales in size. For this reason, hardware designers will often use a *register transfer level* (RTL) computational model for constructing components. In RTL, a component is defined as a series of state elements (mapped to registers in the fabric) and a series of combinational transfer equations. A register's value in the next state is defined as the result of a transfer equation applied to the value of the register(s) in the the current state.

The distinction between current and next state is dictated by the register, which is in turn driven by a system clock. The clocked nature of RTL design introduces a complicating factor into the construction of hardware components using this model. For the value at the input for a register to be valid at a clock edge, the transfer equation implemented as a combinational circuit from the register's output, must meet timing

[3]This is an idealized view of hardware fabrics. In practice, the actual number of gates is finite, and it may be necessary to limit the number of duplicated resources to fit space constraints by introducing additional control logic.

constraints, in that the propagation delay through the combinational circuit must be less than the clock period. In other words, the propagation delay for the transfer equations dictates the clock period.

The clock period is determined by the *longest* combinational path for a register whose input is dictated by a collection of transfer equations. It makes little sense to optimize the timing for one combinational circuit if there is another circuit with worse timing; consequently, the designer must take into account timing as a *global* performance property. There are local approaches to reducing circuit propagation delay (e.g. splitting a combinational circuit by introducing pipeline registers) but these approaches have global ramifications (e.g. the need to insert additional delay registers in sibling circuits).

The concurrency abstractions of a hardware computational model is nearly the inverse of that in software. Whereas in software, concurrency is achieved by multiplexing several threads of control on a single CPU, in hardware concurrency permeates the model. In software, the main programmer effort is in simulating concurrency on an essentially non-concurrent fabric in hardware concurrency is primitive (all elements in the fabric are always executing concurrently) and the challenge is imposing sequentialization of control.

The standard way to impose sequential control within a hardware circuit is using clocked registers. However, introducing a register to buffer data transfer between concurrently operating circuits has the unfortunate consequence that the timing behavior for each component is externalized, and every component must satisfy the same timing requirements. This is analogous to the “leaking” of timing behavior of individual combinational circuits within a component into the other combinational circuits where the performance of every circuit is bounded by the timing of the *worst* performing circuit.

The culprit for the globalization of timing requirements is the clocked nature of the registers used for inter-component communication. One solution to this problem is to introduce multiple clock *domains*, so that each component is operating at its own clock frequency. While this solution simplifies the implementation of circuits *within* a component, it complicates the interactions *between* components. Instead of a simple global clock, it is necessary to include a more sophisticated control protocol at the junction between clock domains.

1.2 Design Process Ramifications

The differences between hardware and software implementations of components is rooted in the disparity between the abstractions used to implement those components. Software presents an exceptionally regular architecture, with the downside that there are inefficiencies introduced due to the fixed set of computational resources and the impedance mismatch that results from exploiting concurrency on a fundamentally sequential abstraction. On the other hand, while hardware offers much more design flexibility by giving the implementer control over the number and size of computational resources and concurrency "for free", the model of computation is at such a low level that it requires the implementer to repeatedly construct many abstractions offered by the software model of computation. At the same time, the hardware level of abstraction emphasizes non-functional performance requirements, such as timing, which are hidden in the software computational model.

Given these differences it is instructive to examine the consequences of the properties of the different models of computation within the larger scope of the system engineering process. This includes the secondary influences of the abstractions upon the various phases of the engineering life cycle, including design, deployment, and maintenance. Moreover, while there may be engineering practices that are naturally induced by the model of computation of the underlying target fabric, these practices are often at odds when employed when constructing a heterogeneous system composed of components in mixed target fabrics.

Design impacts The system design process begins with the collection of functional and performance requirements. The functional requirements define the correct computational behavior of the system, while the performance requirements define constraints – cost, size, security, etc – that must be satisfied while performing the computation. The functional requirements are fabric-agnostic, while the performance requirements are abstractions of the fabrics the function will be implemented in.

The next task in the system design process is a decomposition of the functional requirements into a architecture that that places various portions of the functionality of the system into different components, maps each of those components into a target fabric, and defines the communication interfaces between the components. The decomposition task is advised by the performance requirements, as the non-functional behavior exhibited by a component mapped to a particular technology must satisfy the constraints not only of that component, but of the system as a whole.

A system architect will use gross estimates and heuristics when performing the decomposition. This is because the actual performance of the system may vary widely depending on the exact implementation of a component, and the number of implementations satisfying a particular set of functional requirements may also vary widely. Consequently, it is the task of the system architect to constantly re-evaluate the suitability of the system decomposition towards the performance requirements as the functional requirements are refined to implementations. If during this refinement it is discovered that the system decomposition will result in a system that violates performance constraints, it is necessary to re-architect the system to a different configuration which can satisfy the performance constraints. This process may repeat several times.

Ideally, changing a system decomposition by moving a component from one target fabric to another should be possible without abandoning the implementation effort expended to arrive at the decision to reconfigure the system architecture. Sadly, this is rarely the case, due to the contrasting models of computation used in the refinement of the components. Once a target fabric has be identified for a component, it is likely that the abstraction used in the implementation of that component will make it difficult to transfer that implementation to a different fabric.

This difficulty is especially pronounced in the case of component mapped to hardware components. Because of the tight coupling of functional behavior and non-functional behavior (such as timing) it is often difficult to migrate a component from one target fabric to another *even if the computational model is the same*. Consider, for example, the gate-level model of a circuit mapped to FPGA. As FPGA manufacturing technology advances, the basic building blocks of the FPGA fabric continues to support increasingly-complex boolean circuits. A gate-level design mapped to an older technology may not be easily adapted to utilize new FPGA features, simply because the design is too closely coupled to the implementation technology. This yields serious difficulties in the maintenance life-cycle of the system as existing designs are adapted to new technology.

The initial system decomposition is performed with only gross estimates of the performance properties of the functional requirements mapped to an implementation fabrics. These estimates are honed as the functional requirements are refined to implementations. Because engineering effort (or conversely, the cost to correct errors) increases dramatically – often by orders of magnitude – as the system moves from the design to implementation and then to the validation stages, it is important to detect and correct errors as early as possible in the process. Not only is desirable to be able to re-target implementation artifacts, it is also of major advantage to move validation as early as possible in the engineering process. However, in heterogeneous system design, this is complicated by the often informal nature of the computational models utilized, the informal understanding of the interactions between the different computational models, and the interactions between non-functional performance properties of the computational models.

Deployment and maintenance considerations Software components can be updated “in the field”. Replacing a software component with an augmented version may be as simple as restarting a program or rebooting the system. Components implemented in hardware are rarely so easy to update, as the substrate *is* the component. To update such a component may require removing and replacing the physical circuit, an undertaking that will often require removing the component from the deployment environment. Consequently, bug fixes and feature enhancements for software components entail considerable less expense than the same modifications for hardware components.

The ease with which software can be modified at run-time forces the system designer consider the implications of this malleability and adjust the system design accordingly to adapt. For example, because software can be modified at run time, it may be necessary to make additional security assurances. Not only is it sufficient to guarantee that the system that is deployed satisfies security requirements, but it is also necessary to ensure that the system that is executing continues to be the system that was deployed, and not some altered version that may violate those security requirements. This typically requires some degree of run-time monitoring or measurement, adding additional complexity and execution overhead to the system. Moreover, it remains necessary to ensure that the component providing run-time integrity assurance is itself not compromised. On the other hand, the rigidity of hardware components make them less susceptible to attacks where the deployed component is replaced with a compromised component.

Long after the design and deployment of a system, the relative adaptability of the software vs. hardware models of computation continues to influence the maintenance of the system, even after the components have stabilized and are no longer actively being updated with bug fixes. As the system ages, it becomes necessary to replace components simply because of the physical wear-and-tear on the physical hardware (regardless of whether the hardware is a CPU or custom circuitry).

Historically, technology has increased in speed and storage at an exponential rate. As embedded systems are often deployed in applications where reliability and stability are of high importance, it may be years or decades between the initial deployment of system and the need to replace a physical element of the system. Due to the long delay between the design and servicing, the technology used for the components in the initial design may not be available for purchase, or may be so expensive that it is more economical to redesign the system for new technology. For example, a system which when designed required a number of CPUs (for executing software components) and ASIC (hardware components) may be possible to implement using a single modern CPU with a single hardware interface to legacy elements of the system.

The challenges when attempting to use newer technology to implement legacy components are similar to the challenges used in the initial design process, except that working implementation are already available. Because the existing implementation has been tested and deployed, it is of great benefit to reuse those components. For software components, this involves adapting the existing programs to the ISA of the new technology and executing multiple components on the same CPU. This is more complicated when adapting hardware components. To migrate a hardware implementation to software requires a change of model of computation. Moreover, the hardware implementation may rely on a variety of non-functional performance properties of the original target fabric to insure correct functional behavior. It is likely that these performance assumptions in the original implementation may not be documented, and even less likely that the new technology will exhibit the same properties. Therefore, it is necessary to perform a great deal of post-implementation validation to insure that the new system design respects the performance constraints of the original design and implementation.

1.3 Improving the Design Process

Building hardware and software systems requires modeling, implementing, and integrating components in disparate models of computation. The gap between the hardware and software models of computation makes it difficult to move components between models of computation. Software implementations introduce run-time inefficiencies and constrain concurrency, yet increase the design and maintenance flexibility. Hardware implementations allow a implementer maximum control of computational resources and concurrency, at the cost of increased prominence of non-functional performance properties in the component design and reduced maintenance flexibility.

Improving the process of developing heterogeneous systems requires the identification of a mediating model of computation which satisfies the following desiderata:

- The ability to quickly re-target implementations to fabrics with different models of computation.
- The ability to take advantage of the computational resources of the fabric.

- The ability to define concurrent communication independently of the medium and irrespective of low-level performance considerations.
- The ability to predict performance properties of a computational artifact.

The thesis of this work is that a functional language, extended with monadic effects for imperative and message-passing based concurrency, is a suitable basis for constructing systems that will be synthesized to efficient hardware or software fabrics.

This thesis focuses on the synthesis of mixed-fabric systems in the context of embedded applications. While this is the motivating use case, it should not be seen to be a limited case. Many of the complications that embedded applications emphasize remain relevant in typical desktop applications. This is increasingly so as modern CPUs address the abundance of transistors not by making high-clocked processors, but by adding more CPUs on a single die. This is a response to the decreasing returns, at the CPU design level, yielded by addressing performance with faster and faster processors. This change is fundamental, as it shifts the computational model from the sequential von Neumann architecture with a central store for data and control (as a program) to a concurrent model where data and control are distributed throughout the system. From a historical perspective, this shift was first foretold by Backus over 30 years ago in his lecture about the 'von Neumann' bottleneck. Backus' response to the bottleneck was to use functional programming languages to escape the inherent difficulties in exploiting the new computational model.

The approach outlined in this thesis is tightly related to that advocated by Backus (1978), where a functional programming language was identified as an escape from the "von Neumann bottleneck". However, it goes beyond by highlighting the role of concurrency in systems. Backus' FP was primarily concerned with the parallelization of common sub-elements in a data processing language – SIMD style processing.

In contrast, constructing MIMD – and concurrent applications in general – the primary challenge is in the coordination of the various processing elements. To address this problem, the thesis once again looks back over 30 years to the foundations of concurrent models of computation, Hoare's CSP (Hoare 1978) and Milner's CCS and π-calculus (Milner 1999). These models of computation for concurrency use message-passing as the primitive notion of concurrent computation. Processes are isolated computational units that operate independently until a interaction with the external world is needed. This interaction is accomplished by either sending or receiving a message along a channel. Because all actors in a concurrent system are modeled as processes, every send must have a matching receive from another process, and vice versa. If a process attempts to send (resp. receive) a message and there is not a matching receive (resp. send), the process will "block" until there is a match. Within this model, communication *is* coordination, with control being converted to data.

Backus' FP forms the "functional" core of the model advanced by the thesis, and Hoare and Milner's models of message-passing concurrency form the foundation of the concurrency model. The final piece of the thesis involves the use of *monads* to encapsulate computational effects, such as imperative (stateful) computation and reactive (message-passing) concurrency. Moggi identified monads (Moggi 1990, 1991) as a useful mathematical principle for structuring computational effects that allowed for higher levels of abstraction than operational models, while offering more composability than traditional denotational models (Stoy 1981). From a mathematical viewpoint, monads are algebraic structures and can be manipulated as such, much as Backus advocated an "algebra of programming" for reasoning about and manipulating functional programs.

In addition to offering formal reasoning capabilities, monadic specifications can be interpreted in a pure functional language, giving rise to executable specifications. Wadler(Wadler 1992, 1993) identified the suitability of monads for structuring programs – not just specifications – an observation that has been leveraged to great success in structuring computational effects in pure functional languages such as Haskell.

1.4 Motivation: Software defined radio

The motivating application for this work is the software defined radio (SDR) domain. Software defined radios are an approach to constructing radio systems so that they can be reconfigured and adapted, perhaps on-the-fly, for new applications. Despite the name, a software defined radio isn't exclusively constructed of software components: often a SDR is built upon an existing platform that includes dedicated hardware for specific radio tasks, because the performance of a pure-software system cannot meet the processing demands of a radio application.

The design of a radio may vary depending on the characteristics of the deployment environment. For example an individual operating in the field, communicating with a handheld radio to a fixed central tower, will have vastly different performance constraints for power and size than the tower it is communicating with. The mobile device may need to sacrifice cost for power and speed, placing much of the functionality of the radio in custom hardware, while the central tower, with a dedicated power source, can use a software implementation of much of the radio. However, the two radios are using the same functionality to communicate, and it is desirable to be able to develop a single implementation that can be used both in the software and hardware components.

A second intended benefit of software defined radios is that a radio platform can be reconfigured for a variety of different waveforms as the environment demands. In this usage model, the software defined radio platform provides a computational basis for implementing a given waveform, and may include a variety of different computational elements, including general purpose CPUs, DSPs, and FPGAs (Minden et al. 2007). A component may be used in multiple different waveforms. However, the computational demands of the application may vary, and in one configuration a component may need to be implemented in hardware to satisfy performance requirements, while in a second waveform configuration the same component may be implemented in software, freeing hardware resources for a different component.

The SDR domain requires digital components to interface, at some point, with analog RF components. Analog hardware will have fixed performance requirements, such as power consumption and timing. The computational elements must respect those requirements. Moreover, an SDR is a discrete approximation of an analog waveform; the waveform may include continuous elements that are digitized and processed by the SDR components. It is critical that those elements respect the analog model of the waveform as well.

The final property of the SDR domain that exemplifies the challenges of developing heterogeneous systems is that the lifespan of a deployed SDR platform may be measured in decades, making it necessary to be able to adapt portions of the platform to new computational technology. However, this cost of deploying the platform based on new technology can be high, since the correct implementation of a waveform component can rely on low-level performance constraints, which are often not reflected in the implementation. The cost of not deploying new technology, however, may be greater as legacy parts may cost too much or may simply not be available. To successfully manage the lifespan of an SDR platform, it is necessary to model not only the component behavior but also the performance characteristics of the platform elements, easing the transition to new technology.

1.5 Overview of related work

There is a large body of existing literature describing the use of functional programming languages and notations for describing hardware circuitry. Lava (Bjesse et al. 1998) and Hawk (Matthews et al. 1998) are two such examples. While these languages use a functional programming language to construct circuits, they maintain a distinction between the host language (Haskell, in the case of Lava and Hawk) and the object language, implemented as a library of data structures and combinators.

Embedding the hardware description language within a lazy functional language such as Haskell allows the developer of the language to leverage a great deal of the host language capabilities. Lava circuits are Haskell data that can be manipulated with Haskell functions. Moreover, Lava circuits can be given a variety of interpretations. This includes Haskell stream transformation functions, enabling a basic simulation capability. Other representations include a VHDL back-end, which allows Lava circuits to generate structural VHDL netlists, and as boolean forms to be used as input to SAT solvers for verification purposes.

The distinction between Haskell programs generating Lava circuits is similar to the static/dynamic stage distinction found in the partial evaluation (Jones 1996) and multi-stage programming languagesTaha (1999). However, Lava differs significantly with these systems in that the host language and the object language are different, while most work on multi-stage languages uses the same language for each stage.

A disadvantage of changing from Haskell to Lava between stages is that the Haskell abstractions cannot necessarily be mapped directly onto Lava abstractions. While Haskell is a full-fledged programming language, a Lava circuit is basically a structural representation of a circuit as a graph. Haskell idioms must be replicated as Lava library components, such as a multiplexer instead of a case expression for control flow.

Moreover, Haskell data abstractions are replaced with lower-level data structures, as Lava operates primarily on streams of boolean values representing bits. Composite data structures must be manually constructed as collections of individual bitstreams. Finally, some host-language idioms, such as function abstractions, are simply not present in Lava. As we demonstrate in chapter 3, functions in Oread provide a simply structuring mechanism for controlling resource usage.

VHDL (IEEE) and Verilog (IEE 1995) are the two dominant hardware description languages. Both languages offer similar capabilities for defining hardware, although VHDL traditionally has offered more extensive data abstraction capabilities, as it was originally designed as a specification language, rather than an implementation language. The languages use a discrete-event model of computation, where hardware signals are modeled as a ordered stream of events. Each event contains a specific timestamp, enabling the ordering. The close connection between the discrete event model of computation and the clocked nature of most hardware implementation fabrics has led to an emphasis on strict control of timing within a design. This thesis, in contrast argues that it is necessary to *de-emphasize* the role of timing in the construction of system, relying instead on an asynchronous control protocol for handling circuit coordination. Decoupling a component's functional behavior from its associated non-function properties (such as timing) allows a functional implementation to be more easily adapted to new implementation fabrics as demanded by design requirements.

VHDL and Verilog both support two major forms of system definition: structural and behavioral. A structural description of a system describes the constituent components and the connections between those components. Structural description is roughly analogous to a textual form of the traditional graphical schematic design. Components, selected from a library of predefined parts, are instantiated the ports of components connected via external signals. Circuit descriptions are necessarily low-level, and there is minimal support for capturing circuit patterns, in the style of Lava, as meta-programs, although VHDL does provide a `generate` construct for performing some restricted structural meta-programming.

Because a structural system description is defined at a low level, with components drawn from a predetermined library, they are often easy to synthesize to target fabrics. On the other hand, both VHDL and Verilog offer behavioral modeling capabilities that allow a system to be described at a higher-level of abstraction, but with a trade-off that some systems that can be expressed in the behavioral subset of the language cannot be synthesized. Consequently, users of the behavioral subset of the languages, and the vendors of synthesis tools, often resort to idiomatic programming styles that result in synthesizable designs.

Currently the most dominant of these idioms is that of Register-Transfer Level (RTL) modeling, where circuits are expressed as state, implemented as registers, connected by combinational circuitry that defines a transfer equation between the current values of each register to the next value. Control in such systems are implemented as state machines, with the registers determining the state of the system and the transfer equations determining the next state. A significant disadvantage of RTL modeling is that it implicitly includes the notion of a global clock coordinating the register/transfer equations. The global clock becomes burdensome when connecting two independently-developed components, as it necessitates globally satisfying the timing constraints of all components, or else defining custom coordination circuitry between those components to localize timing considerations.

In contrast, the thesis of this dissertation claims that inter-component communication should be structured asynchronously, and the communication protocol divorced from the internal workings of the component. Using a message-passing model of concurrency, we argue that the need for a global coordination signal based on a single clock is obviated. More importantly, the interaction between components can be described as an asynchronous exchange of a sequence of messages according to a specific protocol.

The thesis expressed in this dissertation, and the implementation strategy employed in its execution, is largely based of that of the Statically-Allocated Functional Language (SAFL) (Sharp and Mycroft 2000; Mycroft and Sharp 2000, 2001b,a; Sharp 2004). In this work, a functional language is as the basis for resource-aware hardware system construction. An extension to SAFL, called SAFL+, includes imperative state and first-class channels, much in the same way that we use imperative effects and message-based concurrency.

The fundamental distinction between SAFL and the work described in this thesis is that we have chosen to use monads, and monadic computations, as the basis for our language, while the SAFL work uses an operational model for the semantics of the language. We believe that using monads as a semantic basis provides an advantage when validating program manipulations, as described in chapter 4, as they provide an algebraic basis (Harrison 2006a) for relating the transformation of the programs to the underlying semantics, eliminating the need to jump to an external semantic formalism.

1.6 Dissertation contributions and organization

The thesis of this work is that a functional language, extended with monadic effects for imperative and message-passing based concurrency, is a suitable basis for constructing systems that will be synthesized to efficient hardware or software fabrics.

To support this thesis, this dissertation makes the following contributions:

- Oread, a functional language extended with *monadic* imperative and message-based concurrency effects.
- A framework for defining concurrency and communication protocols from within Oread.
- A compilation scheme for a subset of Oread that maps to hardware using an explicit control protocol.
- A series of source-to-source transformations that convert non-synthesizable Oread programs to those that satisfy the synthesizable subset of the language.
- A mapping of Oread to C, an imperative language, and Pthreads, a standard library for preemptive concurrency.

The remainder of this dissertation is organized as follows:

Chapter 2 gives an overview of Oread, with an emphasis on its monadic features. In addition to a description of the core language, we detail a general architecture for the combination of monadic Oread components. Later chapters demonstrate the flexibility of this architecture in two separate realizations, one in hardware and one in software.

Chapter 3 describes the compilation of a subset of Oread to hardware via the hardware description language. The compilation scheme uses a small set of simple VHDL components, described behaviorally, and generates a structural VHDL model of a Oread program as an instantiation of those components. The resulting implementation can be both simulated and synthesized. Chapter 4 describes the use of source-to-source program transformations to map general Oread programs that do not adhere to the hardware synthesis restriction required in chapter 3 into programs which are capable of being synthesized.

Chapter 5 details the compilation from Oread to C. The compilation results utilize none of the more sophisticated features of C, relying on the target language more as a high-level assembly, but with run-time support for dynamic memory allocation. The component combination architecture, described in 2, is implemented using a small collection of POSIX threads primitives.

Chapter 6 presents a case study using Oread in two example problems. The first example details the design of a single component, described in Oread and mapped to both hardware and software implementations according to the compilation schemes in chapters 3 and 5. The second example describes the integration of multiple components, both described in Oread, but targeting different implementation fabrics. This example explores the suitability of Oread for system-level heterogeneous design.

The dissertation concludes in chapter 7, where we revisit the motivation for the thesis and the suitability of the solution presented in the intervening chapters to support the dissertation's thesis. Finally, we identify important directions of future work that extend the development in this dissertation.

Chapter 2

Oread: A language for mixed-target synthesis

To exercise the thesis of this dissertation, we have defined a language, Oread, a higher-order functional language reminiscent of ML or Haskell. Oread has a static type system, with polymorphic types and user-defined algebraic data types. This core language is extended with constructs for imperative state and reactive concurrency. Effects in Oread are modeled monadically, allowing a clear delineation between the pure aspects of the language and those aspects that use imperative or concurrency constructs. Oread has been designed as a minimal language, with just the core features needed to demonstrate the thesis.

This chapter introduces Oread. We discuss the motivations that drove the design of the language and analyze how the language features present in Oread relate to the language features highlighted in the thesis. To this end, we provide an overview of the major structuring feature of Oread: the use of monads to encapsulate effects. Monads permeate Oread and the accompanying compilation tool-set, serving both as the computational model for programs written in the language, as well as the structuring mechanism for constructing the tool-set for processing Oread programs, including type-checking, interpretation, and compilation to both an imperative language to target software and VHDL to target hardware fabrics.

In many respects, Oread can be treated as a core intermediate language that is used in the compilation of a more full featured – or programmer-accessible – language. Nevertheless, to simplify the description of the language, we define a concrete syntax that allows programmers to write Oread programs directly. This concrete syntax is necessarily spare, and does not have much of the syntactic sugar that programmers would expect in a typical programming language. The constructs of the concrete syntax are introduced here as they are used, with a complete specification of the grammar in appendix 7.1.

A major contribution of this dissertation is the use of a reactive monadic computation to structure concurrency within a system. Because Oread is designed as a source language for both software and hardware targets, we have defined this concurrency abstraction so that it is not unnecessarily biased towards either target. The following two chapters demonstrate the suitability of this abstraction, as we demonstrate that the same concurrency model can be easily mapped to existing abstractions in both software, using operating system threads, and hardware, using a simple communication protocol.

However, we note that in designing system with concurrent components, it is not feasible to assume that all components will communicate using the same protocol. To this end, we have defined an architecture for defining interaction protocols between components within Oread itself. This architecture allows components to be defined using successively higher-levels of abstraction for constructing the communication and coordination protocols between components. At the simplest level, Oread provides a universal primitive for performing concurrent operations. At a slightly higher level, a programmer can define specific protocols by having components generate encoded messages and by assigning an interpretation to those messages. Finally, at an even higher level, a programmer can define functionality which to adapt protocols, allowing a component which behaves according to one protocol to be adapted for use in a context requiring a different protocol. The two higher levels of abstraction are facilitated by the ability to define the concurrency protocols within Oread itself, using the concurrency architecture we have developed.

The remainder of this chapter is structured as follows. First, we discuss the motivation for the design of Oread and the relationship between the Oread features and those outlined in the thesis. This includes a description of the Oread concurrency architecture, which details how interaction protocols are defined in the system and the mechanism for assembling Oread components into a complete system. Next, we provide define the static and dynamic semantics of Oread. The language semantics are presented in monadic style. In the interest of making this dissertation self-contained, we provide a brief overview of monads.

2.1 Oread Design Motivation

2.1.1 A Functional Language

The functional subset of Oread is a strict higher-order functional language extended with algebraic data types, similar to the second-order lambda calculus (Reynolds 1974). Oread resembles modern functional programming languages such as Standard ML (Milner et al. 1997) and Haskell(Peyton Jones 2003). However, unlike Standard ML, Oread is pure, without effectful operations such references and call/cc. Similarly, Oread differs from Haskell in that it has strict evaluation semantics, as opposed to Haskell's normal-order evaluation semantics.

Oread departs from Standard ML by disallowing effectful computation within the functional subset. On the other hand, Oread *does* allow effectful computation, but the extent of the effects are encapsulated as monadic computations. The reliance on monads resembles Haskell, which has used monads to great success to model effects in a pure language (Wadler 1993; Peyton Jones 2002; Swierstra and Altenkirch 2007).

Haskell has a normal-order evaluation semantics, which arguably leads to a more declarative style of programming (Hughes 1989). The decision to sacrifice more declarative normal-order semantics for strict evaluation semantics is pragmatic. Normal-order evaluation requires the dynamic allocation of space for function arguments that may (or may not) eventually be evaluated. While this strategy works well on a CPU with a significant amount of memory, it is impractical in hardware targets that require precise control over the allocation of resources, including memory allocation, and may (as does the Oread VHDL compilation) disallow dynamic allocation. Strictness analysis (Mycroft 1980) can identify safe transformations of normal-order programs into strict programs, yet it remains possible that some programs may not be transformable, which would retain the need to dynamically allocate space.

This core functional language is *pure*, in that it does not allow side effects within the functional subset of the language. The purity is important because it allows the safe application of fold/unfold transformations. An unfold transformation replaces call to a function with its definition, and conversely a fold transformation abstracts an expression into a new function and then replaces the original occurrence of the expression with a call to the new function.

Unfold transformations allow Oread to take advantage of the inherent parallelism of functional languages. Rather than perform data-independent (SIMD) operations in sequence, as may be the case on a CPU, the common portion of the computation shared among the independent portions of the overall computation can be folded into a top-level function. Applying that operation across a list of elements can be defined using a higher-order function that captures the recursion behavior. Finally, the higher-order function can be unfolded, and then the individual calls to the top-level function.

Example: Consider a program that maps defines an `inc` function across a list of elements. The `List` data declaration defines a polymorphic List data type, with two constructors, `Cons` and `Null`. The **`map`** function takes a function, `f`, and a list, `l`, and generates a new list formed by applying `f` to each element of the input list.

```
(data List [a] (Cons a (List a)) (Null))

(define (map (f (-> a b)) (l (List a)) (List b))
  (case l
    ((Null) (Null))
    ((Cons x xs) (Cons (f x) (map f xs)))))

(define (inc (x Int) Int)
        (+ x 1))
```

If both the **`map`** and the `inc` functions are unfolded onto a list with elements `a,b,c`:

```
(map inc (Cons a (Cons b (Cons c Null))))
```

The unfolded operation inlines all of the calls to **`map`** and to `inc`:

```
(Cons (+ a 1) (Cons (+ b 1) (Cons (+ c 1) Null)))
```

This unfolding – when performed on a program targeting hardware – will result in a duplication of circuitry. However, each copy of the circuit can be active concurrently, yielding a net performance gain at the expense of circuit area. If the functions being manipulated using fold/unfold transformations were not pure, then the transforms would not be safe, because they may result in a duplication of, or reduction in, the computational effects of the original program.

Being able to perform program manipulations is critical to target Oread specification to both hardware and software implementations from a single source. In chapter 1 we argued that a major difference between hardware and software design is in the ability for the designer to precisely control the number of computational resources that are available. In software, the compiler has no choice but to use the resources offered by the CPU, thus it will time-multiplexing data parallelism across those limited resources, generally using a loop. On the other hand, hardware allows the designer to dictate exactly how many of a given computational resource you wish to use, and incorporate that into your circuit.

It is possible to take a pure Oread program and inline it into its primitive components, resulting in a large combinational circuit in hardware or a large program text in software. This is unwise from a design perspective because there is no accounting for space or time limitations in the target model. Consequently, Oread includes two constructs for controlling the time and space behavior exhibited by a pure program: shared function blocks and a parallel let expression.

An Oread program consists of a set of top-level function definitions, along with a "main" expression that can call the defined functions. A top-level function delimits a shared resource. In hardware, the body of the function will be implemented as a single circuit, regardless of the number of calls to the function. If there are, in fact, multiple calls to the function, the function block contains arbitration logic that will process those calls in some undetermined sequential order. This sharing of circuitry is a space vs. time trade-off. The extremes between complete unfolding of all calls to a function (resulting in increased area usage) and multiplexing all calls onto a single shared block (resulting in increased contention) can be explored using fold/unfold program transformations. A designer can take a single shared function block and duplicate it, and distribute the calls to the original function amongst the various duplicated blocks.

Example: Returning to the unfolding example from above, rather than unfold the `inc` function, we can duplicate it to define *two* increment functions, and then map different calls to the various top-level definitions. The two calls to `inc1` will occur sequentially with each other, but in parallel with the call to `inc2`.

```
(define (inc1 (x Int) Int) (+ x 1))
(define (inc2 (x Int) Int) (+ x 1))

(Cons (inc1 a) (Cons (inc2 b) (Cons (inc1 c) Null)))
```

Second, Oread includes a let construct that introduces sequentiality and sharing into a program. Suppose that a particular Oread expression will, when compiled to a hardware target, result in a very long critical path. Inserting a pipeline register into the circuit, and thus reducing that critical path, is as simple as selecting a sub-expression and adding a named binding via a let expression. Likewise, a common sub-expression can be factored into a single let binding. Rather than a circuit for each instance of the sub-expression, the synthesis scheme will generate a single circuit that can be shared among all references to that sub-expression.

The functional subset of Oread is extended with a small collection of primitive types (integers and floating-point numbers) and operations on those primitives. This collection was selected largely because they are commonly used in implementing standard embedded systems. The range of data types representable in Oread is not limited to this set because the algebraic data type facilities of Oread allow a programmer to define new types of data as well as operations upon that data.

The set of primitive types and operations in Oread can be easily extended, as long as those operations remain pure. The design of the tool-set developed for Oread provides additional support for extending (or contracting) the language with new primitive types and operations by be constructed using modular monadic semantic techniques (Espinosa 1995; Liang et al. 1995; Liang and Hudak 1996). The choice of primitive types is skewed neither towards hardware or software compilation targets. For example, the language includes support for floating-point arithmetic, an operation that is typically quite space-expensive to implement in hardware. On the other hand, tuple selection and construction operations are trivial to implement in hardware yet are considerably more complex on a CPU with a fixed word size.

2.1.2 Monadic encapsulation of effects

The ability to perform program transformations gives justification to the pure functional subset of Oread. However, the pure functional subset of the language is simply too restrictive to be practical for building embedded applications. This is because the pure functional model is heavily skewed towards data-flow computation, which is a poor abstraction when constructing control-intensive or reactive systems, both of which are common traits of embedded systems.

We take the viewpoint that control implies a notion of state. This, taken with the interaction with external entities implied by reactive components, forces Oread to include a way to perform both stateful and reactive effects. However, we wish to add constructs for building these sorts of computations without breaking the purity that we rely on to justify transformations that allow us to target both hardware and software. Therefore, we use monads to structure effectful computations.

By structuring computational effects using monads, we get a *static* delineation between pure operations and effectful computations that is expressed at the type level. An effectful computation can include pure computations, but the converse is not true. This means that the extent that transformations are valid are clearly delimited by the monadic encapsulation of effects. The synthesis tool-set is free to transform programs within a monadic computation, but the transformations may never escape those monad boundaries.

Data transformation components, such as those typically found in the example software defined radio domain, have a standard structure. A component is a loop that receives data from some external entity via the reactive concurrency construct, performs some processing on that data, sends the data onto another component via the reactive concurrency construct, and repeats. The component may perform several loop iterations between receives/sends. Also, the component may carry some state across iterations. The primary purpose of this state is to simplify the specification of control logic.

The Oread monadic constructs are segmented into two categories. The first category contains two constructs for performing imperative effects, and the second category has one construct for reactive concurrency. The imperative, or state, constructs include a **get** function for accessing state, and a **put** function for mutating the state. Both of these functions use addressable state.

Example The `stateFun` function below reads a value from address `0` and uses that value as an address to which the input parameter `val` is written.

```
(define (stateFun (val Int)
         (monad [(@st (State Int Int))] Int))
  (do (addr <- (get @st 0))
      (put @st addr val)))
```

2.1.3 A monadic concurrency architecture

Oread includes a **signal** construct for performing reactive concurrency. This construct takes a message and routes that message to an external entity. From the viewpoint of a component, the outside world is accessible only via the signal construct, and can only be affected by signaling a request. Once a computation signals a request, the computation will block until a response is returned from an external entity. Signaling a request and then interpreting the response is the only way to observe the outside world from a reactive computation, and so comprises the complete Oread concurrency semantics within the language. These semantics leave much undefined: the content of the request/response messages, how are the messages are interpreted, and how are they routed between components.

Oread does not include special semantics for different types of reactive communication simply because the possible range of different inter-component communication is boundless. As examples, the inspiration for this work used the reactive monad construct to model an operating systems kernel (Harrison 2006b), while in contrast we have defined a series of point-to-point communication behaviors that include a mailbox, bounded FIFO, and unbounded FIFO. The basic concurrency constructs allow us to model each of these, but the specific behavior for each communication is defined within Oread as a collection of functions, rather than as special language constructs. These functions are arranged according to a regular pattern, thus we have developed a nomenclature to refer to the various elements.

First, a **thread** is a processing component. A thread may consist of a combination of pure functional, imperative, and reactive expressions. The single point of interaction with outside entities for a thread is via the reactive constructs. However, a thread may interact with several different external entities, with each interaction accomplished with a different **protocol**. A named interaction point for a **thread** is called a **port**. Finally, each ports is connected to a **service**, which is a collection of Oread functions that implements the protocol the connected threads use.

For example, consider a component that operates in a stream processing manner: consuming data from an input source, manipulating the data, than sending the data on to an output sink. Figure 2.1 has two examples of this architecture. The boxes labeled source, transform, and sink are Oread threads. Each thread has ports, indicated by the grayed semicircles where arrows enter and exit the thread. The rectangles labeled F in the top figure and Bus in the bottom are services. In the top figure, the transform component has two ports, each of which is connected to a separate service. In the bottom figure, the transform component has a single port, connected to the Bus service.

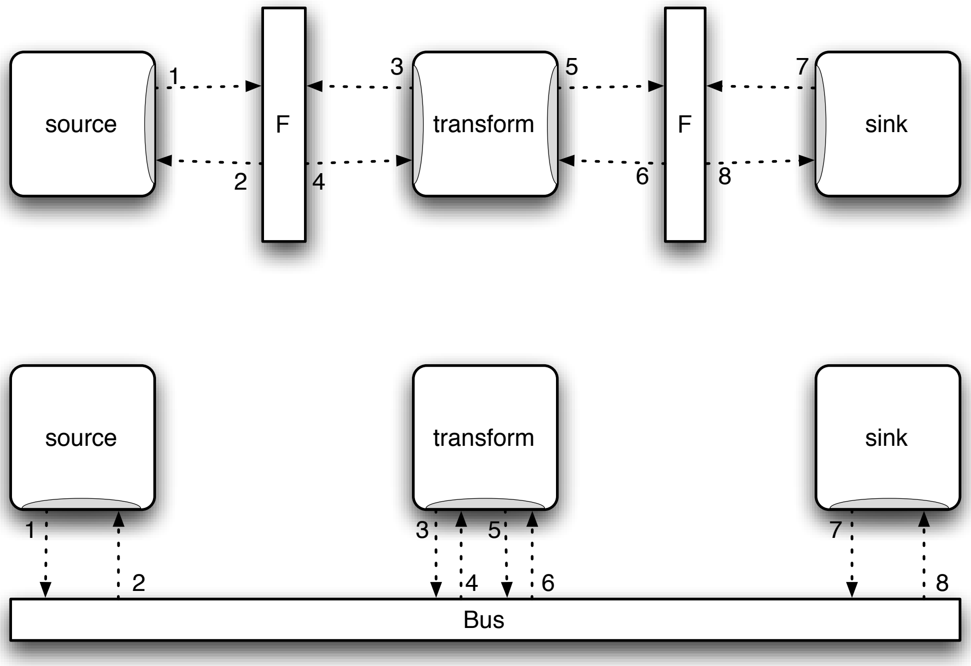

Figure 2.1: Example system architecture

The transmission of messages between threads and services is indicated in the figure by the dotted arrows. The arrows are numbered to indicate the ordering of messages. For example, in the top figure, the source will (1) issue a Send request to the F service that includes the data it wishes to send to the transform thread. The thread then blocks until the service (2) responds with an Ack response. The transform component (3) issues a Receive request to the service F, and then block until F (4) sends a response that includes the data originally transmitted by `source`. The series of messages 5,6,7,8 uses the same protocol to transfer data from `transform` to `sink`.

It is critical to note that the ordering of messages is only for expository purposes. Because the threads `source`, `transform`, and `sink` are operating concurrently, each may issue request at any time, subject the protocol restriction. Temporally, there will be an ordering between request/response pairs: 1 comes before 2, 3 before 4, and so on. In the Oread concurrency architecture, a request/response message pair is called a **service transaction**. The logic of a thread will also dictate the ordering of service transactions – for example, the transaction (3,4) necessarily comes before (5,6), since it is necessary for the `transform` service to receive data to process before processing it and sending it on to the `sink` thread.

The lower figure has all three threads connected via a single `Bus` service. Consequently, the `transform` thread has a single port for communicating with the Bus service, in contrast to the two ports in the point-to-point model with FIFO services. Moreover, the reorganization of the system architecture requires each thread to add additional information to the requests issued to the Bus service indicating the intended destination for the message. In contrast, the architecture in the top diagram allows communication addressing to be implied by the port.

The `F` and `Bus` services in the diagrams implement a given protocol. This is accomplished with a pair of Oread functions: a **handler** function that takes a request issued by a thread and generates the appropriate response, and a **scheduler** function that determines when a response generated by the handler can be returned to the appropriate blocked thread. These functions can include imperative effects, which allow the service to implement the control portion of its protocol. This can be illustrated using the top example architecture from above. The service that handles communication between the `source` and `transform` thread will initially be in a state waiting for a request from either thread. Upon receiving a request, the service invokes the handler function on the request. If the request is from `source` and is a `Send` request, the service will store the value included in the request and generate an `Ack` response. The scheduler function will then be invoked, which will return that `Ack` response to the source thread. Alternatively, if the request is a `Receive` request from the `transform` thread, the handler will note that a receive is pending, but cannot generate a response immediately, because there is no previous send request from the source thread. In this case, the scheduler function will indicate that there are no pending responses to transmit to threads, and the scheduler will wait for the next request, which can only come from the `source` thread, as the `transform` thread will remain blocked until the service returns a response to its receive request.

Hence, the service implements a very simple mailbox protocol: a receive request will cause the issuing thread to block until there has been a matching send request. Moreover, if a thread issues two consecutive send requests, and the service does not receive an intervening receive request, the sending thread will block on the second send until a receive is issued. This is a degenerate case of a bounded FIFO, which would allow an arbitrary (but fixed for a particular instance) number of sends without an intervening receive, simply by using a larger imperative state for buffering sent data. The bounded FIFO is, in turn, a specialization of an unbounded FIFO. However, the unbounded FIFO requires that the service be able to dynamically allocate storage, which is typically not possible in a direct hardware implementation. This suggests a Oread design process: first, a system is modeled as a collection of threads communicating with each other via a given protocol. The system developer generates a service definition which provides the loosest bounds on implementation, such as the unbounded FIFO implementation of point-to-point communication, and uses that model for simulation and early design testing. Then, as threads are mapped to various implementation targets, technology specific knowledge is used to drive the transformations of Oread thread definitions to get implementations which can match the capabilities of the target platform.

2.2 Oread semantics

Oread is a statically typed functional language, extended with monadic features for imperative and concurrent effects. We define the semantics of Oread below, giving both the static semantics and the dynamic semantics for the language. These semantics are written in monadic style, with the resulting semantics, when realized in a pure functional host language such as Haskell, being both definitional and executable specifications.

2.2.1 Monads

Monads are used extensively in Oread, both in the programming model as well as in the implementation of the tools used to process Oread programs. Viewed in another way, Oread is simply a thin veneer used to construct monadic computations. The tool-set we have defined then is just an interpretation of these monadic computations in either hardware or software.

A monad is a mathematical structure which encapsulates effects. When modeled in a functional language, a monad is a type constructor, `T`, paired with two *monad morphisms*, `unit :: a →T a` and `bind :: T a →(a →T b) →T b` [1]. A monadic computation `m :: T a` will yield a value of type `a`.

[1] In Haskell, the language that the Oread tool-set is implemented in, `unit` and `bind` are written as **`return`** and >>=, respectively

Informally, the type constructor `T` encapsulates the effects of a given monad. The `unit` morphism lifts a value into a monadic computation, without causing any effects. The `bind` morphism allows computations to be sequenced, with effects from the first computation to be propagated to the second computation. These informal properties of `unit` and `bind` morphisms formalized in three monad laws, with the `unit` serving as left- and right-identity of `bind`, and `bind` is associative.

In Oread, monadic functions are distinguished by their types. For example, the top-level binding below defines a monadic function, `f`, which takes an `Int` parameter and returns a computation which yields a value of type **`Int`**.

```
(define (f (x Int) (monad Int))
  ...
  )
```

The `unit` monad morphism is written as **`return`** in Oread.

```
(define (g (x Int) (monad Int))
  ...
  (return 1)
  )
```

The `bind` morphism is written using a special **`do`** syntax. The Oread statement below is interpreted, using the `bind` notation directly, as `(`**`bind`**` e1 (lambda (x) e2))`.

```
(do (v <- e1)
    e2)
```

Oread computations are sequenced by the **`do`** notation. The following function `g` calls the monadic computation `f`, waits for the result, binds it to the variable `y`, then again calls the function `f`, but with the result of the first call, `y`.

```
(define (g (x Int) (monad Int))
  (do (y <- (f x))
      (f y)
      (return 1)))
```

The function `g` returns a constant value `1`. However, the effects of the two calls to `f` are propagated through an invocation of `g`, making it unsafe to optimize calls to `g` away at compilation time.

Taken alone, the monad morphisms are largely uninteresting because they offer no mechanism for performing effects in the computation; all `bind` and `unit` can do is lift values into the computation or sequence existing effects. However, the monad signature can be extended with additional *non-proper morphisms*, which model a particular computational effect. In Oread there are special constructs for two such extensions. The first captures the state monad, which models imperative state, and the second captures the reactive monad, which models message-passing concurrency.

State Monad A computation that has imperative effects can be viewed a pure function that takes a state parameter `s` as an input and yields a pair `(a,s')` as an output. The side effects of the computation are captured in the difference between the input state `s` and the output state `s'`. Written in Haskell, this type can be made into a monad, `State s`[2].

```
type State s a = s → (a,s)
unit :: a → State s a
unit x = λs → (x,s)
bind :: (State s a) → (a → State s b) → State s b
bind m f = λs → let (v,s') = m s in f v s'
```

The `State s` monad can be extended with two non-proper morphisms, `get` for accessing the state, and `put` altering the state. The `get` function simply returns the

[2]The `State` monad is parametrized over the state type.

state as the result of the computation, and the `put` function yields a computation which ignores the input state and replaces it with the new state, passed as a parameter to `put`.

```
get :: State s s
get = λs  →   (s,s)
put :: s  →  State s ()
put s = λ_  →   ((),s)
```

In Oread, these functions are primitive constructs. There are two further differences from the `get` and `put` morphisms above. First, the Oread constructs are addressable state. The **`get`** and **`put`** functions take an address parameter and only return (resp. modify) the stored value at that address. Both the address and the value stored as data in each address can be specified. In the degenerate case, the singleton `Unit` type can be used for the address type, yielding equivalent behavior to the `get` and `put` morphisms defined above.

The second difference between Oread's treatment of the state monad and the simple formulation defined above is that a single monadic computation can refer to multiple independent state instances. This capability is useful for compilation targets with a non-uniform memory architecture, such as hardware fabrics, as it allows each state element to be mapped to a different element of the memory architecture. This model of multiple memories can be reduced to a single memory by constructing, for a set of memories, a single memory element with an address type that is the disjoint union of the address types of each component memory, and likewise a data type is the disjoint union of the data types of the constituent memories.

Example: The function `f` below defines a state-monadic computation which has two memories. It takes two `Int` parameters, `x` and `y` and adds the value `y` to an accumulator memory `@count`. It then stores the value `y` in address `x` of the state instance `@mem`. Finally, `f` returns the previous value stored in `@count`.

```
(define (f (x Int) (y Int) (monad [(@count (state Unit Int))
                                   (@mem (state Int Int))] Int))
  (do (c <- (get @count Unit))
      (put @count Unit (+ c y))
      (put @mem x y)
      (return c)))
```

The parameters `@count` and `@mem` are called **instances** in Oread. They must be passed to a monadic computation, such as `f`, when that computation is invoked. In the Oread concrete syntax, instances are passed in square brackets.

```
(define (g  (monad [(@count (state Unit Int))
                    (@mem (state Int Int))] Int))
  (f 0 0 [@count @mem]))
```

Calling a monadic function in Oread necessitates adding that monad instance parameter to the calling function. These feature parameters are percolated up the call graph until a top-level function is reached. At the top level, Oread monad instances are defined and passed to the top level function in a `configuration` declaration, as shown below. This configuration defines two memories `@c` and `@m`, as well as a top-level invocation of the function `g`.

```
(configuration example
  (memory @c Unit Int)
  (memory @m Int Int)
  (thread (g [@c @m])))
```

Reactive Monad The state monad models imperative effects, but within Oread all state is local to a component, defined as a top-level thread in a configuration declaration above. Inter-thread communication is modeled as a different computational effect, using the *reactive* monad. The model of concurrency employed is synchronous message-passing. When interacting with an external entity, a Oread thread will issue a *request* and then block awaiting a response.

This monad is considerably more complex than the state monad, as it includes a state monad within it. Figure 2.2 shows a definition of a `React r s` monad. The `React` type constructor has two data constructors: the first, `D`, signifies a terminated computation. The second, `P`, represents a suspended computation. It has two fields: the first contains a request issued by the suspended thread, and the second represents a continuation. That is, the remaining computation that an externally-generated entity will pass a response on to.

A computation that is suspended may, after receiving a response, issue another request. Consequently, the co-domain of the continuation argument to the `P` constructor yields another `React` computation. Moreover, a reactive computation may also include imperative effects. Accordingly, the co-domain of the continuation is a `State` computation that generates the eventual `React` computation.

```
data React r s a = D a
                 | P r (r  →  State s (React r s a))

unit :: React r s a
unit_r x = D x
bind :: React r s a  →  (a  →  React r s b)  →  React r s b
bind_r (D x) f = f x
bind_r (P req k) f = P req (λrsp  →  bind_s (k rsp) (λv  →
unit_s (bind_r v f)))
```

Figure 2.2: `React` monad

The definitions for the monad morphism `bind` for the `React` monad includes a use of the `State` monad's `bind` and **`return`** morphism. In the definition for the `React` monad, the different morphisms are distinguished lexically by appending an `_s` and `_r` suffix to the morphism name for the `State` and `React` monads, respectively.

Note that bind_r occurs co-recursively in the definition of bind_r. This allows the React monad to model co-recursive, stream-like of non-terminating computations, of the like that are commonly found in the embedded applications that motivate the design of Oread. The React monad has a single morphism, signal, which constructs a monadic computation that issues a request and returns the externally-generated response.

```
signal :: r → React r s r
signal req = P req (λrsp → unit_s (unit_r rsp))
```

The React monad also allows stateful effects, by "lifting" the get and put morphisms into the React monad. This is accomplished in the style of Harrison (2006b), using a higher-order step function. This function uses a default Step request, which we assume is a constructor of the r message type. Furthermore, step assumes a Ack response as part of the message type.

In practice, the compilation routines for the reactive monad in Oread do not utilize the nested monadic structure of the React type define above. We can assume that either there exists a constraint on the message type r that it contains Step and Ack constructors, or adjoin these two constructors post hoc. As with the definition of the monad morphisms, the get and put morphisms have a suffix to identify which monad's get or put is being referenced.

```
step m = P Step (λAck → bind_s m (λv → unit_s (unit v)))
get_r = step get_s
put_r s = step (put_s s)
```

Example: A Oread program can intermingle state monad **get** and **put** constructs as well as reactive monad **signal** invocations within a function. For example, the following function represents a data processing element in a system. It has two ports – named reactive monad features parameters – as well as a memory element that accumulates the values it has seen. It reads from one input port, by issuing a `Recv` request, adds the value it receives to the accumulator element, and then sends the previously accumulated value to a second port by issuing a `Send` request. Finally, the function loops.

```
(data Msg (Recv) (Send Int) (Val Int))
(define (process
    (monad [(@acc (state Unit Int)) (@in (react Msg))
            (@out (reactMsg))] Unit))
  (do (cur <- (get @acc Unit))
      (val <- (signal Recv))
      (case val
        ((Val x) (do (put @acc Unit x)
                     (signal @out (Send cur))
                     (process))))))
```

The `react` features are instantiated in a configuration in the same way that a `state` feature is instantiated. All requests between threads are proxied through a service element. It is this service that defines the message handling protocol for the messages generated by associated threads.

The following `processExample` instantiates `process` as a thread. The input ports `@in` and `@out` are associated with named services, respectively `@src` and `@sink`. A service instantiation includes functions for handling messages by processing requests and generating responses. The service declaration also includes a `scheduler` function, which allow a service attached to multiple threads to decide which threads should receive responses for any given invocation. Although both `service` instantiations use the same handler and scheduler function, the services will operate concurrently. In the case where this system configuration is synthesized to a hardware target, the logic for both the handler and the scheduler functions would be duplicated, one time for each associated service.

```
(configuration processExample
  (memory @st Unit Int)
  (service @src handler scheduler)
  (service @sink handler scheduler)
  (thread (process [@st @src @sink])))
```

Environment Monad The state and reactive monads form the core monadic features that are exposed to a Oread programmer. However, to model the semantics of Oread a third monad *environment* monad is needed. The environment monad is used to model the computational effect of lexical state. This monad is used to define both the static and the dynamic semantics of Oread. However, the environment monad is statically eliminated in the compilation of Oread programs to both software and hardware, as a metacomputation (Harrison and Kamin 2000). In a Oread program, the environment monad is implicit in the definition and use of Oread functions.

The environment monad, parametrized over an environment type `r`, is simply the type of functions `r → a`. Two non-proper morphisms, **`getEnv`** and `withEnv`, manipulate the effects of the monad. The **`getEnv`** morphism accesses the environment, and the `withEnv` morphism takes an environment value of type `r` and an computation, `m`, and executes `m` in the given environment.

```
type Env r a = r → a
unit a = λ_ → a
bind m f = λr → f (m r) r

getEnv :: Env r r
getEnv = λr → r
withEnv :: r → Env r a → Env r a
withEnv e m = λr → m e
```

2.2.2 Oread Semantics

Having defined the basic elements of the Oread language and three `State`, `React`, and `Environment` monads, the semantics of the language can be defined. To simplify the presentation, we use a basic grammar for the language, as shown in figure 2.3. The full concrete syntax for Oreadcan be found in appendix 7.1.

v	$:=$	$x, y, z \ldots$	
t	$:=$	T	User-defined types
	$\mid$	Int	
	$\mid$	Real	
	$\mid$	monad $i^* t$	Monadic computation type
	$\mid$	$t \ldots t \rightarrow t$	Function type
	$\mid$	$(* \, t^+)$	Tuple type
i	$:=$	State t t	State monad instance
	$\mid$	React t	Reactive monad instance
e	$:=$	n	Integer literal
	$\mid$	f	Real literal
	$\mid$	v	Variable reference
	$\mid$	e op e	Arithmetic operation
	$\mid$	$e\, e^+$	Function application
	$\mid$	$\lambda(v : t)^+ . e$	Function abstraction
	$\mid$	prj e e	Tuple projection
	$\mid$	tuple e e^+	Tuple construction
	$\mid$	$C\, e^*$	Constructor application
	$\mid$	case e of $(C\, v^* \rightarrow e \vert)^+$	Case expression
	$\mid$	let $(v : t)^+$ in e	Let expression
	$\mid$	signal i e	Reactive monad **`signal`**
	$\mid$	get $\imath$ e	State monad **`get`**
	$\mid$	put i e e	State monad **`put`**
op	$:=$	$+ \vert - \vert * \vert / \vert =$	Arithmetic operators

Figure 2.3: Oread core syntax

Figure 2.4 shows the static semantics for Oread, in judgement form. Each rule can be read as an inference, where the conclusion on the bottom is implied the antecedents on the top. A judgement of the form $\Gamma : e \vdash t$ is read as "in the environment Γ, the expression e has the type t". The environment, Γ, maps names to types. In the type rules, the letters s,t are meta-variables representing types, f represents monadic instances, C represents data constructors, and T represents user-defined algebraic types.

An Oread program consists of a set of data type declarations, which introduce data constructors, and a set of top-level function declarations. When type-checking an Oread program, the type for each top-level declaration, whether it be a function or data constructor, is assumed to be in the initial environment.

Figures 2.5 and 2.6 show the dynamic semantics of Oread . The semantics are written in monadic form, where the Oread constructs are mapped to monad – Environment, State, and Reactive – morphisms. As these definitions make clear, Oread maps directly to the underlying monadic forms. The constructs in figure 2.5 capture the semantics of the pure functional subset of the language, and only utilize the Environment monad. The semantic definitions include representations for basic arithmetic values (n for integers, f for reals), products (represented by the Prod value), values of constructed data types (represented by the Cons value), and closures (represented by the Clos value). Oread is lexically scoped, so the Clos value produced by the denotation of a λ expression captures the lexical environment, supplied by the Environment monad, the binding variables, and a monadic computation representing the body of the λ expression.

2.3 Summary

Oread is a pure functional language extended with monadic constructs for imperative state and message-passing concurrency. The pure subset of the language allows programs to be manipulated, using fold/unfold transformations, allowing a simple program to be adapted to better utilize the computational features of the target fabric.

VAR
$$\frac{}{\Gamma, x : t \vdash x : t}$$

APP
$$\frac{\Gamma \vdash e : t_0 \ldots t_n \rightarrow s \qquad \Gamma \vdash e_i : t_i}{\Gamma \vdash e\, e_0 \ldots e_n : s}$$

ABS
$$\frac{\Gamma, v_0 : t_0, \ldots, v_n : t_n \vdash e : s}{\Gamma \vdash \lambda v_0 : t_0 \ldots v_n : t_n . e : t_0 \ldots t_n \rightarrow s}$$

INTOP
$$\frac{\Gamma \vdash e_0 : \text{Int} \qquad \Gamma \vdash e_1 : \text{Int}}{\Gamma \vdash e_0\ op\ e_1 : \text{Int}}$$

REALOP
$$\frac{\Gamma \vdash e_0 : \text{Real} \qquad \Gamma \vdash e_1 : \text{Real}}{\Gamma \vdash e_0\ op\ e_1 : \text{Real}}$$

INT
$$\frac{}{\Gamma \vdash n : \text{Int}}$$

REAL
$$\frac{}{\Gamma \vdash f : \text{Real}}$$

EQUALS
$$\frac{\Gamma \vdash e_0 : t \qquad \Gamma \vdash e_1 : t}{\Gamma \vdash e_0 = e_1 : Bool}$$

PROJECTION
$$\frac{\Gamma \vdash e_0 : (* \, t_0 \ldots t_i \ldots t_n) \qquad 0 \leq i < n}{\Gamma \vdash \text{prj i } e_o : t_i}$$

TUPLING
$$\frac{\Gamma \vdash e_0 : t_0 \qquad \ldots \qquad \Gamma \vdash e_n : t_n}{\Gamma \vdash \text{tuple } e_0 \ldots e_n : (* \, t_0 \ldots t_n)}$$

CONSTRUCTOR
$$\frac{\Gamma \vdash C : t_o \ldots t_n \rightarrow T \qquad \Gamma \vdash e_i : t_i}{\Gamma \vdash C\, e_0 \ldots e_n : T}$$

CASE
$$\frac{\Gamma \vdash e_d : t_d \qquad \Gamma \vdash C_i : s_{i_0} \ldots s_{i_n} \rightarrow t_d \qquad \Gamma, v_{i_0} : s_{i_0}, \ldots v_{i_n} : s_{i_n} \vdash e_i : t_c}{\Gamma \vdash \text{case } e_d \text{ of } \{C_0\, v_{0_0} \ldots v_{0_n} \rightarrow e_0 | \ldots | C_j\, v_{j_0} \ldots v_{j_n} \rightarrow e_j\} : t_c}$$

RETURN
$$\frac{\Gamma \vdash e : t}{\Gamma \vdash \text{return } e : \text{monad } ()\ t}$$

BIND
$$\frac{\Gamma \vdash e_o : \text{monad } \ (f_0)\ t \qquad \Gamma, x : t \vdash e_1 : \text{monad } (f_1) s}{\Gamma \vdash \text{bind } e_0\ x\ e_1 : \text{monad } (f_0, f_1) s}$$

GET
$$\frac{\Gamma \vdash i \ : \text{State } t_a\ t_d \qquad \Gamma \vdash e : t_a}{\Gamma \vdash \text{get } i\ e : \text{monad } (\text{State } t_a\ t_d) t_d}$$

PUT
$$\frac{\Gamma \vdash i \ : \text{State } t_a\ t_d \qquad \Gamma \vdash e_a : t_a \qquad \Gamma \vdash e_d : t_d}{\Gamma \vdash \text{put } i\ e_a\ e_d : \text{monad } (\text{State } t_a\ t_d) t_d}$$

SIGNAL
$$\frac{\Gamma \vdash i \ : \text{React } t_m \qquad \Gamma \vdash e : t_m}{\Gamma \vdash \text{signal } i\ e : \text{monad } (\text{React } t_m) t_m}$$

LET
$$\frac{\Gamma \vdash e_i : t_i \qquad \Gamma, v_0 : t_0, \ldots, v_n : t_n \vdash e_b : t_b}{\Gamma \vdash \text{let } v_0 : t_0 = e_0 \ldots v_n : t_n = e_n \text{ in } e_b : t_b}$$

Figure 2.4: Oread static semantics

$$
\begin{array}{lcl}
[\![v]\!] & = & \text{getEnv} \gg\!= \lambda\rho \rightarrow \text{return} \,(\, \text{lookup}\;\; v\,\rho) \\[1em]
[\![e\, e_0 \ldots e_n]\!] & = & \begin{array}{l} [\![e]\!] \gg\!= \lambda\; (\text{Clos}\; x_0 \ldots x_n\; \rho\; \text{m}) \rightarrow \\ [\![e_0]\!] \gg\!= \lambda v_0 \rightarrow \\ \ldots \\ [\![e_n]\!] \gg\!= \lambda v_n \rightarrow \text{withEnv}\; ([x_i \mapsto v_i]\rho) m \end{array} \\[1em]
[\![\lambda x_0 : t_n \ldots x_n : t_n \rightarrow e]\!] & = & \text{getEnv} \gg\!= \lambda\rho \rightarrow \text{return} \,(\, \text{Clos}\; x_0 \ldots x_n\; \rho\; [\![e]\!] \,) \\[1em]
[\![e_0\; op\; e_1]\!] & = & [\![e_0]\!] \gg\!= v_0 \rightarrow [\![e_1]\!] \gg\!= \lambda v_1 \rightarrow \text{return}\; (v_0\; \text{op}\; v_1) \\[1em]
[\![n]\!] & = & \text{return n} \\[1em]
[\![f]\!] & = & \text{return f} \\[1em]
[\![\text{prj}\; i\; e]\!] & = & [\![e]\!] \gg\!= \lambda(\, \text{Prod}\; v_0 \ldots v_n) \rightarrow \text{return}\; v_i \\[1em]
[\![\text{tuple}\; e_0 \ldots e_n]\!] & = & \begin{array}{l} [\![e_0]\!] \gg\!= \lambda v_0 \rightarrow \\ \ldots \\ [\![e_n]\!] \gg\!= \lambda v_n \rightarrow \text{return} \,(\, \text{Prod}\; v_0 \ldots v_n) \end{array} \\[1em]
[\![C\; e_0 \ldots e_n]\!] & = & \begin{array}{l} [\![e_0]\!] \gg\!= \lambda v_0 \rightarrow \\ \ldots \\ [\![e_n]\!] \gg\!= \lambda v_n \rightarrow \mathit{return}(\, \text{Cons}\;\; \text{C}\; v_0 \ldots v_n) \end{array} \\[1em]
\begin{array}{lll} [\![\text{case}\; e_d & \text{of} & \{C_0\; x_{0_0} \ldots x_{0_n} \rightarrow e_0 \mid \\ & & \ldots \mid \\ & & C_j\; x_{j_0} \ldots x_{j_n} \rightarrow e_j]\!] \end{array} & = & \begin{array}{l} [\![d_d]\!] \gg\!= \lambda(\, \text{Cons}\;\; C_i v_0 \ldots v_n) \rightarrow \\ \text{getEnv} \gg\!= \lambda\rho \rightarrow \\ \text{withEnv}\; ([x_i \mapsto v_i]\rho) [\![e_i]\!] \end{array} \\[1em]
[\![\text{let}\; x_0 : t_0 = e_0 \ldots x_n : t_n = e_n\; \text{in}\; e_b]\!] & = & \begin{array}{l} [\![e_0]\!] \gg\!= \lambda v_0 \rightarrow \\ \ldots \\ [\![e_n]\!] \gg\!= \lambda v_n \rightarrow \\ \text{getEnv} \gg\!= \lambda\rho \rightarrow \\ \text{withEnv}\; ([x_i \mapsto v_i]\rho) [\![e_b]\!] \end{array}
\end{array}
$$

Figure 2.5: Evaluation semantics of Oread

$$
\begin{array}{lcl}
[\![\text{return } e]\!] & = & [\![e]\!] \\
 & & [\![e_0]\!] \gg\!= \lambda v \rightarrow \\
[\![\text{bind } e_0\ x\ e_1]\!] & = & \text{getEnv} \gg\!= \lambda \rho \rightarrow \\
 & & \text{withEnv } ([x \mapsto v]\rho) [\![e_1]\!] \\
 & & \\
 & & [\![i]\!] \gg\!= \lambda inst \rightarrow \\
[\![\text{get } i\ e]\!] & = & [\![e]\!] \gg\!= \lambda addr \rightarrow \\
 & & \text{get } (inst, addr) \\
 & & \\
 & & [\![i]\!] \gg\!= \lambda inst \rightarrow \\
[\![\text{put } i\ e_a\ e_d]\!] & = & [\![e_a]\!] \gg\!= \lambda addr \rightarrow \\
 & & [\![e_d]\!] \gg\!= \lambda dat \rightarrow \\
 & & \text{put } (inst, addr) dat \\
 & & \\
 & & [\![i]\!] \gg\!= \lambda inst \rightarrow \\
[\![\text{signal } i\ e_m]\!] & = & [\![e_m]\!] \gg\!= \lambda msg \rightarrow \\
 & & \text{signal } (inst, msg)
\end{array}
$$

Figure 2.6: Evaluation semantics of monadic constructs

Computational effects in Oread are modeled as monads. The semantics of Oread use three such monads: the State monad, for imperative effects; the Environment monad, modeling lexical scope in the pure subset of the language; and the Reactive monad, used to capture the side-effects of message-passing concurrency. To allow these effects to be localized when implemented in specialized fabrics, monadic functions are parametrized over monadic instances parameters.

Oread provides a concurrency framework that allows independent components, threads, to interact via message-passing concurrency. All thread communication is proxied through Oread services, which define the protocol through which the threads communicate. Rather than define a fixed set of service protocols, the Oread concurrency architecture allows a programmer to define the service logic, as a pair of functions. The handler function defines the interpretation of thread requests and generates responses, while the scheduler function simply determines which threads have responses pending and are ready to resume execution.

Chapter 3

Compiling Oread to Hardware

Oread is designed to be compiled to both hardware and software targets. This chapter details the compilation of a Oread system to VHDL, for input to downstream synthesis tools for the chosen hardware fabric. In an unrestricted form, Oread has properties that make it inappropriate (or impossible) to compile directly to hardware targets. Some language features – unrestricted recursion, recursive data types, closures, and higher-order functions – are unsuitable for hardware targets where resource usage and control flow must be statically determined.

Rather than sacrifice language expressiveness by restricting Oread to eliminate these features, we capitalize on an existing body of program transformations to convert Oread programs that do not satisfy the limitations of the VHDL synthesis into programs defined in a synthesizable subset of Oread. The synthesis subset of the language is a first-order functional language extended with monadic constructs for imperative and concurrency effects.

The compilation of Oread to VHDL requires a small collection of predefined VHDL components. These components, described in detail in section 3.2, are "wired" together by the synthesis scheme to implement Oread functionality in hardware. Although the components are implemented as behavioral VHDL, they are fully synthesizable, as is the Oread synthesis output. Moreover, the choice of VHDL as a target is not crucial; the synthesis routines can easily be adapted to generate Verilog or EDIF output.

3.1 Hardware Control Protocol

The fundamental design decision for VHDL compilation is the use of an explicit "ready" control protocol. A Oread program compiled to VHDL consists of a collection of data signals that hold values calculated by the program, and control signals that coordinate the flow of values through data-transforming blocks.

Each data signal has an associated 1-bit control signal that is persistently low, except in the situation when control exits the data-producing component. When this is the case, the control signal associated with a data signal will be high. In the Oread compilation schemes as implemented and described in this chapter, the control signal will pulse high for one clock cycle, indicating that the value on the associated data signal is valid. When a control signal is *not* high, the synthesis routine makes no guarantee regarding the validity of the associated data signal. A disadvantage of this protocol is that it may introduce redundant control signals and circuit inefficiencies. However, due to the simplicity of the protocol, many of the inefficiencies can be safely eliminated from the generated circuit in a separate optimization pass.

Example: Consider the following Oread fragment.

```
(define (f (x Int) Int) (+ x 1))
(define (g (x Int) Int)
  (+ (f x) (f (+ x 1))))
```

The compilation scheme will generate two *shared function blocks*, one for each function definition. The two calls to `f` within the body of `g` will happen simultaneously, yet will be serviced consecutively as they reference the same shared block function block.

Figure 3.1 shows a graphical depiction of the resulting control and data flow graph for the generated VHDL netlist. In the graph, control signals are shown as dashed edges and data signals are shown as solid edges. The shared `f` function block is delimited by the large rectangular block located in the middle of the graph.

Figure 3.2 shows the simulation result from executing a call `(g 1)`. The call begins with the `sigstart` signal pulsing high, indicating a call request to `g`. Moreover, the `sigargs` signal carries the argument value `1`. As the waveform shows, the value of the data signal associated with the call, `sigresult`, varies across the call to `g`. It is initially `0`, then `3` after the `(f (+ x 1))` call in the body of `g` has completed[1], and finally `5`, the correct value of `(g 1)`. The control signal, `sigdone`, associated with the data signal `sigresult`, pulses high at the instant that both calls have completed, indicating that `sigresult` carries valid data.

The above example demonstrates two assumptions of the control protocol. First, the scheme requires that a call to a shared function block, indicated by the control signal associated with the call's data signal, will maintain the validity of the argument signal until the function block completes the call, which could take an indefinite amount of time. Second, it assumes that the result of a call to a shared function block will be valid only when the call's associated done signal, emanating from the shared function block, is high.

For this reason, the synthesis routine inserts a register at the receiving end of a function call that latches the function block's result signal that is shared by all call sites. Inserting this register has the secondary effect of ensuring that the first invariant

[1]The synthesis scheme makes no guarantees regarding the order that concurrent requests to the same function block are served. Because each call is pure, the order is irrelevant both in the circuit result and overall timing.

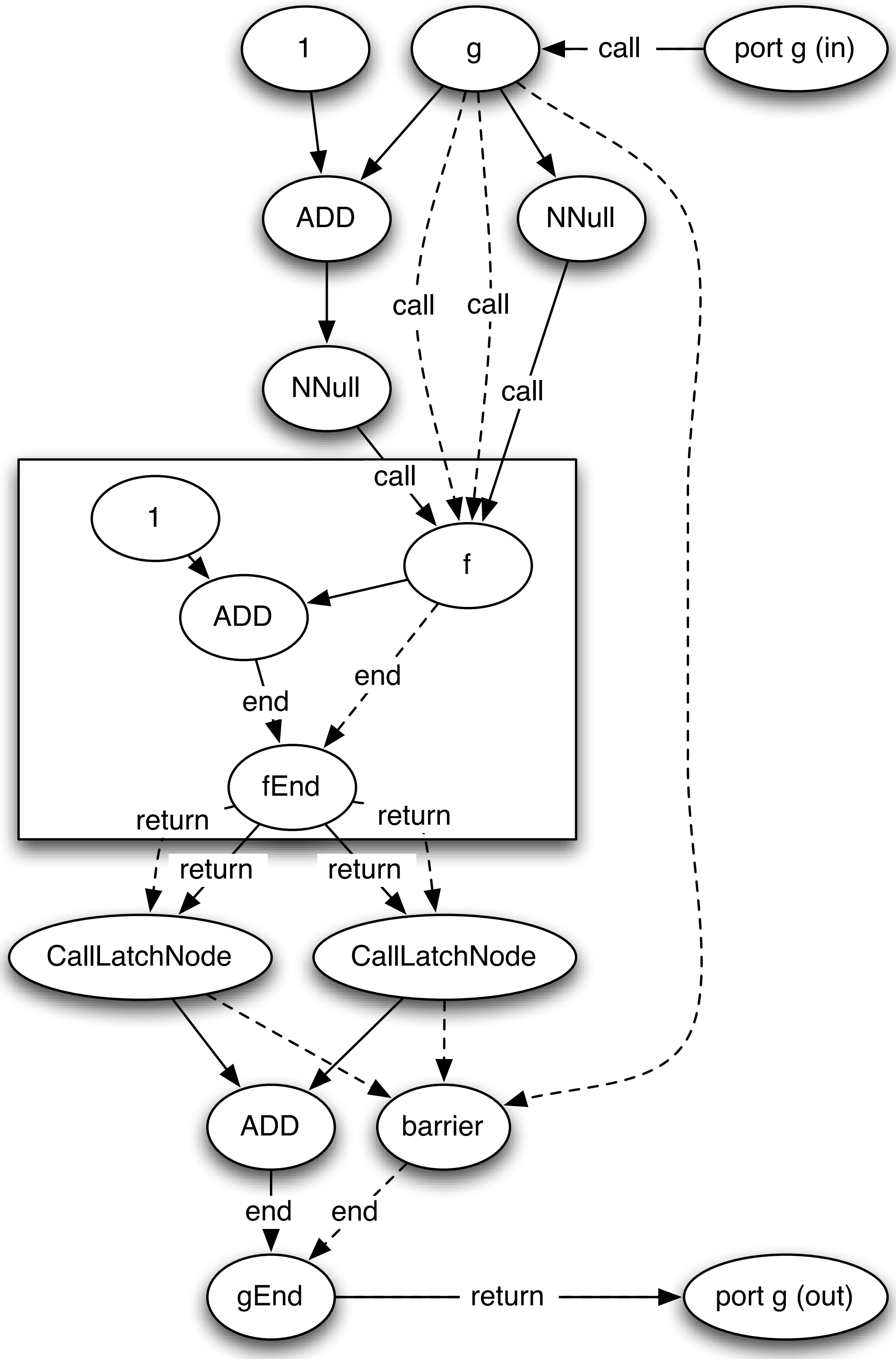

Figure 3.1: Control and data flow graph

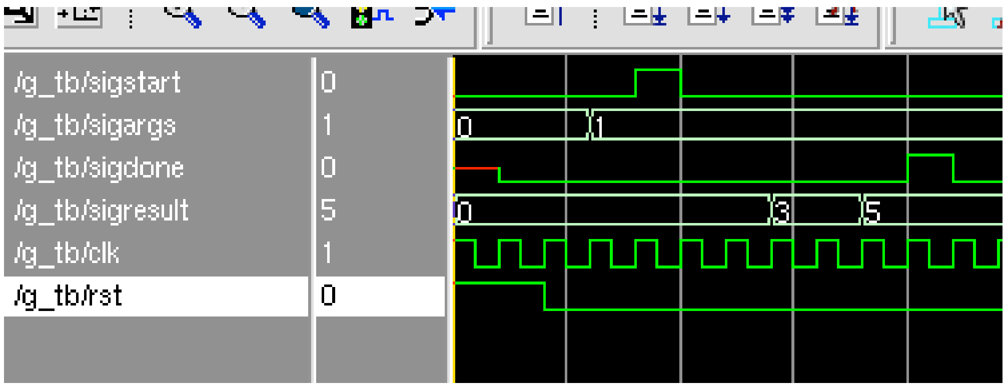

Figure 3.2: Ready protocol example

is maintained, as apart from calls to shared function blocks and monadic operations [2] all data signals are the result of combinational circuits. Therefore, they will not change unless an input signal has changed. This will only occur when control flows through an upstream sequential component. The protocol ignores combinational circuit propagation, assuming that the synthesis result is a clocked circuit which meets timing requirements. A programmer can perform transformations at the Oread source level to control the timing behavior of the system.

Figure 3.3 shows the control protocol with an expanded version of the same example in figure 3.2. For the interface to the `f` shared block, the signals `f/args`, `f/cin`, `f/cout`, and `f/dout` are the call arguments (a concatenation of the arguments from each call), call requests (a concatenation of the requests from each call), call complete (a bitvector with control bit per call site) and call result. Moreover, for each call site the display lists the request control signal, arguments data signal, done control signal, and finally the result data signal.

Figure 3.3 shows a simulation of evaluating `(g 1)`. After some preliminary testbench setup time, the body of the `g` function begins at 80ns, indicated by `sigstart`

[2]Monadic operations are equivalent to calls to shared function blocks from the point of view of the protocol.

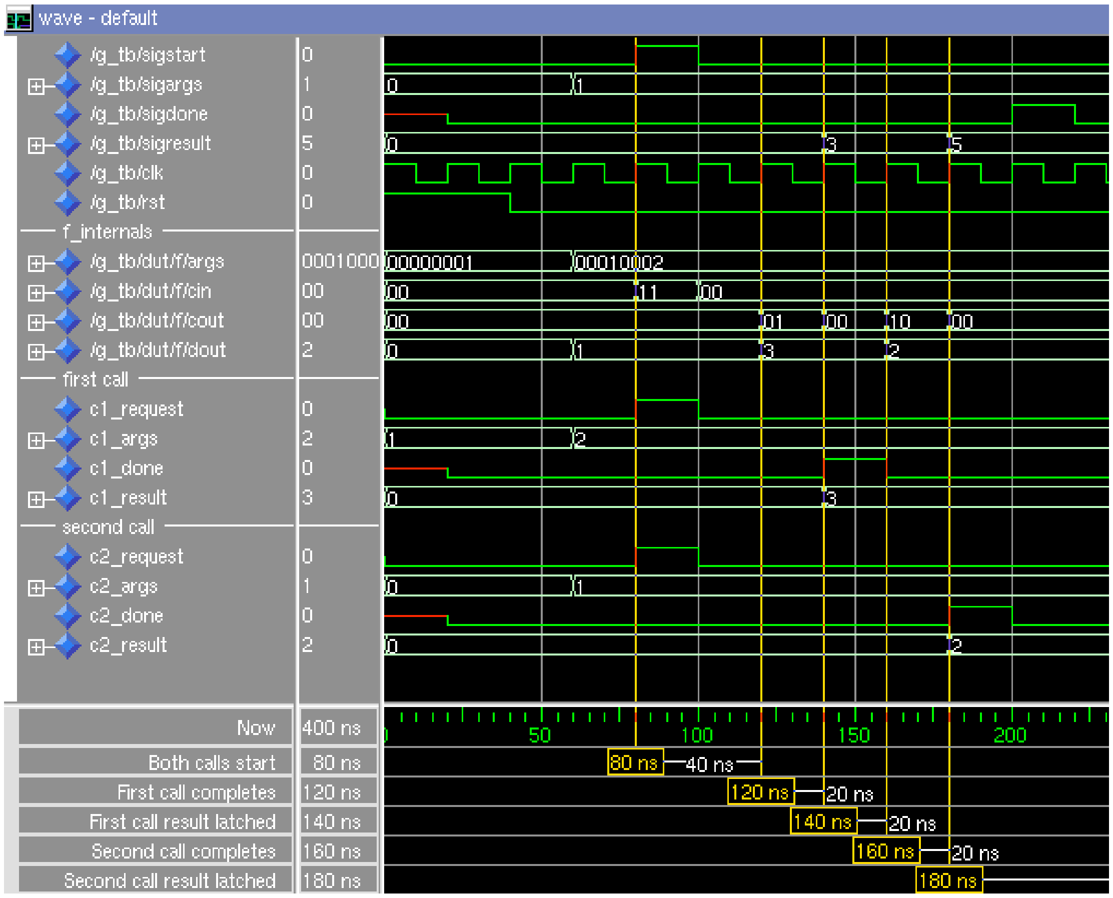

Figure 3.3: Call latching example

pulsing high. As the calls to f do not depend on any sequential components, both call sites request a service at the 80ns time, as indicated by c1_request and c2_request signals at the call site and the f/cin signal at the shared function block. At 120ns, the first call has been completed, indicated by the 1 bit in the f/cout signal, and the value of 3 on the f/dout signal. One cycle later, at 140ns, the f/dout signal is latched the register with the output signal c1_result, and the control signal associated with the latch, c1_done pulses high.

Throughout the time span between 80 ns, when both calls were requested simultaneously, and 140 ns, when the f block finishes servicing the first call, the second call maintains the value of its arguments on signal c2_args. Following the completion of the first call, at time 140 ns, the function block begins servicing the second call. At 160 ns, the second call is completed indicated by the other bit in the f/cout signal being 1. Moreover, the data output signal f/dout for the f function block now contains the result of the second call; however, the output of the first call c1_result is not affected as it had previously been latched. The result of the second call is latched at 180 ns and the ready signal c2_done is pulsed. Finally, the evaluation of (g 1) completes at 200ns, with sigdone pulsing high.

A further consequence of combinational circuits only changing when upstream sequential components change is that the control signal associated with a combinational circuit's data output is equivalent to the control signal for the upstream component. For combinational circuits with multiple inputs, the control signal is associated with the *collection* of upstream components. The library of primitive VHDL components that the Oread compilation scheme relies upon contains a Barrier component which takes any number of these control inputs and only propagates control – by pulsing its control output high – when every control input has pulsed high, each of which may occur in different clock cycles.

As an optimization, the synthesis scheme only inserts a barrier component for a combinational component output when the combinational component's input is consumed by a sequential component. Without this optimization, a chain of combinational components, such as would be generated by the expression `(+ (+ (+ w x) y) z)`, would introduce a barrier for each addition operation. As each barrier component induces a 1 clock-cycle delay in its control output, this would compound to a 3-cycle delay across the expression.

The barrier-insertion optimization diminishes the number of cycles taken to complete a chained combinational circuit, but it has a degenerative effect on the clock speed of the circuit, as it compounds combinational propagation delays. An alternative is to balance clock speed vs. latency with a combination of barrier insertion strategies or program transformations that induce the same effect with a single insertion strategy. Performing the space/time trade off exploration at design time is a principal motivation of the functional-language approach to hardware synthesis advocated by this thesis.

3.2 Primitive VHDL components

Compiling the synthesizable subset of Oread to VHDL proceeds by first generating a control and data flow graph for the program, then generating VHDL entities and architectures corresponding to that graph. These generated VHDL modules are structural VHDL, which simply instantiate a series of hand-written primitive VHDL components.

The primitive components that the Oread synthesis scheme instantiates can be divided along two axes, as shown in table 3.1. The horizontal axis determines if the component is primarily used for managing the control protocol or if it used for calculating data. The vertical axis determines whether the component is combinational or sequential, which has a significant impact on the associated control infrastructure.

Within this taxonomy, the first group the combinational data components, such as the arithmetic operations, tuple projection and creation, and constructor application. The second are the combinational control components, including the split and join components that guard control though a case expression. The barrier component, is the only sequential control component. The remaining components are classified as sequential data. The CallLatch component saves the value of a call to a shared function block. The FunBlock, JumpBlock, MemBlock, and Service components that mediate access to shared blocks.

	Primitive	Oread Construct	Primitive	Oread Construct
Comb	$+,-,*,/$	Arithmetic	Join	**`case`**
	sig (h **downto** l)	`prj`	Split	**`case`**
	sig1 & sig2 & sig3	`tuple,Constructor`		
Seq	CallLatch	Function application	Barrier	N/A
	FunBlock	Functions		
	JumpBlock	Recursive Functions		
	MemBlock	State instances/**`get`**, **`put`**		
	ServiceBlock	Reactive instance/**`signal`**		

Table 3.1: Oread primitive VHDL components

3.2.1 Combinational Data Components

The Oread VHDL compilation schemes utilize a variety of combinational data components that support operations such as arithmetic, tupling and projection, and bit-vector operations. These operations are directly implemented in VHDL without special support for the Oread control protocol.

The synthesis scheme assumes that the combinational data components can yield valid data in a single clock cycle. For this reason, the components do not have specific control logic – in the form of a control signal – associated with their data outputs. Instead, all control signals that would otherwise be used for these components are aggregated at the sequential components which utilize the data output of the combinational components, in the form of an instantiation of a barrier sequential control component.

For many of these components, the instantiation of the data operations comes in the form of a concurrent signal assignment:

```
sigDest <= sigArg0 op sigArg1;
```

Instead of an explicit instantiation of a data precessing component:

```
opInst: entity op
  port map(din0 => sigArg0, din1 => sigArg1, dout => sigDest)
```

The advantage of this scheme is that it reduces the number of primitive VHDL components needed to generate VHDL from a Oread program. Rather than force the Oread synthesis tool to manually instantiate components for combinational data flow – simple wrapper entities around the same concurrent signal assignments – the compilation scheme pushes the synthesis of the combinational circuit down the tool chain to the VHDL synthesis tools that convert the Oread VHDL output into an FPGA net list.

The most significant drawback of this scheme is that there is no explicit connection between the Oread synthesis properties, most importantly that all combinational data flow components must be able to calculate output in one clock cycle, and the resulting VHDL. An example of where this becomes a issue is in the multiplication of large bit-width integers.

The FPGA used for the demonstration of Oread synthesis contains a limited number of 18-bit multipliers. These components are specialized multiplier circuits that are available in the FPGA fabric in addition to the standard collection of LUTs and flip-flops. When mapping VHDL components to FPGA technology, the Xilinx-supplied synthesis tools will utilize one of these 18-bit multiplier blocks when it encounters a multiplication operation with arguments that are less than 18 bits wide.

This scheme is sufficient as long as the inputs to the multiplication satisfy bit width requirements, and there are a sufficient number of multiplier blocks available. The Oread synthesis scheme does not presuppose a specific bit width for integers, and can be updated to support arbitrary widths. However, the current tool set defaults to 16-bit wide integers, allowing the synthesis tools to use those special purpose multiplier blocks.

If the size of integer representations is changed to one that is greater than 18 bits, it is no longer possible to map a Oread multiplication operation into a single 18-bit multiplier block. When the Xilinx synthesis tools discover a multiplier which exceeds the 18-bit limit in a VHDL specification, they generate a pipelined version of the multiplier, using multiple 18-bit multiplier blocks. The pipelined multiplier, in an attempt to minimize the number of multiplier blocks required, inserts registers on the data path. As a consequence, the resulting circuit will not adhere to the restriction the multiply data component will complete in a single clock cycle. Unfortunately, because the restriction is not explicit in the VHDL specification or otherwise communicated to the low-level synthesis tool, the resulting circuit is incorrect with respect to the Oread ready protocol.

One mechanism for dealing with this problem is to eliminate the control independence of the combinational data components by including cycle counts of all combinational components, and inserting the proper number of delay components on the control signal associated with the combinational data output. Each cycle delay introduced by the VHDL to FPGA synthesis tools requires an extra 1-bit flip-flop on the control signal. Given the abundance of flip-flops available on modern FPGA technologies, this seems a reasonable choice. However, tracking cycle delays due to technology mapping of low-level synthesis tools requires the Oread synthesis scheme to have an intimate understanding of how the high-level Oread constructs are mapped to low-level FPGA components.

The control-delay insertion scheme can be extended to account for sub-cycle timing information. For example, the Oread synthesis routines assume that not only will a single combinational operation be completed in a single clock cycle, but any chain of combinational operations can also be completed in a single cycle. While this scheme is straightforward and sound, it suffers in that it may generate an extremely long sequence of combinational circuits, with the associated lengthy propagation delay through the circuit. The resulting circuit may be perform poorly, since the global clock frequency of a component is bounded by the longest combinational path through the circuit, and possibly not even realizable. If a lengthy combinational path occurs on a control branch that is seldom active, then it is likely that inserting a pipeline register on that path and suffering a cycle delay will result in a net speedup as the clock speed increases for the entire circuit.

The ability of the Oread VHDL compilation scheme to account for combinational delay forces the schemes to become even more aware of the properties of the target fabric. As the tools become more precise regarding timing behavior, they must take into account not only the FPGA technology mapping generated by the downstream synthesis tools, but also the spatial properties of the resulting circuit. Integrating this increasingly complex timing analysis has diminishing returns, as the synthesis tools become more technology-dependent and less able to target new fabrics. Rather than going to extremely low-level detail, the Oread schemes currently endeavour to simply limit the construction of obviously bad circuits by performing gross timing analysis and inserting pipeline registers on poor paths. As long as the logic mapping adheres to the constraint that all combinational circuits are mapped to circuits which can be completed in a single cycle – or sequential circuits with a known number of delays, the Oread VHDL compilation scheme will continue to yield correct functionality.

A second issue involved in mapping components to FPGA technology arises when resources are limited, as is the case with the number of multiplier blocks on the demonstration CPU, or for arithmetic operations on floating-point values. In these cases, the design of Oread allows those components to be encapsulated as shared function blocks. When the number of combinational components required by a Oread program exceeds the availability of the FPGA resources, multiple instantiations of a combinational circuit can be wrapped within a single function block, and the uses of that combinational circuits shared amongst the various occurrences of the combinational expression, which are then converted into calls to the shared function block.

Example: Consider the multiplication of complex numbers[3]. A complex number $x = a + bi$ is represented as a tuple `(tuple a b)`.

```
;; x = a + i b
;; y = c + i d
;; x*y = (a*c - b*d) + i (a*d + b*c)
(define (cmult (x (* Int Int)) (y (* Int Int)) (* Int Int))
  (tuple (- (* (prj 0 x) (prj 0 y)) (* (prj 1 x) (prj 1 y)))
         (+ (* (prj 0 x) (prj 1 y)) (* (prj 1 x) (prj 0 y)))))
```

An implementation of complex multiplication uses four multipliers. The corresponding netlist, as a control and data flow graph, for this implementation can be found in figure 3.4. As the figure shows, four independent multiplier components are generated. Moreover, because all of the multiplications are combinational and performed in parallel, control flows immediately from the `cmult` function block beginning to the end.

Alternatively, a single top-level function wrapping the multiplication operation can be defined, and all of the previous multiplication operations replaced with a call to the

[3]This example only considers complex numbers with integer components. When this is relaxed to include real-valued components, the transformation is even more crucial, since floating-point multipliers require much greater hardware resources

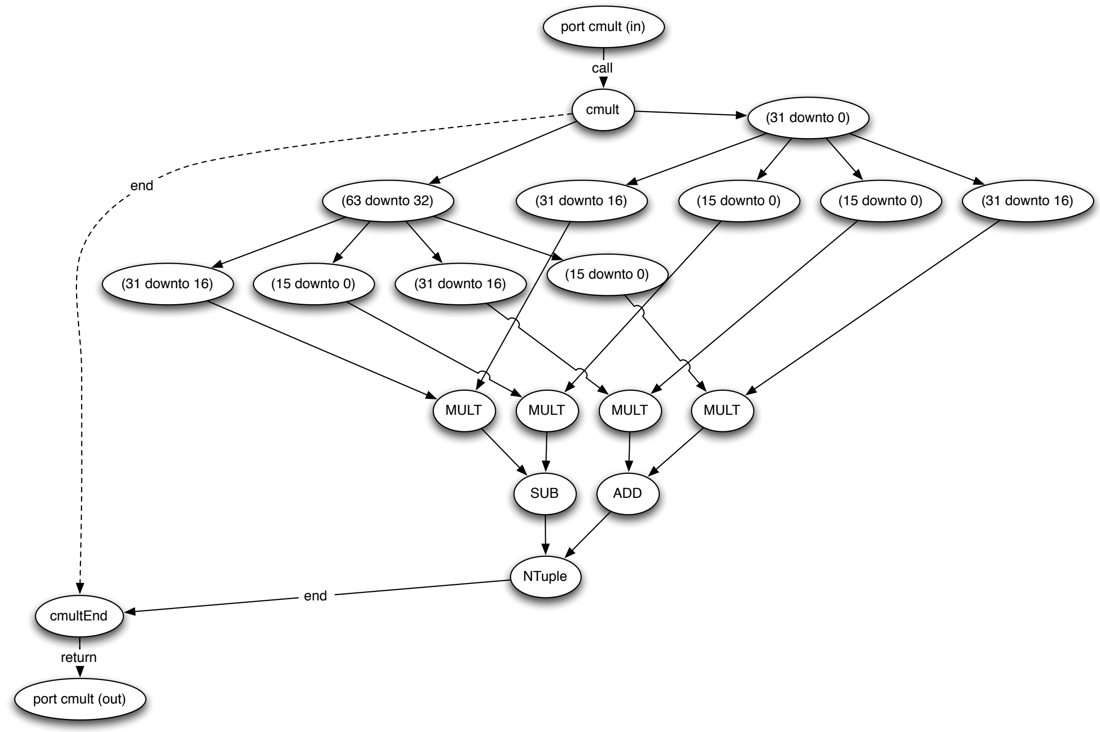

Figure 3.4: Complex Multiplication (Simple)

top-level function, as shown in the `cmultSingle` definition below. Figure 3.5 shows the associated control and data flow graph. The rectangle in the middle represents the shared `mult` function block. Because all four calls to the `mult` block conflict, this circuit will take four times as long to produce data.

```
(define (cmultSingle (x (* Int Int)) (y (* Int Int)) (* Int Int))
  (tuple (- (mult (prj 0 x) (prj 0 y)) (mult (prj 1 x) (prj 1 y)))
         (+ (mult (prj 0 x) (prj 1 y)) (mult (prj 1 x) (prj 0 y)))))

(define (mult (x Int) (y Int) Int)
        (* x y))
```

The timing cost of resource coalescing will introduce delays due to the sequential nature of the shared function block. Moreover, the resource cost of performing this folding of combinational circuits into sequential shared blocks includes that of a single combinational circuit for the function body, the arbitration and multiplexing logic

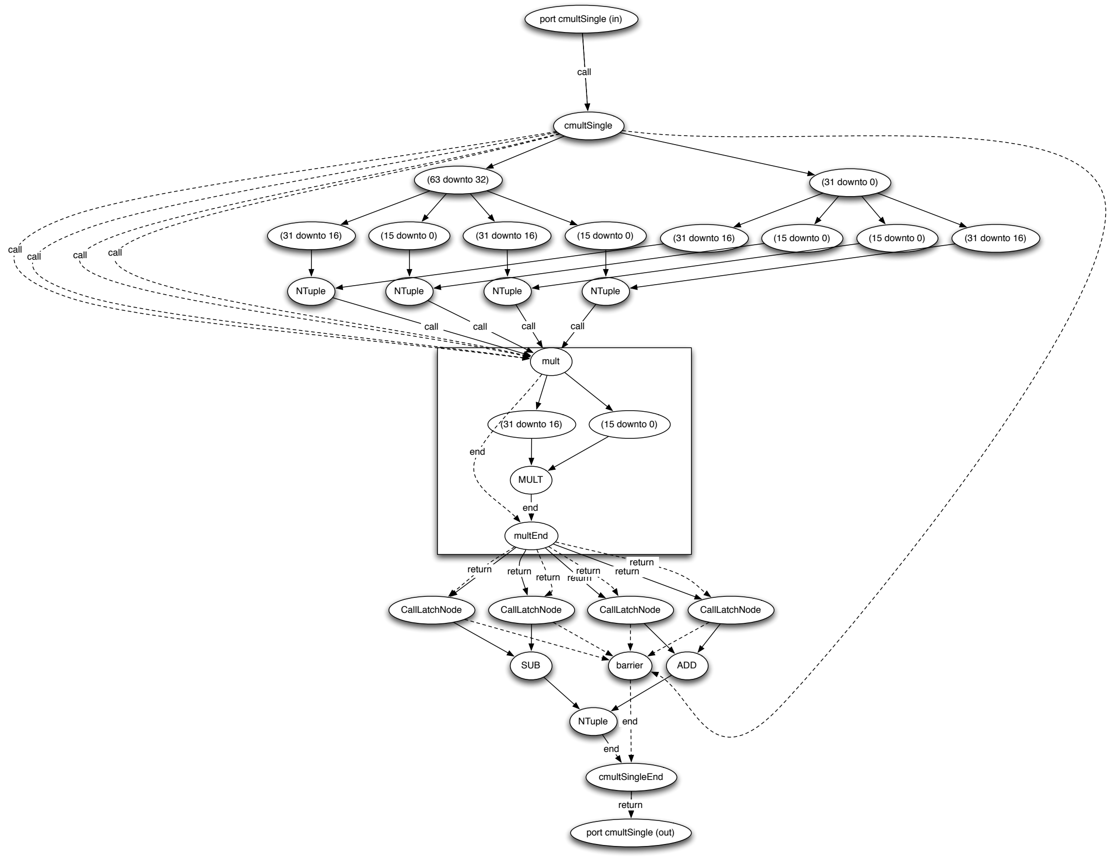

Figure 3.5: Complex Multiplication (Collapsed)

associated with a shared function block, as well as a result register at each call site. The resources necessary to implement this logic are plentiful on the target fabric, so the folding transformation will sacrifice smaller, faster circuits constructed of scarce resources for larger and slower circuits implemented using more plentiful blocks.

The minimum penalty is 2 clock cycles, the lower bound for a call to a shared function block. However, calls may incur an additional penalty when there are other calls that conflict – when a request for a shared function block at one call site occurs at the same time as, or during, a different call to the same shared block.

Example: To balance the extremes of the previous two implementations of complex multiplication, the `mult` function is duplicated into two separate multiplier function blocks. The calls to `mult` are then distributed between the two shared function blocks.

```
(define (cmultDouble (x (* Int Int)) (y (* Int Int)) (* Int Int))
  (tuple (- (mult1 (prj 0 x) (prj 0 y)) (mult2 (prj 1 x) (prj 1 y)))
         (+ (mult1 (prj 0 x) (prj 1 y)) (mult2 (prj 1 x) (prj 0 y)))))
(define (mult1 (x Int) (y Int) Int)
        (* x y))
(define (mult2 (x Int) (y Int) Int)
        (* x y))
```

Figure 3.6 shows the corresponding graph. The two rectangles in the middle of the graph represent the two copies of the `mult` function block. Assuming that the space cost of the multiplication logic dominates the overhead of the wrapper logic, the two function blocks service calls independently, allowing the circuit to complete in roughly half the space and twice the time of the simple circuit, and twice the space and half the time as the collapsed version.

The combinational data operations in Oread map to the VHDL operations shown in table 3.1. The VHDL output of the Oread compiler uses the `std_logic_arith` package, which includes a `std_logic_vector` type and a wide collection of synthe-

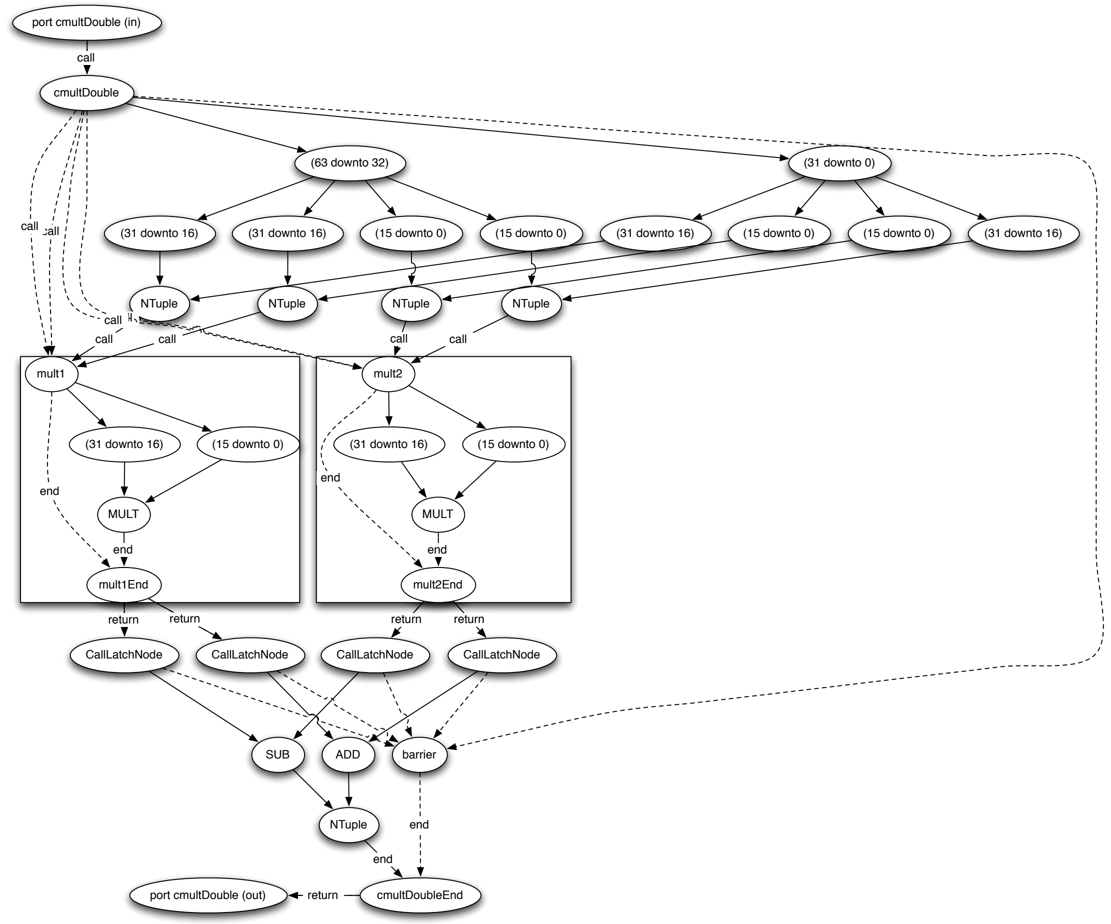

Figure 3.6: Complex Multiplication (Balanced)

sizable arithmetic operations. A `std_logic_vector` signal of the appropriate with is declared for each expression and is used as the output for that expression.

Example: Consider the following Oread expression.

```
(+ (- x 1) (* y 3))
```

This generates a series of VHDL declarations.

```
signal sig0 : std_logic_vector(15 downto 0);
signal sig1 : std_logic_vector(15 downto 0);
signal sig2 : std_logic_vector(15 downto 0);
```

The logic of the expression is implemented as a collection of concurrent signal assignments.

```
sig0 <= sig1 + sig2;
sig1 <= sigX - conv_std_logic_vector(1,16);
sig2 <= sigY * conv_std_logic_vector(3,16);
```

The `sigX` and `sigY` signals are implicitly declared in this code sample. Variables are introduced by functions, let bindings, monadic binding, and case statements, and the appropriate declarations and signal assignments will be generated at the point where those constructs are compiled, rather than where they are used.

Each data output has an associated collection of control output signals. The control output signal associated with `sig0` is the union of the control signals associated with `sig1` and `sig2`. The control output associated with `sig1` is `sigX`, and likewise with `sig2` associated with `sigY`.

The control outputs for a combinational operation are noted, although not used until the data output for the operation is used by a sequential circuit. When a sequential node utilize the data output from a combinational circuit, the control outputs for the combination circuit are calculated, and the set of outputs is refined by adding ancestor nodes.

For example, in the expression `(+ x (f x))`, the control output associated with the expression `x` is used both as a control input to the call to `f`, as well as to the top-level addition. However, when searching the control output ancestry for a combinational circuit, the search ends when a sequential node, such as that generated by the call `(f x)`, is generated. Consequently, the control output for `(+ x (f x))` will depend both on the control output for `x` and the control output for `(f x)`. If this expression were to be used by a sequential call, e.g. `(g (+ x (f x)))`, then the generated VHDL will include a barrier component with inputs for both `sigCtlX` and `sigCtlFX`.

A barrier component with *n* inputs requires *n* 1-bit registers and an *n*-bit comparator, so this barrier would use 2 registers. However, one of the control inputs to the barrier, `sigCtlFX`, depends on the other, `sigCtlx`. The `sigCtlFX` signal is simply some indefinite (due to `f` being a shared function block) delay of `sigCtlx`. In this case, the control dependency of the expression `(+ x (f x))` on the control output `sigX` is extraneous and can be eliminated. Moreover, since the barrier component will then only have one control input, it is no longer necessary, and can be eliminated, with the control output dependency set for `(+ x (f x))` reduced to `sigCtlFX`.

Much as the arithmetic and operations map to VHDL primitive operations, the tupling and projection operations map to VHDL signal concatenation and slicing primitives. Unlike the arithmetic primitives which operate on fixed-sized data signals, the concatenation and slicing VHDL operations are indexed-based. Consequently the compilation scheme requires Oread type information to calculate the proper indices.

The tuple-construction operation used bitvector concatenation to construct the tuple. The size of the resulting bitvector is the sum of the sizes of the input arguments to the tupling operation. Bitvector concatenation in VHDL uses no logic resources (although it may utilize wiring resources), so it is generally an inexpensive operation. As with all of the other combinational operations, the tupling operation may result in extra resource usage if the operation is being used by a sequential component, as it may be necessary to generate a barrier component to handle the control output.

The inverse of a tupling operation is projection. The projection component may require the instantiation of no data logic if the projection index is constant as in the expression `(prj 2 (f x))`. On the other hand, if the projection index is *not* constant, e.g. `(prj i (f x))`, then it becomes necessary to instantiate a multiplexer for the projection operation. Using a tuple projection when the index is not constant is common in recursive functions over *n*-ary products, where the tuple index is the induction variable. A key restriction on these tuples is that the elements have homogeneous type. Contrast this with the usage where the index is statically known: the element types can vary, and the projection operations behave as record selection.

For the simpler (constant-index) case, consider the following example. The expression `e` is of type `(* Int Char)`, and `ord` maps characters to their ASCII encodings. `Ints` are 16 bits wide, and `Chars` 8 bits.

^{0}`(+ `^{1}`(prj 0 e) `^{2}`(ord `^{3}`(prj 1 e)))`

The resulting VHDL includes the following declarations. The numerical suffixes for the signal names correspond to the left-to-right ordering of the operations.

```
signal sig0 : std_logic_vector(15 downto 0);
signal sig1 : std_logic_vector(15 downto 0);
signal sig2 : std_logic_vector(15 downto 0);
signal sig3 : std_logic_vector(7 downto 0);
```

The resulting signal assignments is:

```
sig0 <= sig1 + sig2;
sig1 <= sig_e(15 downto 0);  -- prj 0 e
sig3 <- sig_e(23 downto 16); -- prj 1 e
```

In general, for a projection `(prj i e)` from a tuple `e` of type $(*T_0 T_1 \ldots T_n)$ and constant `i` will result in a VHDL signal declaration and a signal assignment. The indices are calculated using a *size* function mapping Oread types to bitvector widths. The `offset` index is calculated as `offset` = $\sum_{n=0}^{i-1} size(T_n)$.

```
signal sig_p : std_logic_vector(size(T0) - 1 downto 0);
sig_p <= sig_e(size(ti) + offset - 1 downto offset);
```

The Oread type system requires that products with a dynamic projection index have elements of homogeneous type. If this were not the case, it would not be possible to generate a single type for the result of the projection.

The Oread hardware compilation scheme will instantiate a VHDL multiplier component when a dynamic index is found in a projection. For an *n*-ary projection of elements, where each element is *d* bits wide, *d* *n*-to-1 multiplexers will be instantiated. The Oread synthesis library of primitive components contains a generic (*n* x *d*) multiplexer component that instantiates packages the individual *n*-to-1 multiplexer instantiations into a single component. This same multiplexer circuit is used in the function and case expression compilation schemes for multiplexing arguments from multiple call sites and case alternatives, respectively.

A dynamic projection can be converted into a static projection by lifting the projection into a case statement. The expression `(proj i e)`, where `i` is dynamic, can be converted into the equivalent form.

```
(case i of
  ((0) (prj 0 e))
  ((1) (prj 1 e))
  ...
  ((k) (prj k e))) ; e is a k-ary product.
```

The final construct family compiled to combinational data logic is constructor applications. The two constructs in the sums family, case expressions and constructor applications, are handled separately, as case expressions modify the control flow, in terms of which alternative is activated by pulsing the alternative's ready signal. Constructor applications are simply combinational circuits that concatenate the data signals of sub-expressions in addition to adding a constructor tag.

3.2.2 Combinational Control Components

The Split and Join primitive VHDL components are used for compiling Oread case expressions. The Split component is used at the beginning of a group of guarded case alternatives to determine which alternative will have its control input activated. Conversely, the Join component is instantiated at the end of a case statement to collect the control and data output from each alternative and propagate control from the appropriate branch.

In principle, the Split component is simply a decoder. The selection input for the Split component is the tag for the constructor appearing in the case discriminant. The Oread synthesis routines generate a unique numerical tag for each constructor in an algebraic data type. This tag input is used to demultiplex the discriminant's control output onto a collection of control signals, one for each case alternative. Because the data signal emanating from the discriminant is valid at the same instant that the control signal is high, this component can be implemented purely in combinational logic.

The corresponding Join component for a case expression multiplexes data *and* control from the various case alternatives onto a single data and control signal for the entire case expression. As with the Split component, the selection input for the Join component is the integer tag associated with the case discriminant. Because of the control protocol invariants, it is not necessary to latch the value of the discriminant's data output across the control invocation of a case alternative. The Join component consists of a pair of multiplexers, one for the data signal and one for the control signal.

Example: Consider the following Oread program, using a case expression to guard two function calls. This function utilizes the = operation, which generates a value of the type `Bool`, also defined below.

```
(data Bool (True) (False))
(define (h (x Int) Int)
  (case (= x 0)
   ((True) (inc x))
   ((False) (dec x))))

(define (inc (x Int) Int)
  (+ x 1))

(define (dec (x Int) Int)
  (- x 1))
```

Figure 3.7 shows the associated control and data flow graph. Control flows from the beginning of the `h` function block to a Split node. Moreover, there is a data edge from the EQUAL operation (generating a `Bool`) to the split node, to be used as the decoder's selector signal. Control then flows from the Split node to one of two Alt nodes, labeled with the tag associated with that case alternative. In the generated VHDL, the Alt nodes require no logic resources, but simply correspond to different elements of the Split node's control output port.

Control flows from each Alt node to calls to the respective `f` and `g` shared function blocks, enclosed individually in rectangles in the diagram. Each call involves a corresponding CallLatch sequential data node which latches the output of the function block. From there, control and data flow to the Join node. The Join component multiplexes the correct data and control signal, based on the selection value from the discriminant.

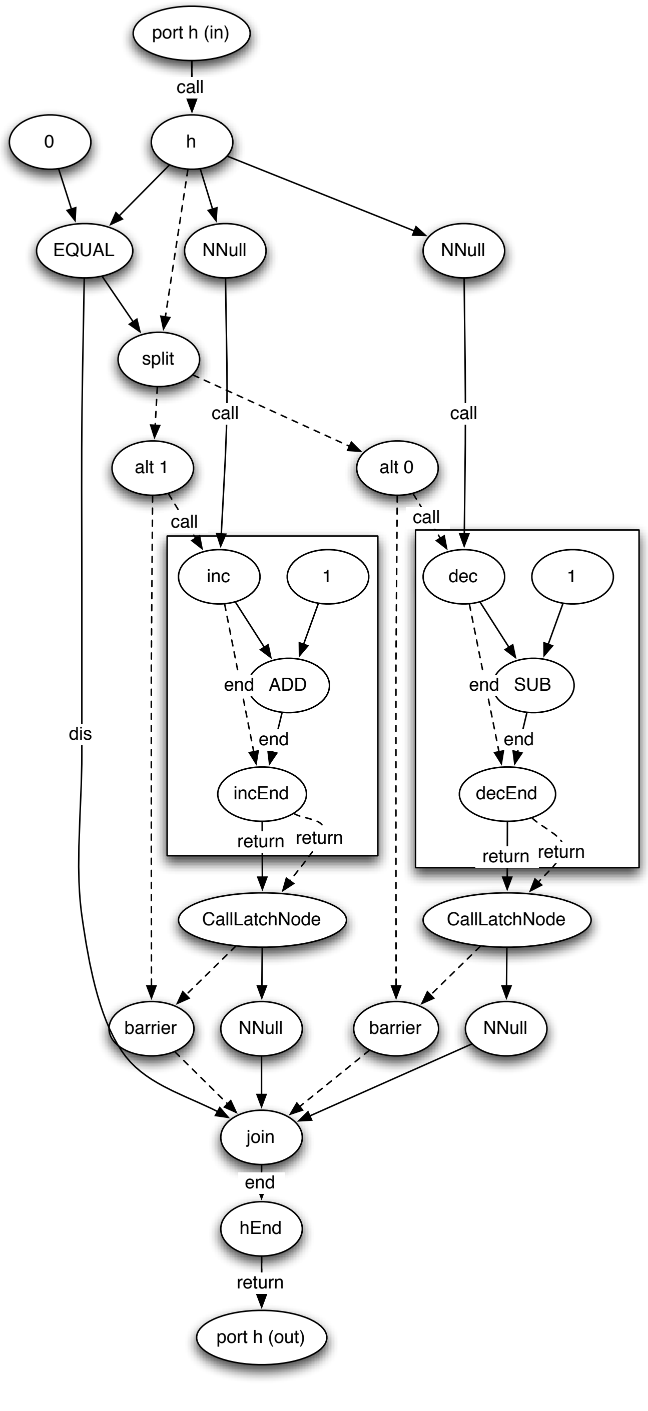

Figure 3.7: Combinational Control

For functions which include a recursive call guarded by a case expression, the control and data output for the recursive call are not routed to the Join node for the case expression. Rather, those signals are routed as jump request to the enclosing shared block. Consequently, the Join node will only contain control and data inputs for alternatives that are not recursive calls. In functions where there is a *single* alternative where this is the case, the VHDL compilation scheme will omit the extraneous Join node, and instead associate that alternative's control and data output signals as the outputs for the entire case expression.

3.2.3 Sequential Components

The set of Oread VHDL sequential primitives includes the Barrier control primitive and a collection of sequential data primitives. Despite the name, the data primitives utilize both the control and data signals generated by the VHDL compilation schemes.

The barrier component takes as input a collection of 1-bit control signals, and generates a 1-bit control output. The purpose of this component is to insure that a collection of data signals – each associated with a control input – *all* contain valid data before control propagates to a following sequential component.

Example: A call to a two-argument function should only be requested, by pulsing the control input on the associated FunBlock component, when both of the argument signals are valid. Consider the following fragment:

```
(define (f (x Int) (y Int) Int)
  (+ x y))

(define (inc (x Int) Int) (+ x 1))

(define (g (x Int) Int)
  (f (inc x) x))
```

In the body of the function `g`, there is a call to `f`. The second argument to `y` is immediately valid upon a call to `g`, yet the first argument will necessarily be delayed because it first must pass through a call to `inc`, which will produce an arbitrary delay. Consequently, the request for the call to `f` within the body of `g` will be guarded by a barrier component.

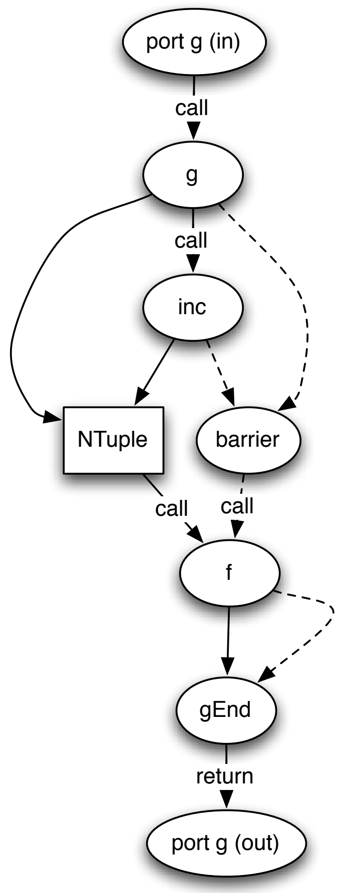

Figure 3.8: Barrier insertion example

Figure 3.8 shows the associated control and data flow graph. Note that the input to the function block `f` has data input from a `NTuple` node, which simply combines the two data input signals into a single data signal, and control input that that originates from a barrier node.

The barrier component is implemented as a collection of 1-bit registers, each connected to a control input signal. On reset, the registers output low. When the associated control input for a register goes high, the register outputs high. The output for each register is connected to an n-to-1 bit gate that calculates the logical `and` of all of its inputs. The 1-bit output of this gate is the control output for the barrier. Moreover, this output is attached to the register reset of each control register, resulting in a 1-cycle pulse when all of the registers have been latched high.

The CallLatch VHDL primitive simply stores the result of a function call. Because function blocks are shared resources, it is necessary to save the result of a call, as it may be invalidated by the function block servicing a subsequent call. The CallLatch primitive is implemented as an n-bit register, where n is the size of the result of the called function. Because the result register will introduce a 1-cycle delay on the function's result data output, the CallLatch component also contains a 1-bit register which delays the function block's control output associated with the call for one cycle.

Shared Blocks

The FunBlock, JumpBlock, MemBlock, and Service components all mediate access to a shared resource. The general structure of each of these components is the same – they all contain an arbiter which chooses one request, of possibly several active, to perform. The block then multiplexes the data input for the chosen request onto the shared circuitry. Once this circuitry has completed, indicated by its control output pulsing high, the block will propagate control indicating that the request has been serviced.

Figure 3.9 shows this behavior as a state machine. The block is initially in the `Wait` state, awaiting requests. Upon receiving a request, the block transitions to the `Execute` state, where control moves to the body of the block. Finally, in the `Finish` state, the block generates a control response for the scheduled request and transitions back to the `Wait` state.

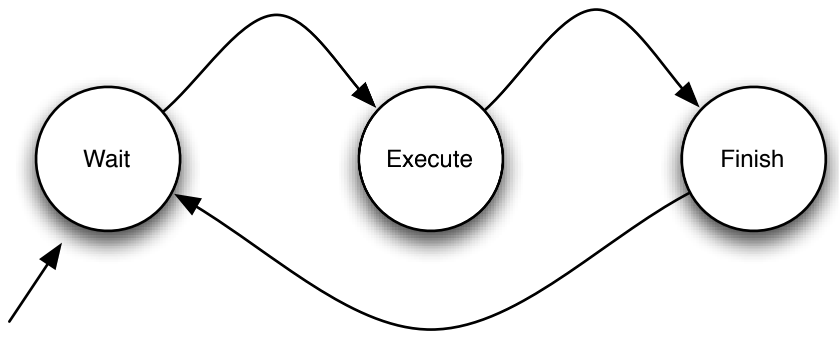

Figure 3.9: Shared Block State Machine

Figure 3.10 shows the shared block structure graphically. At the top is the arbiter component. In the top-left of the arbiter is a series of 1-bit registers, one for each incoming invocation. When an external entity issues a request to the block via a 1-bit control signal pulse, the appropriate register will latch the request. The output of these latches is taken collectively as a data input to a scheduler component.

The scheduler is complicated by the fact that call requests can arrive at any time, including that period of time when the block is executing the body of the shared resource. Consequently, the value produced by the request latch bank may be constantly changing. It is important that the data input to the body of the block to only change when a new service has been scheduled.

To implement this behavior, the scheduler combines three smaller elements. The first is an n-bit `or`, which produces a high output when any of the request registers are high. The output of the `or` is connected to a the second sub-component which indicates whether the body of the function is active. Initially low, the `active` component pulses high for one cycle upon seeing a high output from the `or`; it will then remain low, regardless of the `or` input, until the block's body control output is high, resetting the `active` component. The output of the `active` component is used both to drive the block body's control input, as well as the third `enc` priority encoder sub-component. It is the `enc` that determines which pending request is to be serviced. The output of this `enc` sub-component is used as input both to an argument multiplexer, `mux`, in the arbiter, as well as a control demultiplexer, found in the the block epilogue.

The argument multiplexer selects, based on the scheduler's data output, the correct argument data input to the block's body. This multiplexer component is implemented as *n* *r*-to-1 multiplexers, where *n* is the width of the data input for the body and *r* is the static number of request inputs to the block.

The control and data outputs for the arbiter are connected to the body's control and data inputs, respectively. The control output for the body is connected both to the reset input to the `act` component, indicating that the body invocation has completed, as well as to a control demultiplexer, or decoder. This `demux` component routes the control output to the site of the scheduled request, based upon the stored encoder output. Moreover, the `demux` control output is used as an input to the request register bank, clearing the request register.

Note that the data output of the body is not similarly demultiplexed or saved in a register in the block's epilogue. Rather, block assumes that this data output will be saved at the origin of the request. Because a shared block may service several requests consecutively, the data output for the block will *only* be guaranteed valid for the clock cycle the block's control output associated with the request is high. Compare this with the argument input for the block, which the shared block shared block primitives assume will remain valid for the indefinite time a request is issued until the request is completed, even though the request control input may only be high for a single clock cycle.

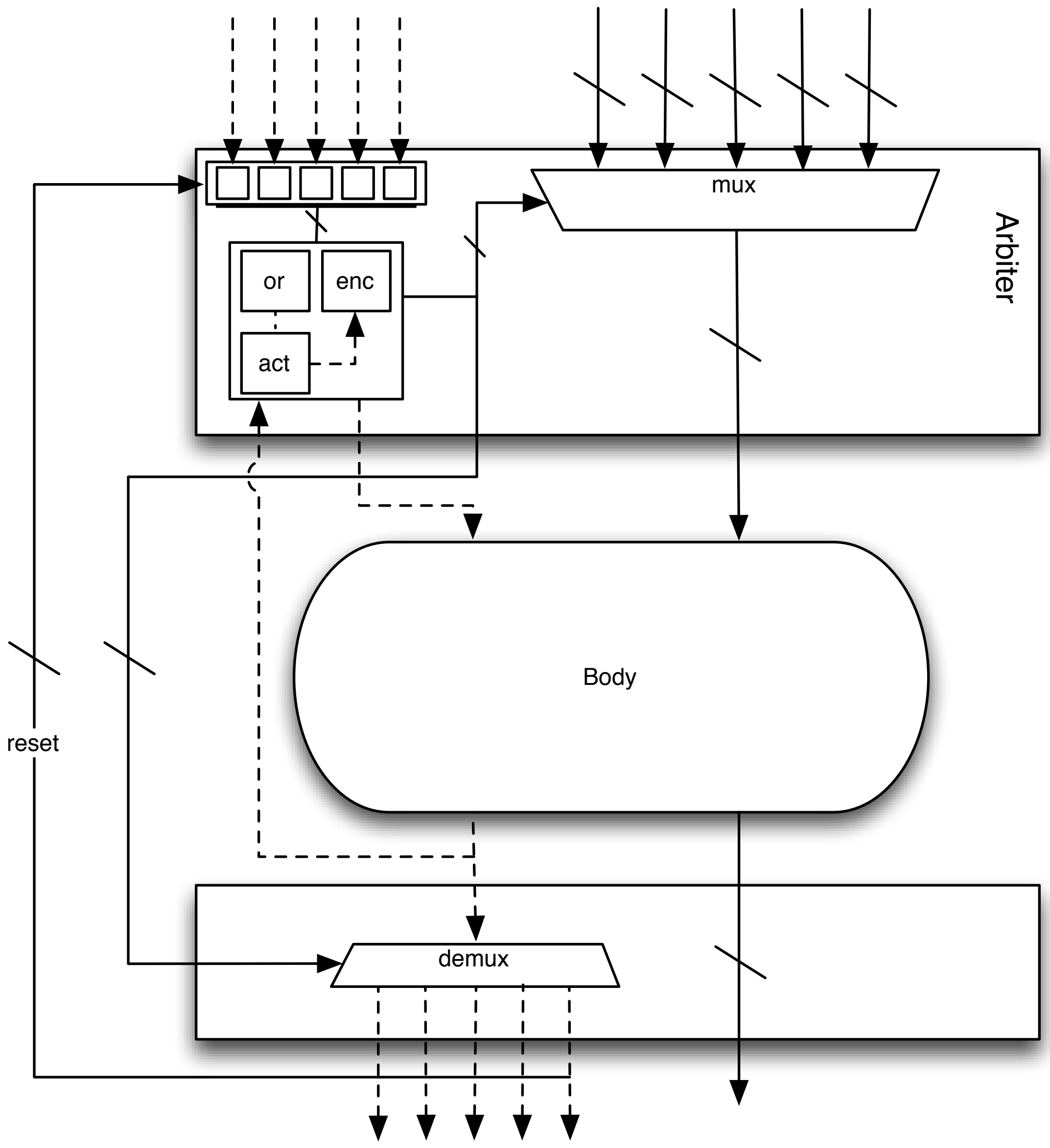

Figure 3.10: Shared Block Structure

This general architecture is used by all of the shared block structures, and mirrors that of the FunBlock component precisely. The other components, are variations on the theme, adding additional logic that implements the specific behavior exhibited by that component.

JumpBlock Oread functions that contain no recursive calls instantiate FunBlock components. However, if a function *does* include recursive calls, or *jumps*, the shared block structure must be modified to include logic to handle those recursive calls. This component insures that external calls are completed sequentially, implying that all recursive calls that result from an external call must complete before another external call is serviced. Consequently, the shared block structure for recursive functions, called a JumpBlock, includes an additional component that adds extra logic for handling those recursive calls.

This logic is similar to the arbiter logic of described in the general shared block architecture. However, this arbiter is slightly less complicated, as it is not necessary to handle re-entrant recursive calls. The Oread hardware compilation scheme requires that all recursive calls are tail calls, so only one can be active at a given time.

Figure 3.11 shows the additional components to support recursive calls in hardware functions. The main addition is on the right-hand side, where there is an additional scheduler component and argument multiplexer. These components are connected to control and data signals at call sites located in the body of the function. The scheduler component is simpler, because language restrictions disallow simultaneous jump requests. Therefore, the `active` register is unnecessary. However, the control protocol for Oread hardware compilation requires that the function body's arguments remain valid across an entire call. Because the multiplexer output in the jump-handled component can change as control passes through the body of the function, it is necessary to introduce a register, called `JumpReg` in the figure.

The `JumpReg` register latches the multiplexer output when a jump is requested. It produces three outputs. The first is a `jump_start` signal, which holds 1-cycle pulse that is connected to the body's control input. The other two outputs are connected to a multiplexer, `jump_mux` that selects between external call arguments and jump arguments to be used in the body. The `jump_sel` output of `JumpReg` is held high (selecting the jump arguments) until it is reset, which is indicated by the body output pulsing high.

MemBlock The Oread hardware compilation scheme represents a state monad instance with a MemBlock component. As with the JumpBlock component, it is a small modification of the general shared block structure. Figure 3.12 depicts the MemBlock component structurally.

At the top of the diagram is an arbiter. Because of monadic sequencing, only one request to a MemBlock component can be active at a time. Consequently, it is not necessary to latch requests. Moreover, the arbiter has two sets of data inputs. The first is for addresses, supplied by both **`get`** and **`put`**, and the second is for data, supplied only by **`put`** requests. Moreover, the type of a request, whether it is **`get`** or **`put`**, is encoded by the numbering of the request control input. Therefore, the scheduler component for the MemBlock contains a comparator, which determines if whether the `re` (for **`get`**) or `we` (for **`put`**) outputs are driven high.

Rather than logic generated by compiling an Oread expression, the body of the MemBlock component has an instantiated BRAM. Because the BRAM does not comply with the Oread control protocol, the MemBlock component uses the `we` and `re` control signals to provide the control output for the MemBlock body. However, as

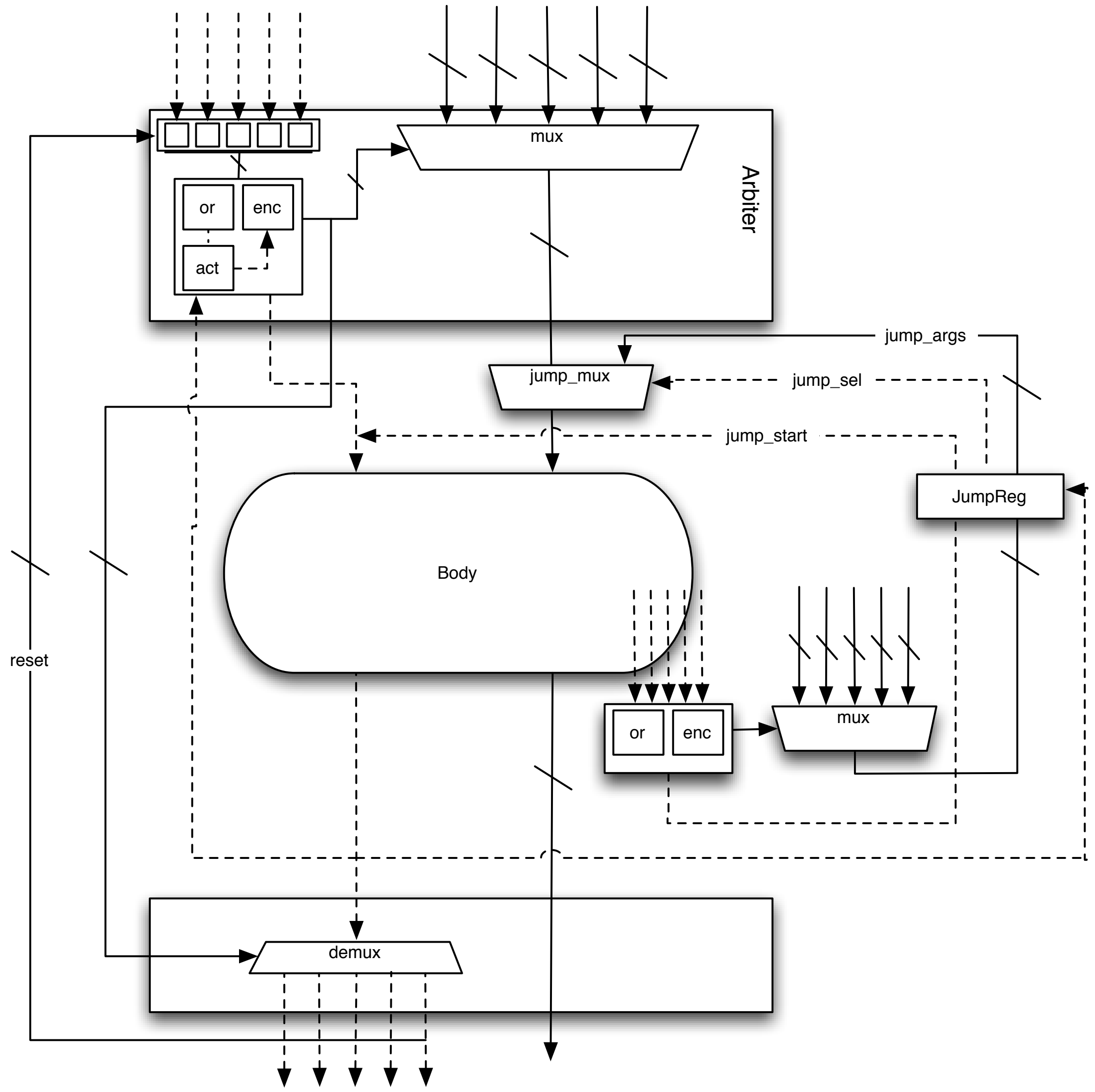

Figure 3.11: Jump Block Structure

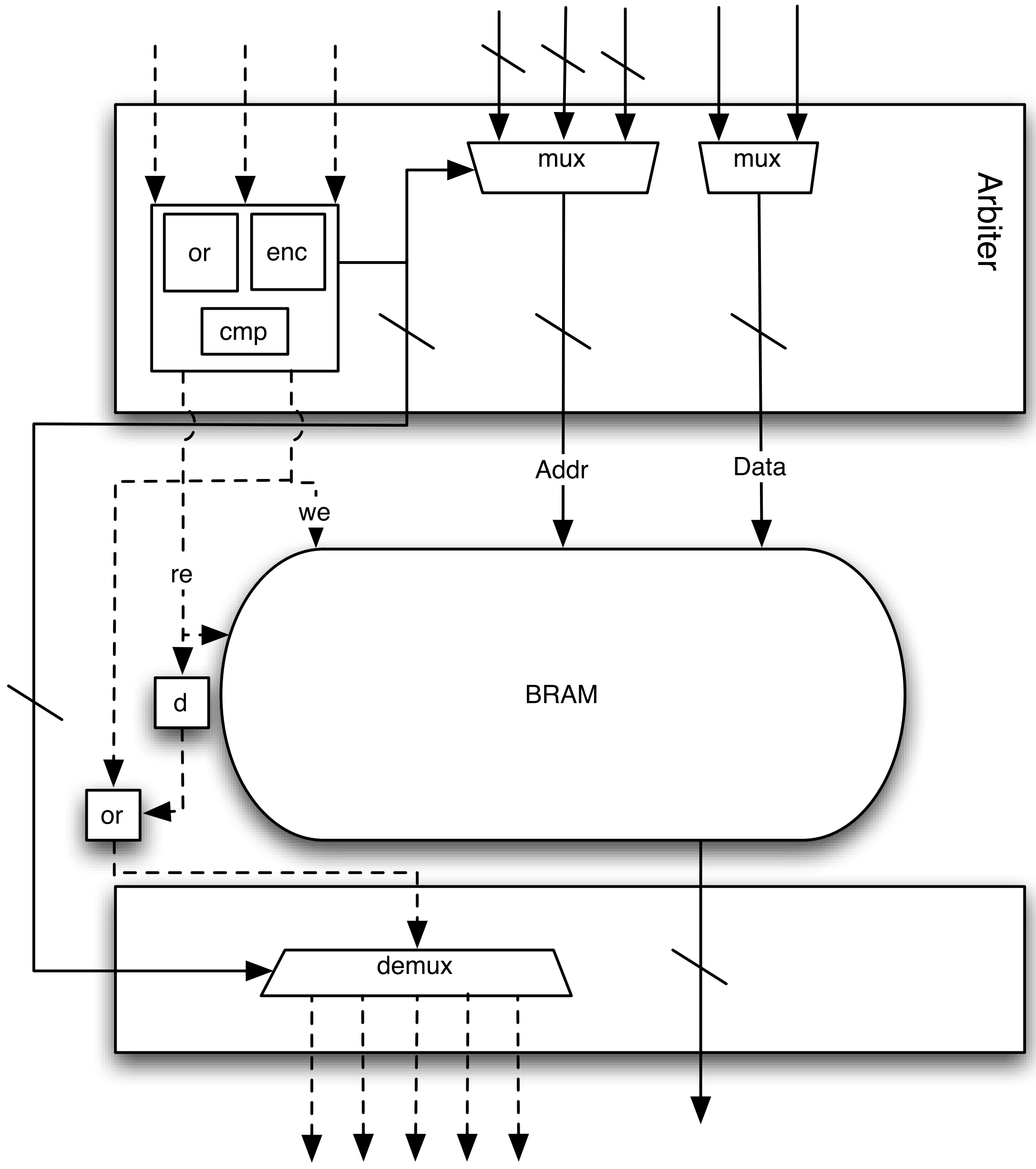

Figure 3.12: MemBlock Structure

reads from the BRAM require two clock cycles, it is necessary to introduce a delay to insure that a read has completed before control is propagated from the MemBlock.

The MemBlock component is, along with the use of 18-bit multipliers, the closest connection between the Oread hardware compilation scheme and the specific FPGA target used in its development. However, because of the limited reliance on low-level hardware properties in the compilation, instead relying on generic primitive blocks and the target-independent control protocol, it is possible to adapt these schemes to new target fabrics by modifying the primitive blocks to account for fabric-specific timing properties, with no change to the compilation schemes.

ServiceBlock The final shared block primitive implements the service element of the Oread concurrency structure, as described in section 2.1.3. Recall that an Oread service utilizes both a handler for processing request messages from connected threads and generating responses, and a scheduler that routes responses back to the appropriate thread. Both the handler and the scheduler that are used in a service are written in Oread, and therefore are compiled using the standard Oread hardware compilation scheme. Most importantly, these functions implement the Oread control protocol. The primary role of the ServiceBlock is to sequence the execution of the handler and scheduler functions.

Figure 3.13 shows the ServiceBlock component, expressed as a state machine. Much as the generic shared block, the ServiceBlock starts in the `Wait` state, awaiting external requests. When a request arrives, it then transitions into the `Handle` state, in which control is passed to the service's handler function. Following the completion of the handler, the ServiceBlock then transitions to the `Sched` state, where the scheduler function is invoked. In the diagram, this state has a transition both back to both a `Resp` state and back to the `Wait` state.

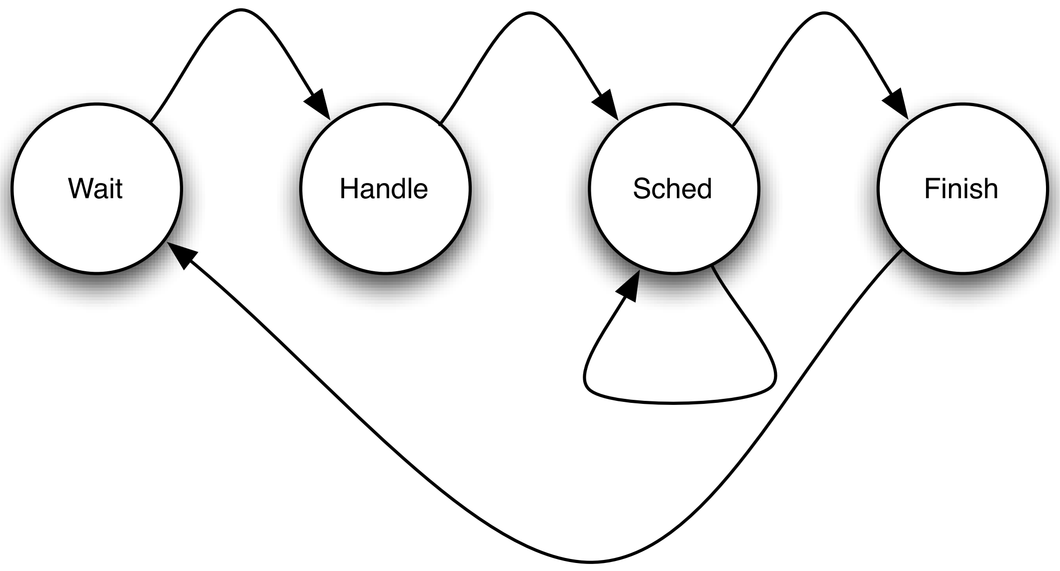

Figure 3.13: ServiceBlock State Machine

The transition the ServiceBlock takes is determined by the output of the scheduler, that indicates if there *is* a response ready for a blocked thread, and if so, the thread identifier and the response message. If there is no response ready, the ServiceBlock transitions back to the `Wait` state. However, if the scheduler indicates that there is a pending response, the ServiceBlock transitions into the `Resp` state, where the response is sent to the indicated thread. Note that this may be a *different* thread than the one that issued the request that forced the transition from the `Wait` state. After routing this response to the appropriate thread, the ServiceBlock transitions back to the `Sched` state, to determine if more threads are ready.

Figure 3.14 shows the ServiceBlock component structurally. As with the other shared blocks, it contains an arbiter that selects between active requests. There are two major differences in this arbiter, however. The first is that a request register is reset as soon as it is scheduled, as it is the role of the handler to ensure that the request is eventually satisfied. Second is that the encoder output is routed connected to both the argument mux and the handler.

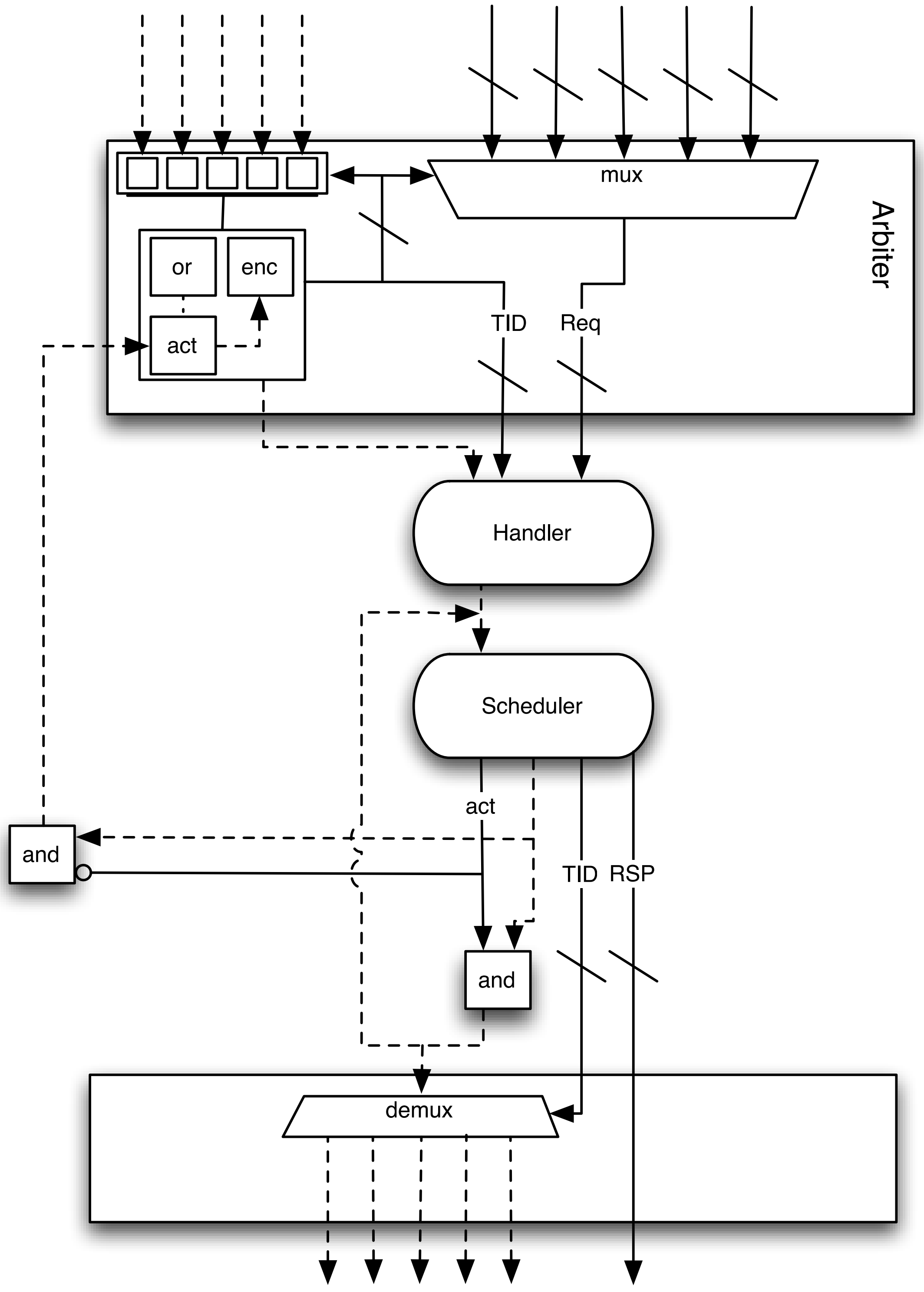

Figure 3.14: ServiceBlock Structure

After control has passed through the handler, which produces no data output, it then enters the scheduler, which consumes no data input. The scheduler has three data outputs: an `act` signal that indicates whether a valid response was generated, a `RSP` that holds a response message, and a `TID` that identifies the thread the response is intended for. The response data signal is exposed as the data output for the ServiceBlock. The thread identity is used as a selection input to the control demultiplexer at the bottom of the diagram. The control output of the scheduler is combined with the `act` signal to determine whether the Service should begin handling another request (indicated by the path from the scheduler to the arbiter) or invoke the scheduler again (indicated by the path back to the control input of the scheduler).

3.3 Summary

Oread is compiled to VHDL that can in turn be synthesized to a hardware implementation fabric. The compilation scheme relies on a simple control protocol that associates each data signal with a 1-bit control signal. The control signal is used to indicate that the associated data signal contains a valid value. Conceptually, this protocol is asynchronous, as it does not rely on precise timing to insure that the generated circuit functions correctly. In practice, the compilation scheme relies on a collection of primitive VHDL components that use synchronous logic to implement the control protocol.

This is not, however, a fundamental limitation of the protocol. The sequential primitive components, consisting of the sequential control barrier and sequential data shared block components described in this chapter can be adapted (Zhuang et al. 2002) to support multiple clock-domain implementations, which in turn enable globally-asynchronous locally-synchronous (GALS) systems (Chapiro 1985).

The VHDL primitive blocks that the Oread hardware compilation scheme utilize can be categorized in two dimensions: combinational vs. sequential, and data vs. control. The combinational data primitives, used by all of the functional subset of the language with the exception of function calls and case expressions, are directly supported in VHDL. Likewise, the combinational control components consist of wrappers around basic multiplexer and demultiplexer circuits. The sequential control barrier component, used to enforce the Oread control protocol, is implemented as a collection of 1-bit registers and an n-bit and gate. Finally, the sequential data primitives include the CallLatch component, implemented as a pair of registers, and the shared block primitives. Each shared block primitive consists of an arbiter circuit that mediates access to the shared resource, a body, and a small amount of control logic.

Chapter 4

Program Transformation and System Generation

A restricted subset of Oread can be effectively compiled into synthesizable VHDL. The ability to compile the complete Oread language into hardware requires extensions in two directions. First, source-to-source transformations are used to turn an otherwise unsynthesizable Oread program into the restricted language subset. Second, the Oread `configuration` construct is used to synthesize a complete system consisting of a number of threads and services.

4.1 Oread program Transformation

When generating a hardware implementation of an Oread program, a key limitation of the target fabric is that the program structure and resource usage must be statically determinable. It is necessary to reconcile the expressiveness of Oread with the fabric limitations. These issues can be addressed at the program level – rather than in tool-specific logic – by performing source-to-source transformations that transform one Oread program into a second program which defines the same functionality but is more amenable to a specific compilation target. Higher-order functions and dynamic allocation of memory resources present two challenges in compiling Oread programs to hardware that can be addressed using program transformations.

4.1.1 Defunctionalization

In a netlist generated from a Oread program, each top-level function is compiled into a separate shared function block. Calls are performed by placing argument values on a data signal and pulsing a control signal indicating a call request. However, it is unclear how a call would pass a function as an argument. Conversely, from within a higher-order function, the application of a function argument should make a request to the appropriate shared function block. A difficulty arises when the higher-order function is called with different values for the function argument, because the compilation scheme cannot statically determine to which function the request and arguments should be routed.

Example: The following program has an increment and decrement function and an apply* wrapper function around each. Figure 4.1 shows the resulting netlists for `applyinc` and `applydec`.

```
(define (inc (x Int) Int)
        (+ x 1))
(define (dec (x Int) Int)
        (- x 1))
(define (applyinc (x Int) Int)
  (inc x))
(define (applydec (x Int) Int)
  (dec x))
```

The apply* functions can be captured as a single higher-order function **`apply`** function. The parameter `f` to **`apply`** determines which function is called.

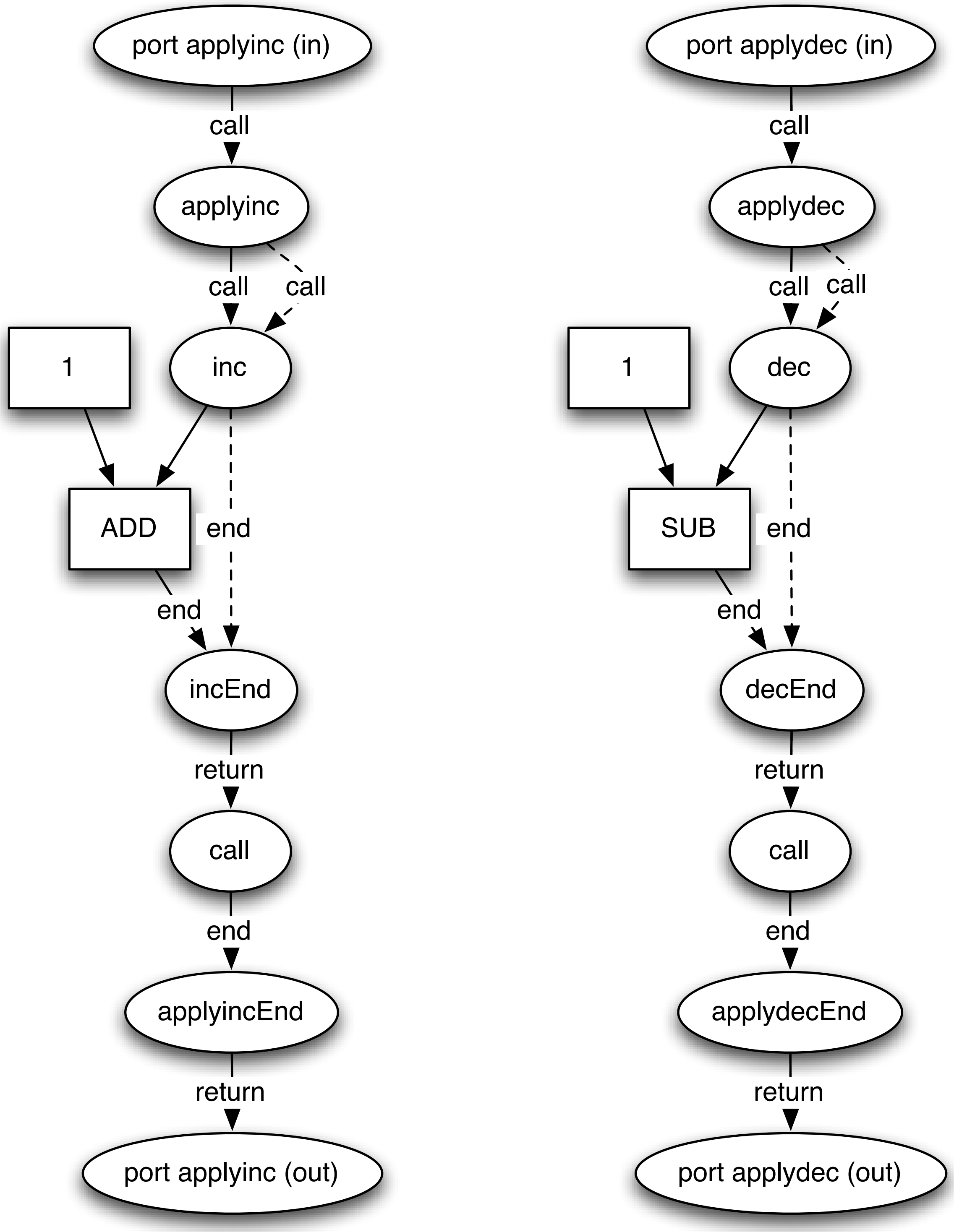

Figure 4.1: `applyinc` and `applydec`

```
(define (apply (f (-> Int Int)) (x Int) Int)
                                (f x))
```

Figure 4.2 shows a hypothetical control and data flow graph for the **apply** function, assuming that it is used in the definitions of applyinc and applydec. The important components are encircled and labeled Split and Join, respectively. Within the Split portion, the calls to inc and dec are performed in either-or fashion: which of the calls is performed depends upon the function parameter f of **apply**. Similarly, in the Join portion, one or the other call returns is latched, as dictated by the f parameter.

This graph resembles that generated when an Oread **case** expression is compiled. In this context, rather than passing a function parameter as a first-class value, we use a first-order tag to label each function value in the program. At each application of a higher-order function, a case expression is generated that selects one of the possible targets, performs the call, and returns the result. This process of converting a higher-order program into a first-order program is well known in the programming language community as *defunctionalization* (Reynolds 1998; Danvy and Nielsen 2001; Hutton and Wright 2006). It is a whole-program transformation, as it requires knowledge of every possible instantiation of a higher-order parameter.

Example: The **apply** function described above can be converted into a first-order function using defunctionalization. Given the following higher-order program:

```
(define (apply (f (-> Int Int)) (x Int) Int)
                                (f x))
(define (applyinc (x Int) Int)
  (apply inc x))
(define (applydec (x Int) Int)
  (apply dec x))
```

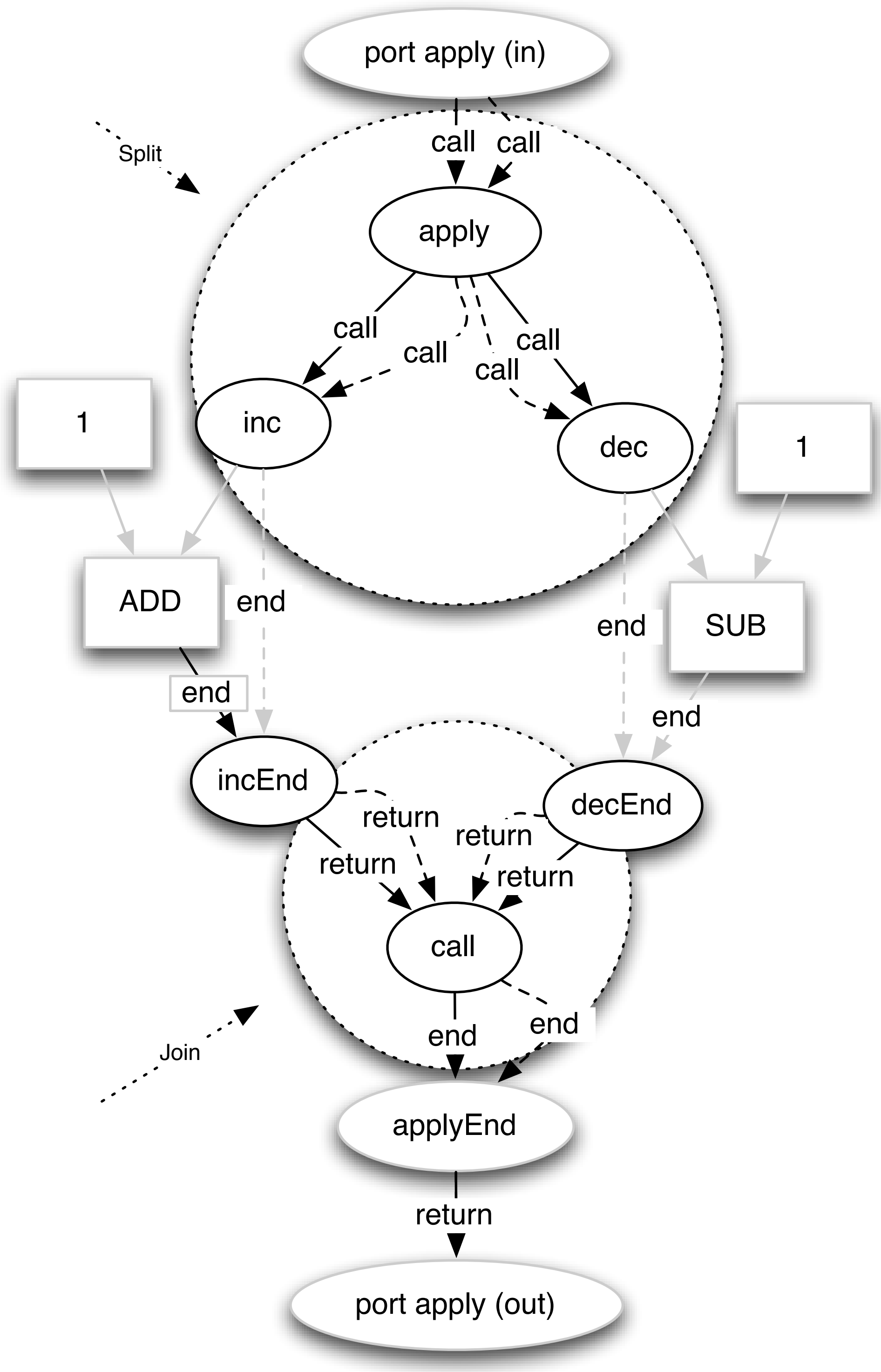

Figure 4.2: Hypothetical **`apply`** graph

Defunctionalization produces the following first-order program.

```
(data FOTY_Int2_Int (FOTY_Int2_Int_C1)
                    (FOTY_Int2_Int_C0))

(define (dispatchFOTY_Int2_Int (var0 FOTY_Int2_Int) (var1 Int) Int)
 (case var0
  ((FOTY_Int2_Int_C1) (dec var1))
  ((FOTY_Int2_Int_C0) (inc var1))))

(define (apply (f FOTY_Int2_Int) (x Int) Int)
 (dispatchFOTY_Int2_Int f x))

(define (applyinc (x Int) Int)
 (apply FOTY_Int2_Int_C0 x))
(define (applydec (x Int) Int)
 (apply FOTY_Int2_Int_C1 x))
```

The defunctionalized program includes a data type `FOTY_Int2_Int`, representing the First Order TYpe `(-> Int Int)`, which is the type of the parameter `f` in the original **`apply`** function. Each call to **`apply`** function results in a new constructor. `FOTY_Int2_Int_C0` and `FOTY_Int2_Int_C1` respectively correspond to calls to **`apply`** with `inc` and `dec`. The connection between the `FOTY_Int2_Int` constructors and the original functions can be found in the `dispatchFOTY_Int2_Int` function. This dispatch function includes a case expression that dispatches a call to the original function, based on the `FOTY_Int2_Int` encoded parameter.

Within the body of the defunctionalized **`apply`** function, the original call to the functional parameter `f` has been replaced with a call to the `dispatchFOTY_Int2_Int` function associated with the defunctionalized `FOTY_Int2_Int` type. Finally, for each call to the defunctionalized **`apply`**, the functional argument is replaced with its first-order `FOTY_Int2_Int` encoding, as found in the defunctionalized `applyinc` and `applydec`. Figure 4.3 shows the resulting netlist.

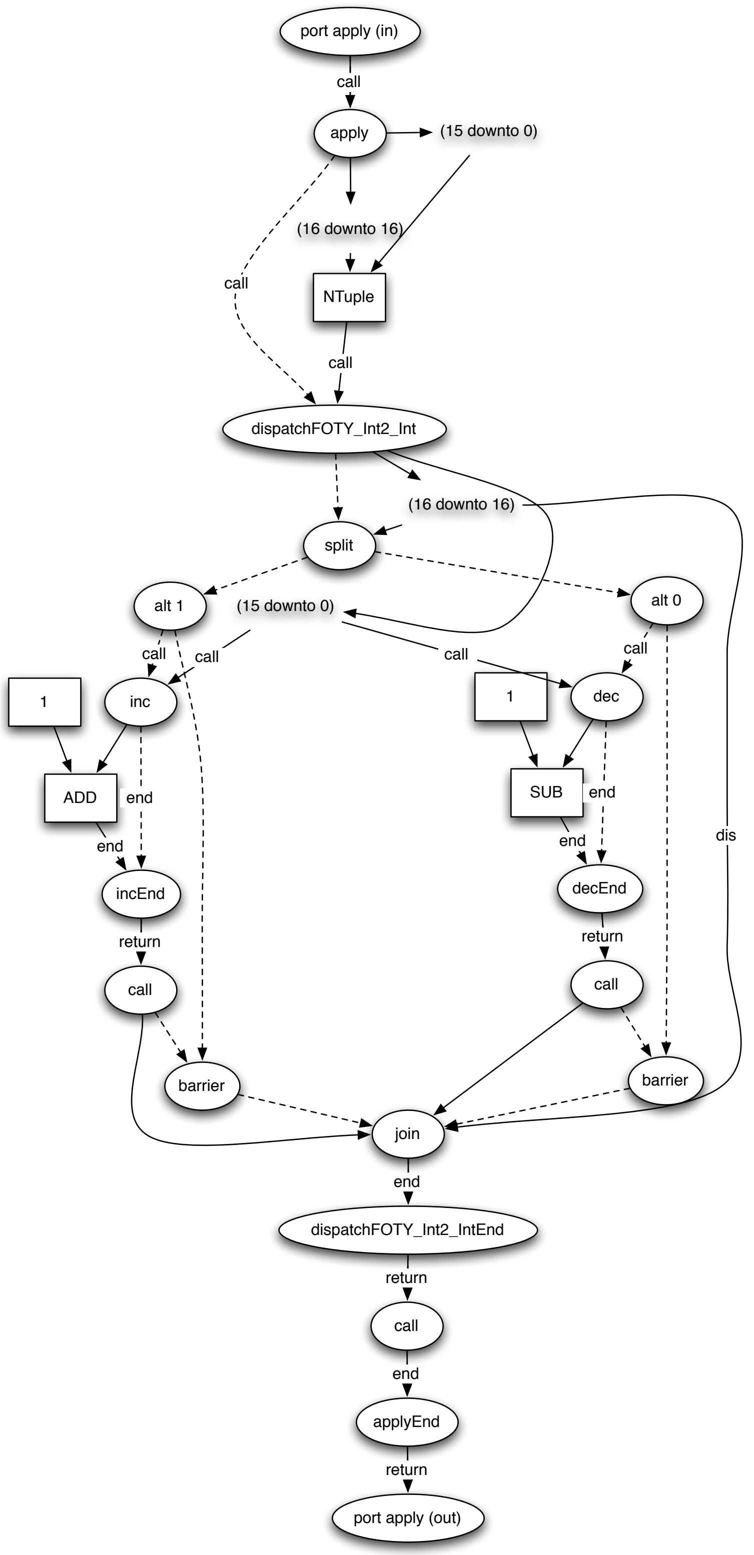

Figure 4.3: Defunctionalized **`apply`**

The defunctionalization transformation is performed in a series of steps by transforming the entire program. First, for each type `(-> T1 ... Tn R)` that occurs as the type of a parameter in a higher-order function, create a data type for the first-order representation of the functional type. Assume a mapping `type_encode` that for each Oread type, `ty`, produces a name `TyEnc`. Second, create a top-level function `dispatch_TyEnc`. Given a function type `(-> T1 ... Tn R)`, the corresponding dispatch function `dispatch_TyEnc` will have the type `(-> TyEnc T1 ... Tn R)`. The body of this function is a case expression, discriminated on the first parameter.

Third, for each call to a higher-order function `(f e1 e2 e3)`, replace each functional parameter to the call with a new constructor for the associated data type. Assume a mapping `fun_encode` that for each functional Oread type, `ty`, and expression `e`, produces a name `EFunEnc`. Using the `fun_encode`, add the constructor `EFunEnc` to the type `TyEnc` and add a case alternative `((EFunEnc) e)` to the `dispatch_TyEnc` function. For example, if in the expression `(f e1 e2 e3)` the arguments `e1` and `e3` have functional type, then the defunctionalized form of the expression is
`(f E1FunEnc e2 E3FunEnc)`. Within a higher-order function, the function-argument transformation should not be performed on arguments to function applications which themselves are higher-order parameters to the calling function.

Finally, within a higher-order function, replace each application of a higher-order function parameter`f` of the form `(f e1 e2 e3)`, with a call to the `dispatch_TyEnc` function, resulting in the defunctionalized expression `(dispatch_FTyEnc e1 e2 e3)`.

The defunctionalization transformation allows a higher-order program to be transformed into a first-order program, a transformation that is critical to getting an Oread program to compile to hardware implementations. However, the transformation as described suffers in that it cannot correctly eliminate closures: functions which capture a portion of their lexical environment.

Example: Consider the `addList` function below that adds a value `v` to each element in a list. The function constructs an anonymous function which captures the value `v` that it then passes to the **`map`** function.

```
(define (map (f (-> Int Int)) (x (List Int)) (List Int))
   (case x
    ((Null) (Null))
    ((Cons a b) (Cons (f a) (map f b)))))

(define (addList (v Int) (l (List Int)) (List Int))
        (map (lambda ((k Int)) Int (+ k v)) l))
```

In the defunctionalization transform defined above, the captured variable `v` is lost. The call to **`map`** will result in a constructor definition representing the anonymous function, but within the generated `dispatch` function, `v` appears free. Using the approach outlined above, the incorrect defunctionalized program is shown below.

```
(data FOTY_Int2_Int (FOTY_Int2_Int_C0))

(define (dispatchFOTY_Int2_Int (var0 FOTY_Int2_Int) (var1 Int) Int)
 (case var0
  ((FOTY_Int2_Int_C0) (+ var1 v)))) ;; error -- v is free

(define (map (f FOTY_Int2_Int) (x (List Int)) (List Int))
 (case x
  ((Null) Null)
  ((Cons a b) (Cons (dispatchFOTY_Int2_Int f a) (map f b)))))

(define (addList (v Int) (l (List Int)) (List Int))
 (map FOTY_Int2_Int_C0 l)) ;; error -- v is lost
```

The traditional approach to dealing with closures is to perform lambda-lifting (Johnsson 1985), which transforms a closure into a top-level function which contains extra arguments, one for each free variable. A similar technique, closure conversion, packages up the free variables captured by a closure into a record structure, and then adds that closure structure as an extra argument to the function. A small modification to the defunctionalization algorithm subsumes both of these techniques.

In the extended defunctionalization algorithm, the method of generating constructors encoding is changed so that it accounts for closure creation. First, this transformation requires a previous analysis pass that calculates, at each program point, the names and types of free variables. Then, when defining a data constructor corresponding with a functional parameter, these free variables are added as constructor arguments. Second, the pattern match in the `dispatch` function case expression introduces bindings for the new constructor argument.

Example: Returning to the `addList` function above, the previous method does not capture the variable `v`. To accomplish this, the `FOTY_Int2_Int_C0` constructor has an `Int` parameter. Similarly, the deconstruction of the `FOTY_Int2_Int_C0` introduces a pattern variable, `v`.

```
(data FOTY_Int2_Int (FOTY_Int2_Int_C0 Int))

(define (dispatchFOTY_Int2_Int (var0 FOTY_Int2_Int) (var1 Int) Int)
 (case var0
  ((FOTY_Int2_Int_C0 v) (+ var1 v))))
```

The final step of the defunctionalization transform is to replace functional parameters to higher-order functions with the first-order encoding. Because the data constructor encoding now account for free variables, it is necessary to supply those values when building the first-order value. This extension is trivial, as the free variables for that functional parameter have already been calculated – they simply need to be passed to the constructor.

Example: Continuing the previous example, the anonymous function parameter to the call to **`map`** in `addList` function has been replaced with the data constructor `FOTY_Int2_Int_C0`, which is applied to the free variable `v`.

```
(define (addList (v Int) (l (List Int)) (List Int))
 (map (FOTY_Int2_Int_C0 v) l))
```

The augmented defunctionalization transformation allows a programmer to utilize standard functional programming techniques, including using higher-order functions to capture common computation patterns, to define programs and then automatically transform those programs into a form capable of being synthesized. This ability is crucial in the design of multi-target programs, as it allows a system implementer to succinctly capture the basic functional structure of a system component, then use source-to-source transformations to allow the compiled representation of the program to take advantage of the computational resources of the target fabric, a primary motivation of this thesis.

4.1.2 Static Data Allocation

The defunctionalization transform handles one of the restrictions of the synthesizable subset of Oread by converting control and data flow of a program into a statically structure. However, the transform does not eliminate the second restriction, that all resource usage be statically determined. The approach to deal with this restriction is to perform transformations on general – unbounded – programs to convert them into bounded versions.

Unlike the defunctionalization transformation, a total transform over whole programs guaranteed to result in first-order program, the data representation transformations are ad hoc and not guaranteed to result in programs that satisfy resource-bounds.

Nevertheless, data transformations allow an Oread programmer to define programs that capitalize on common functional programming idioms and then *manually* transform them into synthesizable programs.

Example: Recall the **`map`** function that applies a function (supplied as a parameter) on every element of a list and returns the resulting list. The **`map`** function uses the recursive `List` data type; the recursive of this type form allows (finite) lists of any size to be represented.

```
(data List [a] (Cons a (List a))
               (Null))

(define (map (f (-> Int Int)) (x (List Int)) (List Int))
 (case x
  ((Null) Null)
  ((Cons a b) (Cons (f a) (map f b)))))
```

It is not possible to statically determine the size of a given value of type `List`, since for any given size, a list one element bigger can be constructed by `Cons`ing on an additional element. However, for a *fixed* size list, an equivalent program can be constructed which uses a homogeneous product representation of lists. For example, a three-element list is represented by the type `(* a a a)`. Given the length restriction, the recursively defined list `(Cons 1 (Cons 2 (Cons 3 Null)))` is equivalent to the fixed-size product `(tuple 1 2 3)`.

Unfortunately, it is not possible to statically type recursive functions *n*-ary products without a type system that is considerably more sophisticated than the Oread static semantics, such as dependent type systems (Barendregt 1992; Augustsson 1999). A dependent type system allows the types to depend on values. This eliminates the typical stratification between types – which are an static abstraction of dynamic values – and the values that types represent. While a dependent type system is strictly more expressive than the Oread type system, that expressiveness comes at a cost. Since statically calculated types can depend on values, and those values can be produced by any well-typed program, then in the presence of general recursion the type checker may fail to terminate. More restricted dependent type systems, such as Dependent ML (Xi 1998) allow for computation at the type level, but restrict the computation to insure that type checking terminates.

An alternative approach to dependently typed languages is to use a staged language, such as MetaOCaml (Taha 1999) and Template Haskell (Sheard and Peyton Jones 2002). These languages have constructs for *meta-programming* – programs that generate programs. In a staged programming language, the stages of a program are clearly delineated. A two-staged language separates a static stage, which when executed yields a new program which is then executed as the dynamic stage. Moreover, the static stage cannot depend on values produced by the dynamic stage. To insure that only well-typed programs are ever executed, a generated program is type checked following each stage.

In MetaOCaml and Template Haskell, stages are explicitly delineated via brackets for constructing later stage programs and escape for moving values from an earlier stage into a later stage. In contrast, partial evaluation (Jones et al. 1993) systems attempt to automatically discriminate between static and dynamic values and to perform all possible static reductions, generating a purely dynamic *residual* program. A variant of partial evaluation, type-directed partial evaluation (TDPE) uses type information (Danvy 1996) to improve the inference of static and dynamic data, reducing the need for explicit *binding time improvements* to improve residualized program (Danvy et al. 1996).

The Oread tool set contains an implementation of TDPE, based on the work of Sheard (1997). The data transformations shown below use this work to perform transformations. Moreover, although the Oread language does not include support for explicit staging, the design of the Oread tool set using modular monadic semantics allows the language to be extended with staging annotations relatively easily (Weaver et al. 2007). Using this partial evaluation capability, a programmer can manually transform data representations in Oread programs.

Example: Consider the previous example of converting a function that performs list manipulation functions into a synthesizable function that uses fixed-size products. Given the `incMap` function defined below:

```
(define (inc (x Int) Int)
  (+ x 1))

(define (map (f (-> Int Int)) (x (List Int)) (List Int))
   (case x
    ((Null) (Null))
    ((Cons a b) (Cons (f a) (map f b)))))

(define (incMap (x (List Int)) (List Int))
  (map inc x)
```

An equivalent program can be constructed using partial evaluation by manually constructing a list value that has elements that are the projection from a product representation. For example, using the equivalence between the list
`Cons a (Cons b (Cons c Null))` and the product `(tuple a b c)`, specialization of the application `(map inc (Cons (prj 0 p) (Cons (prj 1 p) (Cons (prj 2 p) Null))))` – where `p` is a synonym for `(tuple a b c)` – yields the following residual value. The applications of `inc` are reduced, but the additions remain, as they depend on the dynamic free variable `p`.

```
(Cons (+ (prj 0 p) 1) (Cons (+ (prj 1 p) 1) (Cons (+ (prj 2 p) 1) Null)))
```

The variable p appears free both in the input to the specializer and the residual program. If the components of p were defined, then the specializer would statically reduce the the additions.

```
(let ((p (tuple 1 2 3)))
  (map inc (Cons (prj 0 p) (Cons (prj 1 p) (Cons (prj 2 p) Null)))))
```

yields the residual expression:

```
(Cons 2 (Cons 3 (Cons 4 Null)))
```

Finally, the partial evaluation capability can be used to write functions that automatically convert between data representations. The lstToPrd3 function, defined below, takes a function that transforms lists and yields a function that transforms 3-ary homogeneous products. This function requires that the input list transformation function f preserve the list length.

```
(define (lstToPrd3 (f (-> (List a) (List b))) (p (* a a a)) (* b b b))
  (let ((a (prj 0 p))
        (b (prj 1 p))
        (c (prj 2 p)))
    (case (f (Cons a (Cons b (Cons c Null))))
       ((Cons x xs) (case xs
         ((Cons y ys) (case ys
           ((Cons z zs) (tuple x y z)))))))))
```

The prodMap function uses lstToPrd3 to define a 3-ary product map.

```
(define (prodMap (f (-> Int Int)) (p (* Int Int Int)) (* Int Int Int))
 (lstToPrd3 (lambda ((l (List Int))) (List Int) (map f l)) p))
```

Specializing this function yields the following residual definition, with the extraneous list functions statically eliminated:

```
(define (prodMap (f (-> Int Int)) (p (* Int Int Int)) (* Int Int Int))
   (tuple (f (prj 0 p)) (f (prj 1 p)) (f (prj 2 p))))
```

The `lstToPrd3` function defined above only works for lists of length 3. A similar transformation supporting lists of length 4, requires writing a `lstToPrd4` function. Staged meta-programming languages, such as MetaOCaml and Template Haskell, allow the definition of a general `lstToPrdN` function that subsumes the specific functions `lstToPrd1,lstToPrd2`, etc.

Currently, the support for data representation transformations in Oread is limited by tool set support. Recent work Gill and Hutton (2008) places the ad hoc transformations described informally above into a formal framework that supports provable sound transformation of general data transformations.

4.1.3 Static Monadic Instance Resolution

The program transformations in the previous subsections focus on transforming the pure subset of Oread to a synthesizable form. However, the presence of monad instance parameters as Oread function parameters presents similar synthesis challenges. In an invocation of a monadic morphism – **get**, **put**, and **signal** – the monad instance reference appears to the control and data flow graph generation scheme to be the invocation of a higher-order function with a functional parameter.

Example: Consider the following state-monadic function, which zeros the value at a given address in a memory, which is identified as a parameter `@st` to `zero`.

```
(define (zero (x Int) (monad ((@st (State Int Int))) Unit))
   (put @st x 0))
```

If `zero` is used with two different state monad instances, then the body of the `zero` function must route the invocation of the **put** request to the correct instantiation of `@st`.

If `zero` is compiled into a single shared function block, then the `@st` parameter is not statically determined.

```
  (define (t (x Int)
       (monad (@st1 (State Int Int)) (@st2 (State Int Int)) Unit))
     (do (zero [@st1])
         (zero [@st2])))
```

One potential solution to this problem is to duplicate the `zero` function block for each possible monad parameter instantiation, and the original calls replaced with a call to the duplicated block. There is only one possible monad instance target for any given monad morphism, so the data and control flow graph can be statically generated.

```
(define (zero1 (x Int) (monad ((@st (State Int Int))) Unit))
  (put @st x 0))
(define (zero2 (x Int) (monad ((@st (State Int Int))) Unit))
  (put @st x 0))

(define (t (x Int)
    (monad (@st1 (State Int Int)) (@st2 (State Int Int)) Unit))
  (do (zero1 [@st1])
      (zero2 [@st2])))
```

Example: The following top-level configuration makes `t` into a thread. The @ all-caps convention is used signify that @STATE1 and @STATE2 are specific named instances. In the resulting program, the monad instance parameters appearing as targets in monadic constructs are replaced with named instances.

```
(configuration example
  (memory @STATE2 Int Int)
  (memory @STATE1 Int Int)
  (thread (t 9 [@STATE1 @STATE2])))

(define (zero1 (x Int) (monad ((@st (State Int Int))) Unit))
  (put @STATE1 x  0))

(define (zero2 (x Int) (monad ((@st (State Int Int))) Unit))
  (put @STATE2 x 0))

(define (t (x Int) (monad (@st1 (State Int Int)) (@st2 (State Int Int)) Unit))
  (do (zero1 [@STATE1])
      (zero2 [@STATE2])))
```

Performing this expansion can lead to an explosion in generated circuit size. A copy of a function block must be generated for each *combination* of monad parameters. Moreover, the expansion is recursive, so all *called* functions from a duplicated call must also be duplicated. Finally, because any invocation of a monadic operation is within a sequential monadic thread of control, there is no parallelism gain from duplicating code, because only one of the two invocations are ever active at the same time.

If the `t` function defined above were called with two different combinations of monadic instances, then two copies of `t` will be generated and, in turn, four copies of `zero` will be generated.

This approach runs counter to the philosophy of Oread, which uses shared function blocks to control the number of computational resources. This limitation becomes an issue when a function that uses a large amount of computational resources is called with multiple instances. In this case, it may not be possible to duplicate the function definition without exceeding the area constraints of the hardware. Instead, we want to use the same surrounding logic, and route the get/put invocations to different instances.

An alternative approach is to utilize the observation that monad instance parameters are similar to function parameters, and reuse the defunctionalization technique to make the targets of all monadic operations statically identifiable. Much as defunctionalization is a whole-program transformation that relies on knowledge of all call sites for a higher-order function, the monad instance transformation requires static knowledge of all possible monad instances. The Oread `configuration` construct provides this information, including a list of all memories, services, and threads that utilize those monadic instances.

The transformation inserts the appropriate logic in monadic constructs to route requests to the correct logic. As with defunctionalization, an advantage of adding this logic via a source-to-source transformation is that it keeps more of the synthesis process internal to Oread, rather than relying on fabric-specific support. This makes retargeting a new HDL or netlist format simpler, as the routing logic is implemented in Oread, rather than for each netlist target.

The steps in the static monad instance transformation are similar to defunctionalization. First, for each monad instance type that that occurs as a function parameter, generate a new data type to represent encodings of instances of that type.

Example: The following program has monadic instance parameters at two types – `(state Int Int)` and `(state Int Bool)`.

```
(define (zero (addr Int) (monad [(@st (state Int Int))] Int))
  (put @st addr 0))

(define (negateState (addr Int) (monad [(@st (state Int Bool))] Bool))
  (do (v <- (get @st addr))
    (put @st addr (not v))))

(define (xorState (a1 Int) (a2 Int) (a3 Int)
          (monad [(@st (state Int Bool))] Bool))
  (do (v1 <- (get @st a1))
      (v2 <- (get @st a2))
      (put @st a3 (xor v1 v2))))
```

This generates two data types representing the two types.

```
(data StateIntInt (...))
(data StateIntBool (...))
```

The second step uses a configuration declaration to populate the generated data types with nullary constructors, one for each memory in the configuration declaration.

Example: The configuration `bools` defines three memories. The resulting data declarations are shown below the configuration.

```
(configuration bools
 (memory @intmem Int Int)
 (memory @bmem1 Int Bool)
 (memory @bmem2 Int Bool)
 (thread (zero 1 [@intmem]))
 (thread (negateState 0 [@bmem1]))
 (thread (andState 0 1 2 [@bmem2])))

(data StateIntInt (IntMem))
(data StateIntBool (BMem1) (BMem2))
```

The third step is to generate dispatch functions for each instance encoding data type. For those representing state instances, there are two functions generated - one for **`get`** and one for **`put`**. On the other hand, the reactive instance encodings require a single dispatch function. Each dispatch function will perform a case analysis on the input instance encoding, then invoke the appropriate monad morphism on that function. The dispatch functions are named after the monad morphism that they invoke.

Example: Continuing the `bools` example from above, the following dispatch functions are generated:

```
(define (getStateIntInt (enc StateIntInt) (addr Int) (monad Int))
  (case enc
    ((IntMem) (get @intmem addr))))

(define (putStateIntInt (enc StateIntInt) (addr Int) (val Int) (monad Int))
  (case enc
    ((IntMem) (put @intmem addr val))))

(define (getStateIntInt (enc StateIntBool) (addr Bool) (monad Int))
  (case enc
    ((BMEM1) (get @bmem1 addr))
    ((BMEM2) (get @bmem2 addr))))

(define (putStateIntInt (enc StateIntBool) (addr Int) (val Bool)
           (monad Int))
  (case enc
    ((BMEM1) (put @bmem1 addr val))
    ((BMEM2) (put @bmem2 addr val))))
```

The fourth and final step is to replace occurrences of monad constructs with calls to the appropriate dispatch function.

Example: The dispatch functions for the `bools` configuration are shown below.

```
(define (zero (addr Int) (monad [(@st (state Int Int))] Int))
  (putStateIntInt @st addr 0))

(define (negateState (addr Int) (monad [(@st (state Int Bool))] Bool))
  (do (v <- (getStateIntBool @st addr))
      (putStateIntBool @st addr (not v))))

(define (xorState (a1 Int) (a2 Int) (a3 Int)
          (monad [(@st (state Int Bool))] Bool))
  (do (v1 <- (getStateIntBool @st a1))
      (v2 <- (getStateIntBool @st a2))
      (putStateIntBool @st a3 (xor v1 v2))))
```

Creating the instance dispatch functions serializes control, since each dispatch function is a shared function block. This is not a limitation, however, because the transformation is only performed on monadic constructs, which are already serialized within the monadic encapsulation.

This transformation requires a modification of the static type rules for Oread. This is due to the monadic instance parameters having type `(state atype dtype)` or `(react msgtype)`. On the other hand, the dispatch functions expect a value of the data type encoding monadic instances. This is handled in the static semantics by adding a mapping from monadic instance type to encoding type. Moreover, the generated dispatch functions reference static instance names supplied by the configuration declaration, which must be added to the environment.

The Oread monadic constructs are parametrized over the target monad instance, which offers flexibility when generating hardware implementations of programs. Instances – whether state or reactive – can be localized, rather than requiring a single, global instance. This allows, for example, memory resources to be instantiated independently. However, this parametrization complicates the hardware compilation scheme, as it makes the program control flow statically indeterminate. The adapted defunctionalization algorithm allows the reuse of a standard program transformation to satisfy a critical constraint of the Oread hardware compilation schemes.

4.2 System Generation

The program transformations defined above, combined with the hardware compilation schemes described in chapter 3, establish the foundation for compiling complete Oread systems for hardware targets. The last remaining challenge is to define a system from a configuration. The Oread tool set does this by constructing a collection of VHDL entities, each corresponding to an Oread thread or service, then instantiating the collection together to form a complete system.

Figure 4.4 shows an example configuration, and figure 4.5 contains a graphical depiction of the associated system. In the diagram, threads correspond to squares and services correspond to diamonds. Memories are shown as circles within threads and services. Connections between threads and services correspond to channels.

The procedure for compiling a configuration, with an associated set of top-level bindings, to a VHDL system involves a variety of steps.

- Each thread is compiled into a VHDL block through the following steps:

    1. Calculate the set of live functions – those that are referenced, directly or indirectly, by the top-level thread function. All functions that are not live can be discarded.

    2. Perform the defunctionalization and static monad instance resolution transformations, as described in section 4.1.

    3. Instantiate a MemBlock component for each memory instance referenced in the top-level thread.

```
(configuration system
  (memory @m1 Int Int initM1)
  (memory @m2 Int Int initM2)
  (memory @m3 Int Int initM3)
  (memory @m4 Int Int initM4)
  (memory @m5 Int Int initM5)


  (service @s1 MType
    ; service handler
              handler1
    ; service scheduler
              sched1
              initSvc1 contSvc1)

  (service @s2 mType
    ; service handler
              handler2
    ; service scheduler
              sched2
              initSvc1 contSvc1)

   (thread (t1 [@m1 @s1]))
   (thread (t2 [@m2 @s1 @s2]))
   (thread (t3 [@m3 @s1 @s2])))
```

Figure 4.4: System Configuration

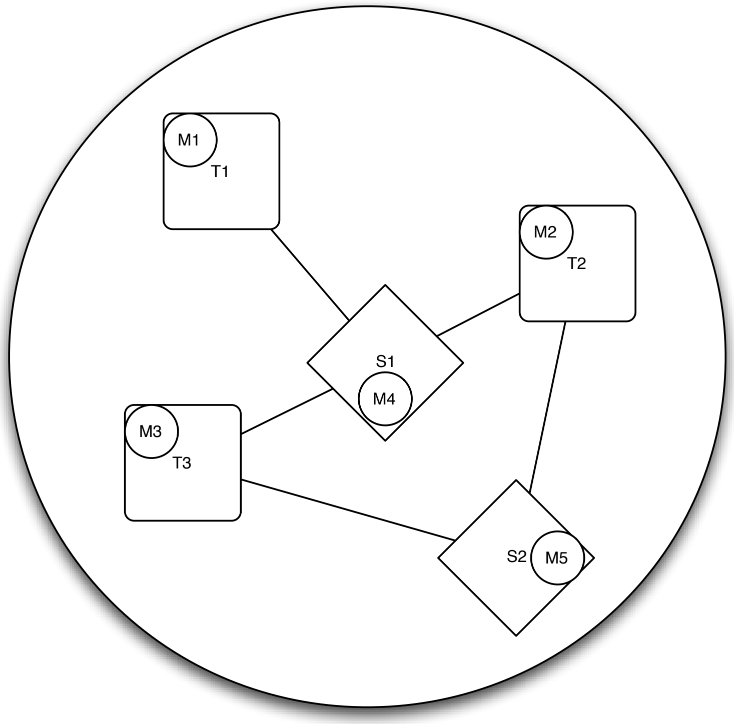

Figure 4.5: System Structure

4. Generate a top-level VHDL entity representing the thread by applying the Oread hardware compilation scheme, described in chapter 3, to the transformed functions.

5. Add VHDL ports to the generated entity for each service that is referenced by the thread. For each referenced service this will include an output request data port, an input response data port, and 1-bit control ports associated with each signal.

- Each Oread service is compiled to a VHDL block through the following steps.

  1. Calculate the set of live functions referenced by the handler and scheduler functions.

2. Perform defunctionalization and static monad instance resolution transformations on the set of live functions.

3. Apply the Oread hardware compilation scheme from chapter 3 to the transformed definitions, creating a top-level VHDL entity representing the service.

4. Add VHDL ports to the generated entity for each thread. For each attached thread this will include an input request data port, an output response data port, and 1-bit control ports associated with each signal.

The system generation procedure defined above is complete with exception that it does not account for memory initialization. The system compilation scheme handles this for memories referenced by threads by generating an auxiliary top-level monadic function definition the simply calls each initialization function in turn, then completes with a tail-call to the function definition. The procedure is similar for service definitions, except that it generates a single top-level function which calls the initialization functions for each memory. This top-level initialization function is called for each service at system reset.

Example: Consider the following Oread configuration fragment.

```
(configuration init
  (memory @m1 Int Int init1)
  (memory @m2 Int Bool init2)
  (thread (t1 2 True [@m1 @m2]))
  ...)
```

The auxiliary function `t1_top` will be generated that performs memory initialization before invoking the main body of the thread.

```
(define (t1_top (x Int) (y Bool)
  (monad [(@mem1 (State Int Int)) (@mem2 (State Int Bool))] Unit))
  (do (init1 [@mem1])
      (init2 [@mem2])
      (t1 2 True [@mem1 @mem2])))
```

4.3 Summary

The Oread compilation scheme defined in chapter 3 only operates on a restricted subset of the language. One restriction is that only first-order programs can be compiled. Using defunctionalization, a higher-order Oread program can be automatically transformed into a first-order program. Moreover, by assuming the perspective that invocations of monadic constructs on parametrized instances, a slightly adapted defunctionalization transform can be used to eliminate dynamic instances.

Oread programs that are compiled to hardware must use static resources. This eliminates the possibility of using common functional programming idioms such as recursive types. The static semantics of Oread are too restrictive to allow automatic conversion between functions written over recursive types to functions over fixed-sized representations of those types. Specific higher-order functions can be defined in Oread that when partially applied, yield residualized programs that can be compiled to hardware. In contrast to staged programming languages like MetaOCaml and Template Haskell, there is no support directly in the language for performing these staging transformations, so the tool set develop relies on manual staging using tools not defined within the language.

Chapter 5

Compiling Oread to Software

The thesis of this dissertation is that a functional language, extended with monadic constructs for imperative state and reactive concurrency, is a suitable basis for developing systems that can be compiled to both hardware and software targets. In the previous two chapters, we described the compilation of Oread to VHDL netlists.

In this chapter, we describe the process of compiling Oread to a software computational model. In many respects, this compilation task is simpler than the hardware compilation scheme, as the language restrictions are not in place. Compiling functional languages to stock microprocessors is a well-studied topic (Appel 1992; Jones 1992), and a great deal of background work has focused on making the resulting programs compile to efficient executables.

The hardware computational model presents a greater challenge for compilation, as it places restrictions on Oread programs that allow those programs to be synthesizable. This dissertation does not claim any new contributions with respect to compiling a functional language to a software computational model. Rather, we seek to demonstrate simply that a language with the features of Oread is a reasonable basis for compilation to varied target fabrics. To this end, the compilation schemes described below may use naive approaches that result in less efficient implementations. Nevertheless, the software compilation scheme serves as a foundation upon which to apply the vast body of research on efficient compilation of functional languages.

The development in this chapter is not without novel contributions, the most important of which is the method for compiling the Oread concurrency model, which uses message-passing concurrency based on the reactive monad, to existing concurrency primitives. In section 5.3, we describe a method for mapping the Oread concurrency model to the Pthreads (IEE 2004) library, based on shared-state concurrency.

The Oread software compilation scheme has three primary components. The first is the data model, which describes how Oread values are represented in memory. The second component describes the mapping of Oread constructs to an imperative language. We have chosen to target the C (ISO 1999) language for this mapping, as C is often used as a universal assembly language. The compilation scheme doesn't generate implementations that utilize any esoteric features of C, so it is entirely reasonable that the compilation target a different imperative language or a processor-specific instruction set. The final component is the implementation of an Oread system as a collection of POSIX threads. Again, this mapping utilizes Pthreads for convenience, and the mapping can be re-targeted towards other preemptive shared-state concurrency models.

5.1 Oread Software Data Model

Recall from section 2.2.2 that Oread programs operate over a limited set of types: integers, reals, algebraic types, products, closures (functions), and monadic instances (both state and reactive). The data model for Oread software compilation requires a mapping of these types to standard software types. In fact, the data model utilizes a more restricted set of software types.

Integers map to the native word size of the target architecture – for example, 32 or 64-bits. Because the software compilation scheme generates C, rather than assembly, the results are architecture agnostic in the particular bit widths. Similarly, Oread reals are mapped to the native floating point representation. Because the compilation scheme assumes that all floating point values are unboxed, they must also fit within the target-specific word size. Both values of algebraic data types and products map to heap-allocated arrays of words. A constructor for a data type with n fields will map to an array of $n+1$ pointers. The array element at position 0 in this array will contain the integer tag associated with that constructor. An n-ary product will map to an n-element array of pointers.

The software data model does not provide a representation for closure data structures, as the defunctionalization transformation 4.1.1 can be used to eliminate closures from the input program. The data model represents of closures using the combination of algebraic data types and dispatch functions generated by the defunctionalization transformation.

The final two Oread data types that must be modeled using software constructs are state and reactive monad instances. Both of these are statically allocated structures. For a state instance, this is simply an array of pointers. The Oread language allows a state monad instance to have any address or data type. In the software compilation scheme, we assume that the values of the address type can be enumerated, allowing the compilation scheme to use native pointer arithmetic for accessing and mutating data within a state instance. The details of the reactive instance representation is deferred to section 5.3, as it is closely tied to the underlying concurrency implementation.

Table 5.1 summarizes the Oread to C data mapping. The mapping between Oread type and C type is specified as $\mathcal{T}[\![ty]\!]$ in the compilation scheme for the Oread expression language shown below. It is important to note that the Oread assumes no particular au-

Oread term	Oread type	C type	Size
`n`	Int	int	Native word width
`f`	Real	float	Native word width
`C i0 .. in`	T	void**	n+1
`tuple i0 .. in`	(* t0 .. tn)	void**	n
`@st`	state a b	b**	size(a)

Table 5.1: Oread to C data mapping

tomatic memory system. This is not because there is no need, as the resulting software implementations will allocate liberally, a problem especially in often resource-limited embedded environments. On the other hand, it is unlikely that any single collection scheme will be suitable in all environments. Consequently, the current work focuses on generating executable software – we propose that efficient use of memory resources requires an additional research effort.

5.2 Oread Compilation Schemes

When compiling an Oread expression to C, each expression results in (i) a list of C declarations, (ii) a list of C statements, and (iii) a value, which can either be a C identifier, an integer literal, or a floating point literal. The Oread tool set uses a Haskell data type to represent C AST elements for declarations and statements. To simplify the presentation of the compilation scheme, we use the C syntax directly, rather than the intermediate AST data structure.

The compiler targets a small collection of C templates that define the structure of the resulting C code. These templates contain variables, identifiers proceeded by a $, which correspond to compiler-supplied syntax. For example, figure 5.2 show the template used for an Oread top-level function declaration. Within the template contains template variables for the function name `$FUNAME`, an arbitrary number of function parameters

`$xi` with types `$ti`. Within the body of the function is a template variable `$DECLS` for local declarations, whichare temporaries created by the expression, a template variable `$STMTS` representing the C statements that define the logic of the function, and finally a `$VAR` template variable representing the value, generally a variable name, where the result of the function is stored by the function's statements.

Compilation schema are written in the same monadic style as the evaluation semantics from section 2.2.2. When generating the resulting declarations, statements, and value for an Oread expression or declaration, the compilation scheme may instantiate a template. Template instantiation is performed by macro-expanding the template variables in the template with the arguments (declarations, statements, value) supplied by the compilation scheme.

When compiling an Oread expression with nested sub-expressions, the compilation scheme will generate a unique temporary value for each computed expression. Each temporary is only assigned a value one time. This behavior corresponds to the common practice in compiler construction of initially compiling to an idealized instruction set with an infinite number of registers, then using a separate register allocation pass to map temporary registers to physical registers or stack locations. The Oread software mapping defers to the C compilation scheme to perform this register allocation. To insure names of temporaries are unique, a monadic `gensym` function yields a fresh name each time it is called. The `gensym` function is implemented using the state monad in the meta-language.

The C arithmetic operations are overloaded, thus it is necessary to use type information to generate the appropriate declarations so that these operations resolve correctly. The compilation schemes assume that the Oread type information is available for every construct. In the compiler implementation this type information is implicitly provided from a monadic type checker (Weaver et al. 2007).

5.2.1 Compiling Algebraic Data Types

When compiling algebraic type declarations, the compilation scheme generates template code for each constructor that performs the appropriate memory allocation and copies any constructor arguments to the allocated memory. Each value carries a tag representing the constructor in the first element of the allocated array.

Example: Consider the following data type declaration.

```
(data Instr (Add Int Int) (Negate Instr))
```

Constructor templates for `Add` and `Negate` are shown below. The `Instr` type is simply a **`typedef`** for **`void`**`**`, and `PTR` a **`typedef`** for **`void`**`*`. Note that the template for `Foo` stores the tag `0` at element 0, and the template for `Negate` store the tag `1`.

```
// Template for (Foo x y)
Instr var0 = (Instr)(malloc (12)) ;
    ((var0) [0]) = ((PTR)(0));
    ((var0) [1]) = ((PTR)(x));
    ((var0) [2]) = ((PTR)(y));

// Template for (Negate x)
Instr var1 = (Instr)(malloc (8)) ;
    ((var1) [0]) = ((PTR)(1));
    ((var1) [1]) = ((PTR)(x));
```

Compiling an algebraic data type declaration simply generates a series of top-level C function declaration. Two compilation schemes are used: the $\mathcal{D}[\!]$ scheme compiles a data type consisting of a collection of defined constructors, and the $\mathcal{C}[\!]$ scheme compiles a constructor. Although each constructor yields a new function declaration, the uses of the function declaration can be safely inlined.

$$
\begin{array}{lcl}
\mathcal{D}[\text{ data } DT = C_0\, t_0 \ldots\, t_n | \ldots | C_i\, t_0 \ldots t_m] & = & \begin{array}{l} \mathcal{C}[C_0 t_0 \ldots t_n] 0 DT \gg\!= \lambda decls_0 \rightarrow \\ \ldots \\ \mathcal{C}[C_i t_0 \ldots t_m] i DT \gg\!= \lambda decls_i \rightarrow \\ \text{let type} = \texttt{typedef void** DT} \text{ in} \\ \text{return } (type, decls_0, \ldots, decls_i) \end{array} \\
\\
\mathcal{C}[Cons\, t_0 \ldots t_n] tagty & = & \begin{array}{l} gensym \gg\!= \lambda \rightarrow arg_0 \\ \ldots \\ gensym \gg\!= \lambda \rightarrow arg_n \\ \text{return} \\ \quad (\text{ ConstructorTemplate } \texttt{DT}= ty, \texttt{N}= n, \\ \qquad\quad \texttt{ti}= \mathcal{T}[t_i], \texttt{TAG}= tag) \end{array}
\end{array}
$$

```
$DT c($t0 $v0, $t1 $v1, .. $tn $vn) {
  $DT t = malloc($N+1 * sizeof(PTR));
  t[0] = (PTR) $TAG;
  t[1] = (PTR) $v1;
  ...
  t[n+1] = (PTR)$vn;
  return t;
}
```

Figure 5.1: Constructor Application Template

5.2.2 Compiling Function Declarations

Compiling a top-level Oread function declaration uses the $\mathcal{D}[]$ compilation scheme to yield a C function declaration. This declaration in turn calls an $\mathcal{E}[]$ scheme, which generates the declarations, statements, and return value for the function body. The $\mathcal{E}[]$ scheme for λ expressions takes an extra f parameter, which is the name of the function to generate a template for. After generating unique parameter names and setting up an environment using `withEnv`, the $\mathcal{E}[]$ scheme recursively invokes $\mathcal{E}[]$ on the body of the lambda.

$$
\begin{array}{rcl}
\mathcal{D}[\![\lambda\ x_0 : t_0 \ldots x_n : t_n\ \ t_r \to e]\!] f & = & \text{gensym} \gg\!= \lambda v_0 \to \\
& & \ldots \\
& & \text{gensym} \gg\!= \lambda v_n \to \\
& & \text{withEnv}\ ([x_0 \mapsto v_0 \ldots, x_n \mapsto v_n]) \mathcal{E}[\![e]\!] \gg\!= \lambda(d, s, rval) \to \\
& & \text{return} \\
& & \quad FunctionTemplate(\\
& & \qquad \texttt{RET=} \mathcal{T}[\![t_r]\!], \texttt{FUNNAME=} f, \\
& & \qquad \texttt{ti=} \mathcal{T}[\![t_i]\!], \texttt{xi=} v_i, \\
& & \qquad \texttt{DECLS=} d, \texttt{STMTS=} s, \texttt{VAL=} rval),
\end{array}
$$

```
$RET $FUNNAME($t0 $x0, ...,  $tn $xn) {
  $DECLS
  $STMTS
  return $VAL
}%$
```

Figure 5.2: Function Declaration Template

5.2.3 Compiling Expressions

The $\mathcal{D}[\![]\!]$ compilation scheme generates top-level declarations, while the $\mathcal{E}[\![]\!]$ scheme, described below, defines the compilation scheme for each Oread expression. In each case, the compilation scheme maps an Oread expression to a triple of declarations, statements, and a value. The value can be an integer or real literal, or else a Var representing a declared temporary.

Function Applications The function application compilation scheme has three steps. First, it invokes the compilation scheme on each function parameter. Next, it declares a destination for a result value then constructs a C call for the declared function. Finally, it collects the declarations and statements for all parameters with the generated statement. The Oread compilation procedure assumes that the input program has been defunctionalized, all calls will be to a named top-level function. Consequently, the result of the $\mathcal{E}[\![]\!]$ scheme for the called function will always be a Var.

$$
\begin{array}{rcl}
\mathcal{E}[\![e\ e_0\ \dots\ e_n]\!] & = & \mathcal{E}[\![e_0]\!] \ggg\!= \lambda(d_0, s_0, v_0) \rightarrow \\
 & & \dots \\
 & & \mathcal{E}[\![e_n]\!] \ggg\!= \lambda(d_n, s_n, v_n) \rightarrow \\
 & & \mathcal{E}[\![e]\!] \ggg\!= \lambda([], [], Var f) \rightarrow \\
 & & \text{gensym} \ggg\!= \lambda res \rightarrow \\
 & & \text{let call =} \\
 & & \quad (FunctionCallTemplate \\
 & & \qquad \texttt{DEST}= res, \texttt{Ai}= v_i \\
 & & \qquad \texttt{FUNNAME}= f) \\
 & & \text{in return} \\
 & & \quad (d_0 \dots d_n\ res, \\
 & & \quad s_0 \dots s_n, \text{call}, \\
 & & \quad res)
\end{array}
$$

```
$DEST =  $FUNNAME($A0, ... , $AN);
```

Figure 5.3: Function Application Template

Variables Compiling an Oread variable reference simply looks up the (previously declared) variable in the environment and returns it, generating no new declarations or statements.

$$\mathcal{E}[\![v]\!] \quad = \quad \text{getEnv} \ggg\!= \lambda\rho(\text{ return (lookup } v\rho))$$

Arithmetic Compiling Oread arithmetic operations linearizes nested operations, resulting in C code that resembles three-address code, by declaring a separate temporary variable for each operation. Compiling integer and real literals simply returns the literal.

$$
\begin{array}{lcl}
\mathcal{E}[\![e_0\ op\ e_1]\!] & = & \text{gensym} \gg\!= \lambda(Var\ res) \rightarrow \\
 & & \mathcal{E}[\![e_0]\!] \gg\!= \lambda(d_0, s_0, r_0) \rightarrow \\
 & & \mathcal{E}[\![e_1]\!] \gg\!= \lambda(d_1, s_1, r_1) \rightarrow \\
 & & \text{let stmt} = \texttt{res = r\_0 op r\_1;} \\
 & & \text{in return} \\
 & & \quad (res, d_0, d_1; \\
 & & \quad s_0, s_1, \text{stmt} \\
 & & \quad res) \\
\mathcal{E}[\![n]\!] & = & \text{return (I n)} \\
\mathcal{E}[\![f]\!] & = & \text{return (F f)}
\end{array}
$$

Tupling The `tuple` operation constructs a tuple by allocating memory and storing the generated value for each tuple element. The `prj` operation simply performs an array index on a previously-allocated tuple.

$$
\begin{array}{lcl}
\mathcal{E}[\![\ \text{tuple}\ e_0\ \ldots\ e_n]\!] & = & \mathcal{E}[\![e_0]\!] \gg\!= \lambda(d_0, s_0, v_0) \rightarrow \\
 & & \ldots \\
 & & \mathcal{E}[\![e_n]\!] \gg\!= \lambda(d_n, s_n, v_n) \rightarrow \\
 & & \text{gensym} \gg\!= \lambda res \rightarrow \\
 & & \text{let tup} = \\
 & & \quad (TupleTemplate \\
 & & \qquad \texttt{DEST=}\ res, \texttt{vi=}\ v_i) \\
 & & \text{in return} \\
 & & \quad (d_0 \ldots d_n res, \\
 & & \quad s_0 \ldots s_n, \text{tup}\ , \\
 & & \quad res)
\end{array}
$$

```
  $DEST = malloc($n * sizeof(PTR));
  t[0] = (PTR) $v0
  t[1] = (PTR) $v1;
  ...
  t[n] = (PTR) $vn;
```

Figure 5.4: Tuple Template

```
$DEST = $SRC[$IDX];
```

Figure 5.5: Projection Template

$$
\begin{array}{rcl}
\mathcal{E}[\![prj\ i\ e]\!] & = & \mathcal{E}[\![e]\!] \gg\!= \lambda(d, c, \text{Var}\ r) \rightarrow \\
 & & \text{gensym} \gg\!= \lambda res \rightarrow \\
 & & \text{let project} = \\
 & & \quad ProjectTemplate \\
 & & \qquad (\texttt{SRC}= r, \texttt{DEST}= res, \texttt{IDX}= i) \\
 & & \text{in return}\ (d\ res, c\ \ \text{project}\ , res)
\end{array}
$$

Case expressions A case expression deconstructs a constructed value. In the software data model, a constructed value is heap-allocated, with the tag for the value stored in the first word. A case expression is compiled to a C switch statement. Each case alternative is compiled separately using an $\mathcal{A}[\![]\!]$ compilation scheme, which generates the declarations for the pattern variables, augments the environment, and compiles the associated expression using the $\mathcal{E}[\![]\!]$ compilation scheme.

Within each case alternative, the bindings introduced by the pattern are mapped to a set of local declarations. Each pattern variable within the entire case expression corresponds to a different local variable declaration.

$$
\begin{array}{lcl}
\mathcal{E}[\![\text{ case } e_d \text{ of } \{A_0|\ldots|A_n\}]\!] & = & \mathcal{E}[\![e_d]\!] \gg\!= \lambda(d, s, Vardis) \rightarrow \\
 & & \text{ gensym} \gg\!=\lambda res \rightarrow \\
 & & \mathcal{A}[\![A_0]\!]dis \gg\!= \lambda(d_0, s_0, Varr_0) \rightarrow \\
 & & \ldots \\
 & & \mathcal{A}[\![A_n]\!]dis \gg\!= \lambda(d_n, s_n, Varr_n) \rightarrow \\
 & & \text{ let stmt =} \\
 & & \qquad CASETEMPLATE \\
 & & \qquad\quad (\texttt{DIS}= dis, \texttt{DECLS}= d, res, d_0 \ldots d_n \\
 & & \qquad\quad \texttt{DISSTMTS}= d, \texttt{RET}= res \\
 & & \qquad\quad \texttt{SA0}= s_0 \ldots \texttt{SAN}= s_n \\
 & & \qquad\quad \texttt{V0}= r_0 \ldots \texttt{VN}= r_n) \\
 & & \text{ in return} \\
 & & \qquad (d, d_0 \ldots d_n, \\
 & & \qquad s, \text{ stmt }, \\
 & & \qquad res) \\
 & & \\
 & & \text{ gensym} \gg\!=\lambda v_0 \rightarrow \\
 & & \ldots \\
 & & \text{ gensym} \gg\!=\lambda v_n \rightarrow \\
 & & \text{ getEnv} \gg\!=\lambda \rho \rightarrow \\
 & & \text{ withEnv } [x_0 \mapsto v_0, \ldots, x_n \mapsto v_n]\rho \\
\mathcal{A}[\![C\, x_0 \ldots ; x_n]\!]dis & = & \quad \mathcal{E}[\![e]\!]_0 \gg\!= \lambda(decls, stmts, \text{Var } res) \rightarrow \\
 & & \text{ let pvars = } v_0\texttt{= dis[1];}\ldots\texttt{;}v_n\texttt{= dis[n+1];} \\
 & & \text{ in return} \\
 & & \qquad (v_0 \ldots v_n, decls, \\
 & & \qquad \text{ pvars }, stmts, \\
 & & \qquad res)
\end{array}
$$

Let Expressions An Oread let expression introduces local bindings in parallel. However, the let expression is not recursive, simplifying the compilation of the construct. In the compilation scheme, each of the bindings is evaluated, the environment is extended to include the new bindings, and the body evaluated in the extended environment.

```
$DISSTMTS
switch *($DIS) {
  case 0:
     // Statements for the case alternatives
     $SA0;
     // Save the result
     $RET = $V0
     break;
   ...
  case n:
   // Statements for the case alternatives
   $SAN;
   //
   $RET = $VN
   break;
}
```

Figure 5.6: Case Template

$$
\begin{aligned}
\mathcal{E}[\![\text{let } x_0 : t_0 = e_0 \ \ldots \ x_n : t_n = e_n \text{ in } e_b]\!] \quad = \quad & \mathcal{E}[\![e_0]\!] \gg\!= \lambda(d_0, s_0, v_0) \rightarrow \\
& \ldots \\
& \mathcal{E}[\![e_n]\!] \gg\!= \lambda(d_n, s_n, v_n) \rightarrow \\
& \text{getEnv} \gg\!= \lambda\rho \rightarrow \\
& \text{withEnv } [x_0 \mapsto v_0, \ldots, x_n \mapsto v_n]\rho \\
& \qquad \mathcal{E}[\![e_b]\!] \gg\!= \lambda(d, s, v) \rightarrow \\
& \text{return } (d_0, \ldots, d_n, d; s_0, \ldots, s_n; v)
\end{aligned}
$$

5.2.4 Compiling Monadic Forms

The Oread C software target already includes sequencing of statements and imperative features, considerably simplifying the compilation of the monadic forms **`return`**,

`bind`, **`get`**, and **`put`**. A **`return`** compiles the **`return`** argument, while **`bind`** compiles the monadic actions, allocating a temporary to store the result of the first action.

$$
\begin{array}{lcl}
\mathcal{E}[\text{ return } x] & = & \mathcal{E}[x] \\[1em]
\mathcal{E}[\text{ bind } e_0\ x\ e_1] & = & \begin{array}{l}
\text{gensym} \gg\!= \lambda temp \rightarrow \\
\mathcal{E}[e_0] \gg\!= \lambda(d_0, s_0, r_0) \rightarrow \\
\text{getEnv} \gg\!= \rho \\
\text{withEnv } [x \mapsto temp]\rho\, \mathcal{E}[e_1] \gg\!= \lambda(d_1, s_1, r_1) \rightarrow \\
\text{return } (d_0, d_1, temp; \\
\quad s_0, \texttt{temp =} r_0, s_1; r_1)
\end{array}
\end{array}
$$

In the Oread software compilation scheme, a memory instance is a statically allocated array of values. The software compilation scheme assumes that the address type can be converted into the native pointer type of the target architecture. Under this assumption, **`get`** and **`put`** map directly to array loads and stores. In the $\mathcal{E}[]$ scheme, the compilation of a monadic instance yields a list of declaration, statements, and a value that is the pointer to the statically-allocated memory. Because the memory is statically allocated at system start time, the declarations and statements corresponding to the memory instance are ignored.

$$
\begin{array}{lcl}
\mathcal{E}[\text{ get } i\ e] & = & \begin{array}{l}
\mathcal{E}[i] \gg\!= \lambda(d_i, s_i, Vararr) \rightarrow \\
\mathcal{E}[e] \gg\!= \lambda(d, s, addr) \rightarrow \\
\text{gensym} \gg\!= \lambda temp \rightarrow \\
\text{return } (d; s, \texttt{temp = arr[addr];}; temp)
\end{array} \\[1em]
\mathcal{E}[\text{ put } i\ e_a\ e_d] & = & \begin{array}{l}
\mathcal{E}[i] \gg\!= \lambda(d_i, s_i, Vararr) \rightarrow \\
\mathcal{E}[e] \gg\!= \lambda(d_a, s_a, addr) \rightarrow \\
\mathcal{E}[e] \gg\!= \lambda(d_d, s_d, dat) \rightarrow \\
\text{return } (d_a, d_d; s_a, s_d, \texttt{arr[addr]=dat;}; temp)
\end{array}
\end{array}
$$

5.3 Oread Concurrency Model Compilation

The $\mathcal{E}[]$ compilation scheme generates a series of C declarations from an Oread program. It does not provide for combining those functions into an executable system, based on an Oread configuration. In this section, we describe a method for mapping the Oread concurrency architecture to the POSIX constructs, a shared memory threading library.

In the Oread to POSIX threads mapping, Oread threads are mapped to Pthreads threads. The Oread service definitions are mapped to a C data structure that contains the functions for the Oread service handler and scheduler in addition to a Pthreads mutex and a series of condition variables for controlling atomic access to the service. In the Oread Pthreads mapping, services are not spawned to Pthreads threads. Rather, when an Oread user thread (each mapped to a separate Pthreads thread) it obtains exclusive access, by locking a Pthreads mutex, to the service state until the request handler and scheduler have been invoked. Because a service may not immediately produce a response for the requesting thread, the mapping uses a condition variable to signal to the requesting thread a response is pending.

Figure 5.7 shows the C representation of an Oread service. The `mutex` is a single `pthread_mutex_t` used to control serialize access to the service user threads. Following this is a `threadResponse` array containing Oread values for thread responses, one per thread attached to the service. When a service schedules a request, it will place the response in the appropriate position of the `threadResponse` array. Moreover, the service will also signal to the thread that the response is available using the respective element of the `threadResponseActive` array, which contains a Pthreads condition variable for each attached thread.

The `threadResponse` and `threadResponseActive` arrays are integer indexed. A Pthreads thread has a unique thread identifier assigned to it by the library, but there is no guarantee that the Pthreads identifier will correspond to an appropriate index into the `threadResponse` and `threadResponseActive` arrays. Therefore, the Service data structure maintains the `index4Pthread` array, which provides the mapping from Pthreads identifier to Oread identifier. To simplify the initialization of a Pthreads-based Oread system, the thread identifier mapping is constructed dynamically, using the bounds established by the `numThreads` and `allocatedIndices` fields of the Service struct.

The `ServiceInterface` data structure contains a `serviceMemory` field, which is an Oread state monad instance representation local to the service. Finally, the service structure contains function pointers for the associated initialization function `handlerFn` for the local memory, the service handler (`handlerFn`), and the scheduler (`schedulerFn`).

The `ServiceInterface` is passed as a monadic instance parameter to threads. The Oread run-time system contains a `signal` function that takes a ServiceInterface instance and a message and executes the handler and scheduler in turn, using the mutex and the condition variables of the ServiceInterface parameter.

The control flow of this `signal` function is the same as that of the ServiceBlock state machine in figure 3.13. While the ServiceBlock VHDL component that implemented that state machine used the Oread control protocol to ensure exclusive access to the ServiceBlock, the `signal` C function uses Pthreads. Figure 5.8 shows the definition of the run-time `signal` function.

The `signal` function first performs the Pthreads thread id to Oread thread id mapping using the `serviceIndexForThread` function. It then locks the mutex for the service and calls the handler function. After the handler has returned, the `signal` function enters a loop, where it repeatedly invokes the scheduler function there is no

```
typedef struct {
  // mutex makes service access atomic
  pthread_mutex_t* mutex;
  DLANG_PTR   threadResponse;

  pthread_cond_t** threadResponseActive;

  // Pthreads thread id to Oread thread id
  int numThreads;
  pthread_t*  index4Pthread;
  int allocatedIndexes;

  // Service specific data.
  DLANG_PTR  serviceMemory;

  // Unit initFn(int numThreads, DLANG_PTR serviceState)
  DLANG_Unit  (*initFn)(int, DLANG_PTR);

  // Unit handlerFn(int threadId, int chan,
  //                DLANG_PTR msg, DLANG_PTR serviceState)
  DLANG_Unit  (*handlerFn)(int, int, DLANG_PTR, DLANG_PTR);

  // schedulerFn(int dummy, DLANG_PTR serviceState)
returns Maybe (AThread(threadId, msg))
  DLANG_PTR  (*schedulerFn)(int, DLANG_PTR);

} ServiceInterface;
```

Figure 5.7: Service Structure

responses are generated. The result of the scheduler function is an Oread `Maybe` type – indicating if there is (`Just (tuple tid msg)` or is not (`Nothing`) pending.[1] Whenever a response is ready, the condition variable associated with the indicated thread is signaled, and the loop repeats.

After the scheduler indicates that no responses are ready, the loop terminates. At this point, the requesting thread then blocks on its condition variable. Due to the semantics of Pthreads condition variables, it is necessary to check to see if the value is not NULL, indicating that the scheduler loop has already signaled the condition variable for that thread.

The call to `pthread_cond_wait` will unlock the service mutex, thus allowing other threads to issue requests. This call will not return until another thread's invocation of the `signal` function results in the current thread's condition variable being called with `pthread_cond_signal`. However, after the call to `pthread_cond_wait` returns, the `signal` function copies the generated response for the current thread, resets the response field, [2] unlocks the mutex [3], and returns the generated response.

5.4 Summary

Oread can be compiled to C by utilizing a small collection of C templates, each corresponding to an Oread construct. The compilation of an expression yields a series of declarations, statements, and a value. The compilation scheme allocates a new tempo-

[1]The code within this loop depends closely on the data representation of the `Maybe` type. Although simple to change, it presents a brittle connection between the run-time and the compiler.

[2]The Oread concurrency semantics prevent a race condition between the invocations of `signal`, because it is not possible for a thread to have two pending requests at any given time, and there is still one request outstanding until `signal` returns.

[3]This unlock is necessary if the thread reached this point without calling `pthread_cond_wait`, which automatically unlocks the mutex. If another thread has already obtained the mutex, the call to unlock will fail safely but silently.

```
DLANG_PTR signal(DLANG_PTR serviceInterface, DLANG_PTR msg) {
    int isActive;
    ServiceInterface *si = (ServiceInterface *)serviceInterface;

    // Capture the threadId of the calling thread, as a unique identifier.
    int  threadId = serviceIndexForThread(si);

    // Lock the mutex to serialize access to the handler and scheduler
    pthread_mutex_lock (si->mutex);
    si->handlerFn(threadId, msg, si->serviceMemory);

    // Repeatedly invoke the scheduler until no new responses are ready
    do {
    DLANG_PTR nextThread = si->schedulerFn(dummy, si->serviceMemory);
    switch ((int)((nextThread) [0]))
             {case 0 : // Nothing ready to run
                     {isActive = 0; break;}
              case 1 : // A response is ready
                {DLANG_PTR var3 = (DLANG_PTR)((nextThread) [1]) ;
                 {switch ((int)((var3) [0]))
                    {case 0 :
                       {int tid = (int)((var3) [1]) ;
                        DLANG_PTR x = (DLANG_PTR)((var3) [2]) ;
                        si->threadResponse[tid] = x;
                        pthread_cond_signal(si->threadResponseActive[tid]);
                        isActive=1; break;}}}
                 break;}}
    } while (isActive);
    // Block waiting on a response. pthread_cond_wait unlocks the mutex.
    while (si->threadResponse[threadId] == NULL) {
        pthread_cond_wait (si->threadResponseActive[threadId], si->mutex);
    }

    DLANG_PTR response = si->threadResponse[threadId];
    si->threadResponse[threadId] = NULL;
    // Unlock the mutex, in case pthread_cond_wait was not called.
    pthread_mutex_unlock (si->mutex);
    return response;
}
```

Figure 5.8: Run-time `signal` function

rary to hold intermediate values, relying on the downstream C compiler to map those temporaries to registers or stack locations.

The state monadic features of Oread map directly to the imperative features of C. The concurrency features, on the other hand, require additional compilation and runtime effort, as the C does not provide a native threading capability. We defined a mapping from the Oread **`signal`** construct to the standard POSIX Pthreads library, which uses a shared-state model of concurrency. The mapping uses standard concurrency control constructs such as mutexes and condition variables, but does not use any Pthreads-specific features, allowing the Oread compilation scheme to be mapped to similar shared-state concurrency models.

Chapter 6

Case Study: Software Defined Radio

The target application domain for Oread is software defined radio (SDR) synthesis. Software defined radio is an approach to constructing radios that allows a common radio platform, with a variety of computational resources, to be quickly re-targeted to meet changing application demands. Because the radio signal processing is implemented in software, or alternatively, in a digital hardware substrate like FPGA, the radio processing components is implemented using digital signal processing techniques, rather than analog hardware components.

A typical SDR platform, such as GNURadio (Blossom 2004) will contain an analog radio frequency (RF) front-end that handles receive and transmit of modulated data. The operating frequency of the RF front-end is very high, often in the gigahertz band. Consequently, it is necessary to perform a frequency conversion via digital sampling and down-conversion to shift the signal to a lower intermediate frequency (IF). The bandwidth of the digital IF signal is typically low enough to allow digital processing using specialized hardware, but is still generally too high to be processed in software. Thus, the IF signal is digitally down-sampled to yield a base band signal, which is capable of being processed using software. Figure 6.1 shows this configuration graphically.

The flow described above, from RF to IF to base band, is performed on the receive side of the radio; the inverse flow is used in transmission.

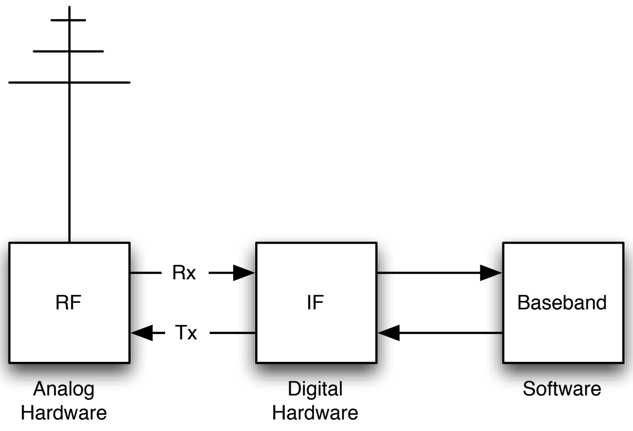

Figure 6.1: SDR configuration

The capability of Oread to compile a design to both hardware and software is important in the SDR domain because potential platforms include both hardware and software targets. A given radio waveform may include analog requirements, such as operating frequency and modulation scheme; signal integrity requirements, implemented using error detection and correction; confidentiality requirements, implemented using cryptographic techniques; and application-level protocol requirements. These requirements may be altered due to changing application demands or due to changing platform capabilities.

To satisfy the changing application requirements, it may be necessary to re-target a given component from hardware to software, or vice versa. Oread is designed to minimize the engineering effort needed when performing implementation re-targeting. Components are encapsulated as Oread threads which can be compiled to either a software target or a hardware target. Inter-component communication is performed using the Oread message-passing concurrency model, divorcing the component implementation from low-level fabric details, such as timing.

To demonstrate this ability, we use Oread to define a common error correction component based on Hamming codes (Hamming 1950). A Hamming coding scheme is a linear error correction scheme that allows the detection of transmission errors with two erroneous bits, and correction of transmission errors with a single erroneous bit.

An error-correction component such as a Hamming encoder or decoder is often included as part of a pipeline of digital signal processing components that may include other components like compression and encryption. We define an Oread service to allow two such components to communicate using a FIFO communication scheme. The FIFO scheme allows the system components to abstract away from waveform-specific details, such as the data bit-rates consumed and produced by each component, further simplifying development of re-targetable radio components.

6.1 Hamming Encoding

The Hamming encoding scheme allows for the detection and correction of transmission errors that result in inverted bits in the received signal. The encoding scheme calculates a number of parity bits from the word to transmit, then transmits those parity bits interleaved with the original word. When received, the parity bits are separated from the data bits, the parity calculation is performed, and the parity bits calculated by the receiver are compared with the received parity bits. This comparison will reveal one-bit errors and allow those errors to be corrected. Moreover, the comparison will be able to detect two-bit errors, although it will be unable to correct the error.

Figure 6.2 shows the Hamming coding scheme graphically. At the top of the figure is a representation of the data word to transmit. Each box corresponds to a single bit; the bits are numbered starting at index 1. The bottom figure shows the transmitted word with parity bits, shaded gray, included. The arrows from the data bits to parity bits indicate the data bits that contribute to each parity bit. The bit index scheme for a parity bit n is to drop the first $n-1$ data bits, take the next n bits, drop n bits, and so on. Rather than treat the entire data input as a single stream of bits, a hamming encoder will generally break the input stream into a fixed word size of w bits, and restart the encoding after processing every w bits. This eliminates the need to receive the entire stream before beginning the parity calculation.

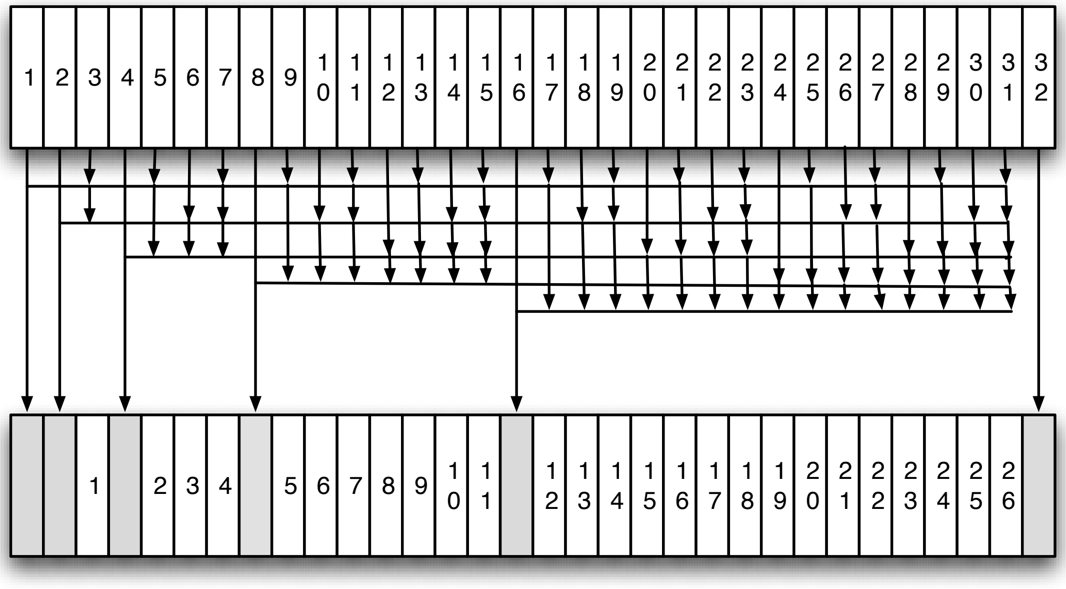

Figure 6.2: Hamming encoding scheme

As the graphical representation of the coding scheme exhibits, there is a great deal of data parallelism in the parity bit calculations. For example, each parity bit can be calculated using a separate circuit, offering a maximal amount of parallelism, but also consuming the most computational resources as the parity of n bits requires $n-1$ XOR operations. Alternatively, the parity calculations can be cascaded. Every group of 2^n bits in a parity bit n block consists of 2^{n-1} bits not factored into the previous parity bit block, and then 2^{n-1} bits that are calculated by the previous parity bit. Using this structure in hardware would allow for a significant reduction in circuit resources, while adding a small amount of timing overhead, as the calculation of parity bits are cascaded. Finally, a purely software implementation of the coding scheme on a CPU with a single XOR ALU operation may calculate each parity bit individually, without sharing the results of prior parity bits, sacrificing the maximal amount of time for reduced resource usage.

To allow a system designer to target each of these different configurations, we use a generic Hamming encoder component. Encoder input data is defined as a list of bits and the encoding is performed in three parts. First, generate the input bit indices for each parity bit. Second, calculate the parity bits by folding an `xor` function over each collection of bits. Finally, combine the data bits and the parity bits into a transmittable bitvector.

The `big` function defines the index generation function. Given an input bitvector `numbits` wide and a parity bit index `i`, the resulting list contains the indices for that parity bit. The `big` function uses a library of functions such as **`map`** and `enumFromTo`. The **`map`** function will apply a function to each element in a list, while `enumFromTo` will generate the sequence of integers between its two arguments as a list.

```
(define (big (numbits Int) (i Int) (List Int))
       (let ((stepsize (exp 2 i)))
        (let ((numtake (/ stepsize 2))
              (maxindex (+ numbits (+ (ceilingLog2 numbits) 1))))
          (let ((numsteps (+ 1 (/ (+ maxindex numtake) stepsize))))

  (tail
   (flatten
    (map (lambda ((step Int)) (List Int)
                 (map (lambda ((offset Int)) Int
                              (+ (- (* step  stepsize) numtake) offset))
                      (enumFromTo 0 (- numtake 1))))
       (enumFromTo 1 numsteps)))))))
```

Example: Calling `(big 32 2)` generates the bit indices for the second parity bit.

```
(Cons 3 (Cons 6 (Cons 7 (Cons 10 (Cons 11 (Cons 14 (Cons 15 (Cons 18
(Cons 19 (Cons 22 (Cons 23 (Cons 26 (Cons 27 (Cons 30 (Cons 31 (Cons 34
(Cons 35 (Cons 38 (Cons 39 (Cons 42 (Cons 43 Null)))))))))))))))))))))
```

The `big` function yields the bit indices for a single parity bit. However, these indices are into the resulting encoded word, which includes parity bits. Thus, the indices are adjusted by subtracting out the number of parity bits prior to each index, yielding an index of a data bit into the original bitvector. Moreover, indices that are outside the range of the word size are eliminated using the `filter` function.

```
; adjust x = x - (log2int x)
(define (adjust (x Int) Int) (- x (ceilingLog2 x)))

(define (inputBIG (numbits Int) (i Int) (List Int))
  (filter (lambda ((x Int)) Bool (lessthaneq x numbits))
          (map adjust (big numbits i))))
```

The `inputBig` generates indices into the original data vector that each parity bit will be calculated from. The parity function itself is calculated by performing an accumulating fold (`foldl`) over the bits at those indices.

```
(data Bit  (Low) (High))

(define (parity (l (List Bit)) Bit)
  (foldl xorb Low l))

(define (xorb (a Bit) (b Bit) Bit)
  (case a
    ((High) (notb b))
    ((Low) b)))

(define (notb (a Bit) Bit)
  (case a
    ((High) Low)
    ((Low) High)))
```

Finally, the Hamming encoding operation for an entire bitvector is defined by mapping the parity function over each set of indices generated by the `inputBIG` function. To simplify the presentation, the `encode` function simply appends the parity bits to the original bitvector, rather than interleaving those parity bits as shown in the graphical representation of the Hamming encoder in figure 6.2.

```
(define (encode (bv (List Bit)) (List Bit))
 (let ((len (length bv)))
  (let ((indices (map (lambda ((i Int)) (List Int) (inputBIG len i))
                     (enumFromTo 1  (+ 1 (ceilingLog2 len))))))
   (append bv
    (map (lambda ((idxs (List Int))) Bit (parity (sel idxs bv)))
        indices)))))
```

Using the `encode` function, we can specialize the function for a bit vector represented as a product of bits. To shorten the length of the residual program, shown in figure 6.3, we do not specialize the `xorb` function.

```
(define (encode16
   (p (* Bit Bit Bit Bit Bit Bit Bit Bit Bit Bit Bit Bit Bit Bit Bit Bit))
   (List Bit))
  (encode (prod16ToList p)))
```

Thus defined, the generic `encoder` component yields a Hamming encoder for arbitrary width data words. Because the encoder component is defined using strictly the functional subset of Oread, it can be transformed to fit the computational resources of the target architecture. In the residualized version in figure 6.3, the `xorb` function has not been unfolded, so it would result in a single shared function block. However, the `xorb` block can be duplicated, and calls to those duplicated instances distributed, trading circuit space for improved performance.

Figure 6.4 shows a portion of the generated C output from the software compilation scheme, and figure 6.5 shows a portion of the generated VHDL. These examples are

```
(define (encode16
   (p (* Bit Bit Bit Bit Bit Bit Bit Bit Bit Bit Bit Bit Bit Bit Bit Bit))
   (List Bit))


(Cons (prj 0 p) (Cons (prj 1 p) (Cons (prj 2 p) (Cons (prj 3 p)
(Cons (prj 4 p) (Cons (prj 5 p) (Cons (prj 6 p) (Cons (prj 7 p)
(Cons (prj 8 p) (Cons (prj 9 p) (Cons (prj 10 p) (Cons (prj 11 p)
(Cons (prj 12 p) (Cons (prj 13 p) (Cons (prj 14 p) (Cons (prj 15 p)
; Begin parity bits
(Cons (xorb (prj 15 p) (xorb (prj 13 p) (xorb (prj 11 p) (xorb (prj 10 p)
            (xorb (prj 8 p) (xorb (prj 6 p) (xorb (prj 4 p) (xorb (prj 3 p)
            (xorb (prj 1 p) (xorb (prj 0 p) Low))))))))))
(Cons (xorb (prj 13 p) (xorb (prj 12 p) (xorb (prj 10 p)
      (xorb (prj 9 p) (xorb (prj 6 p) (xorb (prj 5 p) (xorb (prj 3 p)
      (xorb (prj 2 p) (xorb (prj 0 p) Low)))))))))

(Cons (xorb (prj 15 p) (xorb (prj 14 p) (xorb (prj 10 p) (xorb (prj 9 p)
      (xorb (prj 8 p) (xorb (prj 7 p) (xorb (prj 3 p) (xorb (prj 2 p)
      (xorb (prj 1 p) Low)))))))))

(Cons (xorb (prj 10 p) (xorb (prj 9 p) (xorb (prj 8 p) (xorb (prj 7 p)
      (xorb (prj 6 p) (xorb (prj 5 p) (xorb (prj 4 p) Low)))))))

(Cons (xorb (prj 15 p) (xorb (prj 14 p) (xorb (prj 13 p)
      (xorb (prj 12 p) (xorb (prj 11 p) Low)))))
Null)))))))))))))))))))))
```

Figure 6.3: Residualized Hamming encoder for 16-bit words

included to provide a sample of the generated output. The complete output for even a simple function is often too lengthy for human consumption.

6.2 FIFO-style Message Service

The `encoder` function defines a data transformation that takes an input bitvector, expressed as a list of bits, and generates an output bitvector that includes the Hamming encoding of the input. The `encoder` function achieves a great deal of data parallelism because it can perform the various parity bit computations concurrently. One way to utilize the encoder function within a system is to simply use function composition. For example, if the waveform requirements included Hamming encoding followed by data compression, then assuming a generic `compress` function exists, the combined form could be defined by composing `compress` with `encoder`.

```
(define (encodeAndCompress (bv (List Bit)) (List Bit))
  (compress (encoder bv)))
```

Because the `compress` and `encoder` functions operate sequentially, there is no concurrency across the function blocks. Recall that the Hamming encoding operation is typically performed on fixed-width blocks of a data stream, rather than the entire data stream. Segmenting the input data stream allows the `compress` and `encoder` pipeline to regain parallelism by pipelining the two functions. When the encoder is calculating the $n+1$ data segment, the compression function can be processing the encoder output for segment n.

Arranging these two functions in a pipeline requires lifting each into its own independent thread, then using a first-in-first-out (FIFO) Oread service to handle the coordination of data between the two components. To lift the `encoder` function into a stream-processing thread, we simply wrap it with a pair of **`signal`** calls, one invoked

```
Bit xorb (Bit a,Bit b)
  {Bit var0 = a ;
   switch ((int)((var0) [0]))
     {case 1 :
        {return (notb ((Bit)(b)));}
      case 0 :
        {return (b);}}}
Bit notb (Bit a)
  {Bit var0 = a ;
   switch ((int)((var0) [0]))
     {case 1 :
        {return (&tag_Low);}
      case 0 :
        {return (&tag_High);}}}
void** hamming (void** p)
  {void** var0 = (void**)(malloc (88)) ;
   ((var0) [0]) = ((PTR)(0));
   // Similar assignments removed
   ((var0) [16]) = ((PTR)((p) [16]));


   // Calculation of the parity bits
   ((var0) [18]) = ((PTR)(xorb ((Bit)((p) [14]),(Bit)(xorb ((Bit)((p)
      [13]),(Bit)(xorb ((Bit)((p) [11]),(Bit)(xorb ((Bit)((p) [10]),(Bit)
      (xorb((Bit)((p) [7]),(Bit)(xorb ((Bit)((p) [6]),(Bi t)(xorb ((Bit)((p)
      [4]),(Bit)(xorb ((Bit)((p) [3]),(Bit)(xorb ((Bit)((p)
      [1]),(Bit)(&tag_Low))))))))))))))))))));

   ((var0) [19]) = ((PTR)(xorb ((Bit)((p) [16]),(Bit)(xorb ((Bit)((p)
      [15]),(Bit)(xorb ((Bit)((p) [11]),(Bit)(xorb ((Bit)((p) [10]),(Bit)
      (xorb ((Bit)((p) [9]),(Bit)(xorb ((Bit)((p) [8]),(Bit)
      (xorb ((Bit)((p) [4]),(Bit)(xorb ((Bit)((p) [3]),(Bit)
      (xorb ((Bit)((p) [2]),(Bit)(&tag_Low))))))))))))))))))));

   return (var0);}
```

Figure 6.4: 16-bit Hamming encoder C compilation output

```
entity hamming is
  port (hamming_din : in std_logic_vector(15 downto 0);
        hamming_cin : in std_logic; hamming_cout : out std_logic;
        hamming_dout : out std_logic_vector(19 downto 0);
        clk : in std_logic; rst : in std_logic);
end entity hamming;
architecture str of hamming is
signal sigResult0 : std_logic_vector(0 downto 0);
signal sigDone0 : std_logic_vector(39 downto 0);
signal sigCtl0 : std_logic;
-- signal declarations removed
begin
-- shared XOR function block. Majority of port mappings removed for clarity.
  xorb : entity funblock
    generic map(C_NUMBER_CALLS => 40,C_ARG_WIDTH => 2,
                C_SCHED_WIDTH => 6,C_RESULT_WIDTH => 1)
    port map(args(1 downto 0) => sig160,args(3 downto 2) => sig158,...
            cin(0) => sigCtl163,cin(1) => sigCtl164,cin(2) => sigCtl165,...
            cout(0) => sigCtlIn161,cout(1) => sigCtlIn159,cout(2) =>
            sigCtlIn157,dout => sigResult0,body_start => sigCtl0,
            body_done => sigCtl12, body_args => sig0,body_result => sig12,
            clk => clk,rst => rst);
  hamming : entity funblock
    generic map(C_NUMBER_CALLS => 1,C_ARG_WIDTH => 16,
                C_SCHED_WIDTH => 1,C_RESULT_WIDTH => 20)
    port map(args(15 downto 0) => sig200,cin(0) => sigCtl200,
             cout(0) => sigCtlIn201,dout => sigResult2,
             body_start => sigCtl2,body_done => sigCtl198,
             body_args => sig2,body_result => sig162,clk => clk,rst => rst);
  sig56 <= sig42&sig55;
  inst57 : entity callLatch
    generic map(C_RESULT_WIDTH => 1)
    port map(din => sigResult0,cin => sigCtlIn57,dout => sig57,
             cout => sigCtl57,clk => clk,rst => rst);
  inst163 : entity barrier
    generic map(C_CIN_WIDTH => 2)
    port map(cin(0) => sigCtl2,cin(1) => sigCtl159,cout => sigCtl163,
            clk => clk,rst => rst);
  sig200 <= hamming_din;
  sigCtl200 <= hamming_cin;
  hamming_dout <= sigResult2;
  hamming_cout <= sigCtlIn201;
end architecture;
```

Figure 6.5: 16-bit Hamming encoder VHDL compilation output

on the input stream, and one invoked on the encoder output, which will be subsequently used as the input for the compression thread.

The `Msg` data type encodes the protocol for the FIFO service. When a thread dequeues data from the service, it issues a `ReqRecv` request and blocks until it gets a `RspRecv` response containing the head of the FIFO . Similarly, when the thread wishes to enqueue data, it issues a `ReqSend` request containing the data to enqueue, and gets a `RspAck` in response.

```
(data Msg [a]  (ReqRecv)
               (ReqSend a)
               (RspAck)
               (RspRecv a)
```

The `encodeThread` function issues a `ReqRecv` request on `@instream`, calls `encoder` on the resulting value, and issues a `ReqSend` request with the encoded value on `@outstrm`.

```
(define (encodeThread
 (monad [(@instrm (React (Msg (List Bit))))
         (@outstrm (React (Msg (List Bit) )))] Unit))
  (do (i <- (signal @instrm ReqRecv))
      (case i
       ((RspRecv x)
        (do (signal @outstrm (encoder x))
            (encodeThread [@instrm @outstrm]))))))
```

Likewise, the `compressThread` reads from an input stream, modeled as an Oread reactive monad instance, and writes to an output stream.

```
(define (compressThread
 (monad [(@instrm (React (Msg (List Bit))))
         (@outstrm (React (Msg (List Bit) )))] Unit))
  (do (i <- (signal @instrm ReqRecv))
      (case i
       ((RspRecv x)
        (do (signal @outstrm (compress x))
            (encodeThread [@instrm @outstrm]))))))
```

Having lifted the `encoder` and `compress` functions into the Oread concurrency model, all that remains is to define the FIFO service and then connect the encoder and compression threads in an Oread configuration. An Oread service requires both a handler, for interpreting requests and generating responses, and a scheduler for indicating that a response is pending.

The FIFO service uses a state monad instance to manage the protocol. This state has two elements: the first is a `Maybe` that indicates whether there is a pending request that could not be satisfied because there is not a matching request from the other thread. The second element of the state is a ready-to-run queue containing a list of pending responses. For both elements of the `KernelState` type, the service uses a `AThread` type that holds a thread id, as an `Int`, and a message that is the request.

```
(data AThread [a]   (AThread Int (Msg a)))
(data KernelState [a] (KS  (Maybe (AThread a)) (List (AThread a))))
```

The `dequeue` function serves as the scheduler for the FIFO service. It simply drains the ready-to-run queue.

```
(define (dequeue (monad [(@kstate (state Int (KernelState a)))]
                        (Maybe (AThread a))))
 (do  (ks <- (get @kstate 0))
      (case ks
        ((KS wait rtr)
           (case rtr
             ((Null) (return Nothing))
             ((Cons r rest) (do (put @kstate 0 (KS wait rest))
                                (return (Just r))))))))))
```

The `msgHandler` function dispatches requests to one of two handler functions. Each of these functions checks to see if there is a pending request from the matching thread. If so, then it calls `scheduleSendRecv` which will enqueue both responses into the ready-to-run queue. Figure 6.6 shows the definition for the handler functions and `scheduleSendRecv`.

Finally, a `configuration` connects the compression and encoder threads using the FIFO service. The configuration assumes that there are two additional threads, `src` that produces raw input, and `sink` that consumes encoded and compressed output.

```
(configuration waveform

  (service @src2encoder (Msg (List Bit)) msgHandler dequeue)
  (service @encoder2compress  (Msg (List Bit)) msgHandler dequeue)
  (service @compress2sink  (Msg (List Bit)) msgHandler dequeue)

  (thread (src [@src2encoder]))
  (thread (sink [@compress2sink]))
  (thread (encoder [@src2encoder @encoder2compress]))
  (thread (compress [@encoder2compress @compress2sink]))
  )
```

6.3 Summary

Using Oread, we implemented an archetypal data processing component that calculates the Hamming encoding of an input bit stream. The implementation is constructed using the pure functional subset of Oread, allowing program transformations which exploit the parallelism inherent in the Hamming coding algorithm. These transformations allow a system implementer to explore various space and time trade-offs possible in an implementation fabric.

The Hamming encoder component can be included as part of a larger system that allows for pipelined concurrency by lifting the pure functional implementation to use the Oread reactive concurrency model. Input and output data streams are modeled as reactive monad instances, and reading from or writing to these streams is performed by

```
(define (msgHandler (tid Int) (msg (Msg a))
  (monad [(@kstate (state Int (KernelState a)))] Unit))
    (case msg
      ((ReqSend value) (handleSend tid msg [@kstate]))
      ((ReqRecv) (handleRecv tid msg [@kstate]))
     ))
(define (handleSend (sendTid Int) (msg (Msg a))
          (monad [(@kstate (state Int (KernelState a)))] Unit))
   (do (ks <- (get @kstate 0))
       (case ks
         ((KS wait rtr)
           (case wait
             ((Nothing)
               (do (put @kstate 0 (KS  (Just (AThread sendTid msg)) rtr))
                   (return Unit)))
             ((Just thd)
                (case thd
                  ((AThread tid m)
                   (scheduleSendRecv sendTid msg tid [@kstate])))))))
       (return Unit)))
(define (handleRecv (recvTid Int) (msg (Msg a))
          (monad [(@kstate (state Int (KernelState a)))] Unit))
   (do (ks <- (get @kstate 0))
       (case ks
          ((KS wait rtr)
            (case wait
              ((Nothing)
               (do (put @kstate 0 (KS  (Just (AThread recvTid msg)) rtr))
                   (return Unit)))
              ((Just thd)
               (case thd
                ((AThread tid m)
                 (scheduleSendRecv tid m recvTid [@kstate])))))))
       (return Unit)))
(define (scheduleSendRecv (sendTid Int) (sendMsg (Msg a)) (recvTid Int)
    (monad [(@kstate (state Int (KernelState a)))] Unit))
  (case sendMsg
    ((ReqSend value) (do (put @kstate 0
                              (KS  Nothing
                                   (Cons (AThread sendTid RspAck)
                                   (Cons (AThread recvTid (RspRecv value))
                                  Null))))
                         (return Unit)))))
```

Figure 6.6: FIFO handler functions

invoking the Oread **`signal`** construct on the instance. Inter-thread communication is accomplished using a FIFO service, implemented with a simple handler and scheduler function.

Chapter 7

Conclusions and Future Work

Designing systems that can be targeted towards either software or hardware is necessary to adapt to changing system requirements as well as new implementation technology. Doing this sort of design is complicated by two major factors. The first challenge is that hardware and software exhibit vastly different computational models. The second challenge is that hardware implementations are often defined at a much lower level of abstraction than their associated software counterparts, often incorporating low-level non-functional system constraints such as timing into the functional system models.

The software model of computation is, in modern commercial processors, sequential. The processor supplies a fixed number of computational resources (e.g. ALU and branch units), which the programmer controls via a sequential stream of instructions. Data-independent operations, which could possibly be performed in any order or potentially in parallel, are instead performed in the sequence of the program order. Although a particular program may not need a specific computational resource, the CPU provides a common instruction set and dedicated logic to support that instruction set, regardless of the demands of any given executing program.

In contrast, the hardware model of computation is inherently concurrent. Rather than a limited set of macro-level resources, such as an ALU, the hardware fabric provides a “sea” of lower-level resources such as boolean gates and registers that will, by default, execute concurrently. The system implementer is free to utilize these resources as a specific application demands, but often will have to define intermediate computational resources from primitive components. Moreover, the hardware designer must provide a capability for coordinating the various computational resources.

The most common method for performing coordination on modern hardware fabrics is to use a global system clock. Within this model, all combinational circuitry must have a propagation delay that is less than the clock period. However, as the clock signal is global, so are the timing constraints for all of the components in a system. To localize constraints, additional clocks can be added, at the expense of complicating the interface between components defined over separate clock domains.

This dissertation proposed a functional language, extended with monadic effects for imperative state and concurrency, as an mechanism for reconciling the different models of computation and eliminating the dependency on low-level fabric behavior in the implementation of hardware components. By simplifying the distinction between hardware and software implementations of components for both of these challenges, we can advanced toward a goal of efficiently targeting either hardware or software fabrics.

Oread is the language we have designed to demonstrate this thesis. Because Oread is based on a functional language, it allows a wide range of program transformations that a designer can employ to maximize utilization of a particular target fabric. As chapter 2 describes, most important of these is the fold/unfold transformation which allows a function abstraction to be replaced with its definition, and vice versa. When developing a component, Oread functions can be used to logically structure the function of the component. However, when targeting a specific implementation fabric, functions also serve as a resource structuring mechanism. A function can be duplicated, and calls to that function replaced with calls to the duplicated instances if there are sufficient computational resources available. Conversely, a primitive operation such as floating point multiplication can be encapsulated into a function, allowing the computational resources of that operation to be shared, trading longer execution time for reduced area.

Oread includes a model of concurrency based on message-passing, structured using a reactive monad. Within the concurrency framework provided by Oread, individual components are modeled as threads that communicate with external services using a specific protocol. A thread completely encapsulates the state of the associated top-level function; there is no way for one thread to affect another thread except through a service proxy. The service defines a handler that interprets requests from connected threads, and a scheduler that determines if a response is pending for a prior handled request. The handler and scheduler are written within Oread, allowing complex communication and coordination protocols to be defined within the framework.

Chapter 3 described the compilation of Oread to VHDL. The hardware compilation scheme yields a VHDL netlist, a graph with nodes representing instantiated primitive VHDL blocks and edges representing signals connecting those components. The hardware compilation protocol uses an asynchronous "ready" protocol to coordinate sequential components. A barrier insures that all inputs to a component are ready before control propagates to that component. Shared resources, such as function blocks and monadic instances, exhibit a common structure that includes an arbiter that insures that requests to the component are handled in the proper order.

The hardware compilation routines can only handle a subset of Oread. To minimize the impact of this limitation, we use a series of program-to-program transformations to convert a non-conforming program into a synthesizable program. Most importantly is *defunctionalization*, that allows us to convert a higher-order program into a first-order program. Moreover, a slightly adapted version of defunctionalization allows the Oread compilation scheme to handle functions which reference multiple different monad instances using the same monad instance parameter.

Oread can also be compiled into software, by generating a C program that can then be used as input to a standard C compiler. The C compilation scheme uses a collection of C templates, which are parametrized statements that, when given specific values from the software compilation scheme, yield legal C syntax. C itself does not include a concurrency model, but the Pthreads threading library provides capabilities that support the compilation of the Oread message-based concurrency model. The mapping uses only common shared-state concurrency constructs, and could potentially be adapted to support other threading libraries or to execute on bare metal.

Oread can be used to construct components in our designated target domain, software defined radio. We demonstrated the ability to define a generic Hamming encoding component to perform error detection and correction. Then, using the Oread concurrency model, we define a point-to-point communication protocol within Oread to construct a data pipeline between the Hamming encoder component and a hypothetical compression component.

7.1 Future Work

The work outlined in this dissertation has laid the foundation for a variety of future work directions.

The control protocol described in chapter 3 should be extended to integrate fabric-specific timing information, allowing the compilation scheme to eliminate extraneous control circuitry. Moreover, the implementation currently assumes that there is a global clock; this should be modified to allow clock domain changes at the interface between shared blocks.

The program transformations described in chapter 4 describes briefly the transformation of recursive data structures into equivalent forms that are synthesizable, using specialization techniques. This transformation capability should be extended and formalized, perhaps using the worker/wrapper (Gill and Hutton 2008) transformation.

The software compilation schemes of chapter 5 are admittedly naive. These should be modified to incorporate modern techniques for functional language compilation Appel, as well as to provide standard run-time services. Moreover, the data model for the software scheme utilizes dynamic allocation extensively. Static analyses, such as linear types (Wadler 1990), may allow dynamic allocations to be eliminated.

Oread, as described in this dissertation, attempts to address the challenge of targeting varied implementation targets from a single program source. In an embedded system, it is more likely that a final implementation will include both hardware and software components. Oread can be extended to address the problem of *co-design*: mapping a system into disparate fabrics and synthesizing the interfaces between components in those different fabrics. With this capability, Oread would become a powerful tool for architecture exploration, allowing a system designer to quickly determine the various trade-offs of a given system architecture early in the design cycle, reducing re-engineering costs.

Bibliography

A. Appel. *Compiling with Continuations*. Cambridge University Press, 1992. Cited on 129, 169

L. Augustsson. Cayenne-a language with dependent types. *ACM SIGPLAN Notices*, 34(1):239–250, 1999. Cited on 113

J. Backus. Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, 1978. ISSN 0001-0782. Cited on 25

H. Barendregt. Lambda calculi with types. *Handbook of Logic in Computer Science*, 2:117–309, 1992. Cited on 113

P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: Hardware design in Haskell. In *International Conference on Functional Programming*, pages 174–184, 1998. Cited on 29

E. Blossom. GNU radio: tools for exploring the radio frequency spectrum. *Linux Journal*, 2004(122), 2004. Cited on 149

D. M. Chapiro. *Globally-asynchronous locally-synchronous systems (performance, reliability, digital)*. PhD thesis, Stanford University, 1985. Cited on 100

O. Danvy. Type-directed partial evaluation. In *Annual Symposium on Principles of Programming Languages: Proceedings of the 23 rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, volume 21, pages 242–257. Springer, 1996. Cited on 114

O. Danvy and L. R. Nielsen. Defunctionalization at work. In *PPDP '01: Proceedings of the 3rd ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 162–174, New York, NY, USA, 2001. ACM. Cited on 105

O. Danvy, K. Malmkjaer, and J. Palsberg. Eta-expansion does the trick. *ACM Trans. Program. Lang. Syst.*, 18(6):730–751, 1996. ISSN 0164-0925. Cited on 114

D. Espinosa. *Semantic Lego*. PhD thesis, Columbia University, 1995. Cited on 41

A. Gill and G. Hutton. The worker wrapper transformation. Submitted to the Journal of Functional Programming, January 2008. Cited on 117, 169

R. Hamming. Error detecting and error correcting codes. *Bell System Technical Journal*, 29(2):147–160, 1950. Cited on 151

W. Harrison. Proof abstraction for imperative languages. In *Proceedings of the 4th Asian Symposium on Programming Languages and Systems (APLAS06)*, 2006a. Cited on 32

W. Harrison. The essence of multitasking. In *11th International Conference on Algebraic Methodology and Software Technology (AMAST 2006)*, pages 158–172, July 2006b. Cited on 44, 54

W. L. Harrison and S. N. Kamin. Metacomputation-based compiler architecture. In *Mathematics of Program Construction*, pages 213–229, 2000. Cited on 56

C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–77, 1978. Cited on 26

J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989. Cited on 38

G. Hutton and J. Wright. Calculating an Exceptional Machine. In H.-W. Loidl, editor, *Trends in Functional Programming volume 5*. Intellect, Feb. 2006. Selected papers from the Fifth Symposium on Trends in Functional Programming, Munich, November 2004. Cited on 105

IEEE. *IEEE Standard VHDL Language Reference Manual*. New York, NY, 1994. Cited on 30

Standard Verilog Hardware Description Language Reference Manual. IEEE, New York, NY, 1995. Cited on 30

Standard for information technology - portable operating system interface (POSIX). Shell and utilities. IEEE and The Open Group., 2004. Cited on 16, 130

ISO. The ANSI C standard (C99). Technical report, ISO/IEC, 1999. Cited on 130

T. Johnsson. Lambda lifting: transforming programs to recursive equations. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 190–203, New York, NY, USA, 1985. Springer-Verlag New York, Inc. ISBN 3-387-15975-4. Cited on 110

N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993. Cited on 114

N. D. Jones. An introduction to Partial Evaluation. *ACM Comput. Surv.*, 28(3):480–503, 1996. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/243439.243447. Cited on 29

S. L. P. Jones. Implementing lazy functional languages on stock hardware: The spineless tagless g-machine. *Journal of Functional Programming*, 2:127–202, 1992. Cited on 129

S. Liang and P. Hudak. Modular denotational semantics for compiler construction. In *Programming Languages and Systems – ESOP'96, Proc. 6th European Symposium on Programming, Linköping*, volume 1058, pages 219–234. Springer-Verlag, 1996. Cited on 41

S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In ACM, editor, *Conference record of POPL '95, 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: San Francisco, California, January 22–25, 1995*, pages 333–343, New York, NY, USA, 1995. ACM Press. ISBN 0-89791-692-1. Cited on 41

J. Matthews, B. Cook, and J. Launchbury. Microprocessor Specification in Hawk. In *ICCL '98: International Conference on Computer Languages*, pages 90–101, 1998. Cited on 29

R. Milner. *Communicating and Mobile Systems: The π-calculus*. Cambridge University Press, 1999. Cited on 26

R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (revised)*. MIT Press, Cambridge, MA, 1997. Cited on 37

G. J. Minden, J. Evans, L. Searl, D. DePardo, V. Petty, R. Rajbanshi, T. Newman, Q. Chen, F. Weidling, J. Guffey, D. Datla, B. Barker, M. Peck, B. Cordill, A. 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. Cited on 28

E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Dept. of Computer Science, Edinburgh Univ., 1990. Cited on 26

E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1): 55–92, 1991. Cited on 26

A. Mycroft. The theory and practice of transforming call-by-need into call-by-value. In *Proceedings of the Fourth 'Colloque International sur la Programmation' on International Symposium on Programming*, pages 269–281, London, UK, 1980. Springer-Verlag. ISBN 3-540-09981-6. Cited on 38

A. Mycroft and R. Sharp. A statically allocated parallel functional language. In *ICALP '00: Proceedings of the 27th International Colloquium on Automata, Languages and Programming*, pages 37–48. Springer-Verlag, 2000. ISBN 3-540-67715-1. Cited on 32

A. Mycroft and R. Sharp. Hardware/software co-design using functional languages. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 236–251, 2001a. Cited on 32

A. Mycroft and R. Sharp. Hardware synthesis using SAFL and application to processor design. *Lecture Notes in Computer Science*, 2144:13+, 2001b. Cited on 32

S. Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell. Tutorial at Marktoberdorf Summer School, 2000., March 2002. Cited on 38

S. Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, 2003. Cited on 37

J. C. Reynolds. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 408–423, London, UK, 1974. Springer-Verlag. ISBN 3-540-06859-7. Cited on 37

J. C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher Order Symbol. Comput.*, 11(4):363–397, 1998. ISSN 1388-3690. Cited on 105

R. Sharp. *Higher-Level Hardware Synthesis*. Springer, 2004. Cited on 32

R. Sharp and A. Mycroft. The flash compiler: Efficient circuits from functional specifications. Technical Report tr.2000.3, AT&T Research, 2000. Cited on 32

T. Sheard. A type-directed, on-line partial evaluator for a polymorphic language. In *In Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 22–35. ACM Press, 1997. Cited on 114

T. Sheard and S. Peyton Jones. Template meta-programming for Haskell. *ACM SIGPLAN Notices*, 37(12):60–75, 2002. Cited on 114

J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, MA, USA, 1981. ISBN 0262690764. Cited on 26

W. Swierstra and T. Altenkirch. Beauty in the beast. In *Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, pages 25–36, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-674-5. Cited on 38

W. Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999. Cited on 29, 114

P. Wadler. Linear types can change the world. In *Programming Concepts and Methods*, pages 347–359. North, 1990. Cited on 169

P. Wadler. The essence of functional programming. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, Albequerque, New Mexico, 1992. Cited on 27

P. Wadler. Monads for functional programming. In M. Broy, editor, *Program Design Calculi: Proceedings of the 1992 Marktoberdorf International Summer School*. Springer-Verlag, 1993. Cited on 27, 38

P. Weaver, G. Kimmell, N. Frisby, and P. Alexander. Constructing language processors with algebra combinators. In *GPCE '07: Proceedings of the 6th international conference on Generative programming and component engineering*, pages 155–164, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-855-8. Cited on 115, 133

H. Xi. *Dependent types in practical programming*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1998. Chair-Frank Pfenning. Cited on 114

S. Zhuang, W. Li, J. Carlsson, K. Palmkvist, and L. Wanhammar. An Asynchronous Wrapper with Novel Handshake Circuits for GALS Systems. In *Proceedings of the The IEEE International Conference on Communication Circuits And Systems (ICCCAS'02), Cheungdu, China, August*, 2002. Cited on 100

The Oread external representation

Oread is primarily intended to be an intermediate representation for compiling to multiple targets. However, to simplify the use of the Oread synthesis toolset, as well as to make the toolset independent of any particular concrete representation, we have developed a simple concrete syntax similar to that of Scheme or Lisp.

Table 1 lists the lexical elements of the Oread concrete representation.

`<typeid>`	:=	`<ucase>` `<identifier>`
`<typevar>`	:=	`<lcase>` `<identifier>`
`<consid>`	:=	`<ucase>` `<ident>`$^?$
`<ucase>`	:=	A | .. | Z
`<lcase>`	:=	a | .. | z
`<digit>`	:=	0 | .. | 9
`<letter>`	:=	`<ucase>` | `<lcase>`
`<idchar>`	:=	`<letter>` | `<digit>` | `<_>`
`<number>`	:=	**-**$^?$ `<integer>` .`<integer>`$^?$
`<integer>`	:=	`<digit>` `<digit>`*
`<identifier>`	:=	`<lcase>` `<idchar>`*
`<bvlit>`	:=	**0b**`<bit>`$^+$
`<bit>`	:=	**0** | **1**
`<monadinstance>`	:=	@`<identifier>`

Table 1: Oread external representation lexical elements

Table 2 lists the BNF describing the Oread expression language, as well as the productions for defining algebraic data types and top-level functions.

Table 3 lists the grammar for defining Oread configuration directives, used for describing as system as a collection of components and the interconnections between those components.

<def>	:=	<datadef> | <typedef> | <fundef> | <configdef>
<datadef>	:=	(**data** <typeid> <typevarlist>$^{?}$ (<constructordef>$^{+}$))
<typevarlist>	:=	[<typevar>$^{+}$]
<constructordef>	:=	(<consid> <type>*)
<typedef>	:=	(**type** <typeid> <type>)
<type>	:=	<typeid> | <typevar> | **int** | (***** <type>**+**) | ($\rightarrow$ <type> <type>**+**) | (<typeid> <type>**+**) | (**bits** <integer>) | (**monad** <monadcstlist> <type>)
<fundef>	:=	(**define** (<ident> <arglist>$^{?}$ <range>) <expr>)
<arglist>	:=	(<arg>$^{+}$)
<arg>	:=	(<ident> <type>)
<range>	:=	<type> | <monadrange>
<monadrange>	:=	(**monad** <monadparamlist>$^{?}$ <type>)
<monadparamlist>	:=	[<monadparam>$^{+}$]
<monadparam>	:=	(<monadinst> <monadcst>)
<monadcstlist>	:=	(<monadcst>$^{+}$)
<monadcst>	:=	(**state** <type> <type>) | (**react** <type>)
<monadinstlist>	:=	[<monadinst>$^{+}$]
<expr>	:=	<number> | <bvlit> | <ident> | <consid> | <app> | <caseexpr> | <dostmt> | <monadexpr>
<app>	:=	(<op> <expr>*) | (<expr>$^{+}$ <monadinstlist>$^{?}$)
<op>	:=	**tuple** | **prj** | **shl** | **shr** | **rotl** | **rotl** **selectbit** | **setbit** | **and** | **or** | **xor** | **not**
<caseexpr>	:=	(**case** <expr> (<alt>$^{+}$)
<alt>	:=	(<pattern> <expr>)
<pattern>	:=	(<consid> <ident>*)
<monadexpr>	:=	(**put** <monadinst> <expr> <expr>) | (**get** <monadinst> <expr>) | (**signal** <monadinst> <expr>)
<dostmt>	:=	(**do** <stmt>$^{+}$)
<stmt>	:=	(**return** <expr>) | (<ident> $\leftarrow$ <expr>) | <expr>

Table 2: Oread external representation grammar

`<configdef>`	:=	(**configuration** `<ident>` `<mem>`* `<serv>`* `<thd>`*)
`<mem>`	:=	(**memory** `<monadinst>` `<type>` `<type>` `<expr>`$^?$)
`<serv>`	:=	(**service** `<monadinst>` `<type>` `<expr>` `<expr>` `<expr>`)
`<thd>`	:=	(**thread** `<expr>`)

Table 3: Configuration grammar