Thesis for The Degree of Licentiate of Engineering

Designing a Functional Programming Architecture for the Internet of Things

Jeremy Pope

Division of Computing Science
Department of Computer Science & Engineering
Chalmers University of Technology | University of Gothenburg
Gothenburg, Sweden, 2022

Designing a Functional Programming Architecture for the Internet of Things

Jeremy Pope

Department of Computer Science & Engineering
Division of Computing Science
Chalmers University of Technology and Gothenburg University
Gothenburg, Sweden

This thesis has been prepared using LaTeX.
Printed by Chalmers Digitaltryck,
Gothenburg, Sweden 2022.

“It’s just a small tool to help with the design; it’ll be done in a couple weeks.”
- Jeremy

Abstract

As the Internet of Things (IoT) grows, so too do security concerns: as well as typically having access to sensors and actuators, IoT devices are often programmed using bug-prone, low-level languages. Such a combination results in vulnerabilities that pose risks to privacy and safety.

This thesis aims to address this by making it possible to run high-level functional programs on IoT devices, a daunting prospect with traditional hardware due to the overheads of functional programming runtimes. To accomplish this, an architecture and partial implementation of a "natively functional" processor for IoT, named Cephalopode, is presented. The processor performs both graph reduction and garbage collection directly, without requiring an expensive software runtime.

Implementing Cephalopode raised several opportunities for improving the process of hardware design. To that end, this thesis presents the finite state machine editor Stately and the high-level language Bifröst. Stately raises the level of abstraction of finite state machines enough to avoid a proliferation of edges during design, while maintaining efficiency and low-level control. Bifröst offers a higher-level approach to hardware design, allowing complex algorithmic processes—in particular those that communicate extensively with other components—to be described in an imperative language and compiled to an RTL-level circuit model.

Keywords

Functional Programming, Internet of Things, Architectures, Hardware Description Languages, High-Level Synthesis

Acknowledgments

This work was partially supported by the Wallenberg Autonomous Systems and Software Program (WASP) funded by Knut and Alice Wallenberg Foundation and by a grant from the Swedish Foundation for Strategic Research (SSF) under the project Octopi (Ref. RIT17-0023).

Appreciation goes to the developers and engineers who have inadvertently provided the tools for this work, including: the GNU Project, the Linux Foundation, the Haskell community, Mozilla, Bram Moolenaar, Sun Microsystems, P. G. Krolmeister, and G. Mahler.

I would like to thank my supervisor Carl Seger for his knowledge, guidance, patience, vehicular assistance, and faith in my detours. Likewise my co-supervisor Mary Sheeran, for her support and ideas about broader applications of my research. Thanks goes to my pals at Chalmers, among them: my original office mates Agustín, Nachi, and Matti; my MSc co-conspirator Irene; the dynamic duo of Benjamin and Alexander; the philosophical Ivan; the irreverant Max; the good people of Octopi; and the wider FP, ISec, and FM rabbles.

And finally, I give my endless thanks and love to my dear friends and family all around the world, near and far, who brighten my life and give me boundless energy!

List of Publications

Appended publications

This thesis is based on the following publications:

[A] Jeremy Pope, Jules Saget, Carl-Johan H. Seger "Cephalopode: A Custom Processor Aimed at Functional Language Execution for IoT Devices"
MEMOCODE, 2020.

[B] Jeremy Pope, Jules Saget, Carl-Johan H. Seger "Stately: An FSM Design Tool"
MEMOCODE, 2020.

[C] Dorian Lesbre, Jeremy Pope, Carl-Johan H. Seger "Designing an Arbitrary-Precision ALU for an IoT Processor"
Under submission.

Research Contribution

For Paper A, the general Cephalopode architecture was jointly designed by myself and Carl Seger. I was responsible for the design of the snapshot memory, and both the design and implementation of the core of the reduction engine.

For Paper B, I did the majority of the design work for Stately (with input and feedback from Carl Seger and Jules Saget), and all of its implementation.

For Paper C, I designed the Bifröst language with substantial input and feedback from Carl Seger and Dorian Lesbre, and implemented its compiler.

Contents

Chapter 1

Introduction

1.1 Background

1.1.1 IoT

The Internet of Things (IoT) is a quickly growing network of interconnected devices. It is distinct from the traditional internet in that the devices are not communicating at the behest of humans (e.g. a web browser on a personal computer), but rather of their own accord as appliances (e.g. a toaster). Example IoT appliances include "smart" versions of lightbulbs, coffee makers, speakers, door locks, smoke alarms, and pet feeders; as well as inherently smart technologies such as voice assistants.

IoT devices often interact with the physical world—their utility as networked appliances would be questionable otherwise—by way of sensors or actuators. This puts them in a position of comparative security importance: if taken over by an attacker, they may reveal sensitive information or allow the attacker to disrupt the physical world in ways that can cause damage or harm.

In the interests of power, size, and cost savings, IoT devices typically use highly resource-constrained microcontrollers. An unfortunate side-effect of this is that it encourages the use of low-level languages such as C and C++ to program the devices. These languages typically lack any form of memory safety, and as a result security bugs abound [1–3].

Additionally, low-level languages offer little abstraction, making a variety of conceptually simple tasks difficult to implement. A lack of automatic memory management further adds to program complexity and introduces another avenue for bugs, use-after-free vulnerabilities [4]. Fixed-sized integers result in overflow in arithmetic operations (e.g. $255 + 1 = 0$), which many languages do not detect and can have security implications [3].

Several approaches have been taken to address the IoT endpoint security issues described above. One possibility is to continue to use low-level languages, but accompany their use with extensive testing or formal verification. Neither is without drawbacks: the former rarely gives a complete guarantee of correctness, and the latter is typically time-consuming or even infeasible. Furthermore, they do not address the difficulty of writing complex software in a low-level language in the first place.

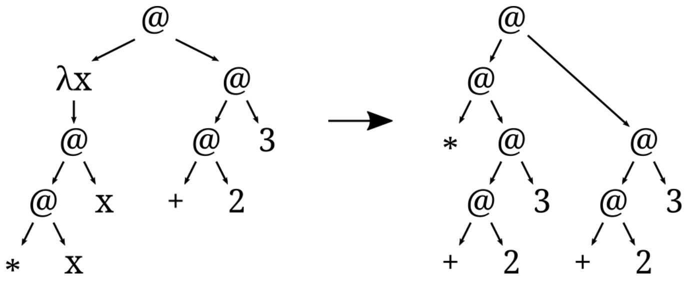


Figure 1.1: Tree reduction step.

1.1.2 Functional programming

An alternative approach is to use a higher-level language. Functional programming languages in particular offer several advantages over other paradigms [5]. Higher-order functions allow concepts such as composition of functions and traversing data structures to be represented in a concise and reusable manner. Data flows are explicit, allowing one to more easily reason about functions' behavior. Combining these with a suitably expressive type system (such as that of Haskell [6]), meaningful security properties can be enforced at compile-time [7]. It will be assumed that the reader has familiarity with functional languages and the basics of the λ-calculus.

In this thesis we exclusively consider functional languages with *non-strict* semantics, where sub-expressions are not evaluated unless needed. This approach offers several benefits over the more typical *strict* semantics, where all sub-expressions are evaluated even if they are not necessary for the final expression's value. First, non-strict semantics only expands the set of programs that produce a result; the set of programs that terminate under non-strict semantics is a superset of those that do so under strict semantics. Second, a concrete example of the previous: under non-strict semantics it is possible to manipulate infinite data structures such as streams (infinite lists), provided that only a finite portion is ever consumed, simplifying producer-consumer relationships [8]. Third, non-strict semantics alleviates the need to meticulously manage control flow within functions, as only necessary sub-expressions (in particular let-bindings) will be computed.

One way to implement a non-strict functional language would be to represent a program (in effect, a large lambda expression) as a syntax tree, and evaluate it in "normal order": functions are applied prior to reducing their arguments. This would produce duplicate work when unevaluated arguments are substituted into functions: $(\lambda x.x{*}x)(2{+}3)$, as seen in Fig. 1.1, would first reduce to the expression $(2{+}3){*}(2{+}3)$, after which subsequent reductions would compute the sum $2 + 3$ twice. From a performance standpoint this is quite poor compared to a strict approach, where $2+3$ could be reduced to 5 first, and only then substituted into $x * x$. (A careful reader may also notice that arithmetic operations such as addition and multiplication—unlike functions defined by λ abstraction—*do* in fact require their arguments to be evaluated first. This detail will be revisited later.)

If the program is instead represented as a *graph*—rather than a tree—then it becomes possible for a single sub-expression to be referenced in multiple places. Consider the effect on the previous example: rather than creating two separate copies of the argument $2 + 3$, with a graph it is possible to instead create two *references*

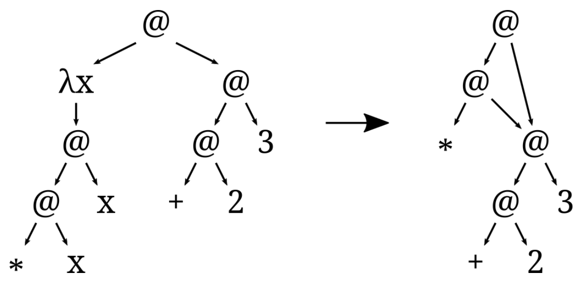


Figure 1.2: Graph reduction step.

to a single instance of $2 + 3$, as shown in Fig. 1.2. This avoids duplication of data, but in order to avoid duplication of work a further modification is made: when a sub-expression is evaluated (e.g. $2 + 3$ is reduced to 5), the original sub-expression is overwritten *in place* with the resulting value. This overwriting step makes the result available to all references to the sub-expression, meaning that its value will not be unnecessarily recomputed, avoiding duplication of work. This approach to lazy evaluation, first described by Wadsworth [9], is called *graph reduction* and is the basis for many (if not all) lazy functional programming runtimes.

Graph reduction comes in several flavors, with different tradeoffs. A naïve approach, where lambda expressions are represented directly in the graph, has difficulties with function application and recursion due to the need to copy and instantiate function bodies. The Spineless Tagless G-Machine [10] (currently used by the Glasgow Haskell Compiler) instead keeps track of variable bindings using closures, rather than performing substitution directly, which adds complexity but is well-suited for traditional processors. Alternatively, one can eliminate free variables altogether by transforming a program into one where all functions are at the top level (effectively replacing closures with explicit function application), as is the case with the Super-combinator [11] technique and lambda lifting in the original G-Machine [12].

1.1.3 Combinators

Combinatory logic, developed by Schönfinkel [13] and Curry [14] and used in a programming context by Turner [15], allows one to take the elimination of variables to an extreme: entire programs can be transformed into applications of a fixed set of combinators (well-behaved functions with no free variables), at the possible expense of program size. A simple example is the SKI combinator calclulus, which introduces the combinators S, K, and I that behave as follows:[1]

$$\begin{aligned} \mathrm{S}\,x\,y\,z &\mapsto (x\,z)\,(y\,z) \\ \mathrm{K}\,y\,z &\mapsto y \\ \mathrm{I}\,z &\mapsto z \end{aligned}$$

While initially perhaps a bit cryptic, the choice of these combinators makes more sense when one considers that a function $\lambda z.e$ ultimately has the job of taking an argument and placing it in some location(s) in the expression e. Lambda abstractions

[1]In the interest of clarity, the choice of variables departs from the standard presentation.

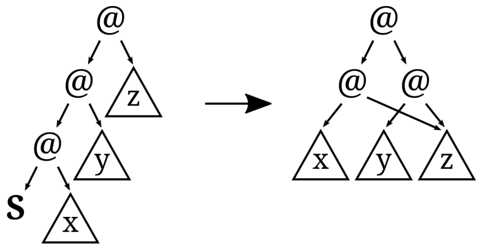


Figure 1.3: Graph reduction with the S combinator.

use variables (e.g. z) to indicate this directly, whereas combinators accomplish it by handing the argument down the expression tree: $\mathrm{S}\,x\,y$ takes an argument z and sends it to both branches x and y, $\mathrm{K}\,y$ discards the argument z rather than passing it down further, and I places the argument z as a leaf. This insight leads to a recursive algorithm for translating an arbitrary lambda expression into an expression consisting solely of S, K, I, application, and variables that were free in the original expression, if any—no more lambda abstractions. Graph reduction can then be carried out on this transformed program, using the definitions of S, K, and I; an example is shown in Fig. 1.3. The lack of variable name bindings and the comparatively local nature of combinator reductions (versus the substitution used in β-reduction) makes this a simple and attractive approach, especially for hardware implementation.

Less attractive, however, is the potentially exponential growth in size of the original program when transformed to use the SKI combinators [16]. Likewise the enormous number of reductions required to accomplish even a simple rearrangement of arguments. So, it is common for the set of combinators to be expanded into a more rich set [15–18], offering better guarantees about growth [19] and decent performance in practice.

1.1.4 Hardware for functional programs

The 1980s brought several software approaches to running functional programs on traditional architectures (abstract machines such as TIM [20] and TIGRE [21]). At the same time, though, special-purpose computers were also being developed to carry out graph reduction natively. Examples include the SKIM machines [22] and NORMA [23], microcoded combinator graph reduction machines.

Interest apparently waned thereafter, likely due to the tendency for software implementations on traditional processors (which were riding Moore's law to greater and greater speeds) to quickly outstrip FP-oriented architectures.

As Moore's law ceases to bear fruit and attention turns toward low-power computing and special-purpose "accelerators", renewed interest in FP architectures is evident. The Reduceron [24] takes advantage of parallelism to bring high-speed graph reduction to FPGAs. PilGRIM [25] uses a complex but performant model similar to STG [10], and aims to be deeply pipelined.

A recent development in architectures for functional programs is the Lambda-One platform [18], which includes a combinator-based graph reduction engine and a memory management unit that performs garbage collection . While the reduction engine appears quite efficient, garbage collection requires halting the running program. If the entirety of the graph resides in the 64KiB on-chip memory then the latency is quite small (approximately 1.64 microseconds, according to the authors), however it is unlikely that the 512MB external DRAM—being 8,000 times larger, and off-chip—can be collected without a more significant delay. Power consumption is also not clear with respect to IoT applications.

1.1.5 Garbage collection

As mentioned previously, high-level languages typically use some form of automatic garbage collection to spare the programmer from error-prone manual memory management. On an ordinary processor, this garbage collection is implemented in software.

Simple algorithms such as mark-and-sweep or Cheney's algorithm [26] require the program to be paused until garbage collection is complete, possibly introducing a significant delay. More complex algorithms such as generational garbage collection [27] can reduce the time spent on objects likely to be still in use, but require complex bookkeeping.

Yuasa [28] proposes an incremental garbage collection algorithm, spreading the work more evenly over time. However, it comes with a non-negligible overhead when updating an object during the marking phase, due to the need to save old pointers for later exploration. In general, garbage collection algorithms designed for incremental or parallel use need to perform special behavior in response to some read and write operations made by the program, in order to maintain certain invariants (e.g. black-gray-white). This necessitates read- and write-barriers, which by nature are costly to implement in software.

Hardware acceleration of garbage collection can help decrease costs, both for concurrent and stop-the-world approaches. The Integrated Hardware Garbage Collector [29] extends a traditional processor with a hardware garbage collection system that runs concurrently to the main processor, avoiding delays and wasted clock cycles. Azul Systems' C4 [30] is a concurrent, compacting, generational garbage collector for use with the Java Virtual Machine, and has both hardware-accelerated and pure software implementations. Maas [31] presents a hardware garbage collector that requires no changes to the processor, but requires pausing computation while it runs (a concurrent approach is proposed but not implemented). The previously mentioned graph reduction processors SKIM [22], NORMA [23], Reduceron [24], PilGRIM [25], and Lambda One [18] all perform garbage collection, but (with the exception of PilGRIM where details are not given) require pausing computation during the marking phase.

1.1.6 Hardware design

Even with an architecture in mind, implementing hardware for it is a substantial task in itself, and has driven the creation of many tools and languages to assist in this process.

Embedded domain-specific languages (EDSLs) based on functional languages, such as Lava [32] and Chisel [33], make use of libraries and higher-order functions to allow an elegant structural description of hardware. This lends itself to combinational circuits, or relatively orderly sequential ones, but is not as well suited to describing complex "algorithmic" processes (where a more behavioral description is preferred).

CλaSH [34] allows hardware to be described *as* a subset of Haskell, rather than merely *within* Haskell (e.g. via an EDSL). This allows the use of Haskell constructs that are not normally available for describing the behavior of a circuit, such as pattern matching and case expressions. An awkward aspect is that state must be passed into functions and returned explicitly, complicating descriptions and introducing a possible avenue for error.

TL-Verilog offers a pragmatic abstraction over Verilog, in particular adding mechanisms for pipelining and tracking data validity. These features make it well-suited for high-performance, pipelined systems, but less so for implementing complex serial processes.

High-level synthesis tools such as Vivado HLS allow one to write algorithms in a familiar and comparatively abstract way (C++), and generate hardware from this code. Such tools typically come from the commercial sector and are difficult to evaluate fully without a license, however they seem to share the trait that they are designed for implementing a single unit, with comparatively shallow interaction with other hardware. Function calls, for example, seem to always be "downward" to child RTL blocks compiled from other functions in the unit, rather than horizontally, or out to external components. This restricts the high-level abstraction of function calls to internal use only, limiting composition.

Bluespec [35] offers an interesting way of describing hardware, where modules expose interfaces called "methods" which are invoked via "atomic actions". Atomic actions are exactly that—they only run when all requisite methods are ready, and when they run they do so instantaneously. This approach of driving transactions using rules offers a very elegant way to describe hardware, particularly pipelined systems. A downside is that describing a complex process (e.g. division of arbitrary-precision integers) requires breaking it across many rules, obfuscating the algorithm and likely requiring some manual tracking of control flow.

The statecharts [36] formalism allows a behavior specification similar in nature to finite state machines (e.g. Moore [37] and Mealy [38] machines), but far more powerful in practice due to hierarchical organization, history, and orthogonal state constructs. They are visually friendly, and well-suited to synthesis [39]. Their semantics are complicated, however, due to the ability to have orthogonality (multiple simultaneous states) on several levels of the machine.

1.2 Contributions

This thesis presents three main contributions toward IoT security and embedded hardware design. The first is Cephalopode, an IoT-oriented processor for running functional programs through graph reduction. The second and third are tools created to help with the design and implementation of Cephalopode: Stately, a finite state machine editor, and Bifröst, a high-level language that can be compiled to hardware.

1.2.1 Cephalopode

Graph reduction

Cephalopode is an microprocessor for running functional programs on IoT devices via combinator graph reduction. The latter means that in contrast to traditional processors, Cephalopode represents a program as a graph in memory (rather than

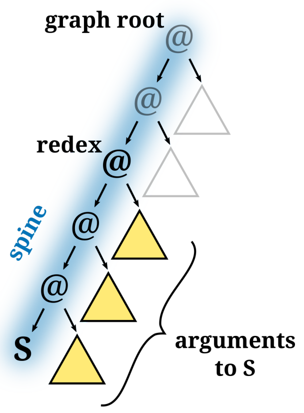


Figure 1.4: Anatomy of a graph.

a sequence of instructions), and the hardware iteratively applies reduction rules to this graph.

By taking this approach, there is no need for any software runtime system to manage the graph reduction process. Instead, the hardware "speaks" graph reduction natively, eliminating a layer that is costly in terms of memory and computation. Futhermore, the hardware for combinator graph reduction is relatively simple, as there is no need to support different addressing modes, operand sizes, or any of the other complications that traditional architectures are usually burdened with.

Central to the process of reduction is the notion of the "spine" of a graph; this is the chain of nodes that represents the outermost function application, running from the root of the graph down and to the left until a non-application node is reached. Since normal order evaluation is used, reduction takes place at the tip of the spine, where the outermost function (e.g. a combinator) is applied. Typically a combinator will only consume some of the arguments on the spine, leaving the remainder to be used by the resulting expression. The combinator's arity therefore determines how many arguments are required; these are obtained by walking back up the spine and collecting the right-hand children. The application node that supplies the last argument used by the combinator serves as the root of the sub-expression being reduced, and is known as the "redex". It is this node that is overwritten when the reduction is carried out. Fig. 1.4 shows the spine and redex of an example graph.

Cephalopode's reduction engine maintains a stack as it walks down the spine, and once it reaches the tip it simply unwinds the stack as necessary to obtain the arguments and redex. A decoder is used to determine how many arguments are to be consumed (the combinator's arity), and the reduction engine pops them from the stack. The decoder also determines how many new nodes are required, and the reduction engine allocates them. A unit dedicated to combinator reductions takes in the combinator and the addresses of the redex, arguments, and new nodes, and uses these to compute an updated value for the redex node as well as values for all of the newly-allocated nodes.

By delegating the reduction rules to a separate "combinator unit" it becomes very easy to add combinators: only the simple, combinational logic in this unit and the decoder need to be changed, not the main reduction state machine. In fact, through the benefits of FL both the combinator unit and decoder take combinator definitions from a separate file, meaning that adding a combinator requires only two or three lines of code. This will facilitate experimenting with different sets of combinators: currently twelve combinators are supported (among them the usual suspects S, K, I, and Y), and as realistic IoT programs are profiled various combinators will be added or replaced.

Primitive functions

In addition to combinators, there are also "primitive functions" that can be applied. These consist of arithmetic, comparisons, if-then-else, and (as the processor is developed further) list manipulation and input-output functions. Many of these present a difficulty, however: they are *strict*, in that they require their arguments to have already be evaluated to some degree. Addition, for example, cannot be carried out until both arguments are evaluated fully to integers. Normally this would require recursively evaluating arguments, requiring another stack (or special markings in the original stack) to keep track of the different layers of evaluation.

To avoid this complication, Cephalopode uses a trick seen in the Reduceron [24]: if a value (integer, list, etc) is applied as if it were a function, then it is swapped with its first argument. This rule, the "swap rule", is written as $v\, f \mapsto f\, v$. The rule gives the processor strictness for free: arguments can be placed on the left, causing the reduction process to evaluate them and only swap them into their correct positions thereafter. As an example, consider a unary primitive function NEGATE that negates an integer. If it were applied to a non-trivial expression e, reduction of NEGATE e would fail since e is not yet an integer. However if the application were instead e NEGATE, then reduction would first take place in e, eventually resulting in v NEGATE (where v is the integer value produced by evaluating e), which the swap rule would rearrange to the more successful NEGATE v. The same trick can be used for primitive functions of higher arity, for example to compute the sum $e_1 + e_2$ one would use the expression e_2 (e_1 ADD). This translation can be done in a software library or compiler to spare the programmer from needing to consider these implementation details.

The reduction engine handles primitive functions in a manner similar to combinators, in terms of using the decoder to determine their arity and the number of new nodes to allocate, and delegating the remaining work to a dedicated "primitive function unit". Because the primitive functions are more dynamic in nature than combinators, though, the modules that perform these computations tend to be state machines that interact with main memory (rather than combinational logic, as was the case for combinator reduction rules).

Arithmetic

Arithmetic in Cephalopode is arbitrary-precision: integers are stored as linked lists of word-sized "chunks", allowing them grow as needed to store large numbers. Barring memory exhaustion, this eliminates integer overflows, avoiding bugs and removing the need to handle overflows at runtime or prove them impossible through static analysis.

It also means that the processor can use a word size closer to the average case, rather than the worst case: if almost all values are 8 bits, then this can be used without producing erroneous results if an occasional 9- or 10-bit value is needed. Related to this, the programmer does not need to be aware of the word size, except perhaps when fine-tuning performance.

The arbitrary-precision arithmetic logic unit (ALU, described in 4) supports comparison, addition, subtraction, multiplication, and division/mod. An integer square root operation is planned but not yet complete.

Memory management

Since Cephalopode performs graph reduction directly (as opposed to using a software runtime system), it also needs to perform memory management duties, such as allocation and garbage collection. As graph nodes are uniformly-sized, there is no risk of memory fragmentation and allocation simply becomes a matter of finding any unused memory address. Garbage collection is less trivial, however, since the delay caused by pausing graph reduction for collection was considered unnacceptable.

To avoid this delay, an approach inspired by Yuasa [28] and Bacon et al. [40] is taken: garbage collection effectively operates on a snapshot of the graph, preventing it from missing pointers that are overwritten as graph reduction continues in parallel. Soundness follows from the fact that allocated addresses that are not reachable in the snapshot (i.e., garbage) remain unreachable as the graph evolves, and are thus safe to free. Unlike the aforementioned systems, though, Cephalopode actually creates a snapshot of the graph. This comes at the cost of halving usable memory (similar to a copying garbage collector), but with the benefit of avoiding the usual write barrier, where the old data would need to be read from memory and copied to the garbage collector's stack prior to being overwritten; instead the garbage collector works on the snapshot as if it were a separate and non-changing graph.

Creating a snapshot by serially duplicating the entire graph would still create an unnacceptable delay, so a redirect-on-write approach is taken. Each memory address, as far as the graph reduction engine is concerned, actually corresponds to two adjacent places in physical memory. This gives room for a current ("live") and snapshot ("shadow") version of each node. Two bits of metadata for each memory address—stored in the processor for rapid access—keep track of which of the two places the current node version is stored in, and which of the two places the snapshot version is stored in (these may coincide or differ). Taking a snapshot updates this metadata in parallel to make the snapshot coincide with the places used to store the current state of the graph. As graph reduction continues, memory writes to a given address go to whichever place is not holding the snapshot version (and the metadata for the address is updated accordingly), so that the snapshot remains intact. This approach allows instantaneous snapshots without increasing main memory traffic, since the write barrier only requires metadata available inside the processor, as opposed to a traditional read-modify-write.

The lifecycle (allocation and marking) of each memory address is managed via a small finite state machine in the processor, allowing the sweep phase to be fully parallelized and simplifying allocation. Keeping this out of RAM also makes it possible for memory allocation to occur at the same time as other memory operations.

1.2.2 Stately

Motivation

When approaching Cephalopode's reduction engine—the unit that performs the graph traversal and manipulation—it became apparent that a fairly complex finite state machine would be needed. Due to the expected number of states, as well as difficult transition rules (some stages of reduction would be skipped in certain circumstances), the prospect of working out the machine on paper and attempting to transcribe it to HFL (the EDSL used by the VossII platform [17]) was not an appealing one.

In order to make the process more manageable, the finite state machine editor Stately was developed. As the main goal of Stately is to simplify the creation of control logic, its scope is narrow: Mealy machines [38] working with single-bit signals. Emphasis is placed on the design process, as this is the greatest barrier to development of control machines directly in HFL.

Visualization and organization

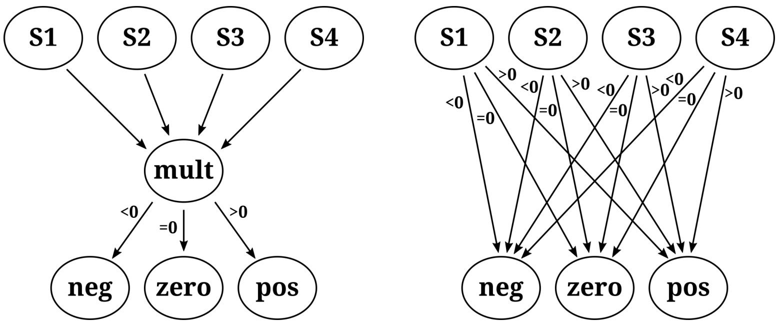
One of the chief complaints when developing such machines in HFL is that they can only be visualized after they are complete (and even then the visualization is sub-optimal), making it needlessly difficult to see the "big picture". This is compounded by output signals being specified separately from states on a signal-by-signal basis, diffusing the behavior of a single state through multiple definitions. Stately takes the opposite approach: the machine is presented visually, and the behavior of each state (a snippet of code including both transitions and outputs) is specified within the state. As well as making the machine far easier to comprehend, this approach makes changing the machine significantly easier, since states' behavior is expressed locally.

Virtual states

When designing Stately with Cephalopode in mind, it became clear that an ordinary finite state machine is slightly too low of a level of abstraction in many cases. In particular, there were several states whose sole function was to check a condition and then transition to one of several possible successor states, wasting a clock cycle on a check that could have easily been done in the previous cycle. An example of this is shown in Fig. 1.5(a), where the "mult" state simplifies the machine at the expense of speed.

However, as there are often multiple predecessors, moving this "gateway" logic into them for the sake of efficiency (in the manner shown in Fig. 1.5(b)) would obfuscate the machine visually and also make it difficult to change the logic later on (as it would need to be updated in all of the predecessors individually). Furthermore, a sequence of such gateway states becomes very difficult to optimize by hand.

Stately resolves this dilemma by raising the level of abstraction, introducing a novel mechanism referred to as *virtual states*. States that are designated as virtual are folded into their predecessors in the compiled machine (and during simulation), effectively automating the process one would perform by hand to eliminate the wasted state. However, the state still exists as part of the machine the designer edits, retaining the clarity benefits that the presence of the extra state confers. Futhermore, states can be switched between virtual and non-virtual very easily, allowing the designer to tune the machine on the balance between speed (favoring virtual states)



(a) Clear, but slow. (b) Fast, but messy.

Figure 1.5: Two approaches when using an ordinary finite state machine.

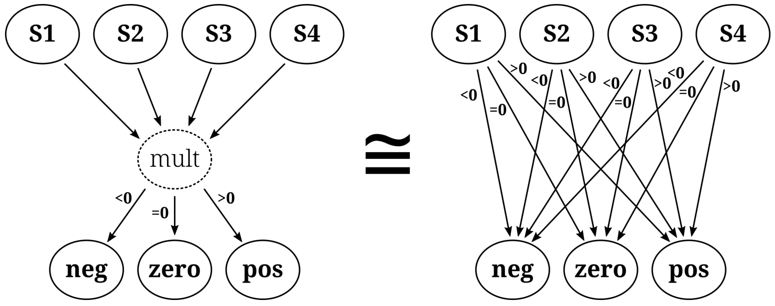


Figure 1.6: The behavior of a machine with a virtual state "mult".

and logic size (usually favoring keeping "gateway" states). Fig. 1.6 demonstrates the use of virtual states for the previous example: the state "mult" is made virtual, resulting in a fast but still easy-to-edit machine.

Correctness

Stately also features a robust error-reporting system that ensures machines are well-formed (particularly when virtual states are used), and makes it easy to locate problematic states/groups of states.

A lightweight form of simulation is included as well, making it possible to check the behavior of localized portions of the machine in response to different combinations of signals. Simulation is initiated at any state via a single click, and user-controlled input signals determine the next state that will be transitioned to. The upcoming transition is shown visually, which is particularly helpful when multiple virtual states are involved (as a transition may pass through several before reaching the actual, non-virtual next state). This is intended to encourage ad-hoc testing of small parts of a machine that is not yet complete, to reduce bugs as early as possible in the design process and to complement the more involved verification provided by VossII [17].

1.2.3 Bifröst

Motivation

Just as the complexity of Cephalopode's reduction engine necessitated a step up in abstraction from raw HFL to Stately, the even greater complexity of Cephalopode's arbitrary-precision arithmetic motivated a further step. This was largely due to the division algorithm, which proved to be sufficiently difficult that a direct approach using Stately, while eventually successful, demonstrated significant drawbacks of the latter.

First, because Stately only produces control logic, it was necessary to create the datapath separately and join them together. The need to keep these synchronized makes the design very difficult to change; some obvious optimizations to the algorithm would have required a large amount of work to integrate into both the state machine and datapath.

Second, FSM-based approaches are simply too low-level when it comes to I/O: operations such as memory accesses are not treated as abstract units but instead need to be performed manually by raising and lowering handshake protocol signals. This is error-prone, obscures the higher-level behavior of the machine, and makes it cumbersome to run several I/O operations in parallel (for example memory allocation and a memory read, which do not conflict in the current Cephalopode architecture). Furthermore, it effectively bakes the handshake protocol into the state machine, making it a significant undertaking to change to a different protocol later on.

Third, although loop structures are visible in finite state machines, their conditions and increments (in e.g. for-loops) are not nearly as visible as in a typical computer program. Likewise for portions of the state machine that are in effect subroutines.

Finally, all attempts at parallelism must be done entirely by hand, requiring careful reasoning about current and future versions of signals. This type of "hand compilation" is by nature error-prone, and the reasoning behind it is not evident afterwards in the design. Small changes to the underlying algorithm may also force the designer to redo some of this work.

Language

The language Bifröst was developed to remedy the aforementioned issues by synthesizing a circuit—both control and datapath—from imperative code. Conventional software programming mechanisms such as variable assignments, if-then-else statements, loops, and subroutine calls make highly sequential and "algorithmic" processes such as arbitrary-precision division straightforward to express, without a disconnect between the control and datapath logic.

For input/output in Bifröst, interaction with the outside world is accomplished through abstract units called *actions*, which behave like function calls to external hardware. The actions are declared by the designer, and the compiler synthesizes the logic required to perform them (input and output wires, handshake protocol circuitry, etc). This, in contrast to manually flipping bits, gives the designer a very convenient and high-level means to perform I/O: rather than twiddling request and acknowledge lines and flip-flop enables each time I/O is needed, the designer can simply write something along the lines of `x = do mem_read addr;` and the compiler will be responsible for the details. This abstraction also makes it trivial to change the protocol: only the initial declaration of the action needs to be modified, and no changes need to be made to the remainder of the code.

Bifröst also includes several miscellaneous quality-of-life features for its intended use with VossII: although Bifröst is strongly-typed, very little information about a given data type is required and as a result it can generate polymorphic hardware for free; a Bifröst-readable list of type declarations can be exported from VossII in order to avoid redundant declarations across the two systems; and Bifröst can incorporate strings of FL code as expressions to enhance the expressiveness of the language.

Example

The following example defines a module that takes an initial value and two memory addresses `bottom` and `top`, and computes a left-fold of the words in memory between `bottom` and `top` (inclusive and exclusive, respectively). The compiled circuit is controlled via a four-phase handshake, and communicates with memory through a two-phase handshake. The operation used for the fold is outsourced to an external component that is combinational (i.e., produces a result immediately and does not require a handshake).

```
name foldl;
protocol fourphase alwayson;

// type declarations
type word => "byte";
alias address = word;
state MEM;
actiontype Read = addr:address -> data:word
    reading [MEM] writing [];
actiontype Combine = x:word -> y:word -> z:word
    reading [] writing [];

// actions
action read:Read provided by external via twophase alwayson;
action combine:Combine provided by external via combinational
    alwayson;

// subroutines
subroutine main : initial:word -> bottom:address -> top:address ->
    result:word
{
    var address p;

    result = initial;
    for(p = bottom; p != top; p = p + 1)
    {
        result = do combine result (do read p);
    }

    return;
}
```

Parallelism

The circuit produced by the Bifröst compiler is a single finite state machine (i.e. serial at the broad level), but within each state it can perform multiple actions and update multiple variables. In contrast to many high-level synthesis tools—and in accord with the applications to Cephalopode—the guiding assumption Bifröst makes about scheduling is that built-in operations are cheap, and the main expense is waiting for the results of actions. The compiler therefore aggressively compacts statements into as few states as possible, using the type of rewriting one would perform by hand. This is limited by actions, for example if the argument to one action depends on the result of another, and also by a special "scissors" statement the designer can insert to prevent too eager scheduling at a particular location in the code (for example if such scheduling produces a circuit with logic too deep to fit timing requirements, or too sizeable in terms of the number of gates).

For the "foldl" module example above, Bifröst's eager attitude results in the first `read` operation being scheduled alone, but subsequent `read` operations being scheduled at the same time as preceding `combine` operations. The main state machine therefore has three states: an idle/done state, a state for the first read operation, and a state that performs a combine and a subsequent read (the latter only if the loop shall continue).

When declaring an action, the designer also roughly characterizes how it interacts with global state through a set of "state aspects" that the action depends on, and a set that it mutates. The compiler in turn uses this information when scheduling invocations actions so as to allow running several actions at once, but not to violate the sequential semantics of the source program. As an example, this semantically-savvy scheduling would prevent the erroneous parallelization of two (hypothetical) actions `reset_packet_count` and `read_packet_count`, since the former presumably mutates a state aspect that the latter depends on. Provided that the declarations are written dutifully, this allows the compiler to get up to the same (if not better) parallelization tricks that a designer would use while crafting a state machine by hand.

Use in Cephalopode

As mentioned previously, the divider was originally made using Stately (for the control logic) and HFL (for the datapath). However, to facilitate future modifications it was re-written using Bifröst, which was completed in half the time of the original implementation (30 days, down from 60). A flaw was later discovered in the time analysis of the algorithm, leading to a substantial design change in order to achieve better performance—this rewrite took only four days (arguably much less, in fact, as the designer was only able to work in brief periods over these days). While optimized circuit size and timing remain to be measured between these different versions of the divider, it is noted that the state machine in the final Bifröst version contains 30 states, approximately one quarter of the number of states in the original, hand-designed FSM.

1.3 Future work

While Stately is largely complete (a few bug-fixes nonwithstanding), Cephalopode and Bifröst offer several avenues for further development.

Cephalopode

In the case of Cephalopode, there are both changes to be made to the existing parts of the processor and also the completion of other components.

The most pressing change is to the memory management system, which currently dedicates a substantial amount of logic to each memory address (in order to accelerate allocation, snapshots, and garbage collection). As this does not scale well as memory grows, alternative designs are being explored.

Another area that is open to improvement is the graph traversal, which currently uses a stack to store the spine and arguments. This requires an additional memory bank whose size is only bounded by the size of the graph, greatly increasing the total memory requirements of the system.

The main parts of Cephalopode that remain to be completed are input/output instructions, and the process scheduler. Both of these depend on the precise choice of programming model, an important consideration for IoT devices as they are likely to be event-oriented rather than to continuously perform computation.

Another aspect of the processor that awaits implementation is path compression, where chains of indirection nodes are removed; an efficient method for this remains to be determined.

A set of combinators will be selected for the final version of the processor to use. These can be taken from other combinator-based systems such as VossII [17] or Lambda One [18], or derived through simulating various realistic IoT programs.

Once it is more complete, Cephalopode will be benchmarked in terms of speed and energy in order to determine the feasibility of this approach for IoT applications.

Bifröst

Similarly, for Bifröst there are both minor changes and extensions that are possible. Changes are primarily to smooth out the user experience: for example, to move the tests for uninitialized variables to earlier in the compilation process, so as to avoid false positives that arise when carrying out the analysis after several program transformations. Another such change is the labelling of components in the compiled circuit in order to separate datapath and control logic during visualization.

An extension that is currently under development is "action arrays", where what appears to be a single action (e.g. `add`) in the program is actually a collection of identical actions (e.g. several adders). The scheduler will then balance work across these, affording additional parallelism. By adjusting the size of the array, the designer can easily control the tradeoff between hardware complexity and speed.

As attention shifts toward computations inside Bifröst modules—or in small subordinate modules such as adders—it may become worthwhile to replace the computational model Bifröst uses with one better suited to juggling overlapping tasks with known duration (more similar to traditional high-level synthesis). This would likely improve performance considerably, if it can be made compatible with Bifröst's approach to actions.

When considering low-power hardware, it can be very helpful to power down portions of the hardware that are not in use—either through clock gating, or disconnecting the supply voltage. As Bifröst already synthesizes logic to carry out handshake protocols, it is not a great stretch to also synthesize logic for *power* protocols, where modules can take requests to power on/off and indicate readiness. This will allow Bifröst modules not only to be easily powered down when appropriate by their parents, but also to power down subordinate modules they interact with via actions.

Finally, there is interest in supporting some form of pipelining in Bifröst, as this has the potential to make better use of hardware resources. It is uncertain how this can be accomplished without lowering the level of abstraction too greatly, or requiring the designer to carry out tedious and error-prone tasks manually (such as accounting for interactions between pipeline stages, or designing forwarding logic).