Towards Porting Operating Systems with Program Synthesis

JINGMEI HU, Harvard University, United States of America
ERIC LU, Harvard University, United States of America
DAVID A. HOLLAND, Harvard University, United States of America
MING KAWAGUCHI, Harvard University, United States of America
STEPHEN CHONG, Harvard University, United States of America
MARGO I. SELTZER, The University of British Columbia, Canada

The end of Moore's Law has ushered in a diversity of hardware not seen in decades. Operating system (and system software) portability is accordingly becoming increasingly critical. Simultaneously, there has been tremendous progress in program synthesis. We set out to explore the feasibility of using modern program synthesis to generate the machine-dependent parts of an operating system. Our ultimate goal is to generate new ports automatically from descriptions of new machines.

One of the issues involved is writing specifications, both for machine-dependent operating system functionality and for instruction set architectures. We designed two domain-specific languages: Alewife for machine-independent specifications of machine-dependent operating system functionality and Cassiopea for describing instruction set architecture semantics. Automated porting also requires an implementation. We developed a toolchain that, given an Alewife specification and a Cassiopea machine description, specializes the machine-independent specification to the target instruction set architecture and synthesizes an implementation in assembly language with a customized symbolic execution engine. Using this approach, we demonstrate successful synthesis of a total of 140 OS components from two pre-existing OSes for four real hardware platforms. We also developed several optimization methods for OS-related assembly synthesis to improve scalability.

The effectiveness of our languages and ability to synthesize code for all 140 specifications is evidence of the feasibility of program synthesis for machine-dependent OS code. However, many research challenges remain; we also discuss the benefits and limitations of our synthesis-based approach to automated OS porting.

CCS Concepts: • **Software and its engineering** → **General programming languages**; *Formal methods*; **Assembly languages**; **Domain specific languages**.

Additional Key Words and Phrases: Program synthesis, assembly languages, operating systems
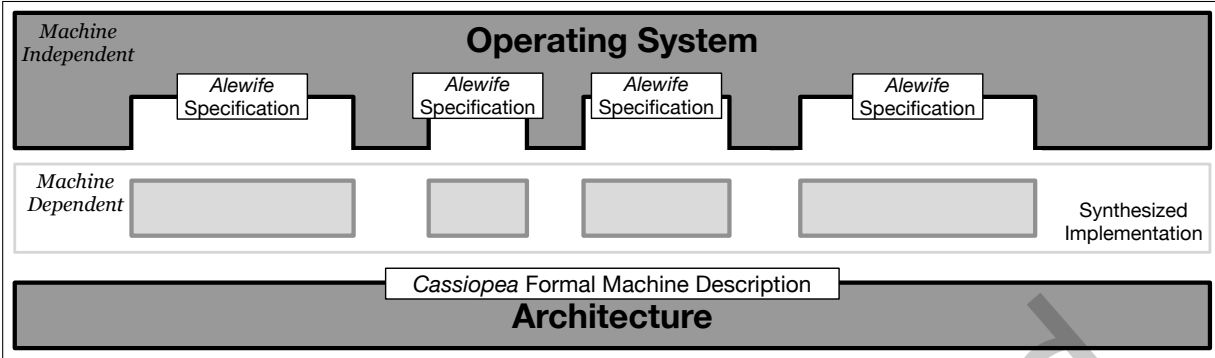
Fig. 1. Porting an operating system (top box in dark gray) to an architecture (bottom box in dark gray) requires re-implementing the machine-dependent parts (middle boxes in light gray).

1 INTRODUCTION

Porting an operating system (OS) to a new machine architecture is expensive.[1] This expense manifests in many ways: 1) direct monetary cost; 2) time to market penalties; and 3) opportunity cost for experienced kernel developers who could be handling more important issues, such as new technology or security [Chen et al. 2020; Ghavamnia et al. 2020; Shinde et al. 2020; Song et al. 2020]. Moreover, the end of Moore's Law has ushered in an era of new hardware [Hennessy and Patterson 2019], which requires new system software. This suggests that porting system software is an increasingly important challenge.

Meanwhile, the convergence of two other trends creates an opportunity. First, some machine architecture vendors are now producing formal specifications of their instruction set architectures (ISAs) [Armstrong et al. 2019]. Second, program synthesis techniques [Bodik et al. 2010; Bornholt and Torlak 2017; Feser et al. 2015; Gulwani 2011; Polozov and Gulwani 2015; Van Geffen et al. 2020], and especially verification-based synthesis such as Counterexample-Guided Inductive Synthesis (CEGIS) [Alur et al. 2013; Solar-Lezama et al. 2008, 2006], have reached a point where we can expect to synthesize small but useful program chunks in reasonable time.

The nature of OS architecture and of verification-based program synthesis suggests that a synthesis-based approach might reduce porting costs and/or provide an efficient way to produce verified operating system ports. Most OSes are structured to be portable as shown in Figure 1; they have clearly delineated machine-independent (top box in dark gray) and machine-dependent parts (middle boxes in light gray) [Custer 1992; Love 2010; McKusick et al. 1996; Rashid et al. 1987; Spier et al. 1973]. Porting an OS requires re-implementing the machine-dependent parts (light gray boxes in Figure 1), not performing deep structural reorganization. Further, some of the machine-dependent parts are written in C; other parts are written in assembler. We focus on those parts written in assembler (typically thousands of lines per port)[2], as those require in-depth knowledge of the assembly language for the processor to which the system is being ported[3].

---

[1]For example, NetBSD's AArch64 (64-bit ARM) port took approximately 300 commits by approximately 20 people (just in the kernel, not including user-level material or toolchain work); this was spread over two and a half years between serious work beginning and first release in 2020. See https://anonhg.NetBSD.org/src/file/tip/sys/arch/aarch64. Note that bugs are still being found; see for example https://gnats.NetBSD.org/56264.

[2]https://anonhg.NetBSD.org/src/file/tip/sys/arch/

[3] Though the C code also requires knowledge of the processor, it does not, by definition, include operations that cannot be expressed in C, such as control register accesses, or code that must not destroy registers that are ordinarily available to the compiler, such as trap handlers. The assembly code covers such material; it is therefore more challenging and is the proper first step for assessing feasibility.

The machine-independent parts of the OS use the machine-dependent parts via a machine-independent interface whose functions each implement a well-defined machine-independent operation, such as turning on interrupts or saving a trapframe. Overall, these assembly language machine-dependent portions exhibit two important characteristics. First, they are usually small and relatively isolated from one another; they *decompose readily into small independent pieces* of code, which each exhibit simple control flow. Each independent piece tends to be either implemented in a single function or combined with related pieces into a sequence of semantically meaningful steps. Second, each machine-dependent portion implements a well-defined machine-independent operation, regardless of the target architecture; while details may vary, *each piece can be given a machine-independent specification*. This uniformity of specification is precisely the definition of the boundary between machine-independent code and machine-dependent code.

Taken together, these characteristics outline an approach for synthesizing the machine-dependent parts of an operating system. Operating system developers provide functional specifications of the machine-dependent parts of their operating system independent of any underlying hardware (white *specification* boxes in Figure 1). Computer architects produce a functional specification of their platform, independent of any operating system (white *formal machine description* box in Figure 1). By combining machine-independent OS specifications with hardware specifications, a synthesis engine produces the appropriate implementation (light gray boxes in Figure 1). Thus, given M operating systems and N hardware platforms, one writes $M + N$ specifications rather than implementing $M \times N$ ports. In our approach we use full functional specifications, rather than, for example, partial specifications based on input/output samples.

CEGIS-based synthesis relies on verification to determine when an implementation satisfies its specification, so components constructed in this manner are correct by construction. Such an approach offers an alternative to building verified operating system ports from the ground up, which can take many person years [Klein et al. 2009].

This approach presents myriad interesting research questions, spanning many different areas of computer science.

(1) What are the requirements for a language that expresses machine-dependent functionality in a machine-independent way?
(2) What are the requirements for a language that describes hardware in a way amenable to synthesis? Must one model the entire ISA to synthesize machine-dependent OS code?
(3) Some parts of an operating system are not only machine-dependent but machine-specific, in that they handle abstractions or structures that do not exist on other machines, such as segment tables on x86. How do we address parts of the operating system that are machine-specific and, thus, cannot be given a machine-independent specification?
(4) How well do modern synthesis techniques work in this domain?
(5) What additional tooling or support is needed to integrate the results of program synthesis into a complete working system?
(6) How well does a synthesized system perform?
(7) Does program synthesis make the porting process faster and/or easier?

This paper proposes and evaluates solutions for the first two research questions and touches upon the fourth. We also designed and built an entire ecosystem in pursuit of the fifth point. A detailed description of that larger ecosystem is beyond the scope of this paper, although we highlight several ways in which the work discussed here leverages it. The other research questions are either current research or intended future work, but this area is wide open, and we invite others to build upon our work and explore alternative approaches.

To tackle these questions we developed a system to explore synthesis of machine-dependent parts of an OS. Two domain specific languages are central to our approach. *Alewife* is a language for machine-independent specifications of machine-dependent OS functionality (white *specification* boxes in Figure 1), and *Cassiopea*[4] is a register transfer language (RTL) style machine description language (white *formal machine description* box in Figure 1).

Alewife expresses machine-independent specifications of machine-dependent functionality by providing an abstract machine to hide details. In Alewife, an OS designer declares abstract predicates (or other functions) and abstract machine state. These describe observable properties of a machine without providing an exact implementation. For example, one might declare the abstract predicate `interrupts_are_on` to indicate whether interrupts are on or `from_usermode` to indicate whether a pending trap came from user mode or kernel mode. One might also declare `crt0_argv_reg` to be the register that `argv` is placed in by the kernel at program startup. These abstract declarations describe concepts that are independent of any particular machine, but can be directly related to the exact machine-dependent state and functionality of each machine. Then, using these abstractions, an OS designer can provide machine-independent pre- and postcondition specifications. For example, the postcondition of a block that is meant to turn interrupts on might require `interrupts_are_on() == true`.

This formulation leads to a beneficial separation of concerns. OS developers can write machine-independent code in terms of the machine-dependent functionality exposed via Alewife specifications. Chip designers can write machine descriptions in Cassiopea, possibly supplying definitions of commonly used abstract declarations, such as interrupt state. A programmer porting an OS to a new machine architecture need only specify how the Alewife abstract declarations used by the OS are defined in terms of the state of the target machine.

Given an Alewife input file, a Cassiopea description of an architecture, and a mapping from Alewife abstract declarations to the state of the target machine, our Alewife compiler turns the Alewife machine-independent specifications into machine-dependent specifications. Our synthesis engine takes these machine-dependent specifications and synthesizes assembly language programs that satisfy the specifications.

We evaluate our languages and tools via 35 synthesis use cases from eight complete machine-dependent procedures in two pre-existing OSes: Barrelfish [Baumann et al. 2009] and the OS/161 instructional OS [Holland et al. 2002], deployed on four real machine architectures: 32-bit MIPS, 32-bit ARM, 32-bit RISC-V and x86_64. Our goal in this evaluation is not (yet) to produce a complete port of either OS, but instead, to show that synthesizing the code for such a port is feasible. For this reason we have selected a variety of code examples with a variety of purposes. Each example is taken from the machine-dependent code for one port of one of the OSes; we synthesize comparable code for each of the other target architectures for which we have written machine specifications. We successfully lower all 35 machine-independent specifications for each architecture with the Alewife compiler, generating 140 machine-dependent specifications in total for synthesis. We validate the efficacy of the synthesis engine by synthesizing and verifying assembly code for all 140 specifications; all synthesis executions finished within a half-hour timeout with optimizations enabled in our synthesis engine.

In summary, the contributions of this paper are:

- A novel approach to synthesizing the machine-dependent portions of an OS.
- The Alewife language, which allows specification of machine-dependent OS components in a machine-independent fashion.
- The Cassiopea language, which allows describing real instruction set architectures to enable synthesis of OS functionality.
- An assembly language synthesis engine and the Alewife compiler, which transforms machine-independent specifications into machine-dependent specifications.
- Several optimization techniques that improve scalability for assembly language synthesis.

---

[4]Cassiopea is named for a jellyfish that features symbiotic photosynthetic algae, which is for some reason spelled without the customary 'i'.

- Identification and discussion of the challenges and payoffs of our synthesis-based approach to automated OS porting.

In the next section, we begin with a brief overview of our approach to program synthesis, named Aquarium, after the ecosystem we built for producing synthesized ports of machine-dependent components of an operating system. We then discuss our languages in detail (Sections 3 and 4) and describe the mechanism for connecting them (Section 5). In Section 6, we describe the implementation of the Alewife compiler and synthesis engine. Then, in Section 7, we present use cases, validation, and experimental results. Section 8 contains further discussion, including when program synthesis is not appropriate for implementation of machine-dependent OS components (and a brief description of other tools we created to address these situations); Section 9 discusses related work, and Section 10 concludes.

2 OVERVIEW: PROGRAM SYNTHESIS IN AQUARIUM

Aquarium is a collection of tools to automatically construct the machine-dependent parts of an OS. In this work, we focus on the languages we developed for 1) describing machine-dependent OS functionality in a machine-independent fashion and 2) specifying a processor's semantics, and on using modern program synthesis in the assembly language realm, where all state is global and data is untyped.

Figure 2 presents an overview of Aquarium's four-step approach to OS synthesis:

Step 1: Writing a Machine Independent Specification. An OS developer writes an Alewife specification for some piece of OS functionality Ⓐ. Alewife reasons about machine-dependent functionality in an abstract machine (Section 3).

Step 2: Writing Machine Descriptions. A computer architect produces a machine model Ⓑ written in Cassiopea, which models the ISA semantics at the assembly language level (Section 4).

Step 3: Lowering to Machine-Dependent Specifications. To specialize a machine-dependent specification for synthesis, Aquarium applies *lowering* files Ⓒ that instantiate the machine-independent Alewife abstractions for the target architecture. The Alewife compiler Ⓓ performs the lowering process, which takes a Cassiopea machine description, an Alewife specification, and the corresponding lowering file to generate the machine-dependent specification Ⓔ for that architecture (Section 5).

Step 4: Synthesis and Verification. Given the Cassiopea machine description and the machine-dependent specification, Aquarium uses the Cassiopea synthesis engine Ⓕ to synthesize a satisfying sequence of assembly instructions Ⓖ and verify it against the specification (Section 6).

A Small Example of Aquarium Program Synthesis

As a running example, we use an excerpt from the exception handling code of the Barrelfish operating system [Baumann et al. 2009]. Figure 3 shows the original ARMv7 Barrelfish code (which we call disp_check). The code sets the return value in r0 to 0 or 1 depending on the relationship between the value in the memory location [r2,#88] and the contents of the special register lr. The value in r0 is used in code that follows this block. Since this code is written in assembly, its implementation is machine-dependent. We discuss this code in more detail and present its machine-independent specification in the sections that follow.

3 WRITING MACHINE-INDEPENDENT SPECIFICATIONS IN ALEWIFE

Each Alewife file (Ⓐ in Figure 2) provides a machine-independent specification for a single piece of machine-dependent functionality in terms of its effects on an abstract machine. This allows specifications to reason about the underlying machine state that is relevant to machine-independent code without having to know machine-dependent details. We assume that these specifications are written by OS designers with general knowledge of machines, but not in-depth knowledge of all the architectures to which their system will eventually be ported.
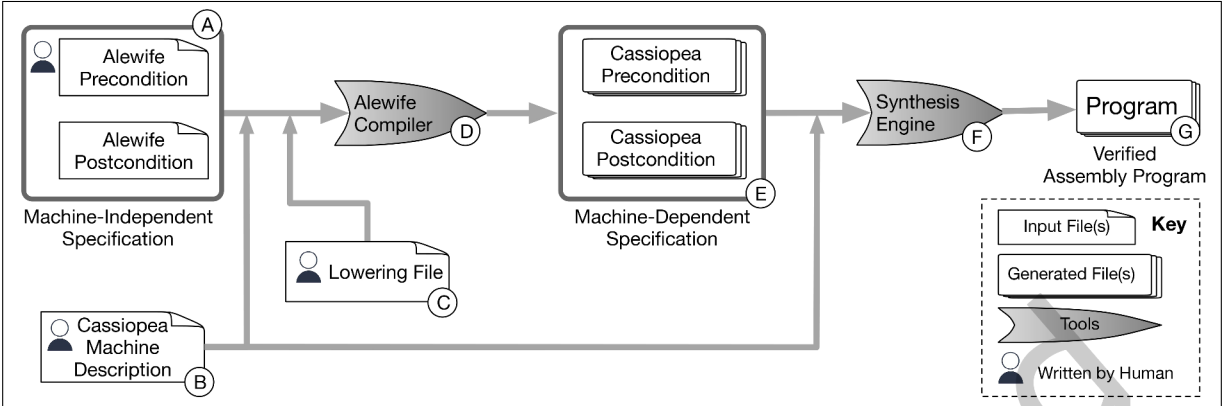
Fig. 2. Aquarium program synthesis workflow. Given a machine-independent Alewife specification Ⓐ (written by OS designers), a Cassiopea machine description Ⓑ for a specific architecture (written by chip designers), and an Alewife-Cassiopea lowering file Ⓒ, the Alewife compiler Ⓓ generates a machine-dependent Cassiopea specification Ⓔ for that architecture. The Cassiopea synthesis engine Ⓕ uses the machine description and the machine-dependent specification to synthesize a satisfying sequence of assembly instructions, verifies them against the specification, and translates them into real assembly programs Ⓖ.

```
1  ldr r0, [r2, #88]
2  cmp lr, r0
3  movhs r0, #0x00000000
4  movlo r0, #0x00000001
```

Fig. 3. ARMv7 disp_check assembly program example from the Barrelfish operating system.

We call the unit of machine-dependent functionality in an Alewife file a *block*. A block is a sequence of assembly instructions that has a single entry point, no backward branches to code in the block, and optionally a single branch to code outside the block. The specification consists of four parts: 1) preconditions, 2) postconditions, 3) let-bindings, and 4) frame conditions. The precondition describes the *initial state* of the abstract machine immediately before the block executes. The generated code may assume that the precondition is true and can behave in arbitrary or undefined ways if the precondition is false. The postcondition describes the *final state* of the abstract machine immediately after the block executes. Provided that the precondition was true, correct generated code (in our case a sequence of assembly instructions) must guarantee that the postcondition is true after it executes. Let-bindings are evaluated against the initial state and allow the postcondition to refer to the initial state. Frame conditions specify what machine state may be modified. An Alewife file may include other files, enabling reuse of declarations shared across multiple OSes.

The rest of this section describes the mechanisms by which Alewife permits machine-independent specification of machine-dependent functionality; Alewife syntax and semantics appear in Appendixes A.5 and A.6.

Figure 4 shows the Alewife specification for Barrelfish's disp_check. This block compares the saved program counter where an exception occurred (previously placed in pc_reg) to the upper bound of the critical region in which the exception is re-entrant and requires special handling. That bound is ordinarily stored in the dispatcher

structure.[5] The dispatcher structure resides in `DispMem` and contains space to save registers; thus, both the number of fields in it and their sizes are machine-dependent and must be abstracted. `get_crit_ptr` takes the base address of the dispatcher structure as an argument and adds in the offset of the proper field in the structure. We make `get_crit_ptr` a function, rather than adding in an abstract offset directly, for illustrative purposes.

The precondition (line 13) indicates that register `disp_reg` (from line 4) must contain a pointer to the base of the region `DispMem`, which is defined on line 8. The postcondition (line 14-15) ensures that the register `disp_area_reg` contains a bitvector whose value is 1 if the variable `crit` (line 12) is `true` or a bitvector whose value is 0 if `crit` is `false`. The variable `crit` is a boolean condition that stores the comparison result between a register (`pc_reg`) and the contents of the memory location computed by the function `get_crit_ptr`.

Memory model. Alewife memory regions, x_{mem}, have finite lengths with types N_1 `bit` N_2 `len` N_3 `ref`; such a region contains N_2 elements, each of which is an N_1-bit value; pointers into the region are N_3-bit values. N_1, N_2, and N_3 here can be either concrete lengths or abstract symbolic constants. For example in Figure 4, the type of the `DispMem` region has three symbolic constants: it has DISP_MAX locations (`DISP_MAX len`), each containing a bitvector with length `wordsize` (`wordsize bit`), whose pointers are `wordsize`-bit values (`wordsize ref`). We discuss the concretization of these abstract values in Section 5.

Memory is addressed by *pointers*, which are pairs of a memory region and offset; a pointer to the nth element of memory region x_{mem} is represented by the pair (x_{mem}, n). Memory regions do not overlap; that is, given two memory regions x_{mem1} and x_{mem2}, a pointer $(x_{mem1}, \_)$ can never alias a pointer $(x_{mem2}, \_)$. This "swiss-cheese" memory model is inspired by DieHard's miniheaps [Berger and Zorn 2006] and array separation logic [Brotherston et al. 2017] and reflects the C standard. Memory regions are second-class; pointers may be passed around, but memory regions may not and every pointer must explicitly name a specific memory region.

A memory region can optionally have a *label*, which is the symbol used to refer to its base address in assembly language text. Note that while this symbol refers to a constant, the value of the constant is not known until the code is assembled and linked into the final output program; or in the case of typical shared libraries or position-independent executables, not until program execution. In general, position-independent code and ordinary code must use different instructions to refer to labels. For this reason, and because Barrelfish links its kernel as position-independent code, we attach an access type to each label.

It is allowable to declare a memory region with `0 len`, which means that we model only the existence of the region but not its internal workings. OS designers generally do not need to know the synthesis algorithm when writing these Alewife specifications. However, limiting memory regions to their minimum size improves performance, since adding extra state slows down synthesis.

Abstract Declarations and Abstract Functions

Values imported with `require` (lines 2-6 in Figure 4), such as the register `pc_reg` and integer `wordsize`, are examples of *abstract declarations*, allowing an Alewife specification to omit details of how an implementation represents and stores data. This is the key to making Alewife specifications machine-independent. Such identifiers must be defined elsewhere for each target machine; we bind the abstract declarations and their corresponding machine-dependent definitions in a later step, before conducting synthesis. Alewife types (the complete specification is in Section A.6) can contain symbolic bit lengths and abstract type identifiers, which will be specialized to different bit lengths and Cassiopea types by Cassiopea machine descriptions and machine-dependent lowerings

---

[5]Barrelfish decomposes its kernel into a machine-dependent CPU driver and a machine independent, user-space Monitor. The CPU driver "performs dispatch and fast local messaging between processes on the core" [Baumann et al. 2009]. The `dispatcher` structure belongs to the CPU driver and is analogous to a process structure or process control block; among other things it contains register save areas. Thus, it is both OS-specific, as it is particular to the Barrelfish architecture, and machine-dependent, as the size of the structure varies with the pointer size, word size, and number of registers of the architecture.

```
1  require type word
2  require value wordsize: int
3  require value pc_reg: wordsize reg
4  require value disp_reg: wordsize reg
5  require value disp_area_reg: wordsize reg
6  require value DISP_MAX: int
7  require function get_crit_ptr: (word) word
8  region DispMem: wordsize bit DISP_MAX len wordsize ref
9  lower-with: disp_defs may_use_flags disp_scratch
10
11 let crit_ptr: wordsize ptr = get_crit_ptr([DispMem, 0])
12 let crit : bool = *pc_reg b< fetch(crit_ptr, wordsize)
13 pre: *disp_reg == [DispMem, 0]
14 post: if crit then *disp_area_reg == (1: wordsize vec)
15 else *disp_area_reg == (0: wordsize vec)
```

Fig. 4. Alewife specification for disp_check.

before synthesis takes place. For example, in Figure 4, the value wordsize, imported with require in line 2, is used to define the dispatcher memory region DispMem on line 8. A machine description, written in Cassiopea, provides the definition for wordsize appropriate for a specific machine. Similarly, the size of the (machine-dependent) dispatcher structure is specified using an abstract integer DISP_MAX. We discuss details of the concretization of these abstract states and functions in Section 4 and Section 5, with examples from two different architectures (MIPS and ARMv7) whose relevant Cassiopea descriptions appear in Figure 5 and the lowering files in Figure 6.

Alewife specifications also use *abstract functions*. For example, line 7 of Figure 4 declares a function get_crit_ptr. This example does not refer to machine state, but others might. The predicate interrupts_are_on mentioned earlier, for example, will.

The keyword lower-with (e.g., line 9 in Figure 4) is used to name modules from which to get the machine-dependent instantiations of the abstract definitions. In this case, there are three: definitions related to disp_check, permission to change the processor flags on machines that have them, and scratch register assignments for this block. These are called *lowering modules* and are discussed in more detail in Section 5.

Frame Conditions

An Alewife specification may optionally provide *frame conditions* that permit a block to modify additional machine state (registers or register fields and memory). By default, synthesized code can modify only machine state explicitly mentioned in the postcondition. All other state elements' final values must equal their initial values. The modify frame condition identifies additional (abstract) state elements that may be modified (e.g., scratch registers). As we discuss in Section 5, lowering modules can also add frame conditions. In practice this is where most frame conditions come from. The frame conditions are more than synthesis heuristics that help to prune the search space; they are a useful shorthand to materially affect the specification.

The default behavior is desirable: most blocks should not modify state irrelevant to their postcondition. However, some computations require scratch registers. Additionally, it is sometimes necessary to read a register and write the same value back. For example, many control registers contain multiple fields but can only be read and written as a unit; changing one field requires reading the whole register, updating the desired field, and writing the whole register back. It is important both to permit these accesses and to ensure that the other fields

are not arbitrarily modified. On MIPS, for example, bit 0 of the status register controls the interrupt state; but the register contains many other fields where arbitrary changes can have unwanted consequences, such as switching between 32-bit and 64-bit mode. A correct MIPS implementation of code to enable or disable interrupts must be able to write to the bit controlling 64-bit mode, but must not change it.

Since we do not prohibit writes entirely, in theory, the synthesizer can generate programs that change registers in arbitrary ways and then restore their initial value. In practice, this is not an issue, because our synthesis technique preferentially produces smaller programs, excluding programs with redundant operations. Another potential problem is that the synthesizer is obliged to waste time considering and ruling out programs that make unwanted changes. We address this in the implementation by *gating* registers (Section 6.3.2) that are irrelevant to the specification; this removes them entirely from consideration.

Limitations of Alewife

Alewife specifications are based on the premise that the machine-dependent code in an operating system is made up of machine-dependent instantiations of machine-independent constructs; this allows the bulk of the specification to be machine-independent with relatively minor parts substituted on a per-machine basis. In practice, this is largely, but not entirely, true. Some machine-dependent code in operating systems is not just machine-dependent but machine-*specific*: it handles concepts that do not exist on other machines, e.g., segment tables on x86. These concepts primarily arise in code that is called directly by the machine (such as trap handlers and the kernel startup code) rather than by the machine-independent part of the OS. It is *possible* to specify such code in Alewife, but only in a degenerate form, where the entire specification is abstracted into a single pre- and postcondition pair in the lowering file.

4 WRITING MACHINE DESCRIPTIONS IN CASSIOPEA

To verify or synthesize programs against a particular machine, Aquarium uses a model of the ISA semantics written in the Cassiopea language (Ⓑ in Figure 2). This section presents an overview of the language, highlighting those design features that improve the efficiency of symbolic execution and synthesis. The complete syntax and semantics appear in Appendices A.2 and A.3.

A Cassiopea machine description provides an executable model of an ISA, declaring *machine state*, such as registers and memory, and defining *operations* that describe its assembly instructions. Cassiopea is a typed, interpreted language. In the following discussions we refer to conditions detected during the initial typechecking pass as "static" and those that are not as "runtime errors". The symbolic execution that occurs during synthesis (see Section 6) explores all execution paths and treats any runtime errors as conditions to avoid; that is, no output program should ever produce a runtime error.

Throughout the rest of this section, we use excerpts from two of our Cassiopea machine descriptions (Figure 5: MIPS (Listing A) and ARMv7 (Listing B)) as running examples. Unless otherwise stated, line numbers refer to lines in both listings from this figure.

Register model. Registers in Cassiopea have types of the form C reg, which indicates that a register can hold C-bit values (line 4). Registers are first-class values and can be passed as arguments to functions and operations. Each register has the text form used in the machine's assembly language associated with it. The synthesis engine uses this to produce the synthesized assembly code; it allows working with non-identifier register names, such as %eax on x86.

Memory model. Cassiopea uses the same "swiss-cheese" memory model as Alewife does (described in Section 3). However, unlike Alewife, Cassiopea memory regions, x_{mem}, have finite, statically known lengths. Unbounded or variable-length memory regions do not typically appear in assembly-language machine-dependent OS components.

Listing A. Cassiopea MIPS machine description

```
1   (* bitvector and register types *)
2   let wordsize: int = 32
3   type word = 32 bit
4   type register = 32 reg
5   (* general purpose registers *)
6   letstate r0: register
7   ...
8   (* control flags: cp0 $12: Status *)
9   letstate control cp0_12_ie: 1 reg ...
10  (* assembly text representation *)
11  let r0.txt = "$0" ...
12  (* machine invariant *)
13  invariant: *r0 == 0x00000000
14  (* instructions *)
15  defop SLTU rd:register rs:register
16    rt:register {
17   txt = format("sltu {1}, {2}, {3}",
18    rd.txt, rs.txt, rt.txt),
19   sem = [ (* check the validity
20   of each operand *)
21     assert (valid_gpreg(rd));
22     assert (valid_gpreg(rs));
23     assert (valid_gpreg(rt));
24     assert (rs != rt);
25     assert (rt != r0);
26     if rd == r0 then skip
27     else if ( *rs b< *rt ) then
28       *rd <- 0x00000001
29     else *rd <- 0x00000000 ]}
30  defop LW rd:register rs:register
31    imm:16 bit {
32   txt = format("lw {1}, {2}({3})",
33    rd.txt, imm.dec, rs.txt),
34   sem = [ ... (* check validity *)
35     if rd == r0 then skip
36     else let addr: word =
37       *rs b+ sign_extend(imm)
38     in *rd <- fetch[addr, 32] ]}
39   ...
```

Listing B. Cassiopea ARMv7 machine description

```
1   (* bitvector and register types *)
2   let wordsize: int = 32
3   type word = 32 bit
4   type register = 32 reg
5   (* general purpose registers *)
6   letstate r0: register
7   ...
8   (* control flags: CPSR *)
9   letstate control Z: 1 reg
10  ...
11  (* assembly text representation *)
12  let r0.txt = "r0"
13  ...
14  (* instructions *)
15  defop MOV_imm rd:register ri:4 bit
16    vi:8 bit cd:4 bit {
17   txt = let wi:word = rotimm(ri, vi) in
18    format("mov{1} {2}, #{3}",
19    armcc(cd), rd.txt, wi.hex),
20   sem = [ (* check validity *)
21     assert (valid_cond(cond));
22     assert (valid_gpreg(rd));
23     assert (valid_rotimm(ri, vi));
24     if checkcond(cd) then
25       *rd <- rotimm(ri, vi)
26     else skip ]}
27  defop LDR_imm rd:register rn:register
28    imm:12 bit cd:4 bit {
29   txt = format("ldr{1} {2}, [{3},#{4}]",
30    armcc(cd), rd.txt, rn.txt, imm.dec),
31   sem = [
32     ... (* check validity *)
33     if checkcond(cd) then
34       assert (is_ptr( *rn ));
35       let addr: word =
36         *rn b+ zero_extend(32, imm)
37       in *rd <- fetch[addr, 32]
38     else skip ]}
39   ...
```

Fig. 5. Excerpts from Cassiopea machine descriptions. Lines of the form (* ... *) are comments.

Cassiopea memory regions have types of the form C_1 bit C_2 len C_3 ref, where C_1, C_2, and C_3 must be concrete constants. As in Alewife, labels have an access type that indicates whether references should be position-independent; instruction definitions that use labels can query or assert about the access type, so that the synthesis engine produces the appropriate code. Note that most memory regions appear in Alewife specifications and not in machine descriptions: we provide a programmer's view of the machine, so the memory regions available are the ones associated with the code at hand and not the machine's entire memory space.

Bitvectors and pointers. In a typical ISA, all values are bitvectors and can be used by any operation. In contrast, as in typed assembly language [Morrisett et al. 1999] and the Compcert backend [Leroy 2009], Cassiopea distinguishes between finite bitvector constants and pointers into memory regions. The distinction is not statically checked, however; the type C bit includes both C-length bitvectors and pointers into memory regions of type C_1 bit C_2 len C ref. It is a runtime error to apply bitvector operations to pointer values and vice versa, though we overload some operations to allow limited pointer arithmetic. We deliberately omit the ability to convert between pointer values and bitvectors, since we do not model absolute memory addresses. This model simplifies reasoning about the semantics of programs, since they cannot create pointers to arbitrary regions. It is also a runtime error to make unaligned or out-of-bounds accesses to memory.

Machine state. All machine state is either registers or memory. Such state is global in scope and declared using **letstate**. Registers and memory can contain only bitvectors and pointers. Cassiopea code can manipulate integers, booleans, and registers to define instruction semantics, but these values cannot be stored directly in machine state.

We refer to the state of a machine $\mathbf{M}$ with the notation $\Sigma_\mathbf{M}$, which is a pair (ρ, σ). ρ is a map from registers to values of bitvector type, and σ is a map from pointers to values of bitvector type.

Our example in Figure 5 declares several pieces of machine state. The general-purpose register r0 for both MIPS and ARM appears on line 6. Its assembly output is specified by binding r0.txt, which appears on line 11 in Figure 5A and line 12 in Figure 5B. We also show one control register for each machine: cp0_12_ie in MIPS (line 9 in Figure 5A) and the Z flag in ARM (line 9 in Figure 5B). (Control registers vary widely across machines.) There are no memory regions in the example because, as discussed previously, memory regions normally arise in program specifications and not in the machine model itself (see Figure 4 line 8).

Operations and programs. Machine state in Cassiopea is manipulated by *operations*. These are typically a single assembly instruction but may represent a group of instruction variants and/or a short sequence. In the examples, lines 15–29 and 30–38 of Figure 5A and lines 15–26 and 27–38 of Figure 5B define operations with **defop**.

Each operation definition has two parts: txt is an expression that constructs the assembly text representation of the operation, and sem is a Cassiopea statement defining the operation's full concrete semantics in terms of the machine state. In the definition, most operations check the validity of each operand first. The MIPS instructions shown are SLTU (unsigned set less than) and LW (load word). The ARM instructions shown are MOV_imm (move immediate to register) and LDR_imm (load with immediate value offset).

Borrowing from JitSynth [Van Geffen et al. 2020], an *operation* is a pair $(op, \mathbf{T})$, where op is an *opcode*, and $\mathbf{T}$ is a map from argument name to type, which can include registers, booleans, concrete bitvectors, and labels. (When used as an argument to an instruction that accepts labels, a label provides a pointer to the first entry in its memory region.) Each operation defines a set of *instructions* $(op, \mathbf{v})$ where $\mathbf{v}$ is a map from argument name to value, where for each argument x, $\mathbf{v}(x)$ is of type $\mathbf{T}(x)$. For a machine $\mathbf{M}$, we denote its set of instructions as $\mathcal{I}_\mathbf{M}$.

Cassiopea machine descriptions define the semantics of an instruction as a partial function (partial due to the possibility of runtime errors) from a machine state to a new machine state accompanied by a *branching state*:

$$\llbracket (op, \mathbf{v}) \rrbracket_\mathbf{M} : \Sigma_\mathbf{M} \rightharpoonup \Sigma_\mathbf{M} \times (\mathbb{N} \cup \{\cdot, \mathsf{ext}\})$$

The branching state indicates the destination of a branch instruction. Operations in Cassiopea may only branch forward, so the branching state indicates that execution should proceed with either the next instruction ($\cdot$), an external assembler label (ext), or the instruction after skipping n instructions forward ($\mathbb{N}$) (where n should be at least 1).

A *program* is a series of instructions. We define the result of running a program P (a list of instructions) on a machine state $(\rho, \sigma) \in \Sigma_\mathbf{M}$ using a partial function $run_\mathbf{M} : \mathcal{P}_\mathbf{M} \times \Sigma_\mathbf{M} \rightharpoonup \Sigma_\mathbf{M} \times bool$ where $\mathcal{P}_\mathbf{M}$ is the set of programs for machine $\mathbf{M}$ and the result includes a boolean representing the branchto value, indicating whether the program jumped to the special external assembler label.

$$
run_\mathbf{M}(P, (\rho, \sigma)) = \begin{cases} (\rho, \sigma, false) & P = nil \\ (\rho', \sigma', true) & [\![head(P)]\!]_\mathbf{M}(\rho, \sigma) = (\rho', \sigma', \text{ext}) \\ run(drop(P, 1), (\rho', \sigma')) & [\![head(P)]\!]_\mathbf{M}(\rho, \sigma) = (\rho', \sigma', \cdot) \\ run(drop(P, n), (\rho', \sigma')) & [\![head(P)]\!]_\mathbf{M}(\rho, \sigma) = (\rho', \sigma', n) \wedge n \le len(P) \\ \bot & [\![head(P)]\!]_\mathbf{M}(\rho, \sigma) = \bot \\ \bot & [\![head(P)]\!]_\mathbf{M}(\rho, \sigma) = (\rho', \sigma', n) \wedge n > len(P) \end{cases}
$$

The function $head(\ell)$ is the first element of a list ℓ, $drop(\ell, n)$ is the tail of the list ℓ after removing n elements, and $len(\ell)$ is the length of the list ℓ.

Other elements. Cassiopea files may define type aliases, values, functions, and procedures and use them in the definitions of operations. (Functions are pure; procedures may alter machine state.) Both the MIPS and ARM descriptions define word as a type of 32 bits (line 3) and register as the type of 32-bit registers (line 4). The value wordsize, discussed in Section 3, is also specified as 32 on both 32-bit MIPS and 32-bit ARMv7 (line 2) and 64 for x86_64 (not shown). Function valid_gpreg checks whether the given operand is a general purpose register (line 21-23 in Figure 5A and line 22 in Figure 5B). This prevents generating code that tries to use special registers or control registers that happen to be the same size as general-purpose registers and thus have the correct type to pass as operands. LW in MIPS uses a function sign_extend (line 37 in Figure 5A, definition not shown), which implements sign extension on bitvectors. LDR_imm in ARM uses a function zero_extend (line 36 in Figure 5B), which implements (unsigned) zero-extension. Common functions and procedures, such as those used for bitvector manipulation, can be defined in separate files and reused by inclusion in different machine descriptions.

It is also possible to define machine invariants. These are predicates that must hold true before and after block execution. The **invariant** statement in line 13 in Figure 5A declares that the value in register r0 is always zero.

Execution model. Cassiopea is not Turing-complete; machine models are finite-state and evaluation of Cassiopea functions, procedures, and operations always terminates. We deliberately restrict Cassiopea's ability to model control flow: Cassiopea operations may only branch forward, so assembly snippets modeled in Cassiopea are loop-free. The semantics of Cassiopea do not include clocks and timers, concurrency, hazards, or weak memory models. The minimalistic design of Cassiopea allows symbolic execution to generate complete logical descriptions of our models and helps to simplify synthesis. We show in Section 7 that Cassiopea remains sufficiently expressive to support assembly synthesis for many OS components.

Because Cassiopea models ISAs at the assembler level, Cassiopea models need not be wire- or bit-level accurate. Our goal is to generate code to be assembled and linked into an (existing) OS, using that OS's compiler and toolchain. Thus, Cassiopea descriptions need not capture phenomena hidden by the assembler (e.g., branch delay slots on MIPS). Moreover, machine descriptions need contain only those parts necessary to synthesize targeted machine-dependent OS components; a surprising finding of this work is how little of an ISA needs to be specified

to synthesize significant parts of an operating system. Although we anticipate a world where vendors provide descriptions of ISA semantics, we currently write Cassiopea machine descriptions manually.

5 LOWERING TO MACHINE-DEPENDENT SPECIFICATIONS WITH THE ALEWIFE COMPILER

Synthesizing an OS component for a particular machine architecture requires a *machine-dependent specification* (Ⓔ in Figure 2), i.e., a pre- and postcondition expressed in terms of a particular machine architecture's state and semantics. Meanwhile, the Alewife specification for an OS component is written in terms of abstract functions and abstract machine state, describing an abstract machine. *Lowering*[6] is the process of bridging this gap, that is, specializing a machine-independent Alewife specification to a machine-dependent specification.

As explained in Section 3, this requires a definition of each abstract Alewife element suitable for the selected machine. One possible source of these definitions is the machine description itself. For OS-independent concepts, it is reasonable to create a single abstract machine-independent model and instantiate it in each machine description. For example, in practice every machine of interest has a stack pointer register; thus we include in each machine description a definition of stackpointer as an alias for the proper register. This is accompanied by a definition for the expected alignment of the stack pointer on entry into C code and a boolean indicating whether stacks conventionally grow up or down. We also have OS-independent models for calling conventions, position-independent code, interrupt enable/disable state, and certain pieces of trap handling state, such as the address at which a trap occurred. OS-independent models for cache and TLB flushes are the subject of ongoing research. Any OS using our machine descriptions is free to use these models or ignore them. For example, our interrupt state model consists of one definition: interrupts_are_on. This is sufficient for the OSes in our use cases, but an OS with traditional interrupt priorities needs a more complex model; for example it might associate an interrupt mask with each priority level and provide a definition that returns the hardware's current interrupt mask. A slightly more complex model might provide a definition that returns the current priority level by interpreting the hardware interrupt mask. That interpretation is itself OS-dependent, and since OS-dependent definitions do not properly belong in the machine description itself, they must appear elsewhere. We place them in *lowering files* (Ⓒ in Figure 2).

Lowering files are Cassiopea-language files that contain OS-specific and machine-specific definitions suitable for instantiating Alewife abstractions. To improve sharing, lowering files are organized in terms of *modules*; each file can contain arbitrarily many modules and each module is a collection of arbitrary Cassiopea declarations and, optionally, a frame clause. Both lowering file modules and Alewife specifications can import zero or more lowering modules by name.

The Alewife compiler (Ⓓ in Figure 2) reads the lowering files, extracts requested modules from them, and lowers Alewife specifications by replacing machine-independent types, definitions, and specifications with their machine-dependent counterparts. The definitions that go into the lowering files must be written as part of producing a new OS port. Most of them derive from OS design decisions about how the OS interacts with machine-dependent abstractions.

Other lowering definitions arise from required machine-*specific* logic: for example, MIPS and RISC-V both have a "global pointer" register, which is used (in some cases) to optimize access to global data. When applicable, it needs to be initialized to a fixed (but linker-chosen) value in key places. This requires a suitable set of abstractions to use in specifications. The lowering files for MIPS and RISC-V supply the needed definitions; for other machines the definitions are empty. Note that while the set of possible machine-specific concepts requiring hooks in machine-independent specifications or code is infinite, and any new machine that appears can produce novel ones

---

[6]In abstract interpretation, a concretization function typically maps an abstract state to a *set* of concrete states. Our lowering chooses a single concrete specification, so we avoid the term "concretization". However, like a concretization function, lowering files map something abstract (an Alewife specification) to something concrete (a machine-dependent specification).

```
                    Listing A. MIPS lowering file                          Listing B. ARMv7 lowering file
1   lowering disp_defs {                              1   lowering disp_defs {
2     let pc_reg: register = r5                       2     let pc_reg: register = r14
3     let disp_reg: register = r6                     3     let disp_reg: register = r2
4     let disp_area_reg: register = r4                4     let disp_area_reg: register = r1
5     let DISP_MAX: int = 268                         5     let DISP_MAX: int = 270
6     def get_crit_ptr base: word->word=              6     def get_crit_ptr base: word->word=
7       base b+ 0x00000058 }                          7       base b+ 0x00000058 }
8   lowering may_use_flags { }                        8   lowering may_use_flags {modify:N Z C V}
9   lowering disp_scratch {modify: r2}                9   lowering disp_scratch { }
```

Fig. 6. Lowering files for disp_check, for use with the Alewife specification in Figure 4.

that require more hooks to be added (or even require OS machine-independent code to be altered to accommodate the new ideas), it should be stressed that this is not a new problem, in the sense that it occurs regardless of what porting technology the OS uses.

Figure 6 shows the MIPS and ARMv7 lowering files used to specialize our example Alewife specification in Figure 4. Each defines the same three modules: disp_defs, may_use_flags, and disp_scratch.

The disp_defs module supplies values for registers, a constant, and a function for lookup in the dispatcher structure. Lines 2–4 supply concrete registers for pc_reg, disp_reg, and disp_area_reg. Line 5 declares the concrete value for DISP_MAX, which gives the dispatcher structure size; it is instantiated with 268 on MIPS and 270 for ARMv7. Lines 6–7 define the abstract function get_crit_ptr, which computes the address of a specific field in the structure. The offset is the same for both MIPS and ARMv7, because both machines have 32-bit words and pointers, but is different for 64-bit machines.

The may_use_flags and disp_scratch lowering definitions relax frame conditions. Different machines can require different numbers of scratch registers to accomplish the same task. Furthermore, on machines that have a flags word, the specification must include explicit permission to modify the flags. The **modify** *frame condition*, as seen in Figure 6, indicates additional machine state (processor flags and scratch registers) that any specification using this lowering module may modify. The MIPS implementation of disp_scratch makes register r2 modifiable, so it can be used as a scratch register (line 9 in Figure 6A), while the may_use_flags module is empty because there are no processor flags on MIPS. ARM needs no scratch register, but may_use_flags makes four control registers N, Z, C and V (the basic processor flags) modifiable (line 8 in Figure 6B).

We use the lowering definitions for two further purposes: first, to define machine-dependent context structures (such as trap frames) and predicates about them, and second, to allocate registers in procedures.

Currently we write all the lowering definitions by hand, including the identity and number of scratch registers. However, we expect other tools in Aquarium can provide some of these definitions automatically. In particular, register allocation (both placing values in registers as they flow from block to block, and choosing scratch registers) is a well-studied problem. Extracting offsets and sizes of structures for use in assembly code is also easily done; almost every OS already has a tool for this[7], and adjusting the output format to make it available to Aquarium is trivial. We have left integrating as-needed use of scratch registers into the synthesis mechanism as future work.

In Section 7, we distinguish lowering definitions that we expect to be automatically generated (those defining context structures and those doing register allocation, including scratch registers) from the other, essential,

---

[7]For example: https://anonhg.NetBSD.org/src/file/tip/usr.bin/genassym/

```
               Listing A. MIPS specification                     Listing B. ARMv7 specification
 1   letstate DispMem:                             1   letstate DispMem:
 2     32 bit 268 len 32 ref memory                2     32 bit 270 len 32 ref memory
 3   frame: modify: r2                             3   frame: modify: N Z C V
 4   let crit_pc: 32 bit =                         4   let crit_pc: 32 bit =
 5     [DispMem, 0] b+ 0x00000058                  5     [DispMem, 0] b+ 0x00000058
 6   let crit: bool =                              6   let crit: bool =
 7       *r5 b< fetch(crit_pc, 32)                 7       *r14 b< fetch(crit_pc, 32)
 8   pre: ( *r6 == [DispMem, 0] ) &&               8   pre: *r2 == [DispMem, 0]
 9     ( *r0 == 0x00000000 )                       9   post: if crit then *r1 = 0x00000001
10   post: (if crit then *r4 = 0x00000001         10       else *r1 = 0x00000000
11     else *r4 = 0x00000000) &&
12     ( *r0 == 0x00000000 )
```

Fig. 7. Generated machine-dependent specifications for disp_check.

lowering definitions that must be written by hand. The context structure lowerings in particular are large (since they include multiple assertions about each register involved, and there are a lot of registers), but only the essential lowering definitions represent porting effort. In fact, we already have a tool in the broader Aquarium ecosystem that generates context structures and the necessary predicates about them [Holland 2020].

Regardless of whether they are automatically generated or manually written, for a given OS and architecture, some lowering definitions are used repeatedly in different procedures. For example, the module may_use_flags (line 8 in Figure 6B) is used in every block that might involve conditional execution, because on ARM and other machines with a flags word, it must be possible to set the flags so conditional executions can test them.

The declarations in a lowering module are specifically permitted to contain memory regions: some specifications require memory regions on some machines and not others. For example, the blocks that load the "global pointer" register on MIPS and RISC-V require a memory region (with an assembler label) that defines the address to be loaded; this is not present on other machines. We do not, however, permit adding new machine instructions in lowering files.

Machine-Dependent Specifications

With the Cassiopea machine descriptions (Ⓑ in Figure 2) and lowering files (Ⓒ in Figure 2), the Alewife compiler (Ⓓ in Figure 2) generates machine-dependent specifications (Ⓔ in Figure 2) for disp_check in MIPS and ARMv7, shown in Figure 7. In Figure 7A, the machine-dependent specification also includes the MIPS invariant in the pre- and postcondition, ensuring that r0 is always zero (line 9 and 12 in Figure 7A). We include more details about the lowering procedures in Appendix A.7.

6 SYNTHESIS WITH THE CASSIOPEA SYNTHESIS ENGINE

The Cassiopea synthesis engine (Ⓕ in Figure 2) operates in two modes. In synthesis mode, it takes as input a Cassiopea machine description (Ⓑ in Figure 2) and a machine-dependent specification generated by the Alewife compiler (Ⓔ in Figure 2) and synthesizes and verifies assembly programs (Ⓖ in Figure 2). In verify-only mode, it takes as input a sequence of operations, a machine-dependent specification, and a Cassiopea machine description and it verifies that the operations satisfy the specification. We use this latter mode extensively to debug both specifications and machine descriptions.

```
          Listing A. MIPS operation sequence                    Listing B. ARMv7 operation sequence
1   (LW r2 r6 0x0058)                             1   (LDR_imm r0 r2 0x058 0b1110)
2   (SLTU r4 r5 r2)                               2   (CMP_reg r14 r0 0b1110)
                                                  3   (MOV_imm r0 0x0 0x01 0b0011)
                                                  4   (MOV_imm r0 0x0 0x00 0b0010)


          Listing C. MIPS assembly program                      Listing D. ARMv7 assembly program
1   lw $2, 88($6)                                 1   ldr r0, [r2, #88]
2   sltu $4, $5, $2                               2   cmp lr, r0
                                                  3   movlo r0, #0x00000001
                                                  4   movhs r0, #0x00000000
```

Fig. 8. Synthesized operation sequences and the verified assembly programs for disp_check.

Figure 8 shows the MIPS and ARMv7 sequences synthesized for the disp_check example. Figure 8A uses LW to load the contents of the memory location [DispMem,88] into scratch register r2 and then uses SLTU to compare and set r4 based on the result of the comparison. By contrast, Figure 8B uses LDR_imm to load the value stored in [DispMem,88] into r0, then uses CMP_reg to compare it with r14. Finally, it uses MOV_imm conditionally twice to set r0 to either 0 or 1. Implicit frame conditions prevent the synthesized code from modifying other general-purpose registers or other fields of the control registers. It is possible that there exist multiple sequences that satisfy the pre- and postconditions; our synthesis engine takes the first specification-satisfying sequence of operations it finds (working from smaller to larger programs) as the synthesized result, and it may produce different results in multiple runs. If the specification is incomplete, the synthesized results can be functionally different. This is common in assembly, because, in general, it does not matter what gets left behind in scratch registers. After verification, our synthesis engine extracts an operation sequence to a syntactically valid assembly program, Figures 8C and D.

Our synthesis engine uses symbolic execution to compile a program to a representation in first-order logic, allowing the use of a satisfiability-modulo-theories (SMT) solver to perform the synthesis itself. It uses several solvers: Boolector [Brummayer and Biere 2009], Z3 [De Moura and Bjørner 2008], and Yices [Dutertre 2014]. Our representation uses the quantifier-free logic of finite bitvectors QF_BV for efficient SMT solving. We developed several novel optimization techniques to improve the scalability of program synthesis, especially synthesis for assembly language. In the rest of this section, we describe the main components of our synthesis engine and several optimizations critical for scaling (Section 6.3).

6.1 Syntax-Guided Synthesis

Our synthesis engine implements syntax-guided synthesis for assembly programs. We model assembly programs as sequences of instructions, each of which has a name and zero or more operands (arguments). As in standard syntax-guided synthesis [Alur et al. 2013; Solar-Lezama et al. 2008], we use a *symbolic program* composed of symbolic instructions to represent sets of possible programs. Symbolic programs are parameterized by *control variables* that represent the choice of operation (i.e., the opcode) and arguments.

Given a symbolic program and a specification, the goal of synthesis is to select a correct program by choosing an appropriate value for each control variable. We first use symbolic execution (Section 6.2) to generate correctness conditions, then use Counterexample-Guided Inductive Synthesis (CEGIS) to find a satisfying assignment of the control variables.

The CEGIS loop iterates *guess* and *verify* phases until it either obtains a correct program or fails. In the guess phase, the solver tries to find an assignment to the control variables, producing a concrete program candidate. In the verify phase, the solver tries to find an initial state, called a *counterexample*, that satisfies the precondition but causes the candidate to falsify the postcondition. If it succeeds in finding such a counterexample, we attempt another guess; otherwise, the candidate is a correct program.

6.2 Symbolic Execution

Symbolic execution for program synthesis explores every execution path to produce a summary of a program's semantics. Since evaluation in Cassiopea always terminates, we do not need any special techniques to approximate loop semantics.

We implemented a symbolic execution engine to efficiently execute assembly programs described in Cassiopea. Like both Sketch [Solar-Lezama 2008] and Rosette [Torlak and Bodik 2014], we represent symbolic values with a DAG and take advantage of this representation to evaluate operations concretely whenever possible. We incorporate simplification techniques from both Sketch and Rosette for efficiency. From Sketch, we incorporate backwards value-set analysis, enabling us to aggressively simplify pre- and postconditions. From Rosette, we incorporate symbolic reflection and structural merging, enabling us to represent higher-level structures (such as Cassiopea values and machine states) separately from SMT constraints, while curtailing the path explosions associated with symbolic pointers.

Although we built our synthesis engine as a standalone system, we believe that we could have extended Rosette, which would have allowed us to take advantage of its symbolic execution engine. For instance, it should be possible to incorporate the simplification techniques from Sketch into Rosette either by extending Rosette's internal symbolic value simplifier or by using Rosette's symbolic reflection interface. Rosette's default type-directed merge alone does not automatically handle the path explosions associated with symbolic pointers, since the merging behavior we desire depends on the pointer's memory region. However, it should be possible to implement our desired merging behavior using Rosette's symbolic reflection interface. Ultimately we chose to implement our own engine, because we found it easier to debug. In retrospect, our optimizations would have been more generally useful had we implemented them as Rosette extensions.

Symbolic pointers. Cassiopea's semantics of bitvectors and pointers requires special treatment for efficient symbolic execution. Symbolic pointers generate two primary problems for symbolic execution: path explosion on reading memory and path explosion due to irregular representation. Careful rewriting is often necessary to prevent combinatorial blowup in symbolic execution; we developed a set of rewriting rules that work well for Cassiopea machine descriptions.

Accessing memory at a symbolic offset. Suppose p is a symbolic pointer (x_{mem}, N) where x_{mem} is a memory region and N is a symbolic offset. To represent the result of reading memory at p, we elected to use an if-then-else (ITE) of the values in x_{mem} at the possible values of N, similarly to ANGR [Shoshitaishvili et al. 2016]. Likewise, when writing to memory at a symbolic offset, we update the memory at each possible offset to an appropriately-guarded ITE of the old and new values.

We considered using uninterpreted functions (UIFs) to represent memory, similar to Serval [Nelson et al. 2019]. However, in early experimentation, we found that UIFs produced significant slowdown in candidate guessing compared to the ITE-based approach. Although UIFs productively accelerate verification for Serval, our synthesis engine spends more than 90% of its run time in candidate guessing. Furthermore, most of our use cases have fairly small memory regions. In combination with our pointer merging approach, this means that the extra merging generated by the ITE-based approach does not significantly affect the performance of symbolic execution.

Path explosion on reading memory. Accessing memory with a symbolic pointer can lead to path explosion. Taming this path explosion requires structural merging for pointer values. However, the merge rule must be selected carefully, as we illustrate with the following example. To merge m concrete pointers q_i for $i \in \{1, \ldots, m\}$, a direct approach would be to merge the regions and offsets, producing a pointer q with a symbolic region and a symbolic offset. But reading memory at q may result in path explosion: since we could end up reading m offsets from each of m distinct regions, the resulting ITE could contain up to m^2 distinct branches. Moreover, all but m of these branches are infeasible.

To avoid this inefficiency, our synthesis engine merges only the offsets of pointers that point into the same region. For example, we rewrite the symbolic value (ite g (R1, x) (R1, y)) (representing the symbolic pointer $(R1, x)$ if g is true and symbolic pointer $(R1, y)$ otherwise) as (R1, (ite g x y)), because the symbolic pointers reference the same memory region. However, we will not rewrite the symbolic value (ite g (R1, x) (R2, y)). If q is produced by this merge strategy, reading from q can produce at most m branches, one for each of the original concrete q_i.

Path explosion due to irregular representation. Representing symbolic values of type C bit requires care to avoid what Bornholt and Torlak [2018] refer to as *irregular representation*. A symbolic C bit value may be a bitvector, a pointer, or a complex ITE that depends on guards or path conditions introduced by state merging. When this structure has a "regular" (canonical) representation, structural merging can efficiently generate compact results. If the representations are not forced to be regular, they will often diverge structurally and execution results will grow combinatorially, producing path explosion.

For example, consider the result of executing a symbolic ADD instruction such as ADD R1 ? R2 (where ? is a symbolic register) on an initial machine state M. Register R2 in the final machine state contains a symbolic value with up to one branch per possible register. The symbolic values in the initial machine state can themselves be ITEs of bitvectors and pointers, so the result of executing this ADD is often a nested ITE structure. Without careful canonicalization, as the number of symbolic instructions grows, the ITEs become deep, leading to path explosion when the resulting value is later used.

Avoiding such blowups is critical for performance. We selected structural merging rules to ensure that symbolic C bit values have a regular representation. Our synthesis engine rewrites symbolic values that contain both bitvector and pointer branches so that bitvector values are in a separate branch from pointer values, in addition to using the pointer merging approach described above. It is necessary to apply this canonicalization recursively to flatten nested ITEs. Besides mitigating path explosion, this also simplifies the path conditions that result from accessing memory at an address that may not be a pointer.

6.3 Synthesis Optimizations

Our synthesis engine employs several synthesis optimizations: read/write constraints, state gating, dependency constraints, and rule-based decomposition. Read/write constraints, dependency constraints, and rule-based decomposition are generally applicable techniques, although our specific rules are mostly assembly language specific. Read/write and dependency constraints are particularly powerful in assembly language synthesis but likely apply to other imperative contexts. Applying these optimizations produces an average speedup of 21× on our use cases, achieving 1380× in one case. We further evaluate these optimizations in Section 7.6.

6.3.1 Read/Write Constraints. We add constraints to restrict the registers that an operation is allowed to read, based on the specification and preceding operations. Specifically, an operation may read only those registers either mentioned by the specification (including frame conditions) or written by preceding operations. We call these *available* registers. For each operation, we add constraints to ensure that any registers it reads are contained in the available register set.

We note that in some machines, operations may read some registers (e.g., control registers) implicitly. We annotate these registers with the **control** keyword (as shown in Figure 5) and include them in the set of available registers for each operation.

Intuitively, registers not in the available register set contain arbitrary values, since they are neither constrained by the specification nor defined by preceding operations. As such, we expect that they should not be read in candidate programs.

6.3.2 State Gating. State gating removes registers and operations that are not needed to produce a correct implementation for a machine-dependent specification. If a register is not mentioned in a precondition, postcondition, or frame condition of a specification, it contains an arbitrary value that is not restricted by the specification, and we consider it irrelevant. A register not mentioned in the postcondition or in a frame condition may not be modified by a correct implementation; registers that are additionally not mentioned in the precondition also do not contain information that needs to be read.

We cannot delete all unnecessary registers immediately, because assembly instructions often access registers implicitly; for example, many instructions in 32-bit ARM have conditional versions that access flag bits in the control register. We analyze the machine description to find the registers that operations can access together with relevant registers and add these to the set of registers that should be retained. We forbid read/write access to all other registers by removing them from the machine description. We additionally remove operations that require an access to an irrelevant register, either explicitly or when all possible values for an operand are irrelevant.

6.3.3 Dependency Constraints. We analyze the specification and machine description to discover dependency constraints to further prune the search space. Where a specification ensures that the final value of a location is uniquely determined by the initial state, we determine which values in the initial state affect it. We then require that guessed programs exhibit these dependencies, i.e., the computation of the final value does indeed depend on the relevant parts of the initial state.

We find locations ℓ that are uniquely determined by the initial state by querying whether there exists an initial state for which the postcondition holds for two distinct values in ℓ:

$$\exists \sigma, \sigma', \sigma'' . \operatorname{pre}(\sigma) \wedge \operatorname{post}(\sigma, \sigma') \wedge \operatorname{post}(\sigma, \sigma'') \wedge \sigma'[\ell] \neq \sigma''[\ell]$$

If this query is UNSAT, then for any initial state, the final value in the location is uniquely determined.

For such a location ℓ, we then perform a series of SMT queries to identify on which universally quantified variables in the precondition the final value of ℓ depends. Specifically, for each variable x in the precondition, we ask whether there exists an initial state σ and distinct values v_1 and v_2 such that $\sigma[x \mapsto v_1]$ and $\sigma[x \mapsto v_2]$ produce two different correct values for location ℓ:

$$\exists \sigma, \sigma', \sigma'', v_1, v_2 . \operatorname{pre}(\sigma[x \mapsto v_1]) \wedge \operatorname{pre}(\sigma[x \mapsto v_2])$$
$$\wedge \operatorname{post}(\sigma[x \mapsto v_1], \sigma') \wedge \operatorname{post}(\sigma[x \mapsto v_2], \sigma'')$$
$$\wedge \sigma'[\ell] \neq \sigma''[\ell]$$

If this query is SAT, then ℓ depends on x.

We add constraints to ensure that guessed programs exhibit these dependencies; we analyze the symbolic program to compute an assertion on control variables that ensures that variable x is used in the computation of location ℓ.[8] Intuitively, these assertions are effective in accelerating guessing, because they enable the solver to quickly reject programs that do not exhibit the appropriate dependencies and therefore cannot possibly be correct.

---

[8]The assertion is a necessary but not sufficient condition that x is used in the computation of ℓ, i.e., our program analysis overapproximates when x might be used.

Algorithm 1 Rule-based decomposition procedure.

1: **procedure** Synthesize(s)   ▷ Given starting specification s.
2:      $S \leftarrow \{\text{Hole}(s, 1)\}$ ▷ Begin with hole for one instruction.
3:      **while** $S \neq \emptyset \wedge$ not timed out **do**
4:          $T \leftarrow \text{TakeBest}(S)$
5:          $h \leftarrow \text{FindFirstHole}(S)$
6:          **if** $h = \bot$ **then**   ▷ No holes remain; S is a complete program.
7:              **return** S
8:          **end if**
9:          $\text{Hole}(g, n) \leftarrow h$ ▷ Otherwise, h is the current hole with spec g and size n.
10:         $P \leftarrow \text{Sygus}(g, n)$
11:         **if** $P \neq \bot$ **then**   ▷ Syntax-guided synthesis succeeded at n instructions.
12:             $S \leftarrow S \cup \{T[h \mapsto \text{Prog}(P)]\}$ ▷ Replace current hole with found program P.
13:             **continue**
14:         **end if**
15:         **for** $(g_1, \ldots, g_m) \leftarrow \text{Decompose}(g)$ **do**   ▷ For each subgoal decomposition, create
16:             $N \leftarrow \text{Seq}(\text{Hole}(g_1, 1), \ldots, \text{Hole}(g_m, 1))$ ▷ a sequence of holes, one per subgoal.
17:             $S \leftarrow S \cup \{T[h \mapsto N]\}$ ▷ Replace current hole with sequence.
18:         **end for**
19:         $S \leftarrow S \cup \{T[h \mapsto \text{Hole}(\sigma, n+1)]\}$ ▷ Increase current hole's search size.
20:     **end while**
21: **end procedure**

6.3.4 Rule-Based Decomposition. Our synthesis engine augments syntax-guided synthesis with a heuristic rule-based decomposition algorithm to achieve performance and scalability. Rule-based decomposition uses a set of machine-independent *decomposition rules* that decompose a specification into specifications for smaller subprograms ("subgoals"). Our algorithm combines decomposition and syntax-guided synthesis in an iterative scheme, alternately using rules to decompose subgoals and using pure syntax-guided synthesis to discharge subgoals. The procedure is described in Algorithm 1.

Rule-based decomposition allows us to improve performance over syntax-guided synthesis. Syntax-guided synthesis scales poorly with increasing program length; decomposing the specification into subgoals for smaller programs reduces the size of each syntax-guided synthesis invocation, significantly accelerating synthesis overall. Meanwhile, using syntax-guided synthesis as a final code generation stage allows our rules to remain machine-independent.

Search Algorithm. During synthesis, our synthesis engine performs a search over possible trees of rule applications to the input specification instance, maintaining a set of trees encountered (S in Algorithm 1). Each tree T corresponds to one possible subdivision of the specification into subgoals; each subgoal has an instruction count n, which is an upper bound on the number of instructions that will be used to satisfy the subgoal. In each iteration, the synthesis engine selects a tree from this set and a subgoal from the tree, and invokes syntax-guided synthesis up to n instructions to try to discharge the subgoal (line 10). The synthesis engine selects the next tree to attempt according to a cost heuristic (TakeBest); our heuristic scores a tree by the lengths of its remaining subgoals, weighting each subgoal exponentially in its length.

If syntax-guided synthesis fails on a subgoal, our synthesis engine attempts to decompose the subgoal by applying each rule (Decompose, line 15). Each new tree obtained is added to the set (line 17). The synthesis

engine also retains the original tree, increasing the subgoal instruction count to $n + 1$ for later exploration (line 19). If syntax-guided synthesis succeeds, the synthesis engine stores the partially discharged tree and continues (lines 11–13). If the synthesis engine discharges a tree's last remaining subgoal, it combines the results to produce a program that is correct with respect to the original goal (line 7).

Rule Selection. Our implementation of rule-based decomposition includes five rules. All of our rules are specific instances of sequential composition; they generate subgoals such that $post(g_i) = pre(g_{i+1})$. As such, subprograms that satisfy the generated subgoals may be composed by sequence to produce a correct program for the input goal.

- LoadMem. Given goal g, if a value v exists in memory that is not in a register, generate two subgoals (g_1, g_2). Subgoal g_1 has $pre(g_1) = pre(g)$ and a postcondition that requires v in a scratch register. Subgoal g_2 has $post(g_2) = post(g)$.
- LoadLbl. Given goal g, if a label ℓ exists whose value (an address) is not in a register, generate two subgoals (g_1, g_2). Subgoal g_1 has $pre(g_1) = pre(g)$ and a postcondition that requires ℓ in a scratch register. Subgoal g_2 has $post(g_2) = post(g)$.
- SetReg. Given goal g, if $post(g)$ contains a register r whose value is uniquely determined and is wrong, then generate two subgoals (g_1, g_2). Subgoal g_1 has $pre(g_1) = pre(g)$ and a postcondition demanding the correct value in register r. Subgoal g_2 has $post(g_2) = post(g)$.
- SetMem is akin to SetReg, but for memory locations.
- FindPtr. When a subgoal g was generated by LoadMem or SetMem, generate subgoals (g_1, g_2). Subgoal g_1 has $pre(g_1) = pre(g)$ and a postcondition demanding that a scratch register contain a pointer to the relevant address. Subgoal g_2 has $post(g_2) = post(g)$.

We base our rule selection on observations of common properties of OS assembly code and assembly languages. First, OS assembly code typically performs simple calculations and otherwise serves only to move or set state. Often, these moves and sets are performed by independent program fragments. Second, many assembly languages support arithmetic over only a subset of their registers; a significant fraction of assembly programs is dedicated to moving values for calculations then storing the results. We use rules that generate (1) independent subgoals for programs to satisfy independent pieces of state and (2) subgoals that move data to and from general-purpose registers. Due to our choice of rules, our algorithm does not speed up synthesis for programs that mainly perform arithmetic.

Our rules are nondeterministic: more than one rule may be applicable to a specification, and one rule may apply several ways. For instance, SetReg may apply to several registers in a specification. Our rules are not, in general, invertible [Liang and Miller 2009]. Applying an invertible rule to a derivable goal will produce only derivable subgoals. Applying non-invertible rules may require backtracking. Our rules are also incomplete: it is possible that no rule applies to a given specification instance, in which case our synthesis engine regresses to normal syntax-guided synthesis. However, our rules are sound: if synthesis succeeds on the generated subgoals, then the resulting program is correct against the input specification instance.

Separation Logic. Internally, we use a separation-logic-like representation of the specification, inspired by SuSLik's [Polikarpova and Sergey 2019] use of separation logic to represent synthesis goals for imperative heap-manipulating programs. This allows our rules to structurally decompose specifications by simple pattern-matching.

7 VALIDATION

We validate the expressiveness of Cassiopea by successfully modeling four real machine architectures—32-bit MIPS, 32-bit ARM, 32-bit RISC-V and x86_64—and using them to synthesize runnable code.

From two preexisting OSes—Barrelfish [Baumann et al. 2009] and OS/161, a BSD-like teaching OS [Holland et al. 2002]—we took eight complete machine-dependent procedures (written either entirely in assembly language or in a mixture of C and assembly language) and split them into semantically meaningful steps. We implement each step with a block of code, which is either a synthesis use case or out of scope for synthesis. We evaluate 50 blocks in all: 35 are synthesis use cases, and 15 are out of scope. Of the latter, 10 are single call or return instructions, two are single system call instructions, and the other three are machine-*specific* exception operations. Section 7.1 discusses the use cases in more detail.

We validate the expressiveness of Alewife by successfully writing machine-independent specifications for each of the blocks not explicitly out of scope.

We validate our implementation as follows:

- We use the Alewife compiler to lower each machine-independent specification for each architecture, generating a total of 140 machine-dependent specifications for synthesis.
- We use our synthesis engine to verify hand-written assembly code for all of these machine-dependent specifications.
- We use our synthesis engine to synthesize and verify assembly code for all 140 machine-dependent specifications.

In the remainder of this section, we explain the targeted machine-dependent procedures and the four real machine architectures modeled in Sections 7.1 and 7.2, respectively. Sections 7.3 and 7.4 report the information about the corresponding files, while Sections 7.5 and 7.6 discuss verification and synthesis performance.

7.1 Use cases

We selected two Barrelfish and six OS/161 complete machine-dependent procedures, most of which were originally written entirely in assembly. Each procedure comes from a specific port of the original OS and performs some specific, recognizable machine-independent task in the appropriate machine-dependent manner for its target machine. We selected the procedures without significantly refactoring the original OS kernel. The procedures include interrupt and exception handling, system call handling, user-level startup code, context switches, and other small OS components. We chose the procedures to be representative of tasks that machine-dependent kernel code and user-level OS code must perform in assembly language: changing processor state and manipulating registers in ways not expressible in C. There are two chief categories missing: one is kernel startup code, which often contains large amounts of machine-*specific* code, and the other is memory management code, such as cache control and TLB management. We successfully synthesized some memory management code blocks, but do not include them in the paper: it is difficult to find an example in existing code bases that is both manageable and explainable without extensive refactoring. We believe the chosen procedures are largely representative of the synthesizable assembly programs in our scope.

Note that refactoring existing OSes to have clean machine-independent interfaces for machine-dependent code is not a goal for our system or our validation. For example, the native Barrelfish trap handling code for ARM and x86_64 is quite different, because each port was handwritten separately. Based on advice from the Barrelfish team, our use cases are based on the ARM version. This means that while we generate x86_64 code, that code cannot be easily incorporated into the existing x86_64 Barrelfish port, which needs a different specification. However, in all cases the code for the original architecture (ARM for Barrelfish, MIPS for OS/161) is suitable for incorporation into the existing OS implementation. (We include handwritten code for out-of-scope blocks.) Reorganizing the OSes with a uniform structure across all ports, and factoring out code that could be machine-independent but is gratuitously different in each port, so that specifications are not needlessly machine-dependent and synthesis results can be seamlessly plugged in for all machines, is a large project in its own right and the subject of ongoing research.

Our goal of synthesizing assembly procedures to implement portions of machine-dependent OS functionality is difficult due to the challenge of scaling assembly code synthesis, while balancing such scalability with the expressiveness and complexity of the languages that describe OS functionality and processor semantics. We adopted a decomposition model, where each procedure is explicitly (and manually) divided into a sequence of machine-independent steps each performing a specific recognizable task, because most of these procedures are too large to synthesize all at once. (One could imagine letting our synthesis engine run for longer periods of time to synthesize complete procedures; unfortunately this inherently requires exponentially longer periods of time.) A given block might be empty for a given machine, but no block is empty for all machines.

As we synthesize only loop-free blocks, we use program labels and external branches for all control flow other than forward branches within a block.

As mentioned earlier, some blocks are out of scope for synthesis or best handled in other ways, so we generate blocks in different ways. Some of these other generators, such as a context-switch compiler and a tool for automatically composing blocks, are described in other work [Holland 2020]. In the following section, we describe each of the blocks amenable to synthesis.

Barrelfish Use Cases

swi-handler (SWI). This is the machine-level system call trap from the ARMv7 implementation. "SWI" stands for "software interrupt", which is ARM terminology for a system call trap. From our machine-independent specifications, we generate comparable system call handlers for MIPS, RISC-V, and x86_64. There are 13 synthesis blocks, of which two are empty on ARM but were necessary for one or more of the other machines.

 (1) spill: save some registers to make room to work.
 (2) check-kern: check whether the trap came from kernel or user mode, and branch to the corresponding code.
 (3) load-got: get the global offset table base address on machines that need it in a register. (The global offset table is used for indirect addressing in position-independent code.)
 (4) get-disp: load the address of the current dispatcher, the Barrelfish process structure equivalent.
 (5) check-disabled: check whether the dispatcher was in "disabled" mode, and if so, branch to the subsequent set-disabled block.
 (6) findpc: load the exception program counter into a general purpose register on machines where this is needed, which does not include ARM.
 (7) check-low: check the exception program counter against the lower bound of a critical region, and branch to the set-enabled block if below it.
 (8) check-high: check the upper bound and branch to the set-disabled block if below it.
 (9) set-enabled: load the address of the "enabled" register save area in the dispatcher.
 (10) set-disabled: load the address of the "disabled" register save area.
 (11) reshuffle: after saving registers, move values around as needed to avoid clobbering them in later blocks. Empty on ARM.
 (12) initstack: load the stack pointer to point to the kernel stack.
 (13) moveargs: set up the arguments for the C system call handler.

cpu-start (CS). This is the entry point for application-core kernels[9] from the ARMv7 port. It consists of two parts. The first part is considered as a synthesis block:
 (1) store-got: on machines that need the global offset table in a register, fetch its base address from the argument structure, and store it where the trap handler expects to find it later.

---

[9]A Barrelfish system uses a single core running a special kernel to initialize the system. The remaining cores then boot with a kernel for cores running applications, including the system Monitor process.

The second part is the same as the initstack block from the swi-handler example (SWI-12), so we reuse the same Alewife specification for this part and do not count it as an extra synthesis block for the evaluation.

OS/161 Use Cases

setjmp (SJ). This is the C standard library function setjmp. It stores callee-saved registers and the stack pointer and returns 0. It is composed of two blocks:
   (1) saveregs: save registers.
   (2) retval: set the return value.

longjmp (LJ). This is the C standard library function longjmp. It reloads the registers saved by setjmp and and arranges to return a caller-supplied value from the setjmp call. It is compsed of two blocks:
   (1) loadregs: load registers.
   (2) retval: set the return value, adjusting it if needed.

crt0 (CRT). This is the startup code (crt0) for user-level programs. It contains seven synthesis blocks:
   (1) initstack: ensure that the stack pointer is aligned correctly according to the function call ABI. As OS/161 is a teaching operating system, it does not assume that the initial stack pointer provided by the kernel is correctly aligned.
   (2) savevals: save the values of argv and environ, passed in registers from the kernel, into the private global variables libc uses to hold them.
   (3) initregs: perform machine-dependent register initializations, such as the global pointer register used by some RISC machines.
   (4) mainargplace: place the arguments for calling main.
   (5) exitsave: save the return value from main in a callee-save register.
   (6) exitargplace: place the return value from main as the argument to exit.
   (7) loop: unconditional branch back to repeat the call to exit in case it returns. (While exit normally does not return, exit is also student code; the loop is a precaution to avoid bizarre behavior when it does not work as intended.)

syscallstub (SYS). This is a user-level system call stub. It has three synthesis blocks:
   (1) loadnum: load the system call number into the appropriate register. Empty on machines where the call number is placed in the system call instruction.
   (2) jump: unconditional jump from per-system-call code to shared code.
   (3) seterrno: set the C-level return value and errno from the kernel-level return value.

cpu-irqoff (IRQ). This procedure turns interrupts off for the current processor. It is just one block:
   (1) irqoff: turn off the interrupts.
(Turning interrupts on is effectively the same and omitted for brevity. A related function for idling the processor consists mostly of interrupt state manipulation, along with a special instruction for idling that we do not attempt to synthesize, and is also omitted.)

thread-switch (TS). This is a complete kernel-level thread switch; it saves the state of the current thread and restores the state of the next thread to run. It consists of six blocks:
   (1) entry: make space on the stack to save registers.
   (2) saveregs: write the old thread's registers to the stack.
   (3) savestack: save the old thread's stack to its thread structure.
   (4) loadstack: load the new thread's stack from its thread structure.
   (5) loadregs: read the new thread's registers from the stack.

(6) cleanup: clean up the stack for return.

7.2 Machine Descriptions

ARM. The 32-bit ARM model has three-operand instructions, 16 registers, and a status register. We do not include Thumb instructions, the 16-bit variant of the ARM ISA. Most ARM instructions are all conditionally executed based on the flags in the status register. We model the status register as multiple smaller register fields, as direct access to the entire status register is rare in ARM code. In ARM, many immediate values are represented as an 8-bit field rotated right by an even 4-bit shift amount between 0 and 30. We model this directly and thus emit only valid immediates. This is necessary, because the ARM assembler will not emit extra instructions to encode impossible immediates. We do not include multiple data transfer instructions such as LDM/STM, since they can always be replaced by multiple single load/store and arithmetic instructions.

MIPS. Our MIPS model has three-operand instructions and 32 general-purpose registers. Most kernel-mode phenomena are handled by control registers, which are composed of fixed-size fields; we model each field separately in Cassiopea and concatenate the fields together when accessed as a register. In MIPS, general-purpose register 0 is always zero, and writes to it are discarded. We handle this by treating register 0 as special in every instruction and adding a machine invariant to the Cassiopea machine description (see Figure 5A). We handle the branch delay slots associated with jumps by using the assembler mode that hides branch delay slots.

RISC-V. Our 32-bit RISC-V model has three-operand instructions, 32 general-purpose registers, and a supervisor-mode status register similar to that of MIPS. We again model the status register by separating it into fixed-size fields, concatenating them together when accessing it as a whole. As in MIPS, general-purpose register 0 is always zero, and we encode this as a machine invariant.

x86_64. We modeled x86_64 with AT&T syntax [Narayan 2007]. Our x86_64 model has two-operand instructions, 16 registers, and a flags register interrogated by certain branch instructions. We model the flags register as separate 1-bit registers.

Architecture	Lines	Operations	General Registers	Special Registers	Flags Registers	Control Registers
ARM	933	34	15		4	9 (141 total bits)
MIPS	495	37	32	2		9 (156 total bits)
RISC-V	618	37	32			31 (193 total bits)
x86_64	567	56	16		4	3 (53 total bits)

Table 1. Size of the Cassiopea machine descriptions for each architecture. Control registers are modeled with separate registers for individual fields, so both the counts and the total addressable space vary widely depending on the layout.

Completeness. We implemented 34 operations in ARM, 37 in MIPS, 37 in RISC-V and 56 in x86_64. All models include assembly operations for arithmetic, bitwise logic and shifts, comparison, moves, memory accesses, conditional branching, and supervisor operations. This is a small fraction of the total instructions available on each machine (especially x86_64) but covers all the basic operations. Each machine model also includes all the general-purpose registers, the basic processor flags on machines that have them, and a selection of control registers as needed for our use cases. Table 1 shows line, operation, and register counts for each machine model.

Shortened Machine Descriptions. Because including the full complement of general-purpose registers makes even the simplest synthesis problems extremely slow in the absence of state gating, for each machine we prepared a

single alternate description with most of the general-purpose registers commented out, retaining only 6–8 of the most commonly used ones. We use this shorter description for all the use cases it can support, that is, those that do not require the registers it comments out. The blocks that use the full descriptions are:

- SJ-1, LJ-1, TS-2 and TS-5, whose context operations inherently require the full register set.
- CRT-5 and CRT-6 on MIPS and RISC-V.
- All the SWI blocks on MIPS except SWI-2 and SWI-12.

This trimming reflects what one might readily do by hand in the absence of state gating. (It is, indeed, what we did by hand before we had state gating.) Note that this change is strictly adverse to our evaluation, in that it makes our baseline considerably more performant.

7.3 Alewife Specifications and Lowering

We were able to express all 35 machine-independent specifications in Alewife. We supplied lowerings for all $35 \times 4 = 140$ combinations of machine-independent specifications and machine descriptions.

Table 2 shows the number of lines of code for each machine-independent specification.[10] Table 3 presents the *total* length of the lowering files for each OS and architecture. As discussed in Section 5, we divide these into two categories: "Manual" lowering includes OS-and-machine-specific definitions that need to be hand-written. "Automatable" lowering includes definitions that can reasonably be generated: register allocation, data structure offsets, and specification fragments for context (register-save) structures. We cannot ascribe a specific length to the lowerings associated with each machine-independent specification, since there is significant sharing and reuse. Centralizing machine-dependent definitions in a small set of files heavily reduces manual effort; we consider it an important element of the expressiveness of Alewife and Cassiopea.

Our use cases include 434 lines of Alewife and 67 lines of manual lowering, for a total of 501 lines of specification. This in turn allows us to synthesize 383 lines of code. Because the incremental cost of a new port is just a new set of lowering files, averaged over the four existing ports the ratio of (manual) lowering files to write to lines of code synthesized is 0.175.

This is perhaps somewhat optimistic, because the generation of automatable lowerings requires some specification of its own, some of which may be machine-dependent and need to be provided for each port. A full discussion of those considerations is beyond the scope of this paper, but an absolute upper bound may be given by including the 588 lines of automatable lowerings in the porting effort, which makes the ratio of lines of lowering files to lines synthesized 1.71.

Including the machine-independent specifications the overall effort for four ports is 1.31 lines of specification per line synthesized. This number is perhaps not as appealing, but we consider the incremental porting cost the primary concern; see Section 8.2 for more context.

7.4 Hand-written assembly programs

To help with validation, we obtained hand-written assembly for all use cases on all machines (we either wrote them manually ourselves or took them from existing implementations)[11]

Table 2 reports the size of the assembly program synthesized for each machine-independent specification under all four architectures. Typically, synthesized assembly programs differ significantly in implementation from our hand-written versions; sometimes, the synthesized programs are shorter. For cases where synthesis timed out after 1800 seconds, the reported number in parentheses is the number of assembly instructions in our hand-written implementation.

---

[10]All lines-of-code counts exclude blank lines, comments, and standalone close-braces.

[11]OS/161 ships with MIPS code; we wrote versions for ARM, RISC-V, and x86_64. Barrelfish ships with ARM and x86_64 versions; we wrote MIPS and RISC-V manually, and restructured the x86_64 version as necessary to match the structure of the ARM implementation.

Spec	Alewife (lines)	Verification Time (ms)				Synthesis Time (s)				Assembly (lines)			
		ARM	MIPS	RISC-V	x86-64	ARM	MIPS	RISC-V	x86-64	ARM	MIPS	RISC-V	x86-64
SWI-1	5	60	130	56	43	5.1	0.65	45	1.0	5	2	7	3
SWI-2	5	41	120	43	34	23	23	9.9	0.70	3	4	3	2
SWI-3	7	41	53	41	32	0.15	0.36	0.036	0.031	1	2	0	0
SWI-4	20	63	150	63	55	15	4.6	3.8	4.1	4	4	4	4
SWI-5	13	45	130	44	37	38	2.4	1.5	1.7	3	2	2	2
SWI-6	6	40	120	43	33	0.047	0.24	0.37	0.043	0	1	1	0
SWI-7	12	58	130	47	42	5.4	1.0	0.81	2.0	3	2	2	3
SWI-8	12	48	140	48	42	15	1.0	0.80	2.1	3	2	2	3
SWI-9	12	42	120	42	33	1.6	1.3	1.1	0.67	3	3	3	3
SWI-10	10	41	120	42	34	1.1	0.62	0.54	0.45	2	2	2	2
SWI-11	5	40	120	42	32	0.047	0.19	0.051	0.043	0	1	0	0
SWI-12	13	43	55	42	33	1.7	0.48	1.7	0.45	2	2	3	2
SWI-13	21	41	120	42	33	0.077	1.9	0.082	0.073	0	3	0	0
CS-1	15	52	68	49	38	1.2	1.6	0.050	0.037	2	3	0	0
SJ-1	10	160	240	290	120	43	33	59	9.5	11	11	14	8
SJ-2	5	41	45	41	32	0.28	0.080	0.083	0.064	1	1	1	1
LJ-1	10	140	220	250	110	54	32	58	9.6	11	11	14	8
LJ-2	13	41	49	44	34	9.4	1.2	2.8	14	2	2	2	3
CRT-1	24	46	65	49	55	8.4	19	0.69	0.44	2	3	1	1
CRT-2	14	48	57	52	40	4.2	2.6	2.4	0.63	4	2	4	2
CRT-3	15	40	50	45	33	0.062	0.20	0.20	0.074	0	1	1	0
CRT-4	30	43	65	47	35	0.066	0.089	0.077	0.079	0	0	0	0
CRT-5	7	41	130	120	33	0.22	0.23	0.19	0.20	1	1	1	1
CRT-6	13	42	150	130	35	0.45	0.44	0.25	0.24	1	1	1	1
CRT-7	4	40	45	41	32	0.042	0.053	0.057	0.060	1	1	1	1
SYS-1	9	39	45	41	32	0.035	0.079	0.082	0.064	0	1	1	1
SYS-2	4	40	45	41	32	0.043	0.054	0.057	0.060	1	1	1	1
SYS-3	26	46	51	47	34	370	160	170	12	6	6	7	3
IRQ-1	4	40	46	42	32	0.057	1.6	0.44	0.047	1	4	1	1
TS-1	14	41	47	41	33	0.17	0.11	0.11	0.16	1	1	1	1
TS-2	19	120	190	220	110	24	24	47	9.1	9	10	13	8
TS-3	12	45	51	47	38	0.17	0.12	0.12	0.092	1	1	1	1
TS-4	12	46	51	47	38	0.17	0.12	0.12	0.094	1	1	1	1
TS-5	19	120	210	240	110	30	25	46	9.4	9	10	13	8
TS-6	14	40	46	41	33	0.30	0.11	0.11	0.16	1	1	1	1

Table 2. Performance of our synthesis engine on our machine-independent specifications. For each, we report: (1) the size of the machine-independent specification, (2) verification time for a hand-written implementation (in milliseconds, averaged across 10 trials), and (3) synthesis time (in seconds, averaged across 5 trials) with all optimizations enabled. We also report (4) the size of the synthesized assembly program.

	Barrelfish				OS/161			
Architecture	ARM	MIPS	RISC-V	x86_64	ARM	MIPS	RISC-V	x86_64
Manual Lowering (LOC)	11	10	6	7	8	8	8	9
Automatable Lowering (LOC)	39	33	39	38	106	110	137	86

Table 3. Length of the lowering files, divided into two categories, for each OS and architecture.

7.5 Performance

We measured the performance of our synthesis engine for verification and synthesis. Our measurement platform is an Intel Core i7-7700K clocked at 4.2 GHz with 64 GB RAM; our tests use only one of the four cores. We ran five trials for synthesis and ten trials for verification, and each trial varied the solver RNG seed. We also randomized the variable names used for solver communication by prepending a random five-character alphabetical string, which we find produces significant variance in solver performance.

Verification. Table 2 reports the verification performance of our synthesis engine on all use cases. We verified all machine-independent Alewife specifications against the hand-written assembly programs, which are taken from existing implementations, to support the validity of our specifications. In most cases, the synthesis engine verifies both our hand-written programs and the synthesized implementations in milliseconds. We are able to verify the whole of thread-switch (TS), our longest hand-written program at 26 instructions, in less than one second. During synthesis, our synthesis engine uses verification as part of the CEGIS loop; because guessing is much slower than verification, this occupies only a small fraction of the total synthesis time, often < 1%.

Synthesis. Synthesis in general is difficult, and synthesis of assembly is particularly hard. The size of the space of possible programs is combinatorial in the number of registers, memory locations, and immediate values, and exponential in the number of instructions [Hu et al. 2019]. This especially affects the performance of syntax-guided synthesis. As a first-order approximation, in our machine descriptions, the overall sizes of the search spaces for programs of length n are $2^{40.3n}$ in ARM, $2^{36.6n}$ in MIPS, $2^{36.3n}$ in RISC-V, and $2^{74.7n}$ in x86_64.

Table 2 summarizes the synthesis performance of our synthesis engine on all machine-independent specifications with all optimizations enabled. We ran the experiments with a half-hour timeout; no cases timed out. Our synthesis times vary widely with architecture and machine-independent specification. Examining these use cases more closely, we see that SWI-1 is a register spill, requiring a series of stores followed by a subtraction that sets the stack pointer appropriately: each instruction is relatively independent and no value in the final state requires multiple arithmetic operations to calculate. In contrast, SWI-5 loads a value from memory, then uses it to perform a conditional branch; this requires that the solver reason about conditional execution and arithmetic to produce a correct program. This leads to entirely different synthesis times under each of the four architectures, especially for ARM and RISC-V. Note that our synthesis engine discharges all two-instruction machine-independent specifications in less than ten seconds.

7.6 Optimizations

Our optimizations are critical for performance and scaling, especially as the size of the program to synthesize increases. To measure the effect of each optimization, we compare a baseline of our synthesis engine with no optimizations enabled against the synthesis engine with each single optimization enabled. Figure 9 and Figure 10 show the results.

Many of our optimizations are heuristic: while accelerating synthesis on the majority of the use cases, they may sometimes cause synthesis to slow down. In particular, read/write constraints and dependency analysis work by providing additional constraints to the SMT solver. Experimentally, we find that these help in most cases, especially as synthesis time increases.

Read/write constraints exhibit a significant tradeoff. They slow synthesis for small programs, but for programs that take our baseline more than 100 seconds to synthesize, read/write constraints produce a geometric mean speedup of 1.6×. We observe a maximum speedup of 12×, which occurs on a use case that takes 851 seconds to run on average in the baseline condition. In the worst case, read/write constraints produce a slowdown of 5.0×, which occurs on a use case that takes 4.9 seconds in the baseline condition.

Dependency analysis is often beneficial. It provides additional constraints when values in the postcondition are uniquely determined by the initial state. Dependency analysis makes our synthesis engine up to 35× faster, which occurs on a use case that takes 307 seconds to run on average in the baseline condition. Dependency analysis produces a geometric mean speedup of 1.4×. Dependency analysis induces an overhead typically less than one second, which is significant only for small programs; the effect is visible as a small upward nonlinearity at smaller times in Figure 9c. In the worst case, dependency analysis makes our synthesis engine 2.5× slower, which occurs on a use case that takes 0.96 seconds in the baseline condition.

(a) All optimizations vs. baseline.

(b) With and without read/write constraints.

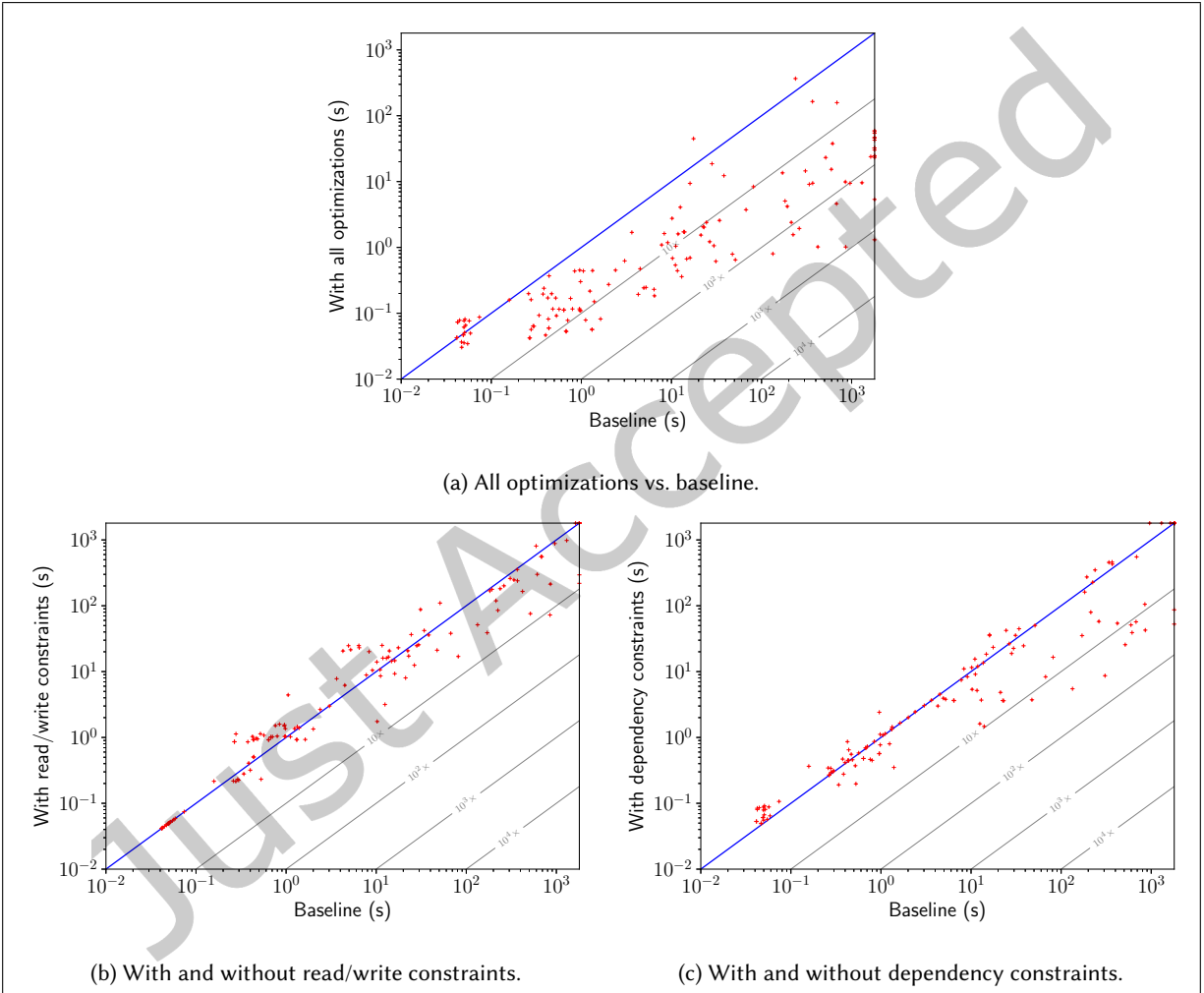(c) With and without dependency constraints.

Fig. 9. Effect of optimizations on synthesis runtimes. Figure 9a shows all optimizations; the other plots show a single optimization enabled, all against our baseline. Each data point represents the runtime under both conditions for one machine-independent specification and architecture, averaged over five trials (timeouts are counted as 1800 seconds). The blue diagonal line represents equal time under both conditions, so that points below/right of the diagonal line demonstrate better performance with the optimization enabled. Gray contours provide guidelines for visually estimating speedup factor. The upper and right boundaries of the plot represent an 1800 second timeout.

(d) With and without rule-based decomposition. (e) With and without state gating.
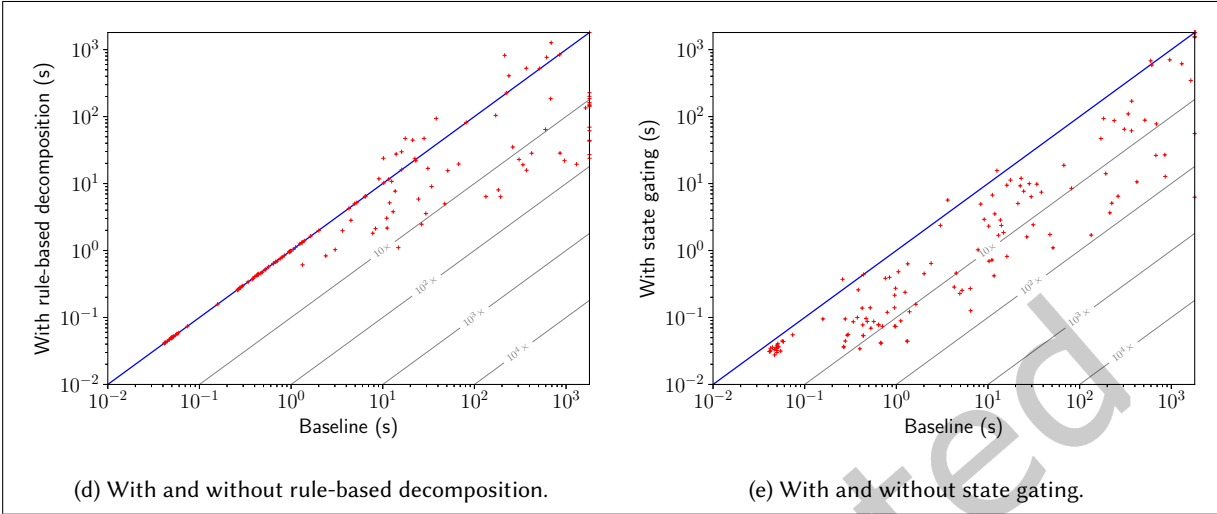
Fig. 9. Effect of optimizations on synthesis runtimes (cont.).

Rule-based decomposition accelerates synthesis when its rules apply and the use case requires a long program; it has sufficiently low overhead that it does not impact performance when its rules do not apply. Rule-based decomposition produces a geometric mean speedup of 2.0×. In the best case, rule-based decomposition synthesizes programs that are not accessible to the baseline, such as context switch code; counting timeouts as 1800 seconds, this is a speedup of 75×. Rule-based decomposition can sometimes slow down synthesis by spending time exploring goal decompositions whose subgoals do not simplify the synthesis task; for example, when a subgoal demands that an unnecessary value be loaded. In the worst case, we observe a slowdown of 3.8×.

State gating is effective, because it reduces the search space directly, limiting the options available to the SMT solver, without ruling out possibly correct programs. State gating makes our synthesis engine up to 290× faster, with a geometric mean speedup of 4.5×. In the worst case, we observe a slowdown of 1.6× on a use case requiring an average to 3.6 seconds to synthesize in the baseline condition.

Used together, our optimizations cut the average total time required to synthesize all of our use cases by 25× (from 38130 seconds to 1511 seconds), with a geometric mean speedup of 9.2×. One use case is accelerated 1380×: while timing out in the baseline condition, our optimizations allow us to synthesize it in just over one second.

Currently, our synthesis engine is single-threaded; we included the *virtual best* curve in Figure 10 as a demonstration of the performance approachable by a *parallelized* implementation, which would run several different synthesis engine configurations and random seeds for a single synthesis problem, taking the first result returned.

7.7 Our Experience Writing Specifications

While we have not conducted an extensive user study to determine how easily architects and kernel designers can produce specifications, we have one related user study and our own experience from which to draw conclusions. In collaboration with colleagues in Human Computer Interaction (HCI), a subset of our team developed an interactive tool that let users give hints to the synthesis engine [Hu et al. 2021]. Although the study did not ask users to produce specifications, they had to understand enough about the specifications and the architecture to make suggestions about synthesis strategies, inclusion/exclusion of instructions, and hypotheses about the
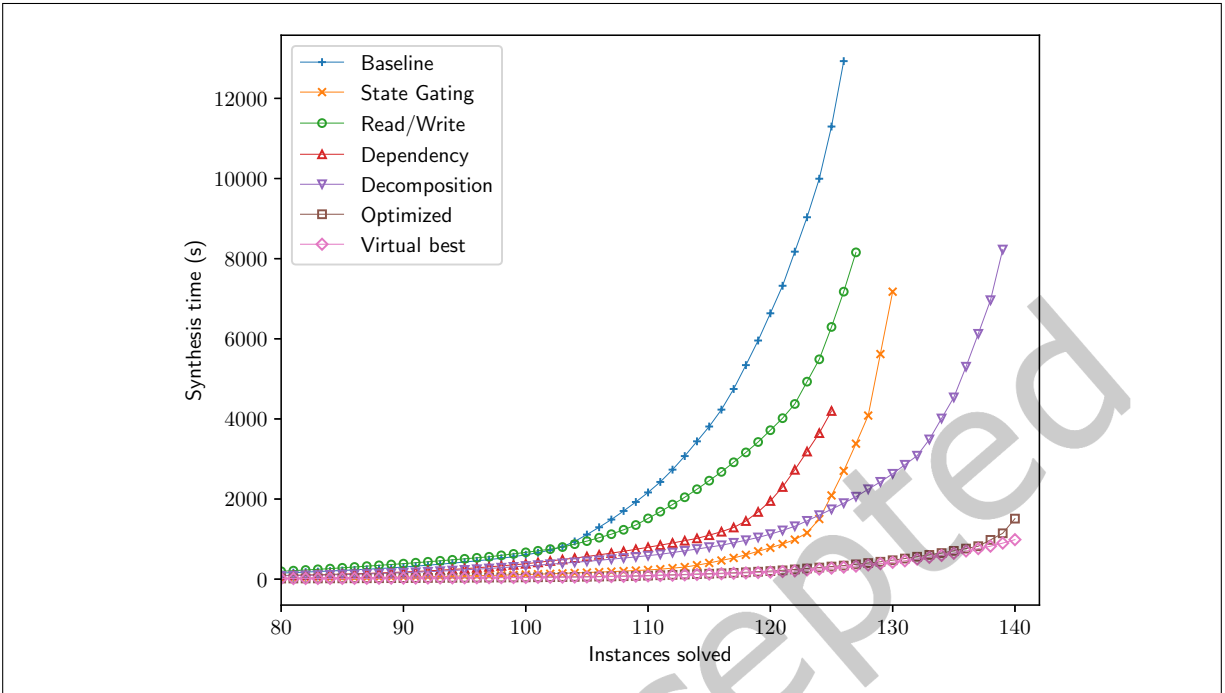
Fig. 10. Cactus plot comparing each optimization against baseline. Each plot shows for all machine-independent specifications the average runtime across each of five trials that do not time out. Machine-independent specifications where all trials timed out are omitted. The *Optimized* curve shows the performance of our system with all optimizations enabled together. The *Virtual best* curve simulates the performance of a system that takes the minimum time observed for each machine-independent specification across all trials.

structure of the code that should be produced. All users, from novice to expert were able to give hints that improved the time to synthesize kernel components.

Our own experience writing specifications is more directly relevant to the work presented here. Recall that synthesis in Aquarium is designed to decouple expertise requirements: computer architects write machine specifications and kernel developers write kernel specifications. The team responsible for writing all the specifications presented in this paper included one experienced kernel developer and two others with minimal kernel development experience. None of the team were computer architects. Nonetheless, we found that it took approximately one day to write and debug a Cassiopea file for a new architecture, even when the architecture was significantly different from others we had written (e.g., producing the Cassiopea file for MIPS after we had produced Cassiopea files for only x86-64 and ARM).

We found that writing the Alewife specifications took us about the same amount of time it takes to manually produce a port of the machine dependent parts of a system. However, once written, every additional port can use these specifications. In other words, using Aquarium is a time saver as soon as you start on a second port. Even our team member who had never written kernel code before found it reasonably straight forward to produce specifications. Further, we discovered that we could use existing ports to debug both our Alewife and Cassiopea specifications. If a Cassiopea specification is incorrect, and we try to verify the code in an existing port, the

verification fails[12]. Similarly, if we write an incorrect kernel specification, verification of the existing code also fails. Overall, we found that Aquarium let us achieve our goal of transforming the N ports for M operating systems problem into something that took time proportional to $N + M$ rather than $N * M$, as is the case for traditional porting.

8 DISCUSSION

8.1 Aquarium Ecosystem

The Aquarium code generation model is to decompose procedures into blocks and generate the blocks one at a time, by synthesis or by other techniques. These blocks must then be recombined into functions and files, connected to other parts of the system via include files, and ultimately compiled and linked using make or a similar tool. Aquarium handles this with separate tools that construct complete procedures from assortments of code blocks [Holland 2020]. We have omitted discussion of this aquarium of tools and languages from this paper, which focuses exclusively on our program synthesis tools.

8.2 Effort/Benefit Tradeoff

As noted in Section 7.3, the line counts of specification and lowering relative to output code are perhaps larger than one might like. When the ratio of specification to output code exceeds 1, it is tempting to think that just writing the code directly might be preferable. Nonetheless, several factors suggest that synthesis is still promising.

First, verifying handwritten code still requires specifications (and lowerings), so these need to be written regardless, and synthesis is always strictly less work.

Second, it is important to interpret these numbers in the correct context. Our specification-to-code ratio is reasonably competitive with other synthesis and verification approaches, especially that of other work in assembly-language synthesis and verification. We ask how much specification a user would need to write to specify assembly for a new architecture. JitSynth [Van Geffen et al. 2020], another assembly synthesis approach, does not report a spec-to-code ratio, but does at least require the user to write the state equivalence between the source and target register machines. Vale [Bond et al. 2017] reports 2014 lines of spec, 3213 lines of implementation, and 13558 lines of proof for 6943 generated lines of assembly (excluding lines of specification, implementation, and proof associated with definitions of the ARM and Intel semantics). This corresponds to a spec-to-code ratio of about 0.75, not counting lines of proof; we are unsure how much additional code would be required to extend Vale to another architecture. The SeL4 proof files in the spec/machine directories contain 2600 lines, while the assembly files in the arch directory contains 3380 lines; trimming comments and newlines, these counts are about 2000 and about 1900 lines respectively. This corresponds to a spec-to-code ratio of about 1.05, which we expect to correspond to the ratio of specification needed to extend SeL4 to a new architecture. We think it is reasonable to claim that the specification effort for a new port is at least comparable to that of similar work.

Third, the cost of producing OS specifications is amortized across multiple ports: the same specifications can be used for multiple architectures. We could dilute this ratio to an arbitrarily low number by including more architectures in our evaluation. Moreover, having machine-independent OS specifications available provides additional benefits, such as enabling verification and analyses to reduce bugs and vulnerabilities. This benefit is also portable to new architectures, which is not true of existing assembly verification approaches. Given that low-level code is often extremely difficult to debug, this is potentially a significant improvement.

Fourth, the synthesis approach reduces the level of mastery of architectural details required of the porter. A porter need only follow the machine manual and not necessarily learn it well enough to write assembly code from scratch. As an example, the authors of this paper had no prior experience writing RISC-V assembly. We found the control register access instructions complex; especially when first starting with the architecture, we

---

[12]This could also be due to a bug in the existing port, but we did not encounter any such situations.

found it convenient to let our synthesis engine figure out how to use them. The required level of knowledge of the ported OS is reduced as well. Currently, porting an OS to a specific architecture using Aquarium requires the porter to provide a lowering for each Alewife specification. Only the lowering file, which relates a particular machine description to the OS specification, requires knowledge of both the machine and the OS, and the lowering file is relatively small, offering considerable savings of time and effort. Many of the blocks in our use cases require few or no block-specific lowering definitions. For many of the rest, the lowering definitions required are straightforward and easily explained; the porter can provide them without necessarily having to fully understand the blocks that use them. However, this does not apply to all the ported code; some code, such as the swi-handler (SWI) example, still requires expert attention. It is expected that with more ports, a considerable fraction of the port code will be derivable from specifications and comparatively simple lowering definitions, especially as the rest of the Aquarium ecosystem comes on line.

Note also that in some cases the need for porters to think about particular pieces of code disappears entirely. For example, the only lowering definitions specific to the OS/161 thread-switch (TS) example are "Data-Structure" definitions that can be automatically generated. Consequently with our approach, the thread switch code no longer requires any attention from the porter: it simply happens, just as all the kernel's machine-independent code simply happens when recompiled.

There are nonetheless cases where the cost/benefit ratio is an issue, especially for blocks that demand additional functionality or expressiveness. One example of the latter is memory barrier instructions, discussed further in Section 8.5.

8.3 Machine-Independence

It is tempting to assume that Alewife specifications cannot "really" be machine-independent and that all the important parts of each specification must actually be in the lowering files. This is, perhaps surprisingly, not true. Operating systems already have a machine-independent model of the abstractions in and functions of the machine-dependent code, which underlies the interface to that code. Our Alewife specifications elucidate and formalize this model. They can also, it turns out, frequently extend it: the kernel's interface to machine-dependent code is necessarily function-level, but our specifications are block-level. Consequently any machine-independent structure in the steps that need to be taken, which is frequently substantial, can be written directly into the Alewife specifications. For example, in the thread-switch (TS) example, there are six steps, and the nature of each and the order in which they appear is necessarily the same on every machine, because at a suitable level of abstraction, they must each do the same thing.

Furthermore, even for operations that cannot be further decomposed, such as turning on interrupts, the specification for the operation is the machine-independent specification for what the kernel expects to happen: some piece of machine state, corresponding to the interrupt state, is toggled on. While the machine-level state and instructions that correspond to machine-independent functionality often vary from machine to machine, higher-level OS code can and will use this block without regard for the implementation details. We can write a machine-independent specification in terms of an abstract version of this state, then use the Alewife compiler to lower the specification so it references the correct control register field(s) for each target machine. Then the synthesis engine can synthesize correct code given the machine description.

An OS architect writing Alewife specifications must understand the abstract model of the underlying machine represented by the machine-independent-to-machine-dependent interface. In the case of Barrelfish, where the kernel is built as position-independent code to facilitate the "multikernel" architecture, this must take into account differences in the handling of position-dependent code on different machines. If porting to a new machine changes the behaviors observable from the machine-independent code, then the machine-independent specifications may need to be strengthened. This is a common problem in OS porting. Every OS embeds a similar understanding in

the structure of its machine-dependent interfaces, and altering it can require altering those interfaces and the OS's machine-independent code.

8.4 Formal ISA Descriptions

We expect machine descriptions to ultimately be supplied by architecture vendors. Architecture vendors already internally maintain extensively tested executable formalizations of their architectures; for instance, ARM internally employs the ARM Specification Language to describe processor semantics, and uses it to test both Verilog and physical hardware. Fully formalizing the semantics of an ISA is a challenging problem; despite this, recent work in formalizing ARM [Reid 2016] and various other architectures [Armstrong et al. 2019] supports the assertion that vendor-supplied ISA descriptions suffice to produce accurate formal descriptions of ISA semantics. Moreover, the RISC-V foundation has officially standardized an executable formal specification of the RISC-V ISA semantics in Sail [RISC-V 2019]. We expect that in the future, ISA semantics formalized by the architecture vendor will become increasingly available and are here to stay.

We used a custom language for machine descriptions, Cassiopea, for two reasons. First, at the time of development, formalized ISA semantics were less available than they are today. Second, our synthesis proof-of-concept requires only a subset of the ISA semantics, and as observed by Bourgeat et al. [Bourgeat et al. 2021], many formal methods projects use simplified ISA semantics. Cassiopea itself, as discussed, is not a complete ISA description language; it is one point on a spectrum of languages with varying levels of expressiveness.

ISA instruction selection. One critical issue is the level of completeness of the ISA description. A more complete description could enable synthesis of more OS components; on the other hand, including more instructions negatively impacts the synthesis performance and the ease of writing machine descriptions. Fortunately for the latter, expressiveness does not require the ISA description to include most of the available instructions. For example, the x86 architecture contains many instructions that are not necessary for full expressiveness, such as the specialized vector operations in the 64-bit version. Excluding these reduced the work required for us to produce ISA descriptions, although we expect that manufacturer-supplied ISA descriptions would be relatively more complete.

Using only a sufficiently expressive ISA description additionally helps with synthesis performance. This gives a reasonably sized search space to allow the synthesis engine to create an initial runnable program. Once we have a runnable program, we can apply superoptimization [Schkufza et al. 2013] or standard compiler techniques [Debray et al. 2000] to obtain a more efficient program that uses a larger machine model.

Our expressiveness decisions were driven by our use case requirements. Consequently, as discussed in Section 7.2, we constructed our machine models to include at least the following general-purpose components:

- Basic arithmetic and logical instructions on register-sized values: addition, subtraction, signed and unsigned comparisons, bitwise operations, shifts.
- Conditional and unconditional branches. When architectures distinguish between short and long jumps, short jumps suffice for intra-block branching.
- Loads and stores for word-sized and byte-sized values, ensuring that we can express any manipulation of an included memory region.
- A mechanism to load the address of an assembler label.

We also include the full set of general-purpose registers for each machine and any special-purpose registers used by ordinary code, such as the flags word.

Beyond these, we include a selection of pertinent control registers and the instructions that manipulate them, as this is a common OS operation. In general, each block we synthesize needs access to conceptually similar pieces of machine state in each architecture.

8.5   Limitations

What we do not synthesize. To strike the balance between expressiveness and complexity of the languages, there are important pieces of OS code that we do not synthesize. We do not synthesize call and return instructions (nor do we synthesize system call instructions). Procedure calls are different from ordinary assembly instructions. There are two ways one could imagine synthesis interacting with procedure calls. The first way is to let the specification explicitly indicate what procedure to call. This is the approach we take; we generate the call and return instructions with other Aquarium tools [Holland 2020]. The second way is to make the specification more expressive about procedure calls and use synthesis to select a particular call from a set of choices. This is a higher-level synthesis task, which is orthogonal to our goal of synthesizing assembly language.

Furthermore, there are small portions of OS code that are not merely machine-dependent, but are machine-specific, which cannot be expressed in a machine-independent way. Well-known examples include the code to set up x86 segment tables and SPARC register window trap handling. A less well-known example that affects our use cases is that ARM exception handlers must manipulate exception-related processor modes that do not exist on other architectures.

Scalability of assembly synthesis. Furthermore, scalability poses a serious challenge. Synthesis of assembly is particularly hard since the characteristics of assembly languages cause a combinatorial explosion in the number of possible instructions that must be considered [Hu et al. 2019]. This detrimentally affects the performance of syntax-guided synthesis.

This difficulty is a general feature of assembly languages. Even though our synthesis is parameterized on a Cassiopea machine description, an approach specialized for a single machine or a low-level intermediate language must still address the fundamental problem of search space size. Although our optimizations significantly reduce the search space, it is still quite easy to write specifications for which synthesis always times out.

Currently, using pure inductive synthesis, we can generate up to five (occasionally only four) instructions in reasonable time; with our rule-based decomposition synthesis technique, we can synthesize programs up to twelve instructions for certain well-structured specifications. Therefore, we expect to be able to handle most small pieces of assembly code; besides the assembly blocks demonstrated in our use cases, Alewife and Cassiopea can probably also handle frame push and pop operations, stack handling, and simple bit manipulations without further extension. Although this is sufficient for many machine-dependent OS components, some require significantly longer implementations. For example, the MIPS general exception handler in NetBSD 9 is about 100 instructions. For these our methodology of decomposition into steps becomes critical.

Ultimately, while blocks of four or five instructions are small, they are still useful, since other tools in our ecosystem allow composing them into larger chunks. For example, as discussed in Sections 7.1 and 8.3, the thread-switch (TS) example is composed of six steps. Two of these we compile; the other four we synthesize. Those four are all one instruction each on most or all architectures. Yet combined, these six blocks are a complete kernel-level thread switch, similar to that found in any operating system kernel. The setjmp (SJ) and longjmp (LJ) examples are similarly decomposed; combining the pieces gives the complete implementations. Other examples are a subject of ongoing research; this includes the crt0 (CRT) use cases.

Concurrency and time. Cassiopea and Alewife do not model concurrency or time. Reasoning about time typically involves temporal logics that are not SMT solvable. This is likely a fatal obstacle for working with real-time operating systems (especially hard real-time), but for conventional systems, machine-dependent code handles time only when interacting with timer hardware. We have not addressed this issue in current work; in future work we plan to investigate whether the temporal aspects of timer control can be adequately abstracted away such that the low-level specifications (and thus code synthesis) can be expressed in Alewife.

We chose not to burden Cassiopea with support for reasoning about concurrent executions. Our use cases do not require concurrency support. This is representative of the majority of the machine-dependent part of an OS: most of it executes below the level where concurrent things happen. (For example, the thread switch code in our use cases executes on a single CPU, on CPU-private data, and with interrupts already disabled by higher-level machine-independent code; it itself need not reason about other CPUs, other threads running time-sliced on the same CPU, or about being interrupted.)

However, this means we cannot model instructions related to concurrent executions. The machine-independent code in a typical OS is written in terms of a small set of concurrency primitives. Producing these is part of generating a new port, but one we decided not to tackle in this work. Currently, they are provided by the porter as prewritten assembly blocks that can be composed with other blocks via other Aquarium tools. For a typical OS this means implementations of spinlocks, a small set of atomic operations, and a small set of memory barriers. Note that all of these are also small blocks of assembly code and likely amenable to synthesis using other tools, though the specifications are nontrivial [Bornholt and Torlak 2017] and (particularly for memory barriers, which are always either zero or one instruction) likely to significantly outweigh the output code.

Performance. Finally, we do not reason about the performance of synthesized code. Although the synthesis engine chooses the shortest sequence of instructions that meets a specification, this may not correlate to performance. We assume that optimization can be performed after synthesis and block composition. Furthermore, low-level OS code typically does not involve expensive computation. We discuss the related problem of synthesizing the fastest possible assembly program, or superoptimization [Massalin 1987], in Section 9.

8.6 Other Approaches Considered

We explored several different approaches to improve scalability of synthesis. We briefly summarize our explorations.

SMT encodings. Before building our synthesis engine, we experimented with an assembly synthesizer written in Rosette [Torlak and Bodik 2014] and explored language designs and theory combinations. We found that combinations of integer and bitvector theories led to poor performance. We also experimented with using the theory of arrays for updates to registers and memory, but found it markedly slower and abandoned it in favor of a direct encoding.

Accumulation algorithm. Since our implementation of syntax-guided synthesis proceeds in stages, we experimented with an *accumulation* algorithm. In general, in stage n, we attempt to synthesize an n-operation program. Starting at stage 0 (which succeeds only if the specification is vacuous), our synthesis engine proceeds to stage $n + 1$ when synthesis at stage n fails, iterating until synthesis succeeds or times out. The accumulation method attempts to reuse work done at stage n in later stages. Specifically, in a later stage $n + k$ we use the last guessed program from stage n as the initial n operations, and attempt to synthesize only the last k operations. Intuitively, the n-operation program might be "almost correct," since it avoids many generated counterexamples. Accumulation accelerates synthesis when the final candidate guessed during synthesis for length n is a prefix of a correct program. Trying to synthesize k operations is typically much faster than synthesizing $n + k$ operations, so accumulation could be worthwhile and relatively cheap. If none of the accumulated program prefixes can be extended to a correct program, stage $n + k$ proceeds to attempt to synthesize all $n + k$ operations. When we evaluated the effectiveness of accumulation we found that it can help, but infrequently. When it does helps, it can result in significant improvements, making syntax-guided synthesis up to 7× faster on a couple of our use cases. However, in the worst case, accumulation caused synthesis to take twice as long.

Instruction dependency graph. We experimented with adding constraints to require specific dependencies between instructions, for example, requiring the locations read by the third instruction to intersect with the locations written by the first instruction. In essence, this enforces a given dependence graph over instructions in a block. Although this can significantly reduce synthesis times, we could not find suitable heuristics for indentifying appropriate dependencies, since implementations of the same block on different machines often requires different dependency graphs.

Multiple counterexamples per loop. We experimented with collecting multiple counterexamples per CEGIS loop iteration, instead of the standard single counterexample per loop. We analyzed the effect on the total number of CEGIS iterations and the synthesis time. Although additional counterexamples consistently reduced the number of iterations, we found it difficult to determine an effect on overall synthesis time. We also tried retaining counterexamples across stages as part of our accumulation approach. This also led to fewer iterations but sometimes longer synthesis times. Ultimately, we found that additional counterexamples increased both the variance and average of the total synthesis time.

Candidate buckets. Due to the high time cost of generating program candidates with multiple counterexamples, we also experimented with grouping candidates across CEGIS iterations into several candidate *buckets* and generating counterexamples that worked for all candidates in a bucket. This method led to less synthesis time for one CEGIS iteration but more iterations in total for some use cases. The total synthesis time tended to increase overall.

Interactive assembly synthesis. Assuage [Hu et al. 2021], based on our synthesis engine, presents a parallel interactive assembly synthesizer that engages the user as an active collaborator. Users, who are familiar with the high-level concepts of assembly languages, provide multiple types of guidance during synthesis, while the synthesizer provides users with different representations of progress feedback. Assuage enables synthesis to scale beyond current limits.

8.7 Correctness versus Security

It is reasonable to ask whether the Aquarium approach with program synthesis improves system security.

Guaranteed correctness can improve security. The machine-dependent parts of an operating system rarely implement policy; they translate policy decisions into hardware implementations. In this sense, a correct implementation does enforce certain security properties, and a verified implementation does provide some extra security assurance. However, in general one must also verify that the machine-independent code invokes the machine-dependent operations at the correct points, which is a much larger issue we do not attempt to address. Also note that many of the possible errors in low-level code do not have subtle consequences: the system hangs or crashes, or violently corrupts memory. Identifying the *source* of such problems can be extremely expensive, and verification can save much debugging time; however, identifying their *existence* is typically less challenging, so they are relatively unlikely to survive long enough to become vulnerabilities in the field.

For example, machine dependent code must properly update page tables. Failure to perform such operations correctly, such as setting the wrong page permissions or using the wrong address-space ID, can create security vulnerabilities. However, verifying the page table updates will not protect against the machine-independent virtual memory system failing to revoke access permissions at the proper times. Correspondingly, setting the wrong page permissions will in general result in denying accesses that should succeed, which will cause programs (or the system) to crash, which in turn is relatively unlikely to be missed during development.

Furthermore, incorporating security related restrictions into synthesis is an interesting direction for future investigation. Integrating them with the specification or even adding as extra constraints can, in general, allow rejecting vulnerable assembly programs during synthesis, which would provide further security guarantees.

Synthesis is not a security panacea. There is a large class of security bugs that arise when microarchitectural implementation details leak through to the instruction level. Spectre and Meltdown [Hill et al. 2019], and Rowhammer [Mutlu and Kim 2020] are, perhaps, the most well-known of such attacks. Synthesis and verification do nothing to address such issues. Cassiopea specifications describe the instruction set architecture; nothing in the Aquarium ecosystem is aware of microarchitectural implementation details, so there is no way to reason about any unwanted effects that might arise. However, given rules for mitigating such vulnerabilities, we expect to be able to synthesize code that adheres to those rules. Further, as always, verification against a specification is only as good as the correctness of the specification.

8.8　Future Work

This work demonstrates the feasibility of assembly program synthesis for machine-dependent OS components. However, there is plenty of opportunity for improvement and many additional directions to explore. In addition to further improving the scalability of assembly synthesis, we see several opportunities to further reduce the effort necessary to port OSes with Aquarium.

Ports have many similarities; it is standard practice to begin a port by copying an existing implementation for a similar machine [Li et al. 2004]. Since an existing OS implementation often exists, it may be possible to synthesize OS specifications of machine-dependent components from existing implementations or to accelerate synthesis by reusing existing implementations. A key challenge is how to appropriately abstract away from the specifics of the machine. We anticipate that accomplishing such synthesis would need a human in the loop. Similarly, we anticipate synthesizing parts of a lowering file from a given Alewife specification and a Cassiopea machine description iteratively with a human in the loop. However, this interaction sacrifices automation and requires further maintenance of the specifications and descriptions.

While developing machine descriptions, we found that a useful way to test the correctness of the descriptions was to create machine-dependent specifications that should be unrealizable and use synthesis to try to find an assembly program satisfying the specification. For example, ARM has certain flag register fields that must not be changed by arithmetic instructions. Given a specification that forced arithmetic operations and required illegal field changes, successful synthesis implies an error in the formalization of the machine. This approach helped us find errors in our Cassiopea descriptions. More generally, the use of program synthesis to find errors in semantics is worthy of further exploration.

There is a truism about machine descriptions, in the context of compilers, to the effect that they always eventually become Turing-complete as they get extended to handle unexpected new cases. It would be interesting to develop a scheme for evaluating how declarative (as opposed to "code-like") a description language is. In general, more declarative descriptions are easier to work with but less expressive and are prone to needing extensions after their initial development, whereas more code-like descriptions are more expensive to use but more expressive.

9　RELATED WORK

Our work lies at the intersection of the programming languages and operating systems.

Program Synthesis. Modern program synthesis began in 2006 with the introduction of SKETCH [Solar-Lezama et al. 2006], in which a programmer provides a partial program and specifications for "holes" in the program,

and SKETCH fills in the holes. Following this work, counter-example guided inductive synthesis (CEGIS) [Solar-Lezama 2008] generalizes sketching for infinite state programs. However, for infinite state programs, the verifier is typically an unsound, bounded model checker. Although we use CEGIS, we restrict ourselves to finite state programs.

Another category of program synthesis, programming by example (PBE), takes input-output examples to demonstrate desired program behaviors [Gulwani et al. 2017; Singh and Gulwani 2015]. In the assembly programming context, examples that include concrete machine state are much more challenging to provide than are formal specifications.

While the DSLs used in early CEGIS systems, e.g., [Solar-Lezama et al. 2005; Solar-Lezama et al. 2006], handle many of the same concepts as does Cassiopea (e.g., precise bit manipulation), the problem of synthesizing assembly code is significantly harder than that of synthesizing high-level languages. OS machine code manipulates only untyped memory and worse, all state is global. While programs in high-level languages contain these features, they are typically used parsimoniously. In contrast, untyped memory and global state are inherent to, and unavoidable in, assembly code. Consequently, assembly code synthesis has enormous search spaces that are exponential in program length and combinatorial in the number of instructions (at least the dozens), registers (also dozens), memory locations (possibly the hundreds or thousands if one considers kernel structures, such as page tables), and immediate values.

Superoptimization and Assembly Synthesis. To the best of our knowledge, the first encoding of assembly language into SMT was the Stoke algorithm for stochastic superoptimization [Schkufza et al. 2013]. The superoptimization problem for machine code [Bansal and Aiken 2006; Massalin 1987; Phothilimthana et al. 2016] is: given a piece of machine code and an execution context, find the lowest cost semantically equivalent piece of code. Stoke formulates correctness and performance improvement into a cost function, then uses a Markov Chain Monte Carlo sampler to find an implementation that, with high probability, outperforms the original (already optimized by LLVM [Lattner and Adve 2004]). Superoptimizers work on straight-line, intraloop, intraprocedural code.

Superoptimization avoids some challenges presented by the general synthesis problem: first, a superoptimizer can fail to optimize a snippet, since the original snippet can be used instead. In contrast, if Aquarium fails to synthesize a snippet, it is a showstopper. Second, superoptimization is bootstrapped by a correct snippet of code to be optimized; in contrast, we have only a declarative, machine independent specification and must produce a working implementation. This rules out, for example, subdividing the problem by computing intermediate machine states to use as subgoals.

Srinivasan and Reps also consider the problem of synthesizing assembly programs from semantic specifications [Srinivasan and Reps 2015; Srinivasan et al. 2016, 2017], developing several synthesis optimizations similar to ours. Like us, they use a divide-and-conquer scheme that attempts to find independent sub-specifications that can be solved separately. Whereas their algorithm divides specifications greedily, our rule-based decomposition (Section 6.3.4) divides lazily, when we find that a specification cannot be solved by a one-instruction program. We use an ordering heuristic to select the split that should be considered next. Their conquer phase exhaustively enumerates instruction opcode and register arguments, using CEGIS to solve for immediate arguments. Using enumeration means that their conquer phase does not scale beyond two- or three-instruction programs, whereas our fully solver-based CEGIS can synthesize certain sequences up to nine instructions long. Their footprint-based pruning approach can be compared to our read/write constraints. Their bits-lost pruner is similar to our dependency constraints (Section 6.3.3) in that it prunes program prefixes that discard necessary dependencies, but it employs finer-grained input tracking – at the level of bits rather than locations – and coarser availability tracking – at the level of whole program state rather than locations. Since their source code is not available, we cannot run a direct performance comparison.

Van Geffen et al. [Van Geffen et al. 2020] synthesize components of JIT compilers that produce sequences of assembly instructions which correspond to instructions in kernel-level DSLs such as eBPF. They use a metasketch-based approach that optimizes synthesis using syntactic constraints on the output assembly programs. Their pre-load sketches load immediate values used by the rest of the program; our rule-based decomposition can do this in some cases, in particular when the program requires a value found at a label or in memory. Their read-write sketches constrain which instructions can be used depending on whether they access registers, memory, or both. This optimization can be compared to our read/write constraints and dependency constraints, which use assertions to restrict the search space according to a sound summary of instruction semantics. Our approach is more precise in that it can constrain instructions based on accesses to specific registers and memory locations as opposed to accesses to any register or any memory location.

Optimization in Synthesis. Our dependency analysis represents one way to use abstraction to constrain search. Other works have found this general approach effective. In some sense CEGIS is such an approach; the abstraction it considers is the program behavior on a specific set of inputs. The Storyboard Programming [Singh and Solar-Lezama 2011] framework starts from sketches and synthesizes imperative programs that manipulate data structures using specifications that constrain the desired abstract behavior.

Cosette [Chu et al. 2017; Wang et al. 2018] searches for input tables that distinguish between two SQL queries by causing them to produce different outputs. To accelerate search, Cosette constrains input tables to contain only those tuples that can possibly affect the query output. Cosette finds these constraints by using the abstract semantics of SQL to obtain *provenance predicates* representing facts about input tuples that could pass through a query. Provenance predicates resemble our dependency analysis inasmuch as they are constructed by following dependencies of output on input; Cosette uses the abstract semantics of SQL to make provenance predicate discovery tractable. Our approach differs in that we reduce a *program* search space by introducing constraints on program abstract behavior.

Much like our Cassiopea DSL is an executable encoding of instruction semantics, Vale [Bond et al. 2017] encodes the semantics of each assembly instruction as an snippet of Dafny [Leino 2010]. While Vale enables verification of assembly snippets using Dafny's pre- and postconditions, we use Alewife, a purpose-built DSL for specifications. Vale does not enable synthesis. An interesting avenue of future work would be to extend our synthesis work to the domain of cryptographic functions with Vale's use cases.

Synthesis for Operating Systems. In the late 1980's and early 1990's, several groups experimented with OS customization [Bershad et al. 1995; Endo et al. 1996; Engler et al. 1995], performance optimization [Liedtke 1995], and specializing code to improve performance [Mosberger and Peterson 1996; Pu et al. 1988]. The specialization approach produced two interesting OS synthesis projects: The Synthesis Kernel [Pu et al. 1988] and Scout [Mosberger and Peterson 1996], both of which were based on the observation that OSes must handle all cases, although the vast majority of the time they are executing specific cases, where many of the decisions throughout a code path are determined a priori by system call parameter values. Creating versions of these code paths specialized to a given parameter improves performance. The Synthesis Kernel was custom-built to support this kind of synthesis. Scout took a similar approach, synthesizing fast-path implementations for network functionality.

The Synthesis Kernel and Scout both differ from Aquarium in two ways. First, they create customized code paths from existing general components, which differs from Aquarium's synthesis using high level, machine-independent specifications. Second, they perform runtime synthesis, relying on knowing specific parameter values. These approaches are orthogonal to ours; it is possible to combine such runtime approaches with modern program synthesis techniques.

More recently, Termite [Ryzhyk et al. 2009] and Termite-2 [Ryzhyk et al. 2014] synthesize device drivers. These use a different approach to synthesis (reactive synthesis rather than CEGIS) and synthesize to a C subset rather than assembly. More importantly, though, they concentrate on figuring out *what* to do, that is, what sequence of

state changes to make, whereas our work concentrates primarily on figuring out *how* to make state changes, that is, the exact assembly instructions. One could imagine a system where something akin to Cassiopea figured out how to issue the device state changes directed by Termite-2, or where something akin to Termite-2 was used to figure out the specifications for the assembly code blocks Cassiopea generates. Thus this work is complementary to ours.

10 CONCLUSIONS

We present the Aquarium approach to program synthesis, part of a collection of languages and tools to automatically construct machine-dependent OS code. This paper focuses on machine-dependent assembly code synthesis. We developed two domain specific languages, Alewife and Cassiopea, to specify OS functionality and describe machine instruction set architectures, respectively. The Alewife compiler compiles machine-independent specifications into machine-dependent specifications, while the Cassiopea synthesis engine synthesizes the implementations in assembly language for specific machines.

With Alewife, we have made a first approach to machine-independent specification of machine-dependent components. We can readily specify some parts of the OS (e.g., interrupt and cache handling) and not so readily specify others (e.g., boot code, drivers, concurrency primitives). We also demonstrated a collection of optimizations for improving synthesis performance. These do not fully address the scalability challenges but do improve program synthesis for an unusual domain.

We validated our tools with 35 synthesis use cases taken from two preexisting operating systems, deployed on four real machine architectures. For most of our use cases, synthesis finishes in a reasonable time, typically a few minutes, although some use cases are beyond our ability to synthesize.

ACKNOWLEDGMENTS

REFERENCES

Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *Proceedings of the 13th Formal Methods in Computer-Aided Design Conference (FMCAD '13)*. 1–8. https://doi.org/10.1109/FMCAD.2013.6679385

Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. 2019. ISA semantics for ARMv8-a, RISC-V, and CHERI-MIPS. In *Proceedings of the 46th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '19)*.

Sorav Bansal and Alex Aiken. 2006. Automatic Generation of Peephole Superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '06)* (San Jose, California, USA) *(ASPLOS)*. 394–403. https://doi.org/10.1145/1168857.1168906

Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. ACM, New York, NY, USA, 29–44. https://doi.org/10.1145/1629575.1629579

Emery D. Berger and Benjamin G. Zorn. 2006. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada) *(PLDI '06)*. ACM, New York, NY, USA, 158–168. https://doi.org/10.1145/1133981.1134000

Brian N. Bershad, Craig Chambers, Susan Eggers, Chris Maeda, Dylan McNamee, Przemysław Pardyak, Stefan Savage, and Emin Gün Sirer. 1995. SPIN—an Extensible Microkernel for Application-Specific Operating System Services. *SIGOPS Oper. Syst. Rev.* 29, 1 (Jan. 1995), 74–77. https://doi.org/10.1145/202453.202472

Rastislav Bodik, Satish Chandra, Joel Galenson, Doug Kimelman, Nicholas Tung, Shaon Barman, and Casey Rodarmor. 2010. Programming with Angelic Nondeterminism. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Madrid, Spain) *(POPL '10)*. ACM, New York, NY, USA, 339–352. https://doi.org/10.1145/1706299.1706339

Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. 2017. Vale: Verifying High-Performance Cryptographic Assembly Code. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security '17)*. USENIX Association, Vancouver, BC, 917–934. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/bond

James Bornholt and Emina Torlak. 2017. Synthesizing Memory Models from Framework Sketches and Litmus Tests. In *Proceedings of the 2017 Programming Languages Design and Implementation Conference (PLDI '17)*.

James Bornholt and Emina Torlak. 2018. Finding code that explodes under symbolic evaluation. In *Proceedings of the 33rd Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '18)*.

Thomas Bourgeat, Ian Clester, Andres Erbsen, Samuel Gruetter, Andrew Wright, and Adam Chlipala. 2021. A Multipurpose Formal RISC-V Specification. *arXiv preprint arXiv:2104.00762* (2021).

James Brotherston, Nikos Gorogiannis, and Max Kanovich. 2017. Biabduction (and related problems) in array separation logic. In *Proceedings of the International Conference on Automated Deduction.* Springer, 472–490.

Robert Brummayer and Armin Biere. 2009. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,* (York, UK) *(TACAS '09)*. Springer-Verlag, Berlin, Heidelberg, 174–177. https://doi.org/10.1007/978-3-642-00768-2_16

Weiteng Chen, Xiaochen Zou, Guoren Li, and Zhiyun Qian. 2020. KOOBE: Towards Facilitating Exploit Generation of Kernel Out-Of-Bounds Write Vulnerabilities. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security '20)*. USENIX Association, 1093–1110. https://www.usenix.org/conference/usenixsecurity20/presentation/chen-weiteng

Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. 2017. Cosette: An Automated Prover for SQL.. In *CIDR*.

Helen Custer. 1992. *Inside windows NT.* Microcomputer Applications.

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340. http://dl.acm.org/citation.cfm?id=1792734.1792766

Saumya K. Debray, William Evans, Robert Muth, and Bjorn De Sutter. 2000. Compiler Techniques for Code Compaction. *ACM Trans. Program. Lang. Syst.* 22, 2 (March 2000), 378–415. https://doi.org/10.1145/349214.349233

Bruno Dutertre. 2014. Yices 2.2. In *Proceedings of the Computer-Aided Verification Conference (CAV '14) (Lecture Notes in Computer Science, Vol. 8559)*, Armin Biere and Roderick Bloem (Eds.). Springer, 737–744.

Yasuhiro Endo, Zheng Wang, J. Bradley Chen, and Margo Seltzer. 1996. Using Latency to Evaluate Interactive System Performance. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation* (Seattle, Washington, USA) *(OSDI '96)*. ACM, New York, NY, USA, 185–199. https://doi.org/10.1145/238721.238775

D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. 1995. Exokernel: An Operating System Architecture for Application-level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (Copper Mountain, Colorado, USA) *(SOSP '95)*. ACM, New York, NY, USA, 251–266. https://doi.org/10.1145/224056.224076

John K. Feser, Swarat Chaudhuri, and Işıl Dillig. 2015. Synthesizing Data Structure Transformations from Input-output Examples. *SIGPLAN Not.* 50, 6 (June 2015), 229–239. https://doi.org/10.1145/2813885.2737977

Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. 2020. Temporal System Call Specialization for Attack Surface Reduction. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security '20)*. USENIX Association, 1749–1766. https://www.usenix.org/conference/usenixsecurity20/presentation/ghavamnia

Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. ACM, New York, NY, USA, 317–330. https://doi.org/10.1145/1926385.1926423

Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program Synthesis. *Foundations and Trends® in Programming Languages* 4, 1-2 (2017), 1–119. https://doi.org/10.1561/2500000010

John L. Hennessy and David A. Patterson. 2019. A New Golden Age for Computer Architecture. *Commun. ACM* 62, 2 (Jan. 2019), 48–60. https://doi.org/10.1145/3282307

Mark D. Hill, Jon Masters, Parthasarathy Ranganathan, Paul Turner, and John L. Hennessy. 2019. On the Spectre and Meltdown Processor Security Vulnerabilities. *IEEE Micro* 39, 2 (2019), 9–19. https://doi.org/10.1109/MM.2019.2897677

David A. Holland. 2020. *Toward Automatic Operating System Ports via Code Generation and Synthesis.* Ph. D. Dissertation. Cambridge, MA, USA. Advisor(s) Margo I. Seltzer and Stephen Chong.

David A. Holland, Ada T. Lim, and Margo I. Seltzer. 2002. A New Instructional Operating System. In *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education* (Cincinnati, Kentucky) *(SIGCSE '02)*. ACM, New York, NY, USA, 111–115. https://doi.org/10.1145/563340.563383

Jingmei Hu, Eric Lu, David A. Holland, Ming Kawaguchi, Stephen Chong, and Margo I. Seltzer. 2019. Trials and Tribulations in Synthesizing Operating Systems. In *Proceedings of the 10th Workshop on Programming Languages and Operating Systems* (Huntsville, ON, Canada) *(PLOS'19)*. Association for Computing Machinery, New York, NY, USA, 67–73. https://doi.org/10.1145/3365137.3365401

Jingmei Hu, Priyan Vaithilingam, Stephen Chong, Margo Seltzer, and Elena L. Glassman. 2021. Assuage: Assembly Synthesis Using A Guided Exploration. In *The 34th Annual ACM Symposium on User Interface Software and Technology* (Virtual Event, USA) *(UIST '21)*. Association for Computing Machinery, New York, NY, USA, 134–148. https://doi.org/10.1145/3472749.3474740

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*. 207–220.

Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75–. http://dl.acm.org/citation.cfm?id=977395.977673

K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*, Edmund M. Clarke and Andrei Voronkov (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 348–370.

Xavier Leroy. 2009. A formally verified compiler back-end. *Journal of Automated Reasoning* 43, 4 (2009), 363.

Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. 2004. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. In *Proceedings of the 6th Conference on Operating Systems Design & Implementation - Volume 6* (San Francisco, CA) *(OSDI'04)*. USENIX Association, Berkeley, CA, USA, 20–20. http://dl.acm.org/citation.cfm?id=1251254.1251274

Chuck Liang and Dale Miller. 2009. Focusing and polarization in linear, intuitionistic, and classical logics. *Theoretical Computer Science* 410, 46 (2009), 4747–4768.

J. Liedtke. 1995. On Micro-kernel Construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (Copper Mountain, Colorado, USA) *(SOSP '95)*. ACM, New York, NY, USA, 237–250. https://doi.org/10.1145/224056.224075

Robert Love. 2010. *Linux Kernel Develoment, Third Edition.* Addison-Wesley Professional.

Henry Massalin. 1987. Superoptimizer: A Look at the Smallest Program. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems (ASPLOS '87)* (Palo Alto, California, USA) *(ASPLOS II)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 122–126. https://doi.org/10.1145/36206.36194

MarshalL Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. 1996. *The Design and Implementation of the 4.4 BSD Operating Systems.* Addison-Wesley.

Greg Morrisett, David Walker, Karl Crary, and Neal Glew. 1999. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 21, 3 (1999), 527–568.

David Mosberger and Larry L. Peterson. 1996. Making Paths Explicit in the Scout Operating System. In *Proceedings of the Operating Systems Design and Implementation (OSDI '96)*. USENIX Assoc., Seattle, WA, 153–167.

Onur Mutlu and Jeremie S. Kim. 2020. RowHammer: A Retrospective. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 8 (2020), 1555–1571. https://doi.org/10.1109/TCAD.2019.2915318

Ram Narayan. 2007. Linux assemblers: A comparison of GAS and NASM. *Tutorial: http://www. ibm. com/developerworks/library/l-gas-nasm. html* (2007).

Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. 2019. Scaling symbolic evaluation for automated verification of systems code with Serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 225–242.

Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. 2016. GreenThumb: Superoptimizer Construction Framework. In *Proceedings of the 25th International Conference on Compiler Construction* (Barcelona, Spain) *(CC '16)*. ACM, New York, NY, USA, 261–262. https://doi.org/10.1145/2892208.2892233

Nadia Polikarpova and Ilya Sergey. 2019. Structuring the synthesis of heap-manipulating programs. In *Proceedings of the 46th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '19)*.

Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: A Framework for Inductive Program Synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '15)*. 107–126.

Calton Pu, Henry Massalin, and John Ioannidis. 1988. The Synthesis Kernel. *Computing Systems* 1 (1988), 11–32.

Richard Rashid, Avadis Tevanian, Michael Young, David Golub, Robert Baron, David Black, William Bolosky, and Jonathan Chew. 1987. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems (ASPLOS '87)* (Palo Alto, California, USA) *(ASPLOS II)*. IEEE Computer Society Press, Washington, DC, USA, 31–39. https://doi.org/10.1145/36206.36181

Alastair Reid. 2016. Trustworthy specifications of ARM® v8-A and v8-M system level architecture. In *Proceedings of the 16th Formal Methods in Computer-Aided Design Conference (FMCAD '16)*. IEEE, 161–168.

RISC-V. 2019. ISA Formal Spec Public Review. https://github.com/riscvarchive/ISA_Formal_Spec_Public_Review.

Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Etienne Le Sueur, and Gernot Heiser. 2009. Automatic Device Driver Synthesis with Termite. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. ACM, New York, NY, USA, 73–86. https://doi.org/10.1145/1629575.1629583

Leonid Ryzhyk, Adam Walker, John Keys, Alexander Legg, Arun Raghunath, Michael Stumm, and Mona Vij. 2014. User-guided Device Driver Synthesis. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, Berkeley, CA, USA, 661–676. http://dl.acm.org/citation.cfm?id=2685048.2685101

Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic Superoptimization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)* (Houston, Texas, USA) *(ASPLOS '13)*. ACM, New York, NY, USA, 305–316. https://doi.org/10.1145/2451116.2451150

Shweta Shinde, Shengyi Wang, Pinghai Yuan, Aquinas Hobor, Abhik Roychoudhury, and Prateek Saxena. 2020. BesFS: A POSIX Filesystem for Enclaves with a Mechanized Safety Proof. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security '20)*. USENIX Association, 523–540. https://www.usenix.org/conference/usenixsecurity20/presentation/shinde

Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. (2016).

Rishabh Singh and Sumit Gulwani. 2015. Predicting a Correct Program in Programming by Example. In *Computer Aided Verification*, Daniel Kroening and Corina S. Păsăreanu (Eds.). Springer International Publishing, Cham, 398–414.

Rishabh Singh and Armando Solar-Lezama. 2011. Synthesizing data structure manipulations from storyboards. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. 289–299.

Armando Solar-Lezama. 2008. *Program Synthesis by Sketching*. Ph. D. Dissertation. Berkeley, CA, USA. Advisor(s) Bodik, Rastislav. AAI3353225.

Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. 2008. Sketching Concurrent Data Structures. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. ACM, New York, NY, USA, 136–148. https://doi.org/10.1145/1375581.1375599

Armando Solar-Lezama, Rodric M. Rabbah, Rastislav Bodík, and Kemal Ebcioglu. 2005. Programming by sketching for bit-streaming programs. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI '05)* (Chicago, IL, USA, June 12-15, 2005). 281–294. https://doi.org/10.1145/1065010.1065045

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '06)* (San Jose, California, USA) *(ASPLOS XII)*. ACM, New York, NY, USA, 404–415. https://doi.org/10.1145/1168857.1168907

Dokyung Song, Felicitas Hetzelt, Jonghwan Kim, Brent ByungHoon Kang, Jean-Pierre Seifert, and Michael Franz. 2020. Agamotto: Accelerating Kernel Driver Fuzzing with Lightweight Virtual Machine Checkpoints. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security '20)*. USENIX Association, 2541–2557. https://www.usenix.org/conference/usenixsecurity20/presentation/song

Michale J. Spier, Thomas N. Hastings, and David N. Cutler. 1973. An Experimental Implementation of the Kernel/Domain Architecture. In *Proceedings of the Fourth ACM Symposium on Operating System Principles (SOSP '73)*. Association for Computing Machinery, New York, NY, USA, 8–21. https://doi.org/10.1145/800009.808043

Venkatesh Srinivasan and Thomas Reps. 2015. Synthesis of machine code from semantics. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. 596–607.

Venkatesh Srinivasan, Tushar Sharma, and Thomas Reps. 2016. Speeding up machine-code synthesis. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '16)*. 165–180.

Venkatesh Srinivasan, Ara Vartanian, and Thomas Reps. 2017. Model-assisted machine-code synthesis. In *Proceedings of the 32nd Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '17)*. ACM New York, NY, USA, 1–26.

Emina Torlak and Rastislav Bodik. 2014. A lightweight symbolic virtual machine for solver-aided host languages. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 530–541.

Jacob Van Geffen, Luke Nelson, Işıl Dillig, Xi Wang, and Emina Torlak. 2020. Synthesizing JIT Compilers for In-Kernel DSLs. In *Proceedings of the Computer Aided Verification Conference (CAV '20)*. Springer International Publishing, Cham, 564–586.

Chenglong Wang, Alvin Cheung, and Rastislav Bodík. 2018. Speeding up symbolic reasoning for relational queries. In *Proceedings of the 33rd Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '18)*. ACM New York, NY, USA, 1–25.

A   CASSIOPEA AND ALEWIFE LANGUAGE DEFINITION

A.1   Notation

In the abstract syntax, type judgments, and semantics judgments we use italics for metavariables (e.g., v) and for words corresponding to types in the abstract syntax (e.g., *declaration*). We use typewriter font for words that correspond to language keywords. The notation $\overline{\alpha_i}$ means "a sequence of one or more α, each to be referred to elsewhere as α_i". If there are no references outside the overbar, the i subscript may be left off. Epsilon (ϵ) appearing in syntax represents an empty production. The notation "..." represents a string literal with arbitrary contents.

Bitvectors (machine integers) may be any width greater than zero. Bitvector *constants* are represented as 0bC, which can be thought of as an explicit sequence of zeros and ones. The number of bits in a bitvector constant (that is, the number of digits) gives its type. Thus, 0b00 and 0b0000 are different. In the concrete syntax, bitvector constants whose size is a multiple of 4 can also be written in the form 0xC, where C is a hexadecimal constant. These are desugared in the parser and not shown further in this document.

The Cassiopea and Alewife syntaxes are disjoint. Some elements are the same in each, but these are specified separately regardless. They use the same metavariables as well, which should not be mixed; any language construct in a judgment should be all Cassiopea or all Alewife. In a few places mixing is needed, in which case the translation defined in Section A.7 is applied to allow inserting Alewife fragments into Cassiopea terms. The Alewife rules in Section A.6 do use the same *environments* as the Cassiopea rules. These should be construed as holding only Cassiopea elements. Further details can be found in Section A.6.

A.2   Cassiopea Overview

This section covers the abstract syntax for Cassiopea.

Cassiopea is a register-transfer-list style language: it models instructions as non-Turing-complete procedures that update a machine state. Its executable semantics are covered in Section A.4.

We model the machine from the assembly-language programmer's perspective. In particular, we do not treat memory as a huge block of address space but instead treat it in small chunks passed in from somewhere else. We model both control registers and general-purpose registers as well as other machine state such as whether interrupts are enabled.

Furthermore we must allow assembler labels, which have addresses, but those addresses are not resolved until after programs are compiled and linked and must be treated as abstract.

Notation. We use the following metavariables:

x, y, z	Program variables (binders)
r	Registers (abstract)
C	Integer constants (written in decimal)
0bC	Bitvector constants (written in binary)
τ	Types
v	Values
e	Expressions
S	Statements
i, j	Rule-level integers

(Other constructs are referred to with longer names.)

A number of constructs are lists (with a null case and a cons case) – these are written out in longhand so that typing and semantic judgments can be applied explicitly to each case, in order to, for example, propagate environment updates correctly.

Identifiers and Variables. Identifiers are divided syntactically into seven categories:

(Cassiopea Types)

$$\tau ::= \quad \tau_{base} \mid \tau_{mem} \mid \tau_{func}$$

$$\tau_{base} ::= \quad () \mid \text{int} \mid \text{bool} \mid \text{string} \mid x_\tau \mid C \text{ bit}$$
$$\mid \quad \tau_{reg} \mid \tau_{regs} \mid \tau_{label}$$
$$\tau_{reg} ::= \quad C \text{ reg}$$
$$\tau_{regs} ::= \quad C \text{ reg set}$$
$$\tau_{label} ::= \quad C \text{ label}$$

$$\tau_{mem} ::= \quad C_1 \text{ bit } C_2 \text{ len } C_3 \text{ ref}$$
$$\tau_{func} ::= \quad \overline{\tau_{base_i}} \rightarrow \tau_{base}$$

(Cassiopea Operators)

$$unop ::= \quad - \mid \text{b}- \mid \neg \mid \text{bnot}$$
$$binop ::= \quad = \mid \neq \mid + \mid - \mid * \mid / \mid < \mid <= \mid > \mid >=$$
$$\mid \quad \&\& \mid || \mid \oplus$$
$$\mid \quad >> \mid >>_S \mid << \mid \text{band} \mid \text{bor} \mid \text{bxor}$$
$$\mid \quad \text{b}+ \mid \text{b}- \mid \text{b}* \mid \text{b}/$$
$$\mid \quad \text{b}< \mid \text{b}<= \mid \text{b}> \mid \text{b}>=$$
$$\mid \quad \text{bs}< \mid \text{bs}<= \mid \text{bs}> \mid \text{bs}>=$$
$$\mid \quad \cup \mid \cap \mid \subseteq \mid \backslash$$

(Cassiopea Values)

$$\tilde{v} ::= \quad v \mid \text{fail}$$
$$v ::= \quad \text{true} \mid \text{false} \mid C \mid \text{0b}C \mid \text{"..."}$$
$$\mid \quad r \mid (x_{mem}, C)$$

(Cassiopea Expressions)

$$e ::= \quad \tilde{v} \mid x$$
$$\mid \quad e.\text{txt}$$
$$\mid \quad x_{func} \, (\overline{e})$$
$$\mid \quad unop \; e$$
$$\mid \quad e_1 \; binop \; e_2$$
$$\mid \quad e[C] \mid e[C_1, C_2]$$
$$\mid \quad \text{let } x : \tau_{base} = e_1 \text{ in } e_2$$
$$\mid \quad \text{if } e_1 \text{ then } e_2 \text{ else } e_3$$
$$\mid \quad (x_{mem}, e)$$
$$\mid \quad {}^*e \mid \text{fetch}(e, C)$$
$$\mid \quad \text{branchto}$$
$$\mid \quad \{x_1, \ldots, x_k\}$$
$$\mid \quad \|e\| \mid e_1 \in e_2$$

Fig. 11. Cassiopea types, values, operators, and expressions.

- x_{mem} are identifiers bound to memory regions, which are second-class.
- x_{func} are identifiers bound to functions, which are second-class.
- x_{proc} are identifiers bound to procedures, which are second-class.
- x_{op} are identifiers bound to instructions ("operations"), which are akin to procedures but distinguished from them.
- x_τ are identifiers for type aliases, which are bound to base types in declarations.
- x_{module} are the names of "modules", which are used to select among many possible groups of lowering elements.
- Other identifiers x are used for other things, and should be assumed to *not* range over the above elements.

Note that all identifiers live in the same namespace, and rebinding or shadowing them is not allowed. All these identifiers can be thought of as variables, in the sense that they are names that stand for other things. All of them are immutable once defined, including the ordinary variables x that contain plain values.

Types. Types are divided syntactically into base types (integers, booleans, strings, bitvectors, etc.) and others (memory regions and functions). Functions may handle only base types. Furthermore, memory regions and functions are second-class for reasons discussed below and are excluded in various places in the syntax and the typing rules.

We use index typing to capture the bit width of values.

Registers. Registers are represented in the specification with the metavariable r, which stands for the underlying abstract identity of a register. Declaring a register, e.g., with `letstate` $x : C$ `reg` as shown in Figure 12, allocates a fresh register r and binds the variable x to it. A subsequent declaration of the form `let` $y : C$ `reg` $= x$ creates another variable y that refers to the same underlying register. One might think of registers as numbered internally. We use the form `letstate control` $x : C$ `reg` to declare specific *control* registers, which are treated differently by the framing rules. The additional keyword `dontgate` inhibits state gating for the register; this should be used for flags registers and anything similar that is implicitly used by all ordinary code.

Some registers have associated with them a text form, which is declared separately and is the form an assembler expects to parse. The synthesis engine uses this to extract an assembly program from Cassiopea's internal representation. It is referred to by attaching the suffix `.txt` to the/a register variable. As some registers are not directly addressable by the assembler (e.g., they might be subfields of some larger addressable unit or non-addressable internal state), not all registers have a text form. This is not readily checked statically without additional information, so invalid `.txt` references fail at assembly code extraction time.

The type of a register is C `reg`, which is a register that holds a C-bit bitvector. The bitvector value in question can be updated by assigning a new value; this is a statement ($e_1 : = e_2$) and can only happen in places where statements are allowed. The construct $^*e$ reads a register.

The reader will note that the semantics rules for machines and declarations do not provide initial values for registers. Instead, executions are defined in terms of some initial register state (and also some memory state), which is required to have the right registers to match the machine definition. This allows reasoning about the execution of programs and program fragments in terms of many or all possible initial states. These issues are discussed further below.

Memory. A memory region has the type C_1 `bit` C_2 `len` C_3 `ref`, shown in Figure 11. This refers to a memory region that has C_2 cells, each of which stores a bitvector of width C_1. This memory region is addressed with pointers of width C_3. Note that we assume byte-addressed machines, and for the purposes of both this specification and our implementation, we assume bytes are 8 bits wide. (This restriction could be relaxed if we wanted to model various historic machines.) Thus a memory region of type `32 bit 4 len 32 ref` has 4 32-bit values in it, which can be addressed at byte offsets 0, 4, 8, and 12. Memory regions and registers are mutable state.

Memory regions are named with identifiers. These names, and memory regions themselves, are not first class; variables are not allowed to range over them. Also note that memory regions are a property of programs (and thus are declared in specifications) and not a property of the machine as a whole.

Pointers. A pointer literal has the form (x_{mem}, C), in which x_{mem} is the name of a memory region and C is the offset, shown in Figure 11. Because memory regions are second-class, x_{mem} must be one of the available declared memory regions. Pointer literals exist in the abstract syntax, but are not allowed in the concrete syntax except in specifications. The only way to get a pointer value is to look up a label (discussed immediately below) or have it provided in a register as part of the initial machine state.

A pointer literal is treated as a bitvector of the width appropriate for pointers to the memory region, so one can appear in a register or in memory. However, we enforce a restriction (not captured in the semantic judgments so far) that no value in the initial machine state, whether in a register or in memory, is a pointer unless required to be so by the precondition part of the specification. All other values are restricted to be plain bitvector values.

Addition and subtraction are allowed on pointers and they change the offset, but other bitvector operations (e.g., multiply) are disallowed and fail. Similarly, attempting to fetch from or store to a plain bitvector that is not a pointer fails. Note however that we do not statically distinguish pointers and plain bitvectors. (We could have used flow-sensitive typing to reason about when registers and memory cells contain pointers and when they do not; but this adds substantial complexity and for our problem domain does not provide significant value.) Instead, we step to failure at runtime. This can be seen in the semantic judgments.

(Cassiopea Declarations)

$decls ::= \qquad\qquad\qquad\qquad\qquad \epsilon \mid decl;\ decls$

$decl ::= \qquad\qquad\qquad\qquad\qquad \text{type } x_\tau = \tau_{base}$

(Cassiopea Statements)

$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \qquad\qquad\qquad\qquad \text{let } x : \tau_{base} = e$

$S ::= \qquad\qquad\qquad\qquad\qquad\qquad\qquad S;\ S \qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \qquad\qquad\qquad\qquad\quad \text{let } x.\text{txt} = e$

$\mid \qquad\qquad\qquad\qquad\qquad\quad x_{proc}\ (\overline{e}) \qquad\qquad\qquad\qquad\qquad \mid \qquad \text{def } x_{func}\ \overline{x_i : \tau_{base_i}} \rightarrow \tau_{base} = e$

$\mid \qquad\qquad\qquad \text{let } x : \tau_{base} = e \text{ in } S \qquad\qquad\qquad\qquad \mid \qquad \text{proc } x_{proc}\ \overline{x_i : \tau_{base_i}} \rightarrow () = S$

$\mid \qquad\qquad\quad \text{for } x \in (C_1 \dots C_2) \text{ do } S \qquad\qquad\qquad\qquad\qquad \mid \qquad\qquad\qquad\qquad \text{letstate } x : \tau_{reg}$

$\mid \qquad\qquad\qquad \text{if } e \text{ then } S_1 \text{ else } S_2 \qquad\qquad\qquad\qquad\qquad \mid \qquad\qquad\qquad \text{letstate control } x : \tau_{reg}$

$\mid \qquad\qquad\qquad\qquad\qquad\quad e_1 := e_2 \qquad\qquad\qquad\qquad \mid \qquad \text{letstate control dontgate } x : \tau_{reg}$

$\mid \qquad\qquad\qquad\quad \text{store}(e_1, C) \leftarrow e_2 \qquad\qquad\qquad\qquad\qquad \mid \qquad\qquad\qquad\quad \text{letstate } x_{mem} : \tau_{mem}$

$\mid \qquad\qquad\qquad\qquad\qquad \text{BRANCH}(e) \qquad\qquad\qquad\qquad\qquad \mid \qquad\qquad\quad \text{letstate } x_{mem} : \tau_{mem} \text{ with } x$

$\mid \qquad\qquad\qquad\qquad\qquad\quad \text{assert}(e)$

$\mid \qquad\qquad\qquad\qquad\qquad\qquad\quad \text{skip} \qquad\qquad\qquad defops ::= \qquad\qquad\qquad\qquad \epsilon \mid defop;\ defops$

$\mid \qquad\qquad\qquad\qquad\qquad\qquad\quad \text{crash} \qquad\qquad\qquad defop ::= \qquad\qquad\quad \text{defop } x_{op}\ \{\text{txt} = e, \text{sem} = S\}$

$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \qquad \text{defop } x_{op}\ \overline{x_i : \tau_{base_i}}\ \{\text{txt} = e, \text{sem} = S\}$

Fig. 12. Cassiopea statements and declarations.

Fetching from a pointer takes the form fetch (e,C). Storing to a pointer takes the form $\text{store}(e_1, C) \leftarrow e_2$. The extra constant C specifies the width of the cell pointed to. (This is not an offset.) Because we do not check pointers statically, we do not know the memory region being pointed to and cannot look up its cell size; thus we need the width explicitly for typing. It is checked at runtime.

Labels. As mentioned above, the term "label" means an assembler label or linker symbol. These are constant addresses that are not known at assembly time, so we must model them abstractly.

When one declares a memory region, one may attach a label to it, which is an additional identifier. This identifier is created as a variable of type C label in Figure 11. The value is a pointer to the first entry in the region, and a single type subsumption rule allows this value to be accessed and placed in an ordinary register or variable of suitable bitvector type. The intended mechanism is that for each machine the preferred instruction on that machine for loading assembler symbols into a register can be defined to take an operand of type C label, and then its value can be assigned to the destination register. This type restriction on the operand is sufficient to synthesize programs that use labels correctly.

Code Positions and Branching. Cassiopea code blocks may contain branches, either forward within the block (branching backwards is forbidden) or to a single external assembler label outside the block. This model is sufficient for our block-oriented synthesis approach; more complex control flow can be handled with multiple blocks.

Consequently, a branch may either go to the single external assembler label (which terminates execution of the current block) or skip zero or more instructions. We model branch targets as an 8-bit skip count. In Figure 12, the statement BRANCH(0bC) skips 0bC instructions; BRANCH(0xff) jumps to the external assembler label. This statement may be used to define both conditional and unconditional branch instructions. Such instructions should be defined to take an operand of type 8 bit to choose the branch destination. This magic number should then

be printed to the output assembly text using the built-in function textlabel, which replaces it with a valid assembler label, either the selected external label string or a scratch label attached to the proper destination instruction.

Specifications do not need to be directly concerned with internal branches, which occur or not as needed. However, external branches are part of a block's specification; typically the purpose of a block with an external branch is to test some condition and then either branch away or fall through to the next block. It is thus necessary to be able both to name the external label to use and to specify the conditions when it should be reached. For this purpose a predicate branchto is provided. It may appear in the postcondition and governs the exit path from the block: if forced to true, the block branches to the external assembler label, and if false, the block falls through from its last instruction. The concrete syntax is branchto(dest) which also sets the assembler label used to dest. It is not valid to name more than one such assembler label.

Note that the assembler labels used in branching are, though also assembler labels, a separate mechanism unrelated to the labels attached to memory regions; they are code labels rather than data labels and inherently work differently.

Register Sets. Register sets are second-class elements intended to exist only as literals and only as the result of lowering machine-independent specifications that cannot directly talk about specific registers. Currently they do not exist in the implementation. Register sets are not allowed to be operands to instructions to avoid state explosions when synthesizing. For simplicity, this restriction is not shown in the abstract syntax or typing rules.

Functions and Procedures. Functions, defined with def in Figure 12, are pure functions whose bodies are expressions. They produce values. They can read registers and memory, and can fail, but cannot update anything. Procedures, defined with proc, are impure and their bodies are statements. They do not produce values, but they may update the machine state. They are otherwise similar, and are intended to be used to abstract out common chunks of functionality shared among multiple instructions in machine descriptions. Functions can also be used for state hiding in specifications.

Functions and procedures are second-class; they may be called only by their own name and may not be bound to variables or passed around. Furthermore, they are only allowed to handle base types: higher-order functions are explicitly not supported.

Operations. Operations (defined with defop in Figure 12) are machine instructions or short sequences of machine instructions. An operation takes zero or more operands and transforms the machine state as defined by one or more statements. Operands are currently defined as expressions, but are restricted as follows:

- They may be values, but not string values, and not fail.
- They may be variables of register type.
- They may be variables of label type.

This restriction affects what the synthesizer tries to generate; a broader set of expressions may be accepted for verification or concrete execution and simply evaluated in place.

In general, we refer to "instructions" and "operations" interchangeably. However, there is an important distinction between them: operations do not necessarily need to be single machine instructions. The text output to the assembler is arbitrary and if desired can be computed on the fly based on the operand values. On some platforms the assembler defines "synthetic instructions" that potentially assemble to multiple machine instructions, or to different sequences based on the constants or registers used. This facility takes that a step further by allowing the writer of the machine description to define their own synthetics. Operations are the units in which Cassiopea reasons about machine operations and the units in which Cassiopea generates programs and code fragments.

Other Constructs. $e[C]$ and $e[C_1, C_2]$ in Figure 11 extract a single bit and a slice, respectively, from a bitvector. The offsets are constants; if variable offsets are needed, the value can be shifted first. The width of the slice must be constant for static typing.

		(Cassiopea Lowerings)
	modules ::=	ϵ | *module*; *modules*
(Cassiopea Machines)	*module* ::=	module x_{module} {*decls*; *frame*}
machine ::= *decls*; *defops*		
		(Cassiopea Specifications)
(Cassiopea Programs)	*frame* ::=	reg-modify : $\overline{x_{reg_i}}$
inst ::= x_{op} | x_{op} $\overline{e}$	|	mem-modify : $\overline{(x_{mem_i}, e_i)}$
insts ::= ϵ | *inst*; *insts*	*frames* ::=	ϵ | *frame frames*
program ::= *insts*	*pre* ::=	e
	post ::=	e
	spec ::=	*decls*; *frames*; *pre*; *post*

Fig. 13. Cassiopea machines, lowerings, programs, and specifications.

Machines, Lowering, Specifications, and Programs.

A *machine* is the full description of a machine architecture; it includes declarations (including types, constants, registers, functions and procedures) and also instructions, shown in Figure 13. This is typically a standalone file but may be a set of files referenced via include.

A (single) lowering is a collection of declarations used to instantiate elements in Alewife translations, shown in Figure 13. These are placed into a *module*, with multiple modules per file, so that the lowerings associated with multiple related Alewife specifications can be kept together. The import x_{module} directive enables sharing common elements in one module x_{module} across multiple specifications. The modules used to lower a specification are selected using the lower-with declaration in Alewife.

A *spec* (specification) is a precondition and postcondition, which are boolean expressions, along with optional permission to modify additional registers (the *frame*), shown in Figure 13. Cassiopea specifications are produced by compiling Alewife specifications. Note that a module can also contain frame declarations; they are added to any frame conditions provided in the Alewife specification. A code block is permitted to modify any register that is either explicitly listed in the frame declarations or mentioned in the postcondition, while it may read any register mentioned in the precondition and any control register. This restriction is currently not adequately captured in the semantics rules.

A *program* is a sequence of instruction invocations, shown in Figure 13.

Built-in Functions. Here is a partial list of the built-in functions in Cassiopea along with their types.

- empty : int $\rightarrow$ C reg set produces an empty register set of bit size C, where C is the value of the first argument. Note that this built-in function is dependently typed and treated specially during type checking. The first argument must be a constant.
- hex : C bit $\rightarrow$ string prints numbers in hexadecimal.
- bin : C bit $\rightarrow$ string prints numbers in binary.
- dec : C bit $\rightarrow$ string prints numbers in decimal.
- lbl : C label $\rightarrow$ string prints labels (it returns the label identifier as a string). This is for data labels attached to memory locations.
- textlabel : 8 bit $\rightarrow$ string prints branch offsets as assembler labels. This is for code labels, as described above.

(Type Well-Formedness)

$$\overline{\Delta \vdash_{\mathsf{wf}} ()} \quad \overline{\Delta \vdash_{\mathsf{wf}} \mathtt{int}} \quad \overline{\Delta \vdash_{\mathsf{wf}} \mathtt{bool}} \quad \overline{\Delta \vdash_{\mathsf{wf}} \mathtt{string}}$$

$$\frac{C > 0}{\Delta \vdash_{\mathsf{wf}} C\,\mathtt{bit}} \quad \frac{C > 0}{\Delta \vdash_{\mathsf{wf}} C\,\mathtt{reg}} \quad \frac{C > 0}{\Delta \vdash_{\mathsf{wf}} C\,\mathtt{label}} \quad \frac{C > 0}{\Delta \vdash_{\mathsf{wf}} C\,\mathtt{reg\ set}}$$

$$\frac{C_1 > 0,\ C_2 > 0,\ C_3 > 0}{\Delta \vdash_{\mathsf{wf}} C_1\,\mathtt{bit}\ C_2\,\mathtt{len}\ C_3\,\mathtt{ref}} \quad \frac{\Delta(x) = \tau \quad \Delta \vdash_{\mathsf{wf}} \tau}{\Delta \vdash_{\mathsf{wf}} x} \quad \frac{\forall i, \Delta \vdash_{\mathsf{wf}} \tau_i \quad \Delta \vdash_{\mathsf{wf}} \tau_r}{\Delta \vdash_{\mathsf{wf}} \overline{\tau_i} \to \tau_r}$$

Fig. 14. Cassiopea type well-formedness.

- format : string $\to$ string $\ldots \to$ string formats strings. The first argument is a format string; the remainder of the arguments are substituted into the format string where a dollar sign appears followed by the argument number (1-based). (A literal dollar sign can be inserted by using $$.) The number of additional arguments expected is deduced from the contents of the format string.
- bv_to_len : int $\to C_2$ bit $\to C_1$ bit returns a new bitvector of size C_1 (where C_1 is the value of the first argument) with the same value as the second argument, up to the ability of the new size to represent that value. Note that this built-in function is dependently typed and treated specially during type checking. The first argument must be a constant.
- bv_to_uint : C_1 bit $\to$ int converts a bitvector to unsigned int.
- uint_to_bv_l : int $\to$ int $\to C_1$ bit converts an unsigned int (second argument) into a bitvector of size C_1, where C_1 is the value of the first argument. Note that this built-in function is dependently typed and treated specially during type checking. The first argument must be a constant.
- isptr : C bit $\to$ bool tests at runtime if a bitvector-typed value is a pointer.

Note that some of these functions have their own typing rules, some of which are polymorphic in bitvector size. We have not complicated the typing rules presented by including all of these as special cases.

Concrete Syntax. We do not describe the concrete syntax in detail here; however, it does not stray very far from the abstract syntax. The operator precedence and most of the operator spellings are taken from C but most of the rest of the concrete syntax is ML-style. There are also a few things desugared in the parser and not shown in the abstract syntax. As already mentioned, bitvector constants whose size is a multiple of 4 can also be written in the form 0xC. Syntax of the form e.hex, e.bin, and e.dec is converted to the built-in functions hex, bin, dec respectively. These print either integers or bitvectors as strings in hexadecimal, binary, or decimal respectively. The syntax x.lbl is similarly converted to the built-in function lbl. This produces the label (that is, the identifier naming the label) as a string. Further the concrete syntax supports include files via an include directive, which is useful for sharing common elements between related descriptions.

A.3 Cassiopea Static Typing

This section describes the Cassiopea type system.

Environments. The type system uses two environments: Δ maps type alias names to their expansions, and Γ maps variable names to types. Recall from the syntax that type alias names may only expand to base types; thus type alias names can be treated as base types.

Well-Formedness. Since types syntactically include type alias names, we need to check the validity of those names. We also insist that the widths of bitvectors be greater than zero. The judgment for this has the form

(Expression Typing)

$$\overline{\Delta, \Gamma \vdash C : \text{int}} \quad \overline{\Delta, \Gamma \vdash "..." : \text{string}} \quad \overline{\Delta, \Gamma \vdash \text{true} : \text{bool}} \quad \overline{\Delta, \Gamma \vdash \text{false} : \text{bool}}$$

$$\frac{C \in \{0,1\}^k}{\Delta, \Gamma \vdash \text{0b}C : k \text{ bit}} \quad \frac{\Delta \vdash_{\text{wf}} \tau}{\Delta, \Gamma \vdash \text{fail} : \tau} \quad \frac{\Gamma(x) = \tau \quad \Delta \vdash_{\text{wf}} \tau}{\Delta, \Gamma \vdash x : \tau} \quad \frac{\Delta, \Gamma \vdash e : \tau_{reg}}{\Delta, \Gamma \vdash e.\text{txt} : \text{string}}$$

$$\frac{\Delta, \Gamma \vdash x_{func} : \left(\overline{\tau_{base_i}} \to \tau_{base}\right) \quad \forall i, \ \Delta, \Gamma \vdash e_i : \tau_{base_i}}{\Delta, \Gamma \vdash x_{func} \ (\overline{e_i}) : \tau_{base}} \quad \frac{\Delta, \Gamma \vdash e : \tau_{base_1} \quad \vDash unop : \tau_{base_1} \to \tau_{base_2}}{\Delta, \Gamma \vdash unop \ e : \tau_{base_2}}$$

$$\frac{\Delta, \Gamma \vdash e_1 : \tau_{base_1} \quad \Delta, \Gamma \vdash e_2 : \tau_{base_1} \quad \vDash binop : \tau_{base_1} \to \tau_{base_1} \to \tau_{base_2}}{\Delta, \Gamma \vdash e_1 \ binop \ e_2 : \tau_{base_2}}$$

$$\frac{\Delta, \Gamma \vdash e : C_2 \text{ bit} \quad 0 \le C_1 < C_2}{\Delta, \Gamma \vdash e[C_1] : 1 \text{ bit}} \quad \frac{\Delta, \Gamma \vdash e : C_3 \text{ bit} \quad 0 \le C_1 < C_2 \le C_3 \quad k = C_2 - C_1}{\Delta, \Gamma \vdash e[C_1, C_2] : k \text{ bit}}$$

$$\frac{\Delta, \Gamma \vdash e_1 : \tau_{base} \quad x \notin \Delta, \Gamma \quad \Delta, \Gamma[x \mapsto \tau_{base}] \vdash e_2 : \tau_2}{\Delta, \Gamma \vdash \text{let } x : \tau_{base} = e_1 \text{ in } e_2 : \tau_2}$$

$$\frac{\Delta, \Gamma \vdash b : \text{bool} \quad \Delta, \Gamma \vdash e_1 : \tau \quad \Delta, \Gamma \vdash e_2 : \tau}{\Delta, \Gamma \vdash \text{if } b \text{ then } e_1 \text{ else } e_2 : \tau}$$

$$\frac{\Delta, \Gamma \vdash e : \text{int} \quad \Delta, \Gamma \vdash x_{mem} : \_\text{bit}\_\text{len } C \text{ ref}}{\Delta, \Gamma \vdash (x_{mem}, e) : C \text{ bit}} \quad \frac{\Delta, \Gamma \vdash e : \_\text{bit} \quad C > 0}{\Delta, \Gamma \vdash \text{fetch}(e, C) : C \text{ bit}}$$

$$\frac{\Delta, \Gamma \vdash e : C \text{ reg}}{\Delta, \Gamma \vdash {}^*e : C \text{ bit}} \quad \frac{\Delta, \Gamma \vdash x : C \text{ label}}{\Delta, \Gamma \vdash x : C \text{ bit}} \quad \overline{\Delta, \Gamma \vdash \text{branchto} : \text{bool}}$$

$$\frac{\forall i \in (1 \ldots k), \ \Delta, \Gamma \vdash x_i : C \text{ reg}}{\Delta, \Gamma \vdash \{x_1, \ldots, x_k\} : C \text{ reg set}} \quad \frac{\Delta, \Gamma \vdash e : C \text{ reg set}}{\Delta, \Gamma \vdash \|e\| : \text{int}} \quad \frac{\Delta, \Gamma \vdash e_1 : C \text{ reg} \quad \Delta, \Gamma \vdash e_2 : C \text{ reg set}}{\Delta, \Gamma \vdash e_1 \in e_2 : \text{bool}}$$

Fig. 15. Cassiopea typing rules for expressions.

$\Delta \vdash_{\text{wf}} \tau$, shown in Figure 14. There is an intended invariant that only well-formed types may be entered into the variable typing environment Γ, so that types taken out of it do not need to be checked for well-formedness again.

In a typing environment comprised of Δ mapping user-defined type names (type aliases) to types and Γ mapping variables to types, we say that a type is well formed when all type names exist and refer to well-formed types, and all indices are of type int and positive.

Expressions. Expressions produce values that have types. Because types appear explicitly in some expressions (e.g., let), we need both environments, so the form of an expression typing judgment is $\Delta, \Gamma \vdash e : \tau$, shown in Figure 15. This means that we conclude e has type τ. Note that the .txt form is restricted to registers; it is specifically for extracting the assembly text form of a register. We have not written out a separate rule for each unary and binary operator. The types of operators are as shown in Figure 16. Note that the bitvector operators are polymorphic in bit size.

(Operator Expression Typing)

$-$				$\text{int} \to \text{int}$
$\text{b}-$				$\forall C,\ C\,\text{bit} \to C\,\text{bit}$
$\neg$				$\text{bool} \to \text{bool}$
bnot				$\forall C,\ C\,\text{bit} \to C\,\text{bit}$

$=$	$\neq$			$\forall \tau_{base},\ \tau_{base} \to \tau_{base} \to \text{bool}$
$+$	$-$	$*$	$/$	$\text{int} \to \text{int} \to \text{int}$
$<$	$<=$	$>$	$>=$	$\text{int} \to \text{int} \to \text{bool}$
$\&\&$	$\|\|$	$\oplus$		$\text{bool} \to \text{bool} \to \text{bool}$
$>>$	$>>_S$	$<<$		
band	bor	bxor		
$\text{b}+$	$\text{b}-$	$\text{b}*$	$\text{b}/$	$\forall C,\ C\,\text{bit} \to C\,\text{bit} \to C\,\text{bit}$
$\text{b}<$	$\text{b}<=$	$\text{b}>$	$\text{b}>=$	
$\text{bs}<$	$\text{bs}<=$	$\text{bs}>$	$\text{bs}>=$	$\forall C,\ C\,\text{bit} \to C\,\text{bit} \to \text{bool}$
$\cup$	$\cap$	$\backslash$		$\forall C,\ C\,\text{reg set} \to C\,\text{reg set} \to C\,\text{reg set}$
$\subseteq$				$\forall C,\ C\,\text{reg set} \to C\,\text{reg set} \to \text{bool}$

Fig. 16. Cassiopea typing rules for *unop* and *binop*.

(Statement Typing)

$$\frac{\Delta, \Gamma \vdash S_1 \qquad \Delta, \Gamma \vdash S_2}{\Delta, \Gamma \vdash S_1;\ S_2} \qquad \frac{\Delta, \Gamma \vdash x_{proc} : (\overline{\tau_{base_i}} \to ()) \qquad \forall i,\ \Delta, \Gamma \vdash e_i : \tau_{base_i}}{\Delta, \Gamma \vdash x_{proc}\ (\overline{e_i})}$$

$$\frac{\Delta, \Gamma \vdash e : \tau_{base} \qquad x \notin \Delta, \Gamma \qquad \Delta, \Gamma[x \mapsto \tau_{base}] \vdash S}{\Delta, \Gamma \vdash \text{let } x : \tau_{base} = e \text{ in } S} \qquad \frac{x \notin \Delta, \Gamma \qquad \Delta, \Gamma[x \mapsto \text{int}] \vdash S}{\Delta, \Gamma \vdash \text{for } x \in (C_1 \ldots C_2) \text{ do } S}$$

$$\frac{\Delta, \Gamma \vdash e : \text{bool} \qquad \Delta, \Gamma \vdash S_1 \qquad \Delta, \Gamma \vdash S_2}{\Delta, \Gamma \vdash \text{ if } e \text{ then } S_1 \text{ else } S_2} \qquad \frac{\Delta, \Gamma \vdash e_1 : C\,\text{reg} \qquad \Delta, \Gamma \vdash e_2 : C\,\text{bit}}{\Delta, \Gamma \vdash e_1 := e_2}$$

$$\frac{\Delta, \Gamma \vdash e_1 : C_1\,\text{bit} \qquad \Delta, \Gamma \vdash e_2 : C_2\,\text{bit}}{\Delta, \Gamma \vdash \text{store}(e_1, C_2) \leftarrow e_2} \qquad \frac{\Delta, \Gamma \vdash e : 8\,\text{bit}}{\Delta, \Gamma \vdash \text{BRANCH}(e)} \qquad \frac{\Delta, \Gamma \vdash e : \text{bool}}{\Delta, \Gamma \vdash \text{assert}(e)}$$

$$\frac{}{\Delta, \Gamma \vdash \text{skip}} \qquad \frac{}{\Delta, \Gamma \vdash \text{crash}}$$

Fig. 17. Cassiopea typing rules for statements.

Arguably the right hand argument of the shift operators should be allowed to be a different width. There is one rule for pointer literals that covers both the expression and the value form. There is no rule (either in the typing or in the semantics) that allows taking a subrange of a memory region as a new smaller region. We have not needed this for our use cases, so for simplicity we do not support it.

Statements. Statements do not produce values. We still need both environments, though, so the form of a typing judgment for a statement is $\Delta, \Gamma \vdash S$, shown in Figure 17. This means that S is well typed.

<div align="center">

(Declaration Typing)

</div>

$$\frac{}{\Delta, \Gamma \vdash \epsilon \rhd \Delta, \Gamma} \qquad \frac{\Delta, \Gamma \vdash \mathit{decl} \rhd \Delta', \Gamma' \qquad \Delta', \Gamma' \vdash \mathit{decls} \rhd \Delta'', \Gamma''}{\Delta, \Gamma \vdash \mathit{decl}; \mathit{decls} \rhd \Delta'', \Gamma''}$$

$$\frac{\Delta \vdash_{\mathsf{wf}} \tau_{base} \qquad x_\tau \notin \Delta, \Gamma \qquad \Delta' = \Delta[x_\tau \to \tau_{base}]}{\Delta, \Gamma \vdash \mathsf{type}\ x_\tau = \tau_{base} \rhd \Delta', \Gamma} \qquad \frac{\Delta, \Gamma \vdash x : \tau_{reg} \qquad \Delta, \Gamma \vdash e : \mathsf{string}}{\Delta, \Gamma \vdash \mathsf{let}\ x.\mathsf{txt} = e \rhd \Delta, \Gamma}$$

$$\frac{\Delta \vdash_{\mathsf{wf}} \tau_{base} \qquad x \notin \Delta, \Gamma \qquad \Delta, \Gamma \vdash e : \tau_{base} \qquad \Gamma' = \Gamma[x \mapsto \tau_{base}]}{\Delta, \Gamma \vdash \mathsf{let}\ x : \tau_{base} = e \rhd \Delta, \Gamma'}$$

$$\frac{\Delta \vdash_{\mathsf{wf}} \left(\overline{\tau_{base_i}} \to \tau_{base}\right)}{x_{func} \notin \Delta, \Gamma \qquad \Gamma' = \Gamma[\forall i, x_i \mapsto \tau_{base_i}] \qquad \Delta, \Gamma' \vdash e : \tau_{base} \qquad \Gamma'' = \Gamma[x_{func} \mapsto \left(\overline{x_i : \tau_{base_i}} \to \tau_{base}\right)]}{\Delta, \Gamma \vdash \mathsf{def}\ x_{func}\ \overline{x_i : \tau_{base_i}} \to \tau_{base} = e \rhd \Delta, \Gamma''}$$

$$\frac{\Delta \vdash_{\mathsf{wf}} \left(\overline{\tau_{base_i}} \to ()\right)}{x_{proc} \notin \Delta, \Gamma \qquad \Gamma' = \Gamma[\forall i, x_i \mapsto \tau_{base_i}] \qquad \Delta, \Gamma' \vdash S \qquad \Gamma'' = \Gamma[x_{proc} \mapsto \left(\overline{x_i : \tau_{base_i}} \to ()\right)]}{\Delta, \Gamma \vdash \mathsf{proc}\ x_{proc}\ \overline{x_i : \tau_{base_i}} \to () = S \rhd \Delta, \Gamma''}$$

$$\frac{\Delta \vdash_{\mathsf{wf}} C\ \mathsf{reg} \qquad x \notin \Delta, \Gamma \qquad \Gamma' = \Gamma[x \mapsto C\ \mathsf{reg}]}{\Delta, \Gamma \vdash \mathsf{letstate}\ x : C\ \mathsf{reg} \rhd \Delta, \Gamma'}$$

$$\frac{\Delta \vdash_{\mathsf{wf}} C\ \mathsf{reg} \qquad x \notin \Delta, \Gamma \qquad \Gamma' = \Gamma[x \mapsto C\ \mathsf{reg}]}{\Delta, \Gamma \vdash \mathsf{letstate\ control}\ x : C\ \mathsf{reg} \rhd \Delta, \Gamma'}$$

$$\frac{\Delta \vdash_{\mathsf{wf}} C\ \mathsf{reg} \qquad x \notin \Delta, \Gamma \qquad \Gamma' = \Gamma[x \mapsto C\ \mathsf{reg}]}{\Delta, \Gamma \vdash \mathsf{letstate\ control\ dontgate}\ x : C\ \mathsf{reg} \rhd \Delta, \Gamma'}$$

$$\frac{\Delta \vdash_{\mathsf{wf}} C_1\ \mathsf{bit}\ C_2\ \mathsf{len}\ C_3\ \mathsf{ref} \qquad x_{mem} \notin \Delta, \Gamma \qquad \Gamma' = \Gamma[x_{mem} \mapsto C_1\ \mathsf{bit}\ C_2\ \mathsf{len}\ C_3\ \mathsf{ref}]}{\Delta, \Gamma \vdash \mathsf{letstate}\ x_{mem} : C_1\ \mathsf{bit}\ C_2\ \mathsf{len}\ C_3\ \mathsf{ref} \rhd \Delta, \Gamma'}$$

$$\frac{\Delta \vdash_{\mathsf{wf}} C_1\ \mathsf{bit}\ C_2\ \mathsf{len}\ C_3\ \mathsf{ref} \qquad \Delta \vdash_{\mathsf{wf}} C_3\ \mathsf{label}}{x_{mem} \notin \Delta, \Gamma \qquad x \notin \Delta, \Gamma \qquad \Gamma' = \Gamma[x_{mem} \mapsto C_1\ \mathsf{bit}\ C_2\ \mathsf{len}\ C_3\ \mathsf{ref}] \qquad \Gamma'' = \Gamma'[x \mapsto C_3\ \mathsf{label}]} {\Delta, \Gamma \vdash \mathsf{letstate}\ x_{mem} : C_1\ \mathsf{bit}\ C_2\ \mathsf{len}\ C_3\ \mathsf{ref\ with}\ x \rhd \Delta, \Gamma''}$$

$$\frac{}{\Delta, \Gamma \vdash \epsilon \rhd \Delta, \Gamma} \qquad \frac{\Delta, \Gamma \vdash \mathit{defop} \rhd \Delta', \Gamma' \qquad \Delta', \Gamma' \vdash \mathit{defops} \rhd \Delta'', \Gamma''}{\Delta, \Gamma \vdash \mathit{defop}; \mathit{defops} \rhd \Delta'', \Gamma''}$$

$$\frac{x_{op} \notin \Delta, \Gamma \qquad \Delta, \Gamma \vdash e : \mathsf{string} \qquad \Delta, \Gamma \vdash S \qquad \Gamma' = \Gamma[x_{op} \mapsto () \to ()]}{\Delta, \Gamma \vdash \mathsf{defop}\ x_{op}\ \{\mathsf{txt} = e, \mathsf{sem} = S\} \rhd \Delta, \Gamma'}$$

$$\frac{\Delta \vdash_{\mathsf{wf}} \left(\overline{\tau_{base_i}} \to ()\right) \qquad x_{op} \notin \Delta, \Gamma \qquad \forall i, \tau_{base_i} \neq \mathsf{string} \wedge \tau_{base_i} \neq () \wedge \tau_{base_i} \neq \tau_{regs}}{\Gamma' = \Gamma[\forall i, x_i \mapsto \tau_{base_i}] \qquad \Delta, \Gamma' \vdash e : \mathsf{string} \qquad \Delta, \Gamma' \vdash S \qquad \Gamma'' = \Gamma[x_{op} \mapsto \left(\overline{x_i : \tau_{base_i}} \to ()\right)]}{\Delta, \Gamma \vdash \mathsf{defop}\ x_{op}\ \overline{x_i : \tau_{base_i}}\{\mathsf{txt} = e, \mathsf{sem} = S\} \rhd \Delta, \Gamma''}$$

<div align="center">

Fig. 18. Cassiopea typing rules for declarations.

</div>

(Machine Typing)

$$\frac{\Delta_{builtin}, \Gamma_{builtin} \vdash decls \triangleright \Delta, \Gamma \qquad \Delta, \Gamma \vdash defops \triangleright \Delta', \Gamma'}{\vdash decls; defops \triangleright \Delta', \Gamma'}$$

(Specification Typing)

$$\frac{\vdash machine \triangleright \Delta, \Gamma \qquad \Delta, \Gamma \vdash decls \triangleright \Delta', \Gamma'}{\vdash machine; decls \triangleright \Delta', \Gamma'}$$

$$\frac{\forall i,\ \Delta, \Gamma \vdash x_{reg_i} : C_i\ \mathsf{reg}}{\Delta, \Gamma \vdash \mathsf{reg\text{-}modify} : \overline{x_{reg_i}}} \qquad \frac{\forall i,\ \Delta, \Gamma \vdash e_i : \mathsf{int} \qquad \forall i,\ \Delta, \Gamma \vdash x_{mem_i} : \tau_{mem}}{\Delta, \Gamma \vdash \mathsf{mem\text{-}modify} : \overline{(x_{mem_i},\, e_i)}}$$

$$\frac{}{\Delta, \Gamma \vdash \epsilon} \qquad \frac{\Delta, \Gamma \vdash frame \qquad \Delta, \Gamma \vdash frames}{\Delta, \Gamma \vdash frame\ frames}$$

$$\frac{\vdash machine; decls \triangleright \Delta, \Gamma \qquad \Delta, \Gamma \vdash frame \qquad \Delta, \Gamma \vdash pre : \mathsf{bool} \qquad \Delta, \Gamma \vdash post : \mathsf{bool}}{machine \vdash decls; frames; pre; post}$$

(Program Typing)

$$\frac{\Delta, \Gamma \vdash x_{op} : (() \to ())}{\Delta, \Gamma \vdash x_{op}} \qquad \frac{\Delta, \Gamma \vdash x_{op} : (\overline{\tau_{base_i}} \to ()) \qquad \forall i,\ \Delta, \Gamma \vdash e_i : \tau_{base_i}}{\Delta, \Gamma \vdash x_{op}\ \overline{e_i}}$$

$$\frac{}{\Delta, \Gamma \vdash \epsilon} \qquad \frac{\Delta, \Gamma \vdash inst \qquad \Delta, \Gamma \vdash insts}{\Delta, \Gamma \vdash inst; insts} \qquad \frac{\vdash machine \triangleright \Delta, \Gamma \qquad \Delta, \Gamma \vdash program}{machine \vdash program}$$

Fig. 19. Cassiopea typing rules for machine, specification, and program.

Declarations. Declarations update the environment. The form of a typing judgment for a declaration is $\Delta, \Gamma \vdash decl \triangleright \Delta', \Gamma'$, and a judgment for a list of declarations has the same form, shown in Figure 18. This means that the declaration (or list) is well typed and produces the new environment on the right.

We impose an additional syntactic restriction on declarations found in a machine description (as opposed to the additional declarations that may appear in a specification): they may not use the expression forms that refer to machine state (registers or memory), because when defining the machine, there is no specific machine state to refer to; any references would need to be quantified. (That in turn is not allowed; while many SMT solvers now support quantified expressions, they generally do not perform well.)

Machines. A machine is some declarations followed by some defops, so the typing rule is just sequencing, shown in Figure 19, but there is a wrinkle: the initial environment for the machine is not an input. $\Delta_{builtin}$ is the (fixed) environment describing the built-in type aliases. (Currently there are none.) $\Gamma_{builtin}$ is the environment describing the types of built-in variables. This notionally includes the built-in functions. (But as mentioned earlier some of them actually have their own typing rules.) The form of a typing judgment for a machine is $\vdash machine \triangleright \Delta, \Gamma$. This means that the machine description is well typed and provides the environment on the right for use of other constructs that depend on the machine. (Specs and programs are only valid relative to a given machine.)

Specifications. For specifications we need two helper rules, shown in Figure 19: one that applies an additional list of declarations to a machine, which has the same form as the judgment on a machine; and one that says that a frame (modifies list) is well typed, which has the form $\Delta, \Gamma \vdash frames$. This lets us write the real rule, which has the form $machine \vdash spec$ and means that the specification is well typed under the machine description.

Programs. A program is a sequence of calls to instructions. We need judgments of the form $\Delta, \Gamma \vdash inst$ for a single instruction and also $\Delta, \Gamma \vdash insts$ for the sequence, shown in Figure 19. There are two cases for a single instruction because of a minor glitch in formulation: because the overbar notation means "one or more", there are two cases in the syntax for instructions, one for zero operands and one for one or more operands; we need typing rules for both cases. Meanwhile the type entered into Γ for a zero-operand instruction is unit to unit, not ϵ to unit, to avoid needing an additional form for types just for this case. (Notice that a one-operand instruction may not have type unit to unit because unit is not allowed as an instruction operand, so the type is not ambiguous.) These rules let us write a judgment for a program, which has the form $machine \vdash program$ and means that the program is well typed relative to the machine.

Soundness. Our type system is sound: we include the necessary checks and failure states in the semantics so that evaluation does not get stuck, even though some properties are not statically checked.

A.4 Cassiopea Semantics

This section defines the semantics of Cassiopea.

Environment. The execution environment Λ maps Cassiopea variables x to values v. For labels on memory regions, each label maps to a pointer that points to the base (offset 0) of the region associated with the label. However, we take advantage of the polymorphism and dynamic typing of paper rules to also store the following in the same environment:

- x_{func} (function names) map to pairs $\{\overline{x_i}, e\}$, which give the list of argument names and the body for functions.
- x_{proc} (procedure names) map to pairs $\{\overline{x_i}, S\}$, which give the list of argument names and the body for procedures.
- x_{op} (operation/instruction names) map to triples $\{\overline{x_i}, e, S\}$, which give the list of argument names, the expression for the text form, and the body for operations.
- r.txt (the form for the text version of a register) maps to a value.
- The word EXTBRANCH maps to a branch state, which must be either ext or $\cdot$. This reports whether, after executing a program, it branched to the external label or not.

Since identifiers are not allowed to overlap in well-typed programs, and register identities are not strings at all, and EXTBRANCH is reserved, this usage creates no conflicts.

Note that x_{mem}, x_τ, and x_{module} do not appear in Λ as these require no translation/lookup at runtime.

Machine State. In addition to the execution environment, we also need a representation of machine state. We define two stores, one for registers and one for memory. The register store ρ maps registers r to values v. The memory store σ is more complicated: it maps pairs (x_{mem}, C) (that is, pointer literals) to pairs (v, C_l), where v is the value stored at that location and C_l is the bit width. The bit widths of memory regions are invariant, both across the region when they are declared and also over time. They are used to check the access widths appearing in fetch and store operations. Also note that new entries cannot be created in either the register store or the memory store, as real hardware does not permit such actions. The values stored in registers and memory regions are restricted by the typing rules to bitvectors (whether constants or pointers) of the appropriate width.

Notice that stepping through the declarations does *not* initialize the machine state. We want to reason about executions over ranges of possible starting machine states; so instead we provide a judgment that uses the typing environments to restricts the stores to forms consistent with the declarations. This is discussed further below.

(Expression Semantics)

$$\frac{\Lambda(x) = v}{\Lambda \vdash (\rho, \sigma, x) \Downarrow v} \qquad \frac{\Lambda \vdash (\rho, \sigma, e) \Downarrow r \qquad \Lambda(r.\text{txt}) = v}{\Lambda \vdash (\rho, \sigma, e.\text{txt}) \Downarrow v} \qquad \frac{\Lambda \vdash (\rho, \sigma, e) \Downarrow r \qquad r.\text{txt} \notin \Lambda}{\Lambda \vdash (\rho, \sigma, e.\text{txt}) \Downarrow \texttt{fail}}$$

$$\frac{\forall i, \Lambda \vdash (\rho, \sigma, e_i) \Downarrow v_i \qquad \Lambda(x_{func}) = \{\overline{x_i}, e\} \qquad \Lambda[\forall i, x_i \mapsto v_i] \vdash (\rho, \sigma, e) \Downarrow \tilde{v}}{\Lambda \vdash (\rho, \sigma, x_{func} \, (\overline{e_i})) \Downarrow \tilde{v}}$$

$$\frac{\Lambda \vdash (\rho, \sigma, e) \Downarrow v_1 \qquad \tilde{v}_2 = unop \, v_1}{\Lambda \vdash (\rho, \sigma, unop \, e) \Downarrow \tilde{v}_2} \qquad \frac{\Lambda \vdash (\rho, \sigma, e_1) \Downarrow v_1 \qquad \Lambda \vdash (\rho, \sigma, e_2) \Downarrow v_2 \qquad \tilde{v}_3 = v_1 \, binop \, v_2}{\Lambda \vdash (\rho, \sigma, e_1 \, binop \, e_2) \Downarrow \tilde{v}_3}$$

$$\frac{\Lambda \vdash (\rho, \sigma, e) \Downarrow \texttt{0b}C \qquad C = b_0 \dots b_{C_i} \dots b_n}{\Lambda \vdash (\rho, \sigma, e[C_i]) \Downarrow b_{C_i}} \qquad \frac{\Lambda \vdash (\rho, \sigma, e) \Downarrow \texttt{0b}C \qquad C = b_0 \dots b_{C_i} \dots b_{C_j} \dots b_n}{\Lambda \vdash (\rho, \sigma, e[C_i, C_j]) \Downarrow b_{C_i} \dots b_{C_j}}$$

$$\frac{\Lambda \vdash (\rho, \sigma, e) \Downarrow (x_{mem}, C)}{\Lambda \vdash (\rho, \sigma, e[\_]) \Downarrow \texttt{fail}} \qquad \frac{\Lambda \vdash (\rho, \sigma, e) \Downarrow (x_{mem}, C)}{\Lambda \vdash (\rho, \sigma, e[\_, \_]) \Downarrow \texttt{fail}}$$

$$\frac{\Lambda \vdash (\rho, \sigma, e_1) \Downarrow v_1 \qquad \Lambda[x \mapsto v_1] \vdash (\rho, \sigma, e_2) \Downarrow \tilde{v}_2}{\Lambda \vdash (\rho, \sigma, \texttt{let} \, x : \tau_{base} = e_1 \, \texttt{in} \, e_2) \Downarrow \tilde{v}_2}$$

$$\frac{\Lambda \vdash (\rho, \sigma, e) \Downarrow \texttt{true} \qquad \Lambda \vdash (\rho, \sigma, e_t) \Downarrow \tilde{v}_t}{\Lambda \vdash (\rho, \sigma, \texttt{if} \, e \, \texttt{then} \, e_t \, \texttt{else} \, \_) \Downarrow \tilde{v}_t} \qquad \frac{\Lambda \vdash (\rho, \sigma, e) \Downarrow \texttt{false} \qquad \Lambda \vdash (\rho, \sigma, e_f) \Downarrow \tilde{v}_f}{\Lambda \vdash (\rho, \sigma, \texttt{if} \, e \, \texttt{then} \, \_ \, \texttt{else} \, e_f) \Downarrow \tilde{v}_f}$$

$$\frac{\Lambda \vdash (\rho, \sigma, e) \Downarrow r \qquad \rho(r) = v}{\Lambda \vdash (\rho, \sigma, {}^*e) \Downarrow v} \qquad \frac{\Lambda \vdash (\rho, \sigma, e) \Downarrow (x_{mem}, C) \qquad \sigma(x_{mem}, C) = (v, C_l)}{\Lambda \vdash (\rho, \sigma, \texttt{fetch}(e, C_l)) \Downarrow v}$$

$$\frac{\Lambda \vdash (\rho, \sigma, e) \Downarrow (x_{mem}, C) \qquad \sigma(x_{mem}, C) = (\_, C_m) \qquad C_m \neq C_l}{\Lambda \vdash (\rho, \sigma, \texttt{fetch}(e, C_l)) \Downarrow \texttt{fail}}$$

$$\frac{\Lambda \vdash (\rho, \sigma, e) \Downarrow (x_{mem}, C) \qquad (x_{mem}, C) \notin \sigma}{\Lambda \vdash (\rho, \sigma, \texttt{fetch}(e, C_l)) \Downarrow \texttt{fail}} \qquad \frac{\Lambda \vdash (\rho, \sigma, e) \Downarrow \texttt{0b}C}{\Lambda \vdash (\rho, \sigma, \texttt{fetch}(e, C_l)) \Downarrow \texttt{fail}}$$

$$\frac{\Lambda \vdash (\rho, \sigma, e) \Downarrow C}{\Lambda \vdash (\rho, \sigma, (x_{mem}, e)) \Downarrow (x_{mem}, C)}$$

$$\frac{\Lambda(\text{EXTBRANCH}) = \texttt{ext}}{\Lambda \vdash (\rho, \sigma, \texttt{branchto}) \Downarrow \texttt{true}} \qquad \frac{\Lambda(\text{EXTBRANCH}) = \cdot}{\Lambda \vdash (\rho, \sigma, \texttt{branchto}) \Downarrow \texttt{false}}$$

$$\frac{\forall i \in (1 \dots k), \Lambda(x_i) = r_i}{\Lambda \vdash (\rho, \sigma, \{x_1, \dots, x_k\}) \Downarrow \{r_1, \dots, r_k\}} \qquad \frac{\Lambda \vdash (\rho, \sigma, e) \Downarrow \{r_1, \dots, r_C\}}{\Lambda \vdash (\rho, \sigma, \|e\|) \Downarrow C}$$

$$\frac{\Lambda \vdash (\rho, \sigma, e_1) \Downarrow r \qquad \Lambda \vdash (\rho, \sigma, e_2) \Downarrow \{r_1, \dots, r_k\} \qquad \exists i \in (1 \dots k), r_i = r}{\Lambda \vdash (\rho, \sigma, e_1 \in e_2) \Downarrow \texttt{true}}$$

$$\frac{\Lambda \vdash (\rho, \sigma, e_1) \Downarrow r \qquad \Lambda \vdash (\rho, \sigma, e_2) \Downarrow \{r_1, \dots, r_k\} \qquad \forall i \in (1 \dots k), r_i \neq r}{\Lambda \vdash (\rho, \sigma, e_1 \in e_2) \Downarrow \texttt{false}}$$

Fig. 20. Cassiopea semantics for expressions.

Expressions. We describe expressions with a large-step operational semantics, shown in Figure 20. The form of an expression semantic judgment is: $\Lambda \vdash (\rho, \sigma, e) \Downarrow v$, which means that given the environment Λ and the machine state ρ, σ, the expression e evaluates to the value v. Expressions may read the machine state, but not modify it. Expressions can fail; in addition to the explicit failure cases seen, some of the operators and built-in functions can fail. For example, as mentioned earlier, attempting bitvector arithmetic other than addition and subtraction on pointers will fail. Furthermore, division by zero fails.

Note that we currently do not statically check (in the typing rules) that the .txt form is present for every register or that it is defined for registers on which it is used. Thus we have an explicit failure rule for when no matching declaration has been seen. We also have failure rules for bad fetch operations: if the length annotation is wrong, if the pointer is not in the machine state (this covers both unaligned accesses and out of bounds accesses), or if the value used is not a pointer. Similarly, we have failure rules for when bit indexing/slicing a pointer. We do not, conversely, need explicit failure checks or rules for the bit indexes in the bit extraction/slicing constructs as they are statically checked.

Also note that we include in the semantics the obvious failure propagation rules for when subexpressions fail. We do not show these explicitly as they are not particularly interesting or informative; however, note that the && and || logical operators short-cut left to right.

Statements. Unlike expressions, statements can change machine state. Thus, the form of a machine state semantics judgment (also large step) is $\Lambda \vdash (\rho, \sigma, S) \Downarrow (\rho', \sigma', S', \xi)$, shown in Figure 21. This means that the statement S evaluates to the irreducible statement S' (which must be either skip or crash) and in the course of doing so changes the machine state from ρ, σ to ρ', σ', and produces a branching state ξ, which can be either an 8-bit bitvector, the reserved value ext, or a dot ($\cdot$). As with expressions, statements can fail. Explicit failure rules are shown for bad stores (corresponding to the cases for bad fetches) and for a failed assertions. We also similarly include, but do not show, the obvious failure propagation rules for cases where sub-statements, or expressions within statements, fail.

Declarations. The semantics for declarations have judgments of the form $\Lambda, (\rho, \sigma) \vdash decl \triangleright \Lambda'$, shown in Figure 22. This means that the given declaration updates Λ as shown. As stated above, we do not *initialize* the machine state while handling declarations; this instead allows us to work with arbitrary (or universally quantified) machine states afterwards. However, because the let-binding declaration evaluates an expression, it potentially needs *access* to a machine state. Consequently we write the rules so they accept a machine state as input, but do not update it. In the case of machine descriptions, where there is no machine state, we pass empty environments; let-binding declarations in machine descriptions are not allowed to reference machine state. In the case of the additional declarations that accompany a specification, we pass in the initial machine state; this allows values from the initial machine state to be globally bound so they can be referred to in the postcondition.

We give first the rules for a list of declarations, then the rules for the various declarations, then the rules for a list of operation definitions and a rule for a single operation definition. Note that several of the declarations do not update Λ, and nothing is placed in Λ for memory regions. For registers, only the mapping of the identifier to its underlying register r is entered; nothing for r is inserted.

Machines. As with the typing rules, the semantics rule for a whole machine description integrates the initial environment and gives a judgment of the form $\vdash machine \triangleright \Lambda'$, shown in Figure 23. We also include a comparable form that includes additional declarations, as it will be used below.

Programs. Instructions (or more precisely, Cassiopea operations) update the machine state, and we chose to represent programs as lists of instructions rather than having dummy instruction forms for skip and sequence. Consequently the form of the judgments is slightly different, and there are several of them, shown in Figure 24.

First, we can execute an individual instruction using the form $\Lambda \vdash (\rho, \sigma, inst) \rightarrow (\rho', \sigma', \xi)$, meaning that the instruction executes and updates the machine state ρ, σ to ρ', σ', producing the branching state ξ. Then, a list of instructions executes using the form $\Lambda \vdash (\rho, \sigma, insts, \xi) \rightarrow (\rho', \sigma', insts', \xi')$, which means that the list steps to a new list and updates both the machine state and the branching state. When the instruction list runs out, these reduce to a shorter form via a judgment of the form $\Lambda \vdash (\rho, \sigma, insts, \xi) \rightarrow (\rho', \sigma')$, which discards the instructions and branching state and produces an output machine state. That in turn allows us to draw conclusions of the form $machine \vdash (\rho, \sigma, program) \rightarrow (\rho', \sigma')$, which means that a machine with the initial state ρ, σ executes the program to produce the new machine state (ρ', σ').

(Statement Semantics)

$$\frac{\Lambda \vdash (\rho, \sigma, S_1) \Downarrow (\rho', \sigma', \mathsf{skip}, \cdot) \qquad \Lambda \vdash (\rho', \sigma', S_2) \Downarrow (\rho'', \sigma'', S_2', \cdot)}{\Lambda \vdash (\rho, \sigma, S_1; S_2) \Downarrow (\rho'', \sigma'', S_2', \cdot)}$$

$$\frac{\forall i, \Lambda \vdash (\rho, \sigma, e_i) \Downarrow v_i \qquad \Lambda(x_{proc}) = \{\overline{x_i}, S\} \qquad \Lambda[\forall i, x_i \mapsto v_i] \vdash (\rho, \sigma, S) \Downarrow (\rho', \sigma', S', \cdot)}{\Lambda \vdash (\rho, \sigma, x_{proc}\ (\overline{e_i})) \Downarrow (\rho', \sigma', S', \cdot)}$$

$$\frac{\Lambda \vdash (\rho, \sigma, e) \Downarrow v \qquad \Lambda[x \mapsto v] \vdash (\rho, \sigma, S) \Downarrow (\rho', \sigma', S', \cdot)}{\Lambda \vdash (\rho, \sigma, \mathsf{let}\ x : \tau_{base} = e\ \mathsf{in}\ S) \Downarrow (\rho', \sigma', S', \cdot)}$$

$$\frac{\forall i \in (C_1, C_1 + 1, \ldots, C_2),\ \Lambda[x \mapsto i] \vdash (\rho_i, \sigma_i, S) \Downarrow (\rho_{i+1}, \sigma_{i+1}, \mathsf{skip}, \cdot)}{\Lambda \vdash (\rho_{C_1}, \sigma_{C_1}, \mathsf{for}\ x \in (C_1 \ldots C_2)\ \mathsf{do}\ S) \Downarrow (\rho_{C_2+1}, \sigma_{C_2+1}, \mathsf{skip}, \cdot)}$$

$$\frac{\Lambda \vdash (\rho, \sigma, e) \Downarrow \mathsf{true} \qquad \Lambda \vdash (\rho, \sigma, S) \Downarrow (\rho_t, \sigma_t, S_t, \cdot)}{\Lambda \vdash (\rho, \sigma, \mathsf{if}\ e\ \mathsf{then}\ S\ \mathsf{else}\ \_) \Downarrow (\rho_t, \sigma_t, S_t, \cdot)}$$

$$\frac{\Lambda \vdash (\rho, \sigma, e) \Downarrow \mathsf{false} \qquad \Lambda \vdash (\rho, \sigma, S) \Downarrow (\rho_f, \sigma_f, S_f, \cdot)}{\Lambda \vdash (\rho, \sigma, \mathsf{if}\ e\ \mathsf{then}\ \_\ \mathsf{else}\ S) \Downarrow (\rho_f, \sigma_f, S_f, \cdot)}$$

$$\frac{\Lambda \vdash (\rho, \sigma, e_1) \Downarrow r \qquad r \in \rho \qquad \Lambda \vdash (\rho, \sigma, e_2) \Downarrow v \qquad \rho' = \rho[r \mapsto v]}{\Lambda \vdash (\rho, \sigma, e_1 := e_2) \Downarrow (\rho', \sigma, \mathsf{skip}, \cdot)}$$

$$\frac{\Lambda \vdash (\rho, \sigma, e_1) \Downarrow (x_{mem}, C) \qquad \sigma(x_{mem}, C) = (\_, C_l) \qquad \Lambda \vdash (\rho, \sigma, e_2) \Downarrow v \qquad \sigma' = \sigma[(x_{mem}, C) \mapsto (v, C_l)]}{\Lambda \vdash (\rho, \sigma, \mathsf{store}(e_1, C_l) \leftarrow e_2) \Downarrow (\rho, \sigma', \mathsf{skip}, \cdot)}$$

$$\frac{\Lambda \vdash (\rho, \sigma, e_1) \Downarrow (x_{mem}, C) \qquad \sigma(x_{mem}, C) = (\_, C_m) \qquad C_m \neq C_l}{\Lambda \vdash (\rho, \sigma, \mathsf{store}(e_1, C_l) \leftarrow e_2) \Downarrow (\rho, \sigma, \mathsf{crash}, \cdot)}$$

$$\frac{\Lambda \vdash (\rho, \sigma, e_1) \Downarrow (x_{mem}, C) \qquad (x_{mem}, C) \notin \sigma}{\Lambda \vdash (\rho, \sigma, \mathsf{store}(e_1, C_l) \leftarrow e_2) \Downarrow (\rho, \sigma, \mathsf{crash}, \cdot)} \qquad \frac{\Lambda \vdash (\rho, \sigma, e_1) \Downarrow \mathsf{0b}C}{\Lambda \vdash (\rho, \sigma, \mathsf{store}(e_1, C_l) \leftarrow e_2) \Downarrow (\rho, \sigma, \mathsf{crash}, \cdot)}$$

$$\frac{\Lambda \vdash (\rho, \sigma, e) \Downarrow \mathsf{true}}{\Lambda \vdash (\rho, \sigma, \mathsf{assert}(e)) \Downarrow (\rho, \sigma, \mathsf{skip}, \cdot)} \qquad \frac{\Lambda \vdash (\rho, \sigma, e) \Downarrow \mathsf{false}}{\Lambda \vdash (\rho, \sigma, \mathsf{assert}(e)) \Downarrow (\rho, \sigma, \mathsf{crash}, \cdot)}$$

$$\frac{\Lambda \vdash (\rho, \sigma, e) \Downarrow 0}{\Lambda \vdash (\rho, \sigma, \mathsf{BRANCH}(e)) \Downarrow (\rho, \sigma, \mathsf{skip}, \cdot)} \qquad \frac{\Lambda \vdash (\rho, \sigma, e) \Downarrow \mathsf{0b}C \qquad \mathsf{0x00} < \mathsf{0b}C < \mathsf{0xff}}{\Lambda \vdash (\rho, \sigma, \mathsf{BRANCH}(e)) \Downarrow (\rho, \sigma, \mathsf{skip}, \mathsf{0b}C)}$$

$$\frac{\Lambda \vdash (\rho, \sigma, e) \Downarrow \mathsf{0xff}}{\Lambda \vdash (\rho, \sigma, \mathsf{BRANCH}(e)) \Downarrow (\rho, \sigma, \mathsf{skip}, \mathsf{ext})}$$

Fig. 21. Cassiopea semantics for statements.

(Declaration Semantics)

$$\overline{\Lambda, (\rho, \sigma) \vdash \epsilon \rhd \Lambda} \qquad \frac{\Lambda, (\rho, \sigma) \vdash decl \rhd \Lambda' \qquad \Lambda', (\rho, \sigma) \vdash decls \rhd \Lambda''}{\Lambda, (\rho, \sigma) \vdash decl; decls \rhd \Lambda''}$$

$$\overline{\Lambda, (\rho, \sigma) \vdash \text{type } x_\tau = \tau_{base} \rhd \Lambda} \qquad \frac{\Lambda \vdash (\rho, \sigma, e) \Downarrow v}{\Lambda, (\rho, \sigma) \vdash \text{let } x : \tau_{base} = e \rhd \Lambda[x \mapsto v]}$$

$$\frac{\Lambda \vdash (\rho, \sigma, x) \Downarrow r \qquad \Lambda \vdash (\rho, \sigma, e) \Downarrow v}{\Lambda, (\rho, \sigma) \vdash \text{let } x.\text{txt} = e \rhd \Lambda[r.\text{txt} \mapsto v]}$$

$$\frac{\Lambda' = \Lambda[x_{func} \mapsto \{\overline{x_i}, e\}]}{\Lambda, (\rho, \sigma) \vdash \text{def } x_{func} \ \overline{x_i : \tau_{base_i}} \to \tau_{base} = e \rhd \Lambda'} \qquad \frac{\Lambda' = \Lambda[x_{proc} \mapsto \{\overline{x_i}, S\}]}{\Lambda, (\rho, \sigma) \vdash \text{proc } x_{proc} \ \overline{x_i : \tau_{base_i}} \to () = S \rhd \Lambda'}$$

$$\frac{\Lambda' = \Lambda[x \mapsto r] \qquad r \text{ fresh}}{\Lambda, (\rho, \sigma) \vdash \text{letstate } x : \tau_{reg} \rhd \Lambda'} \qquad \frac{\Lambda' = \Lambda[x \mapsto r] \qquad r \text{ fresh}}{\Lambda, (\rho, \sigma) \vdash \text{letstate control } x : \tau_{reg} \rhd \Lambda'}$$

$$\frac{\Lambda' = \Lambda[x \mapsto r] \qquad r \text{ fresh}}{\Lambda, (\rho, \sigma) \vdash \text{letstate control dontgate } x : \tau_{reg} \rhd \Lambda'}$$

$$\overline{\Lambda, (\rho, \sigma) \vdash \text{letstate } x_{mem} : \tau_{mem} \rhd \Lambda} \qquad \frac{\Lambda' = \Lambda[x \mapsto (x_{mem}, 0)]}{\Lambda, (\rho, \sigma) \vdash \text{letstate } x_{mem} : \tau_{mem} \text{ with } x \rhd \Lambda'}$$

$$\overline{\Lambda, (\rho, \sigma) \vdash \epsilon \rhd \Lambda} \qquad \frac{\Lambda, (\rho, \sigma) \vdash defop \rhd \Lambda' \qquad \Lambda', (\rho, \sigma) \vdash defops \rhd \Lambda''}{\Lambda, (\rho, \sigma) \vdash defop; defops \rhd \Lambda'}$$

$$\frac{\Lambda' = \Lambda[x_{op} \mapsto \{[\,], e, S\}]}{\Lambda, (\rho, \sigma) \vdash \text{defop } x_{op} \ \{\text{txt} = e, \text{sem} = S\} \rhd \Lambda'}$$

$$\frac{\Lambda' = \Lambda[x_{op} \mapsto \{\overline{x_i}, e, S\}]}{\Lambda, (\rho, \sigma) \vdash \text{defop } x_{op} \ \overline{x_i : \tau_{base_i}} \ \{\text{txt} = e, \text{sem} = S\} \rhd \Lambda'}$$

Fig. 22. Cassiopea semantics for declarations.

(Machine Semantics)

$$\frac{\Lambda_{builtin}, (\{\}, \{\}) \vdash decls \rhd \Lambda \qquad \Lambda \vdash defops \rhd \Lambda'}{\vdash decls; defops \rhd \Lambda'}$$

Fig. 23. Cassiopea semantics for machines.

Note that there are two versions of the judgment for instructions, one specialized for no arguments/operands. Instructions with no operands are declared as taking unit, but invoked with empty operands (not with unit) to correspond to the way assembly languages normally work.

(Program Semantics)

$$\frac{\Lambda(x_{op}) = \{[\ ], \_, S)\} \qquad \Lambda \vdash (\rho, \sigma, S) \Downarrow (\rho', \sigma', \mathrm{skip}, \xi)}{\Lambda \vdash (\rho, \sigma, x_{op}) \to (\rho', \sigma', \xi)}$$

$$\frac{\forall i, \Lambda \vdash (\rho, \sigma, e_i) \Downarrow v_i \qquad \Lambda(x_{op}) = \{\overline{x_i}, \_, S\} \qquad \Lambda' = \Lambda[\forall i, x_i \mapsto v_i] \qquad \Lambda' \vdash (\rho, \sigma, S) \Downarrow (\rho', \sigma', \mathrm{skip}, \xi)}{\Lambda \vdash (\rho, \sigma, x_{op}\ \overline{e_i}) \to (\rho', \sigma', \xi)}$$

$$\frac{\Lambda \vdash (\rho, \sigma, inst) \to (\rho', \sigma', \xi)}{\Lambda \vdash (\rho, \sigma, inst; insts, \cdot) \to (\rho', \sigma', insts, \xi)} \qquad \frac{}{\Lambda \vdash (\rho, \sigma, inst; insts, \mathrm{0x01}) \to (\rho, \sigma, insts, \cdot)}$$

$$\frac{\mathrm{0b}C > \mathrm{0x01}}{\Lambda \vdash (\rho, \sigma, inst; insts, \mathrm{0b}C) \to (\rho, \sigma, insts, \mathrm{0b}C\ \mathrm{b-}\ \mathrm{0x01})} \qquad \frac{}{\Lambda \vdash (\rho, \sigma, \_, \mathrm{ext}) \to (\rho, \sigma, \epsilon, \mathrm{ext})}$$

$$\frac{\vdash machine \rhd \Lambda \qquad \Lambda \vdash (\rho, \sigma, program, \cdot) \to^* (\rho', \sigma', \epsilon, \xi)}{machine \vdash (\rho, \sigma, program) \to (\rho', \sigma', \xi)}$$

Fig. 24. Cassiopea semantics for programs.

We include a final judgment of the form $machine \vdash (\rho, \sigma, program) \to (\rho', \sigma')$ that puts the machine on the left-hand side of the turnstile. It means that under a given machine the program maps ρ, σ to ρ', σ'. There is a limitation in the way we have formulated programs and the rules for programs, which is that there is no easy way to represent failure. (Failure in this might represent triggering an exception and stopping execution, which we do not model, or invoking "unpredictable" or "undefined" behavior in the processor and transitioning to an arbitrary unknown machine state.)

The intended behavior is that a program that fails during execution (that is, the body of one of its instructions steps to crash) enters a state where no postcondition can evaluate to true. We have decided for the moment that working this explicitly into the formalism would result in a lot of complication and obscuration without providing any useful information.

Specifications. For specifications we need three judgments, shown in Figure 25: the first two state what the reg-modify and mem-modify clauses mean, respectively (they are properties on initial and final register and memory states), and the last one says what it means for a program to satisfy a specification. Note that the reg-modify and mem-modify rules as shown are slightly misleading, because the register and pointer list provided in the input specification is implicitly augmented with all registers and pointers mentioned in the postcondition before it gets to this point.

Machine State Validity. As discussed above, we do not initialize the machine state while processing declarations. Instead we treat the starting machine state as an input (e.g., in the final judgment about programs) or quantify it universally as in the specification judgment. This requires that we have a predicate to reject machine states that do not match the machine description. The validity judgment has the form $\Delta, \Gamma, \Lambda \vdash \rho$, shown in Figure 26, and correspondingly for σ (except without Λ) and then for ρ, σ (both stores at once). This means that the given stores match the given environments.

We use this with the typing environments that come from both the machine description and the additional declarations arising from a specification. In the case of registers we need access to Λ to handle the names of registers. We do *not* use the Λ generated from the additional declarations in a specification; this avoids circularity.

(Specification Semantics)

$$\frac{\forall i, \Lambda \vdash (\rho, \sigma, x_{reg_i}) \Downarrow r_i \qquad \forall r \notin \{\overline{r_i}\}, \rho(r) = \rho'(r)}{\Lambda, \rho, \sigma, \rho', \sigma' \vdash \mathtt{reg\text{-}modify} : \overline{x_{reg_i}}}$$

$$\frac{\forall i, \Lambda \vdash (\rho, \sigma, (x_{mem_i}, e_i)) \Downarrow (x_{mem_i}, C_i)}{\forall x_{mem}, C, (x_{mem}, C) \notin \{\overline{(x_{mem_i}, C_i)}\}, \qquad \sigma((x_{mem}, C)) = \sigma'((x_{mem}, C))}{\Lambda, \rho, \sigma, \rho', \sigma' \vdash \mathtt{mem\text{-}modify} : \overline{(x_{mem_i}, e_i)}}$$

$$\frac{}{\Lambda, \rho, \sigma, \rho', \sigma' \vdash \epsilon} \qquad \frac{\Lambda, \rho, \sigma, \rho', \sigma' \vdash frame \qquad \Lambda, \rho, \sigma, \rho', \sigma' \vdash frames}{\Lambda, \rho, \sigma, \rho', \sigma' \vdash frame\ frames}$$

$$\frac{\begin{array}{c} \vdash machine; decls \rhd \Delta, \Gamma \qquad \vdash machine \rhd \Lambda \qquad \forall \rho, \sigma, (\Delta, \Gamma, \Lambda \vdash \rho, \sigma) \implies \qquad \Lambda, (\rho, \sigma) \vdash decls \rhd \Lambda' \implies \\ \Lambda' \vdash (\rho, \sigma, pre) \Downarrow \mathtt{true} \implies \qquad \forall \rho', \sigma', (\Lambda' \vdash (\rho, \sigma, program, \cdot) \to^* (\rho', \sigma', \epsilon, \xi)) \implies \\ (\Lambda'[\mathtt{EXTBRANCH} \mapsto \xi] \vdash (\rho', \sigma', post) \Downarrow \mathtt{true} \ \land \ \Lambda', \rho, \sigma, \rho', \sigma' \vdash frames) \end{array}}{machine, (decls; frame; pre; post) \vdash program}$$

Fig. 25. Cassiopea semantics for specifications.

(Machine State Validity)

$$\frac{(\forall x, \ \Delta, \Gamma \vdash x : C\ \mathtt{reg} \land \Lambda(x) = r) \Leftrightarrow (\exists v, \ \rho(r) = v \land \Delta, \Gamma \vdash v : C\ \mathtt{bit})}{\Delta, \Gamma, \Lambda \vdash \rho}$$

$$\frac{\begin{array}{c} \forall x_{mem}, \ \Delta, \Gamma \vdash x_{mem} : C_1\ \mathtt{bit}\ C_2\ \mathtt{len}\ C_3\ \mathtt{ref} \Leftrightarrow \\ (\forall i \in \{0, C_1/8, \dots, (C_2 - 1) * C_1/8\}, \exists v, \ \sigma(x_{mem}, i) = (v, C_1) \land \Delta, \Gamma \vdash v : C_1\ \mathtt{bit}) \land \\ (\forall i \notin \{0, C_1/8, \dots, (C_2 - 1) * C_1/8\}, \Lambda \vdash (\rho, \sigma, (x_{mem}, i)) \Downarrow \mathtt{fail}) \end{array}}{\Delta, \Gamma \vdash \sigma}$$

$$\frac{\Delta, \Gamma, \Lambda \vdash \rho \qquad \Delta, \Gamma \vdash \sigma}{\Delta, \Gamma, \Lambda \vdash \rho, \sigma}$$

Fig. 26. Cassiopea machine state validity.

This is acceptable, because specifications are not allowed to define new registers. For memory regions we need to enumerate the valid offsets for the region (note the literal 8 that hardwires 8-bit bytes) and check the cell width.

Branching. To handle branching, we allow statements to produce a branching state, which indicates the number of instructions to skip over. Normally this is $\cdot$, which means none and has no effect; however, when an instruction body produces something else we use it to branch. A nonzero bitvector results in skipping the indicated number of instructions; the out-of-band additional value ext causes a branch to the external label. The magic number used to select the external label appears only in one of the statement rules; beyond that point we use ext explicitly.

The program rule in Figure 25 inserts the branch state produced by the program into Λ, where it provides a value for the branchto predicate found in the postcondition. This allows specifications to enforce the external branching behavior.

A.5 Alewife Overview

This section describes Alewife, our specification language for writing abstracted machine-independent specifications of low-level code.

Alewife specifications are abstractions of machine-level Cassiopea specifications; we say that Cassiopea constructs are *lifted* into Alewife and Alewife constructs are *lowered* into Cassiopea. Alewife is only for specifications, so there are no statements, no updates, and no notion of instructions or programs. We refer to the single synthesis problem in one Alewife specification as a *block*.

Notation. We use the following metavariables:

x, y, z	Program variables (binders)
r	Registers (abstract)
C	Integer constants (written in decimal)
$\mathtt{0b}C$	Bitvector constants (written in binary)
N	Symbolic integer constants
τ	Types
v	Values
e	Expressions
i, j	Rule-level integers

(Other constructions are referred to with longer names.)

As noted previously, Alewife types and expressions should be considered distinct from Cassiopea ones (even where they correspond directly). We use the same letters in the hopes that this will cause less confusion (even in the definition of the translation) than picking an entirely different set of letters for Alewife.

Identifiers and Variables. In Alewife there are five syntactic categories of identifiers: As in Cassiopea, x_{mem} name memory regions. x_{func} name functions, and x_τ are type aliases. x_{module} name Cassiopea lowering modules, which are used to instantiate the abstract elements and conduct Alewife–Cassiopea translation for synthesis. Other x are ordinary variables that range over other things, and may be presumed to not range over the above reserved categories. All variables are immutable, in the sense that their values do not change once bound.

Symbolic Constants. In Alewife symbolic constants N are permitted to occur in some places where only integer constants are allowed in the corresponding Cassiopea constructions. In particular, the bit sizes associated with types (and the lengths of memory regions, which are functionally similar) may be given as symbolic values x instead of integer constants. These must be bound to integer constants either directly in the Alewife spec, in the Cassiopea lowering, or by the Cassiopea machine description. This allows the concrete sizes of bitvectors to vary depending on the machine architecture.

Types. As in Cassiopea, Alewife types are divided syntactically into base types and others, shown in Figure 27. The chief difference from Cassiopea is that bit widths (and the lengths of memory regions) can be symbolic constants. However, an additional difference is that pointers (ptr) are distinguished from plain bitvectors (vec). This is reasonably possible in Alewife, because it need not reason about the progression of values through machine registers, only pre- and post-block machine states. Strings and unit are also absent, as they are not needed for specifications.

Values and Expressions. The values in Alewife correspond directly to the values in Cassiopea as do operators and most expressions, shown in Figure 27. Note that the width argument of fetch can be a symbolic size.

Declarations and Frames. Alewife declarations come in two forms: require and provide, shown in Figure 28. The second form declares elements in the ordinary way, while the first form declares an element that must be provided by the Cassiopea lowerings or the Cassiopea machine description. In this case, the type is given, but not the value. This functions as a form of import, and allows an Alewife file to be checked on its own separately from any particular machine description or Cassiopea lowerings. However, we do not currently define or implement

(Alewife Operators)

$$unop ::= \quad - \mid \text{b}- \mid \neg \mid \text{bnot}$$

$$binop ::= \quad = \mid \neq \mid + \mid - \mid * \mid / \mid < \mid <= \mid > \mid >=$$
$$\mid \quad \&\& \mid \mid\mid \mid \oplus$$
$$\mid \quad >> \mid >>_S \mid << \mid \text{band} \mid \text{bor} \mid \text{bxor}$$
$$\mid \quad \text{b+} \mid \text{b-} \mid \text{b*} \mid \text{b/}$$
$$\mid \quad \text{b<} \mid \text{b<=} \mid \text{b>} \mid \text{b>=}$$
$$\mid \quad \text{bs<} \mid \text{bs<=} \mid \text{bs>} \mid \text{bs>=}$$
$$\mid \quad \cup \mid \cap \mid \subseteq \mid \setminus$$

(Alewife Symbolic Constants)

$$N ::= \quad C \mid x$$

(Alewife Types)

$$\tau ::= \quad \tau_{base} \mid \tau_{mem} \mid \tau_{func}$$

$$\tau_{base} ::= \quad \text{int} \mid \text{bool} \mid x_\tau$$
$$\mid \quad N \text{ vec} \mid N \text{ ptr} \mid N \text{ reg}$$
$$\mid \quad N \text{ label} \mid \tau_{regs}$$
$$\tau_{regs} ::= \quad N \text{ reg set}$$

$$\tau_{mem} ::= \quad N_1 \text{ bit } N_2 \text{ len } N_3 \text{ ref}$$
$$\tau_{func} ::= \quad \overline{\tau_{base_i}} \rightarrow \tau_{base}$$

(Alewife Expressions)

$$e ::= \quad v \mid x$$
$$\mid \quad x_{func} (\overline{e})$$
$$\mid \quad unop\ e$$
$$\mid \quad e_1\ binop\ e_2$$
$$\mid \quad e[C] \mid e[C_1, C_2]$$
$$\mid \quad \text{let } x : \tau_{base} = e_1 \text{ in } e_2$$
$$\mid \quad \text{if } e_1 \text{ then } e_2 \text{ else } e_3$$
$$\mid \quad (x_{mem}, e)$$
$$\mid \quad {}^*e \mid \text{fetch}(e, N)$$
$$\mid \quad \text{branchto}$$
$$\mid \quad \{x_1, \ldots, x_k\}$$
$$\mid \quad \|e\| \mid e_1 \in e_2$$

(Alewife Values)

$$v ::= \quad \text{true} \mid \text{false} \mid C \mid \text{0b}C \mid (x_{mem}, C)$$
$$\mid \quad \text{fail}$$

Fig. 27. Alewife symbolic constants, types, values, operators and expressions.

such a check. Note that it is possible to require functions that implicitly depend on machine state or that depend on machine state on some machines and not others. Such functions can also depend on constants or other elements that are not visible in the Alewife specification at all. The lower-with declarations specify all lowering modules that are used to compile this Alewife specification into a Cassiopea specification. The module name x_{module} is used to look up the Cassiopea lowering module to apply. The region declarations declare memory regions, like the memory-typed letstate declarations in Cassiopea. (These are implicitly always provide, because, for memory regions, the corresponding require declaration would be entirely equivalent, requiring duplication in the Cassiopea lowering.) Note that the parameters of the region can be symbolic constants if abstraction is needed.

Frame declarations in Alewife, annotated with reg-modify and mem-modify, are exactly the same as in Cassiopea. Because Alewife files are machine-independent, the registers mentioned must be abstract and concretized via the Cassiopea lowerings.

(Alewife Declarations)		
$decls$ ::=	$\epsilon \mid decl; \, decls$	
$decl$ ::=	$\texttt{require type } x_\tau$	
$\mid$	$\texttt{require value } x : \tau_{base}$	**(Initial State Bindings)**
$\mid$	$\texttt{require func } x_{func} : \tau_{func}$	
$\mid$	$\texttt{provide type } x_\tau = \tau$	
$\mid$	$\texttt{provide value } x : \tau_{base} = e$	**(Alewife Specifications)**
$\mid$	$\texttt{provide func } x_{func} : \overline{x_i : \tau_{base_i}} \rightarrow \tau_{base} = e$	
$\mid$	$\texttt{region } x_{mem} : \tau_{mem}$	
$\mid$	$\texttt{region } x_{mem} : \tau_{mem} \texttt{ with } x$	
$\mid$	$\texttt{lower-with } x_{module}$	
$\mid$	$\texttt{reg-modify} : \overline{x_i}$	
$\mid$	$\texttt{mem-modify} : \overline{(x_{mem_i}, e_i)}$	

Additional right column bindings:

$block\text{-}lets$ ::=	$\epsilon \mid block\text{-}let; \, block\text{-}lets$
$block\text{-}let$ ::=	$\texttt{let } x : \tau_{base} = e$
pre ::=	e
$post$ ::=	e
$spec$ ::=	$decls; \, block\text{-}lets; \, pre; \, post$

Fig. 28. Alewife declarations, block-lets, and specifications.

Block-lets. While Alewife expressions include let-bindings, the scope of those let-bindings is conventional: it lasts until the end of the expression. To refer to values taken from the initial state (that is, the machine state of which the precondition must be true), we need a way to bind these values so their scope extends to the postcondition. The block-lets serve this purpose in Alewife, shown in Figure 28, much like the additional declarations seen in Cassiopea specs can. These are found within a block (because a block corresponds to a synthesis problem, it is meaningful to associate pre- and post-block machine states with it), and the scope is the entire block.

Specifications. A full specification starts with a preamble of declarations, shown in Figure 28. It also includes block-lets and the pre- and postconditions for the block. Common declarations can be shared with include.

A.6 Alewife Typing and Semantics

We do not provide (or implement) a full typechecking pass for Alewife. Instead, when we lower to Cassiopea, we allow the Cassiopea typechecker to reject invalid results, which might be caused by invalid Alewife input or by bad/mismatched Cassiopea lowering definitions. The rules provided here are for doing scans over the declarations sufficient to make the translation to Cassiopea work and no more.

Environments. We retain the Cassiopea typing environments, Δ, Γ. We add an additional environment Σ, which maps identifiers to integer constants. This is a projection of the Cassiopea execution environment Λ: it holds mappings only for variables defined as integer constants and excludes everything else. We include a separate set of rules for extracting these integer constants without doing a full Cassiopea execution. (Among other things, this avoids involving machine state or the machine state stores.)

Translation. The translation (lowering) from Alewife to Cassiopea, defined in the next section, appears cross-recursively in the rules in this section. Because Δ, Γ are Cassiopea environments, they map identifiers to Cassiopea types, not Alewife ones. This means Alewife types must be lowered on the fly to update them correctly.

Integer Constant Extraction. The integer constant extraction rules do a simple pass over Cassiopea declarations to extract the variables defined as integer constants, shown in Figure 29. These populate a substitution

(Cassiopea Integer Constant Extraction)

$$\overline{\Sigma \vdash \epsilon \rhd \Sigma} \qquad \frac{\Sigma \vdash decl \rhd \Sigma' \qquad \Sigma' \vdash decls \rhd \Sigma''}{\Sigma \vdash decl; decls \rhd \Sigma''} \qquad \frac{\Sigma \vdash decls \rhd \Sigma'}{\vdash decls; defops \rhd \Sigma'}$$

$$\overline{\Sigma \vdash \mathsf{type}\ x_\tau = \tau_{base} \rhd \Sigma} \qquad \frac{\Sigma' = \Sigma[x \mapsto C]}{\Sigma \vdash \mathsf{let}\ x : \mathsf{int} = C \rhd \Sigma'} \qquad \frac{e \neq C}{\Sigma \vdash \mathsf{let}\ x : \tau_{base} = e \rhd \Sigma}$$

$$\overline{\Sigma \vdash \mathsf{def}\ x_{func}\ \overline{x_i : \tau_{base_i}} \to \tau_{base} = e \rhd \Sigma} \qquad \overline{\Sigma \vdash \mathsf{proc}\ x_{proc}\ \overline{x_i : \tau_{base_i}} \to () = S \rhd \Sigma}$$

$$\overline{\Sigma \vdash \mathsf{letstate}\ x : \tau_{reg} \rhd \Sigma} \qquad \overline{\Sigma \vdash \mathsf{letstate}\ x_{mem} : \tau_{mem} \rhd \Sigma}$$

$$\overline{\Sigma \vdash \mathsf{letstate}\ x_{mem} : \tau_{mem}\ \mathsf{with}\ x \rhd \Sigma} \qquad \overline{\Sigma \vdash \mathsf{let}\ x.\mathsf{txt} = e \rhd \Sigma}$$

Fig. 29. Cassiopea integer constant extraction.

environment Σ that we use for lowering Alewife types containing symbolic constants. These rules are judgments of the form $\Sigma \vdash decl \rhd \Sigma'$ or $\Sigma \vdash decls \rhd \Sigma'$, plus one of the form $\vdash machine \rhd \Sigma$ for a whole machine description.

Typing. The declaration typing rules are intended to accumulate types for all the declarations in an Alewife specification. They are applied concurrently with the Cassiopea declaration rules to the Alewife specification, the Cassiopea machine description, and the Cassiopea lowering. The declaration typing rules have judgments of the form $\Delta, \Gamma, \Sigma \vdash decl \rhd \Delta', \Gamma', \Sigma'$ and $\Delta, \Gamma, \Sigma \vdash decls \rhd \Delta', \Gamma', \Sigma'$, shown in Figure 30. These mean that the declaration or declarations update the type environment (and integer constant environment) as shown. Note that there is a special-case rule for provide value for when the value is an integer constant; this enters the constant into Σ. The integer constants are in turn used when lowering the types of memory regions, which can be seen in the last two rules.

Block-lets. The rules for block-lets are effectively the same as the rules for declarations, shown in Figure 31. The ways in which block-lets are special mostly do not apply here. Note however that even though we pass through Σ (for consistency of the form of the rules) there is no rule for loading integer constants into Σ from block-lets. Integer constants used in types and defined in the Alewife specification should be defined with provide value; block-lets are intended to provide access to machine state.

Specifications. The rule for the semantics of an entire specification is large and complex. The conclusion is that a given machine, lowering module, and Alewife specification produce a final translation output Ω. The rules work by nondeterministically taking fixpoints over all the material included. *decls* is the combination of all declarations found both in the initial specification and all the included lowering modules, and *frames* is the combination of all frame information (part of the declarations in Alewife; separated in Cassiopea). Similarly, the final set of environments Γ, Δ, Σ represent fixpoints produced by processing all the declarations.

In Figure 32, the first premise expands the Alewife specification as we will need to work with the components. The next two premises generate initial environments: the Cassiopea typing environments induced by the machine description, and its integer constants, and then we require that these are included in the final environments. In the fifth and sixth premises, we then require that the result of processing the declarations from the specification appears in the final environments, and that the translation of these to Cassiopea is included in the final declarations and frame rules. Then for every lowering module requested by the specification, we require that it be provided in the input modules list and that its components appear in the final declarations and frame rules. This is followed

(Alewife Declaration Typing)

$$\frac{\Delta, \Gamma, \Sigma \vdash decl \rhd \Delta', \Gamma', \Sigma' \qquad \Delta', \Gamma', \Sigma \vdash decls \rhd \Delta'', \Gamma'', \Sigma''}{\Delta, \Gamma, \Sigma \vdash decl; decls \rhd \Delta'', \Gamma'', \Sigma''}$$

$$\frac{\Delta \vdash_{\mathsf{wf}} x_\tau}{\Delta, \Gamma, \Sigma \vdash \mathsf{require\ type}\ x_\tau \rhd \Delta, \Gamma, \Sigma} \qquad \frac{\Delta, \Gamma, \Sigma \vdash \mathcal{AC}[\![\tau_{base}]\!] = \tau \qquad \Gamma(x) = \tau}{\Delta, \Gamma, \Sigma \vdash \mathsf{require\ value}\ x : \tau_{base} \rhd \Delta, \Gamma, \Sigma}$$

$$\frac{\Delta, \Gamma, \Sigma \vdash \mathcal{AC}[\![\tau_{func}]\!] = \tau \qquad \Gamma(x_{func}) = \tau}{\Delta, \Gamma, \Sigma \vdash \mathsf{require\ func}\ x_{func} : \tau_{func} \rhd \Delta, \Gamma, \Sigma}$$

$$\frac{\Delta, \Gamma, \Sigma \vdash \mathcal{AC}[\![\tau]\!] = \tau' \qquad \Delta \vdash_{\mathsf{wf}} \tau' \qquad \Delta' = \Delta[x_\tau \to \tau']}{\Delta, \Gamma, \Sigma \vdash \mathsf{provide\ type}\ x_\tau = \tau \rhd \Delta', \Gamma, \Sigma}$$

$$\frac{\Delta, \Gamma \vdash C : \mathsf{int} \qquad \Gamma' = \Gamma[x \mapsto \mathsf{int}] \qquad \Sigma' = \Sigma[x \mapsto C]}{\Delta, \Gamma, \Sigma \vdash \mathsf{provide\ value}\ x : \mathsf{int} = C \rhd \Delta, \Gamma', \Sigma'}$$

$$\frac{\Delta, \Gamma, \Sigma \vdash \mathcal{AC}[\![\tau_{base}]\!] = \tau \qquad \Delta \vdash_{\mathsf{wf}} \tau \qquad e \neq C \qquad \Delta, \Gamma, \Sigma \vdash \mathcal{AC}[\![e]\!] = e' \qquad \Delta, \Gamma \vdash e' : \tau \qquad \Gamma' = \Gamma[x \mapsto \tau]}{\Delta, \Gamma, \Sigma \vdash \mathsf{provide\ value}\ x : \tau_{base} = e \rhd \Delta, \Gamma', \Sigma}$$

$$\frac{\begin{array}{c}\forall i,\ \Delta, \Gamma, \Sigma \vdash \mathcal{AC}[\![\tau_{base_i}]\!] = \tau_i \wedge \Delta \vdash_{\mathsf{wf}} \tau_i \qquad \Delta, \Gamma, \Sigma \vdash \mathcal{AC}[\![\tau_{base}]\!] = \tau \qquad \Delta \vdash_{\mathsf{wf}} \tau \\ \Delta, \Gamma, \Sigma \vdash \mathcal{AC}[\![e]\!] = e' \qquad \Gamma' = \Gamma[\forall i, x_i \mapsto \tau_i] \qquad \Delta, \Gamma' \vdash e' : \tau \qquad \Gamma'' = \Gamma[x_{func} \mapsto (\overline{x_i : \tau_i} \to \tau)]\end{array}}{\Delta, \Gamma, \Sigma \vdash \mathsf{provide\ func}\ x_{func} : \overline{x_i : \tau_{base_i}} \to \tau_{base} \quad = e \rhd \Delta, \Gamma'', \Sigma}$$

$$\frac{\begin{array}{c}\Delta, \Gamma, \Sigma \vdash \mathcal{AC}[\![N_1\ \mathsf{bit}\ N_2\ \mathsf{len}\ N_3\ \mathsf{ref}]\!] = C_1\ \mathsf{bit}\ C_2\ \mathsf{len}\ C_3\ \mathsf{ref} \\ \Delta \vdash_{\mathsf{wf}} C_1\ \mathsf{bit}\ C_2\ \mathsf{len}\ C_3\ \mathsf{ref} \qquad \Gamma' = \Gamma[x_{mem} \mapsto C_1\ \mathsf{bit}\ C_2\ \mathsf{len}\ C_3\ \mathsf{ref}]\end{array}}{\Delta, \Gamma, \Sigma \vdash \mathsf{region}\ x_{mem} : N_1\ \mathsf{bit}\ N_2\ \mathsf{len}\ N_3\ \mathsf{ref} \rhd \Delta, \Gamma', \Sigma}$$

$$\frac{\begin{array}{c}\Delta, \Gamma, \Sigma \vdash \mathcal{AC}[\![N_1\ \mathsf{bit}\ N_2\ \mathsf{len}\ N_3\ \mathsf{ref}]\!] = C_1\ \mathsf{bit}\ C_2\ \mathsf{len}\ C_3\ \mathsf{ref} \\ \Delta \vdash_{\mathsf{wf}} C_1\ \mathsf{bit}\ C_2\ \mathsf{len}\ C_3\ \mathsf{ref} \qquad \Delta \vdash_{\mathsf{wf}} C_3\ \mathsf{label} \\ \Delta, \Gamma, \Sigma \vdash \mathcal{AC}[\![N_3\ \mathsf{label}]\!] = C_3\ \mathsf{label} \qquad \Gamma' = \Gamma[x_{mem} \mapsto C_1\ \mathsf{bit}\ C_2\ \mathsf{len}\ C_3\ \mathsf{ref}; x \mapsto C_3\ \mathsf{label}]\end{array}}{\Delta, \Gamma, \Sigma \vdash \mathsf{region}\ x_{mem} : N_1\ \mathsf{bit}\ N_2\ \mathsf{len}\ N_3\ \mathsf{ref\ with}\ x \rhd \Delta, \Gamma', \Sigma}$$

$$\frac{}{\Delta, \Gamma, \Sigma \vdash \mathsf{lower\text{-}with}\ x_{module} \rhd \Delta, \Gamma, \Sigma}$$

$$\frac{\Delta, \Gamma, \Sigma \vdash \mathcal{AC}[\![\mathsf{reg\text{-}modify} : \overline{x_i}]\!] = \mathsf{reg\text{-}modify} : \overline{x_i} \qquad \Delta, \Gamma \vdash \mathsf{reg\text{-}modify} : \overline{x_i}}{\Delta, \Gamma, \Sigma \vdash \mathsf{reg\text{-}modify} : \overline{x_i} \rhd \Delta, \Gamma, \Sigma}$$

$$\frac{\Delta, \Gamma, \Sigma \vdash \mathcal{AC}[\![\mathsf{mem\text{-}modify} : \overline{(x_{mem_i}, e_i)}]\!] = \mathsf{mem\text{-}modify} : \overline{(x_{mem_i}, e'_i)} \qquad \Delta, \Gamma \vdash \mathsf{mem\text{-}modify} : \overline{(x_{mem_i}, e'_i)}}{\Delta, \Gamma, \Sigma \vdash \mathsf{mem\text{-}modify} : \overline{(x_{mem_i}, e_i)} \rhd \Delta, \Gamma, \Sigma}$$

Fig. 30. Alewife typing rules for declaration.

(Alewife Specification Typing)

$$\frac{}{\Delta, \Gamma, \Sigma \vdash \epsilon \triangleright \Delta, \Gamma, \Sigma} \qquad \frac{\Delta, \Gamma, \Sigma \vdash block\text{-}let \triangleright \Delta', \Gamma', \Sigma' \quad \Delta', \Gamma', \Sigma \vdash block\text{-}lets \triangleright \Delta'', \Gamma'', \Sigma''}{\Delta, \Gamma, \Sigma \vdash block\text{-}let; block\text{-}lets \triangleright \Delta'', \Gamma'', \Sigma''}$$

$$\frac{\Delta, \Gamma, \Sigma \vdash \mathcal{AC}[\![\tau_{base}]\!] = \tau \quad \Delta \vdash_{wf} \tau \quad \Delta, \Gamma, \Sigma \vdash \mathcal{AC}[\![e]\!] = e' \quad \Delta, \Gamma \vdash e' : \tau \quad \Gamma' = \Gamma[x \mapsto \tau]}{\Delta, \Gamma, \Sigma \vdash \texttt{let } x : \tau_{base} = e \triangleright \Delta, \Gamma', \Sigma}$$

$$\frac{\begin{array}{c} \Delta, \Gamma, \Sigma \vdash decls \triangleright \Delta', \Gamma', \Sigma' \quad \Delta', \Gamma', \Sigma' \vdash block\text{-}lets \triangleright \Delta'', \Gamma'', \Sigma'' \quad \Delta'', \Gamma'', \Sigma \vdash \mathcal{AC}[\![pre]\!] = pre' \\ \Delta'', \Gamma'', \Sigma \vdash \mathcal{AC}[\![post]\!] = post' \quad \Delta'', \Gamma'' \vdash pre' : \texttt{bool} \quad \Delta'', \Gamma'' \vdash post' : \texttt{bool} \end{array}}{\Delta, \Gamma, \Sigma \vdash decls; block\text{-}lets; pre; post \triangleright \Delta'', \Gamma'', \Sigma''}$$

Fig. 31. Alewife typing rules for specifications.

(Alewife Specification Semantics)

$$spec = decls_{ale}; block\text{-}lets; pre; post$$
$$\vdash machine \triangleright \Delta_0, \Gamma_0 \quad \vdash machine \triangleright \Sigma_0 \quad (\Delta_0 \subseteq \Delta) \wedge (\Gamma_0 \subseteq \Gamma) \wedge (\Sigma_0 \subseteq \Sigma)$$
$$\Delta, \Gamma, \Sigma \vdash decls_{ale} \triangleright \Delta, \Gamma, \Sigma \quad \Delta, \Gamma, \Sigma \vdash \mathcal{AC}[\![decls_{ale}]\!] \subseteq decls; frames$$
$$(\forall x_{module}, \texttt{lower-with } x_{module} \in decls_{ale} \implies$$
$$\texttt{module } x_{module} \{ decls_{lower}; frames_{lower} \} \in modules \wedge decls_{lower} \subseteq decls \wedge frames_{lower} \subseteq frames)$$
$$\frac{\Delta, \Gamma \vdash decls \triangleright \Delta, \Gamma \quad \Sigma \vdash decls \triangleright \Sigma \quad \Delta, \Gamma, \Sigma \vdash block\text{-}lets \triangleright \Delta, \Gamma, \Sigma \quad \Delta, \Gamma, \Sigma \vdash \mathcal{AC}[\![block\text{-}lets]\!] \subseteq decls}{\begin{array}{c} \Delta, \Gamma, \Sigma \vdash \mathcal{AC}[\![pre]\!] = pre' \quad \Delta, \Gamma, \Sigma \vdash \mathcal{AC}[\![post]\!] = post' \quad \Omega = decls; frames; pre'; post' \end{array}}$$
$$machine, modules, spec \triangleright \Omega$$

Fig. 32. Alewife semantics for specifications.

by two more rules to ensure that these results are represented in the final environments. Later, we include the block-let material in the final environments, include its lowered form in the final declaration list (block-lets lower to declarations), bind the lowerings of the pre- and postconditions, and define the output.

The fixpoint-based evaluation strategy for declarations is required, because the Alewife declarations rely on the Cassiopea lowering file (most notably for resolving symbolic constants), but the Cassiopea lowering file is in turn also specifically allowed to refer to objects declared by the Alewife specification, such as memory regions. In the implementation this circularity is resolved by lifting both the Cassiopea and Alewife declarations (and block-lets) into a common representation and topologically sorting them based on identifier references. (Genuinely circular references among identifiers are prohibited.) From this point, they can be handled in order in a more conventional way.

Complete Output. Note that the output includes the declarations from the Cassiopea lowering modules (each $decls_{lower}$). Apart from symbolic constants, we do not substitute the definitions of the lowering elements, as that would greatly complicate things, especially with functions; instead we include the definitions and let the translation refer to them. In fact, because of the declaration ordering issues, in the implementation the lowering declarations and translated Alewife declarations can be arbitrarily interleaved in the output.

Note furthermore that it would not be sufficient to include *only* the lowering declarations explicitly imported with require declarations, as those may refer freely to other things declared in the lowering module that the Alewife specification itself may have no cognizance of whatsoever.

Alewife – Cassiopea Type Translation

$$\Delta, \Gamma, \Sigma \vdash \mathcal{AC}[\![N]\!] = \begin{cases} C & N = C \\ \Sigma(x) & N = x \wedge x \in \Sigma \\ \bot & N = x \wedge x \notin \Sigma \end{cases} \qquad \Delta, \Gamma, \Sigma \vdash \mathcal{AC}[\![x_\tau]\!] = \begin{cases} \Delta(x_\tau) & x_\tau \in \Delta \\ \bot & x_\tau \notin \Delta \end{cases}$$

$$\mathcal{AC}[\![\texttt{int}]\!] = \texttt{int} \qquad\qquad\qquad \mathcal{AC}[\![\texttt{bool}]\!] = \texttt{bool}$$

$$\mathcal{AC}[\![N \texttt{ vec}]\!] = \mathcal{AC}[\![N]\!] \texttt{ bit} \qquad\qquad\qquad \mathcal{AC}[\![N \texttt{ ptr}]\!] = \mathcal{AC}[\![N]\!] \texttt{ bit}$$

$$\mathcal{AC}[\![N \texttt{ reg}]\!] = \mathcal{AC}[\![N]\!] \texttt{ reg} \qquad\qquad\qquad \mathcal{AC}[\![N \texttt{ reg set}]\!] = \mathcal{AC}[\![N]\!] \texttt{ reg set}$$

$$\mathcal{AC}[\![N_1 \texttt{ bit } N_2 \texttt{ len } N_3 \texttt{ ref}]\!] = \mathcal{AC}[\![N_1]\!] \texttt{ bit } \mathcal{AC}[\![N_2]\!] \texttt{ len } \mathcal{AC}[\![N_3]\!] \texttt{ ref}$$

$$\mathcal{AC}[\![\overline{\tau_{base_i}} \rightarrow \tau_{base}]\!] = \overline{\mathcal{AC}[\![\tau_{base_i}]\!]} \rightarrow \mathcal{AC}[\![\tau_{base}]\!]$$

Fig. 33. Alewife – Cassiopea type translation.

A.7 Lowering Alewife

The semantics of an Alewife specification depend on material taken from a Cassiopea mapping and machine description. This does not preclude defining a semantics for Alewife in terms of that material or even some abstracted concept of what any such Cassiopea material might be. However, doing so is complicated (as can be seen from the material in the previous section, which does not even attempt to handle expression evaluation) and not perhaps very illuminating or rewarding.

So instead, we write only enough typing rules to prepare material for writing a translation (lowering) to Cassiopea, and then apply the Cassiopea typing to the lowered material. This gives Alewife a semantics in terms of the Cassiopea semantics. The translated material goes into the Cassiopea typing environments Δ, Γ, and as discussed in the previous section, we also maintain an additional environment Σ of integer constants used for substituting symbolic constants in types.

This section defines the translation. $\mathcal{AC}[\![a]\!]$ defines the Cassiopea lowering of an Alewife element a. We make the translation polymorphic over the various kinds of element; that is, $\mathcal{AC}[\![\tau]\!]$ is the translation of a type (shown in Figure 33), $\mathcal{AC}[\![e]\!]$ is the translation of an expression, etc. Some of the translation rules rely on the environments; these are written $\Delta, \Gamma, \Sigma \vdash \mathcal{AC}[\![a]\!]$ (shown in Figure 34).

Some of the translation rules produce $\bot$. If these are reached, the translation fails; this can happen if the Alewife spec was malformed and, potentially, if the mapping module failed to declare elements that were expected of it or declared them in an incompatible or inconsistent way. The rules in the previous section exclude some of these cases, but we are not (yet) prepared to argue that they rule out all translation-time failures.

Notice that the translations for require declarations are empty, because the declarations from the mapping module are output along with the translated Alewife specification.

Alewife – Cassiopea Expression Translation

$$\Delta, \Gamma, \Sigma \vdash \mathcal{AC}[\![x]\!] = \begin{cases} x & x \in \Gamma \\ \bot & x \notin \Gamma \end{cases} \qquad \Delta, \Gamma, \Sigma \vdash \mathcal{AC}[\![x_{func}\ (\overline{e})]\!] = \begin{cases} x_{func}\ (\overline{\mathcal{AC}[\![e]\!]}) & x_{func} \in \Gamma \\ \bot & x_{func} \notin \Gamma \end{cases}$$

$$\mathcal{AC}[\![\texttt{true}]\!] = \texttt{true} \qquad\qquad \mathcal{AC}[\![\texttt{false}]\!] = \texttt{false}$$

$$\mathcal{AC}[\![C]\!] = C \qquad\qquad \mathcal{AC}[\![\texttt{0b}C]\!] = \texttt{0b}C$$

$$\mathcal{AC}[\![unop\ e]\!] = unop\ \mathcal{AC}[\![e]\!] \qquad \mathcal{AC}[\![e_1\ binop\ e_2]\!] = \mathcal{AC}[\![e_1]\!]\ binop\ \mathcal{AC}[\![e_2]\!]$$

$$\mathcal{AC}[\![e[C]]\!] = \mathcal{AC}[\![e]\!][C] \qquad \mathcal{AC}[\![e[C_1, C_2]]\!] = \mathcal{AC}[\![e]\!][C_1, C_2]$$

$$\mathcal{AC}[\![\texttt{let}\ x : \tau_{base} = e_1\ \texttt{in}\ e_2]\!] = \texttt{let}\ x : \mathcal{AC}[\![\tau_{base}]\!] = \mathcal{AC}[\![e_1]\!]\ \texttt{in}\ \mathcal{AC}[\![e_2]\!]$$

$$\mathcal{AC}[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!] = \texttt{if}\ \mathcal{AC}[\![e_1]\!]\ \texttt{then}\ \mathcal{AC}[\![e_2]\!]\ \texttt{else}\ \mathcal{AC}[\![e_3]\!]$$

$$\Delta, \Gamma, \Sigma \vdash \mathcal{AC}[\![(x_{mem}, e)]\!] = \begin{cases} (x_{mem}, \mathcal{AC}[\![e]\!]) & x_{mem} \in \Gamma \\ \bot & x_{mem} \notin \Gamma \end{cases}$$

$$\mathcal{AC}[\![\texttt{fail}]\!] = \texttt{fail}$$

$$\mathcal{AC}[\![{}^*e]\!] = {}^*\mathcal{AC}[\![e]\!] \qquad \mathcal{AC}[\![\texttt{fetch}(e, N)]\!] = \texttt{fetch}(\mathcal{AC}[\![e]\!], \mathcal{AC}[\![N]\!])$$

$$\mathcal{AC}[\![\texttt{branchto}]\!] = \texttt{branchto} \qquad \mathcal{AC}[\![\{x_1, \ldots, x_k\}]\!] = \{\mathcal{AC}[\![x_1]\!], \ldots, \mathcal{AC}[\![x_k]\!]\}$$

$$\mathcal{AC}[\![\|e\|]\!] = \|\mathcal{AC}[\![e]\!]\| \qquad \mathcal{AC}[\![e_1 \in e_2]\!] = \mathcal{AC}[\![e_1]\!] \in \mathcal{AC}[\![e_2]\!]$$

Alewife – Cassiopea Block-Lets Translation

$$\mathcal{AC}[\![\texttt{let}\ x : \tau_{base} = e]\!] = \texttt{let}\ x : \mathcal{AC}[\![\tau_{base}]\!] = \mathcal{AC}[\![e]\!]$$

Alewife – Cassiopea Declaration Translation

$$\Delta, \Gamma, \Sigma \vdash \mathcal{AC}[\![\texttt{require type}\ x_\tau]\!] = \begin{cases} \epsilon & x_\tau \in \Delta \\ \bot & x_\tau \notin \Delta \end{cases}$$

$$\Delta, \Gamma, \Sigma \vdash \mathcal{AC}[\![\texttt{require value}\ x : \tau_{base}]\!] = \begin{cases} \epsilon & x \in \Gamma \\ \bot & x \notin \Gamma \end{cases}$$

$$\Delta, \Gamma, \Sigma \vdash \mathcal{AC}[\![\texttt{require func}\ x_{func} : \tau_{func}]\!] = \begin{cases} \epsilon & x_{func} \in \Gamma \\ \bot & x_{func} \notin \Sigma \end{cases}$$

$$\mathcal{AC}[\![\texttt{provide type}\ x_\tau = \tau]\!] = \texttt{type}\ x_\tau = \mathcal{AC}[\![\tau]\!]$$

$$\mathcal{AC}[\![\texttt{provide value}\ x : \tau_{base} = e]\!] = \texttt{let}\ x : \mathcal{AC}[\![\tau_{base}]\!] = \mathcal{AC}[\![e]\!]$$

$$\mathcal{AC}[\![\texttt{provide func}\ x_{func} : \overline{x_i : \tau_{base_i}} \to \tau_{base} = e]\!] = \texttt{def}\ x_{func}\ \overline{x_i : \mathcal{AC}[\![\tau_{base_i}]\!]} \to \mathcal{AC}[\![\tau_{base}]\!]$$
$$= \mathcal{AC}[\![e]\!]$$

$$\mathcal{AC}[\![\texttt{region}\ x_{mem} : \tau_{mem}]\!] = \texttt{letstate}\ x_{mem} : \mathcal{AC}[\![\tau_{mem}]\!]$$

$$\mathcal{AC}[\![\texttt{region}\ x_{mem} : \tau_{mem}\ \texttt{with}\ x]\!] = \texttt{letstate}\ x_{mem} : \mathcal{AC}[\![\tau_{mem}]\!]\ \texttt{with}\ x$$

$$\mathcal{AC}[\![\texttt{lower-with}\ x_{module}]\!] = \epsilon$$

$$\mathcal{AC}[\![\texttt{reg-modify} : \overline{x_i}]\!] = \texttt{reg-modify} : \overline{\mathcal{AC}[\![x_i]\!]}$$

$$\mathcal{AC}[\![\texttt{mem-modify} : \overline{(x_{mem_i}, e_i)}]\!] = \texttt{mem-modify} : \overline{(\mathcal{AC}[\![x_{mem_i}]\!], \mathcal{AC}[\![e_i]\!])}$$

Fig. 34. Alewife – Cassiopea translation.