Aquarium: Cassiopea and Alewife Languages

(Prerelease Version of 20191119)

David A. Holland
Harvard University
dholland@eecs.harvard.edu

Jingmei Hu
Harvard University
jingmei_hu@g.harvard.edu

Ming Kawaguchi
Harvard University
ming@seas.harvard.edu

Eric Lu
Harvard University
ericlu01@g.harvard.edu

Stephen Chong
Harvard University
chong@seas.harvard.edu

Margo I. Seltzer
University of British Columbia
seltzer@cs.ubc.ca

Abstract

This technical report describes two of the domain specific languages used in the Aquarium kernel code synthesis project. It presents the language cores in terms of abstract syntax. Cassiopea is a machine description language for describing the semantics of processor instruction sets. Alewife is a specification language that can be used to write machine-independent specifications for assembly-level instruction blocks. An Alewife specification can be used to verify and synthesize code for any machine described in Cassiopea, given a machine-specific translation for abstractions used in the specification. This article does not include an introduction to either the Aquarium system or the use of the languages. In addition to this version of the article being a draft, the Aquarium project and the languages are work in progress. This article cannot currently be considered either final or complete.

1 Introduction

The Aquarium project's goal is to synthesize the machine-dependent parts of an operating system. This has the potential to greatly reduce the amount of work needed to port an operating system to a new machine architecture. It also potentially reduces the depth of knowledge needed to do that work – currently an OS port requires deep expert knowledge of both the OS and the new machine architecture – and with luck the overall amount of time involved as well.

This document does not discuss either code synthesis or OS porting in any detail. It is intended as a supplement to other project publications, which should in general be read first. While this document includes discussions of and rationales for language features (as well as absence of language features) these discussions assume familiarity with the surrounding context.

The two languages described in this document are Cassiopea, which is a register-transfer-list-style machine description language used for writing down the semantics of processor instruction sets, and Alewife, which is a specification or modeling language used for writing down machine-independent specifications for assembly language code blocks to be synthesized.

Cassiopea also includes material for writing machine-dependent specifications (this is the target for the Alewife translator), a concept of mapping modules used by the Alewife translator, and a simple representation of basic blocks in assembly language. These are described in this document.

This document does not, however, describe any of the other languages or file formats used in the Aquarium project or any of the Aquarium tools.

The document is structured with six parts:

- The Cassiopea language and abstract syntax (Sec. 3)
- Cassiopea types (Section 4)
- Cassiopea semantics (Section 5)
- The Alewife language and abstract syntax (Section 6)
- Alewife types and semantics (Section 7)
- Alewife translation to Cassiopea (Section 8)

2 Notation

In the abstract syntax, type judgments, and semantics judgments we use italics for metavariables (e.g. v) and also 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 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 (aka 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 $0xC$. These are desugared in the parser and not shown further in this document.

The Cassiopea and Alewife syntax should be considered 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

$$\boxed{\begin{array}{rl}
 & \textbf{(Cassiopea Types)} \\
\tau ::= & \tau_{base} \mid \tau_{mem} \mid \tau_{func} \\[1em]
\tau_{base} ::= & \text{()} \mid \text{int} \mid \text{bool} \mid \text{string} \mid x_{\tau} \mid C \text{ bit} \\
\mid & \tau_{reg} \mid \tau_{regs} \mid \tau_{label} \\
\tau_{reg} ::= & C \text{ reg} \\
\tau_{regs} ::= & C \text{ reg set} \\
\tau_{label} ::= & C \text{ label} \\[1em]
\tau_{mem} ::= & C_1 \text{ bit } C_2 \text{ len } C_3 \text{ ref} \\
\tau_{func} ::= & \overline{\tau_{base_i}} \rightarrow \tau_{base}
\end{array}}$$

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 8 is applied to allow inserting Alewife fragments into Cassiopea terms. The Alewife rules in Section 7 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 7.

We have attempted as much as possible to use standard notation in as many places as possible.

3 Cassiopea Overview

This section covers the abstract syntax for Cassiopea.

As mentioned above, Cassiopea is a register-transfer-list style language: it models instructions as non-Turing-complete procedures that update a machine state. (This gives it an executable semantics, which is covered in Section 5.)

Because of our operating domain (which is similar to a compiler, but requires access to all the dusty corners of an architecture compilers can normally ignore) 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 handle it in small chunks passed in from somewhere else. We also need to model control registers as well as general-purpose registers, and machine state like whether interrupts are enabled.

Furthermore, we need to handle assembler labels: these are like pointers, but they are not themselves pointers; they 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)
$\text{0b}C$	Bitvector constants (written in binary)
τ	Types
v	Values
e	Expressions
S	Statements
i, j	Rule-level integers

(Other constructions are referred to with longer names.)

A number of constructions are lists written out in longhand (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, thread environment updates through correctly.

Identifiers and variables. Identifiers are divided syntactically into eight categories.

x_{mem} are identifiers bound to memory regions, which are second-class; x_{label} are identifiers that are assembler labels.

x_{func} and x_{proc} are identifiers bound to functions and procedures, respectively, which are also 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 mapping 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). User functions may handle only base types. Furthermore, memory regions and functions are intended to be 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`, 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. (This is different from another possible world, where the initial declaration creates a register named x, and then the second declaration creates a variable that ranges over registers, and the identifiers x and y are different kinds of things. That is, however, not how Cassiopea works.)

$$
\begin{array}{rl}
& \textbf{(Cassiopea Values)} \\
\tilde{v} ::= & v \mid \texttt{fail} \\
v ::= & \texttt{true} \mid \texttt{false} \mid C \mid \texttt{0b}C \mid \texttt{"..."} \\
\mid & r \mid (x_{mem}, C)
\end{array}
$$

$$
\begin{array}{rl}
& \textbf{(Cassiopea Operators)} \\
unop ::= & - \mid \texttt{b-} \mid \neg \mid \texttt{bnot} \\
binop ::= & = \mid \neq \mid + \mid - \mid * \mid / \mid < \mid <= \mid > \mid >= \\
\mid & \vee \mid \wedge \mid \oplus \\
\mid & >> \mid >>_S \mid << \mid \texttt{band} \mid \texttt{bor} \mid \texttt{bxor} \\
\mid & \texttt{b+} \mid \texttt{b-} \mid \texttt{b*} \mid \texttt{b/} \\
\mid & \texttt{b<} \mid \texttt{b<=} \mid \texttt{b>} \mid \texttt{b>=} \\
\mid & \texttt{bs<} \mid \texttt{bs<=} \mid \texttt{bs>} \mid \texttt{bs>=} \\
\mid & \cup \mid \cap \mid \subseteq \mid \setminus
\end{array}
$$

$$
\begin{array}{rl}
& \textbf{(Cassiopea Expressions)} \\
e ::= & \tilde{v} \mid x \\
\mid & e.\texttt{txt} \\
\mid & x_{func}\,(\overline{e}) \\
\mid & unop\ e \\
\mid & e_1\ binop\ e_2 \\
\mid & e[C] \mid e[C_1, C_2] \\
\mid & \texttt{let}\ x : \tau_{base} = e_1\ \texttt{in}\ e_2 \\
\mid & \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \\
\mid & (x_{mem}, e) \mid x_{label} \\
\mid & {}^{*}e \mid \texttt{fetch}(e, C) \\
\mid & \{x_1, \ldots, x_k\} \\
\mid & \|e\| \mid e_1 \in e_2
\end{array}
$$

Registers, or at least some registers, have associated with them a text form, which is declared separately and is the form an assembler for the machine expects to see. This is used when our internal representation of a program is extracted to an assembly source file. It is referred to by attaching the suffix .txt to the/a register variable. Note that because on some machines some registers may not be directly addressable by the assembler (they might be subfields of some larger addressable unit, or nonaddressable internal state), not all registers necessarily have a text form. And because of the way registers are referenced, we cannot readily check statically whether a given reference with .txt is valid or not, or at least without introducing additional machinery, so if not the failure happens at 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 construction ${}^{*}e$ 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. 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. These values can of course be changed, like values in registers.

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 region name and C is the offset. Because memory regions are second-class, x_{mem} must be specifically 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 same width, so one can appear in a register or in memory. However, we enforce a restriction (not captured in the semantics rules 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 bitector values.

Addition and subtraction are allowed on pointers (this changes 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

<div style="border:1px solid">

(Cassiopea Statements)

$$S ::= \qquad S; S$$
$$| \qquad x_{proc}\,(\overline{e})$$
$$| \qquad \text{let } x : \tau_{base} = e \text{ in } S$$
$$| \qquad \text{for } x \in (C_1 \ldots C_2) \text{ do } S$$
$$| \qquad \text{if } e \text{ then } S_1 \text{ else } S_2$$
$$| \qquad e_1 := e_2$$
$$| \qquad \text{store}(e_1, C) \leftarrow e_2$$
$$| \qquad \text{assert}(e)$$
$$| \qquad \text{skip}$$
$$| \qquad \text{crash}$$

</div>

<div style="border:1px solid">

(Cassiopea Declarations)

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

$$decl ::= \qquad \text{type } x_\tau = \tau_{base}$$
$$| \qquad \text{let } x : \tau_{base} = e$$
$$| \qquad \text{let } x.\text{txt} = e$$
$$| \qquad \text{def } x_{func}\,\overline{x_i : \tau_{base_i}} \rightarrow \tau_{base} = e$$
$$| \qquad \text{proc } x_{proc}\,\overline{x_i : \tau_{base_i}} \rightarrow () = S$$
$$| \qquad \text{letstate } x : \tau_{reg}$$
$$| \qquad \text{letstate } x_{mem} : \tau_{mem}$$
$$| \qquad \text{letstate } x_{mem} : \tau_{mem} \text{ with } x_{label}$$

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

</div>

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 buy much in return.) Instead, we step to failure at runtime. This can be seen in the semantics rules.

Fetching from a pointer takes the form $\text{fetch}(e, C)$. 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, by "label" we mean an assembler label or linker symbol. These have addresses (or depending on how one looks at them, they are addresses) and those addresses are constants, but the constants 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. 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 value can be just assigned to the destination register. The type restriction on the operand is sufficient to synthesize programs that use labels correctly.

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 and so everything about them is a bit fuzzy.

Register sets are not allowed to be operands to instructions to avoid state explosions when synthesizing. This restriction

is currently not captured in the abstract syntax or typing rules.

Functions and procedures. Functions, defined with def, are pure functions whose bodies are expressions. They produce values. They can access registers and memory, and can fail, but cannot update anything. Procedures, defined with proc, are on the other hand 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 only called 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) are essentially instructions, and we refer to these interchangeably. An operation takes zero or more operands and does some transform on the machine state 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.

$$
\begin{array}{rcl}
& \textbf{(Cassiopea Machines)} & \\
\textit{machine} & ::= & \textit{decls}; \textit{defops}
\end{array}
$$

$$
\begin{array}{rcl}
& \textbf{(Cassiopea Mappings)} & \\
\textit{modules} & ::= & \epsilon \mid \textit{module}; \textit{modules} \\
\textit{module} & ::= & \texttt{module } x_{module} \, \{\textit{decls}; \textit{frame}\}
\end{array}
$$

$$
\begin{array}{rcl}
& \textbf{(Cassiopea Specifications)} & \\
\textit{frame} & ::= & \epsilon \mid \texttt{reg-modify} : \overline{x_{reg_i}} \\
& \mid & \texttt{mem-modify} : \overline{(x_{mem i}, e_i)} \\
\textit{pre} & ::= & e \\
\textit{post} & ::= & e \\
\textit{spec} & ::= & \textit{decls}; \textit{frame}; \textit{pre}; \textit{post}
\end{array}
$$

$$
\begin{array}{rcl}
& \textbf{(Cassiopea Programs)} & \\
\textit{inst} & ::= & x_{op} \mid x_{op} \, \overline{e} \\
\textit{insts} & ::= & \epsilon \mid \textit{inst}; \textit{insts} \\
\textit{program} & ::= & \textit{insts}
\end{array}
$$

There is an important distinction between "operations" and "instructions". Operations are the units in which Cassiopea thinks about machine operations, and the units in which Cassiopea generates programs and code fragments, but they do not necessarily need to be single instructions. The text output to the assembler is arbitrary and can be computed on the fly based on the operand value. On some platforms the assembler defines so-called "synthetic instructions" that are potentially multiple real instructions. This facility takes that a step further by allowing the writer of the machine description to define their own synthetics.

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

Machines, Mappings, Specs, and Programs.

A *machine* is a complete description of a machine architecture; it includes declarations (including types, constants, registers, functions and procedures) and also instructions. This is typically a standalone file, or possibly several via `include`.

A (single) mapping is a collection of declarations used to instantiate elements in Alewife translations. These are placed into a *module*, with multiple modules per file, so that the mappings associated with multiple related Alewife specifications can be kept together. The module name is selected by using the Alewife block name.

A *spec* is a precondition and postcondition, which are boolean expressions, along with optional permission to destroy additional registers. (The latter is a *frame*.) Cassiopea specs are produced by compiling Alewife specs. Note that a module can also include frame declarations; these are added to any provided in the Alewife spec. A code block is permitted to destroy any register that is either explicitly listed in the frame declarations or mentioned in the postcondition. (This restriction is currently not adequately captured in the semantics rules.)

A *program* is a sequence of instruction invocations. There are no labels bound in these because currently we only not support a single basic block at a time.

Built-in functions. Here is a partial list of the built-in functions in Cassiopea.

- `empty` (int $C \rightarrow C$ reg set) produces an empty register set of the requested bit size.
- `hex` (int$|C$ bit $\rightarrow$ string) prints numbers in hexadecimal.
- `bin` (int$|C$ bit $\rightarrow$ string) prints numbers in binary.
- `dec` (int$|C$ bit $\rightarrow$ string) prints numbers in decimal.
- `lbl` (C label $\rightarrow$ string) prints labels (it returns the label identifier as a string).
- `format` (string $\rightarrow$ string $\ldots \rightarrow$ 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` ($C_1 \rightarrow C_2$ bit $\rightarrow C_1$ bit) returns a new bitvector of size C_1 with the same value up to the ability of the new size to represent that value.
- `bv_to_uint` (C_1 bit $\rightarrow$ int) converts a bitvector to unsigned int.
- `uint_to_bv_l` (int $C_1 \rightarrow$ int $C_2 \rightarrow C_1$ bit) converts an unsigned int C_2 into a bitvector of size C_1.
- `isptr` (C bit $\rightarrow$ bool) tests at runtime if a bitvector value is a pointer or not.

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. As noted earlier we do not describe the concrete syntax 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 (to avoid violating the principle of least surprise) 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,

<div style="border: 1px solid black; padding: 10px;">

(Type Well-Formedness)

$$\overline{\Delta \vdash_{\mathsf{wf}} ()}$$

$$\overline{\Delta \vdash_{\mathsf{wf}} \mathtt{int}}$$

$$\overline{\Delta \vdash_{\mathsf{wf}} \mathtt{bool}}$$

$$\overline{\Delta \vdash_{\mathsf{wf}} \mathtt{string}}$$

$$\frac{\Delta(x) = \tau \qquad \Delta \vdash_{\mathsf{wf}} \tau}{\Delta \vdash_{\mathsf{wf}} x}$$

$$\frac{\forall i, \Delta \vdash_{\mathsf{wf}} \tau_i \qquad \Delta \vdash_{\mathsf{wf}} \tau_r}{\Delta \vdash_{\mathsf{wf}} \overline{\tau_i} \to \tau_r}$$

$$\frac{C > 0}{\Delta \vdash_{\mathsf{wf}} C \ \mathtt{bit}}$$

$$\frac{C > 0}{\Delta \vdash_{\mathsf{wf}} C \ \mathtt{reg}}$$

$$\frac{C > 0}{\Delta \vdash_{\mathsf{wf}} C \ \mathtt{label}}$$

$$\frac{C > 0}{\Delta \vdash_{\mathsf{wf}} C \ \mathtt{reg \ set}}$$

$$\frac{C_1 > 0, \ C_2 > 0, \ C_3 > 0}{\Delta \vdash_{\mathsf{wf}} C_1 \ \mathtt{bit} \ C_2 \ \mathtt{len} \ C_3 \ \mathtt{ref}}$$

</div>

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_{label}.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 a include directive, which is useful for sharing common elements.

4 Cassiopea Static Typing

This section describes the Cassiopea type system.

Environments. The type system uses two environments: Δ maps type alias names to the types they represent, and Γ maps variables to the types assigned to them. Recall from the syntax that only base types may have alias names, so alias names can be treated as base types.

Well-Formedness. Since types include alias names, we need to check that a proposed alias name is actually a type name. At the same time we insist that the widths of bitvectors be greater than zero. The judgment for this has the form $\Delta \vdash_{\mathsf{wf}} \tau$.

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 program binders (variables) to types, we say that a type is well formed when all type names are well-formed and all indices are of type int.

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 $\Delta, \Gamma \vdash e : \tau$. 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 follows. (Note that the bitvector operators are polymorphic in bit size.)

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

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

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,

(Expression Typing)

$$\frac{}{\Delta, \Gamma \vdash C : \mathsf{int}}$$

$$\frac{}{\Delta, \Gamma \vdash \mathsf{true} : \mathsf{bool}}$$

$$\frac{}{\Delta, \Gamma \vdash \mathsf{false} : \mathsf{bool}}$$

$$\frac{}{\Delta, \Gamma \vdash \text{"}\ldots\text{"} : \mathsf{string}}$$

$$\frac{C = \{0, 1\}^k}{\Delta, \Gamma \vdash \mathtt{0b}C : k \, \mathsf{bit}}$$

$$\frac{\Delta \vdash_{\mathsf{wf}} \tau}{\Delta, \Gamma \vdash \mathsf{fail} : \tau}$$

$$\frac{\Gamma(x) = \tau \qquad \Delta \vdash_{\mathsf{wf}} \tau}{\Delta, \Gamma \vdash x : \tau}$$

$$\frac{\Delta, \Gamma \vdash e : \tau_{reg}}{\Delta, \Gamma \vdash e.\mathtt{txt} : \mathsf{string}}$$

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

$$\frac{\Delta, \Gamma \vdash e : \tau_{base_1} \qquad \vDash unop : \tau_{base_1} \to \tau_{base_2}}{\Delta, \Gamma \vdash unop \ e : \tau_{base_2}}$$

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

$$\frac{\Delta, \Gamma \vdash e : C_2 \, \mathsf{bit} \qquad 0 \le C_1 < C_2}{\Delta, \Gamma \vdash e[C_1] : 1 \, \mathsf{bit}}$$

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

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

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

(More Expression Typing)

$$\frac{\Delta, \Gamma \vdash e : \mathsf{int} \qquad \Delta, \Gamma \vdash x_{mem} : \_ \, \mathsf{bit} \, \_ \, \mathsf{len} \, C \, \mathsf{ref}}{\Delta, \Gamma \vdash (x_{mem}, e) : C \, \mathsf{bit}}$$

$$\frac{\Delta, \Gamma \vdash e : \_ \, \mathsf{bit}}{\Delta, \Gamma \vdash \mathsf{fetch}(e, C) : C \, \mathsf{bit}}$$

$$\frac{\Delta, \Gamma \vdash x_{label} : C \, \mathsf{label}}{\Delta, \Gamma \vdash x_{label} : C \, \mathsf{bit}}$$

$$\frac{\Delta, \Gamma \vdash e : C \, \mathsf{reg}}{\Delta, \Gamma \vdash {}^*e : C \, \mathsf{bit}}$$

$$\frac{\forall i \in (1 \ldots k), \ \Delta, \Gamma \vdash x_i : C \, \mathsf{reg}}{\Delta, \Gamma \vdash \{x_1, \ldots, x_k\} : C \, \mathsf{reg} \, \mathsf{set}}$$

$$\frac{\Delta, \Gamma \vdash e : C \, \mathsf{reg} \, \mathsf{set}}{\Delta, \Gamma \vdash \|e\| : \mathsf{int}}$$

$$\frac{\Delta, \Gamma \vdash e_1 : C \, \mathsf{reg} \qquad \Delta, \Gamma \vdash e_2 : C \, \mathsf{reg} \, \mathsf{set}}{\Delta, \Gamma \vdash e_1 \in e_2 : \mathsf{bool}}$$

and keeping the set of possible regions fixed simplifies a number of things.

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

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. This means that the declaration (or list) is well typed and produces the new environment on the right. One can think of the $\vdash \triangleright$ as the tail and head of an arrow with the element transforming the environments labeling the body.

We impose an additional syntactic restriction on declarations found in a machine description (as opposed to the additional declarations that may appear in a spec): they may not use the expression forms that refer to machine state (registers or memory). This is because when defining the machine there is no specific machine state to refer to; any references would need to be quantified. (And that in turn is not allowed to avoid feeding quantifiers to the solver.)

Machines. A machine is some declarations followed by some defops, so the typing rule is just sequencing, but there's a wrinkle: the initial environment for the machine is not an input. $\Delta_{builtin}$ is the (fixed) environment describing the

<div style="text-align: center;">

(Statement Typing)

</div>

$$\frac{\Delta, \Gamma \vdash S_1 \qquad \Delta, \Gamma \vdash S_2}{\Delta, \Gamma \vdash S_1; \ S_2}$$

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

$$\frac{\Delta, \Gamma \vdash e : \tau_{base} \qquad x \notin \Delta, \Gamma \qquad \Delta, \Gamma[x \mapsto \tau_{base}] \vdash S}{\Delta, \Gamma \vdash \text{let } x : \tau_{base} = e \text{ in } S}$$

$$\frac{\Delta, \Gamma[x \mapsto \text{int}] \vdash S}{\Delta, \Gamma \vdash \text{for } x \in (C_1 \dots C_2) \text{ do } S}$$

$$\frac{\Delta, \Gamma \vdash e : \text{bool} \qquad \Delta, \Gamma \vdash S_1 \qquad \Delta, \Gamma \vdash S_2}{\Delta, \Gamma \vdash \text{ if } e \text{ then } S_1 \text{ else } S_2}$$

$$\frac{\Delta, \Gamma \vdash e_1 : C \text{ reg} \qquad \Delta, \Gamma \vdash e_2 : C \text{ bit}}{\Delta, \Gamma \vdash e_1 := e_2}$$

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

$$\frac{\Delta, \Gamma \vdash e : \text{bool}}{\Delta, \Gamma \vdash \text{assert}(e)}$$

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

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

<div style="text-align: center;">

(Declaration Typing)

</div>

$$\frac{}{\Delta, \Gamma \vdash \epsilon \rhd \Delta, \Gamma}$$

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

$$\frac{\Delta \vdash_{\text{wf}} \tau_{base} \qquad x_\tau \notin \Delta, \Gamma \qquad \Delta' = \Delta[x_\tau \to \tau_{base}]}{\Delta, \Gamma \vdash \text{type } x_\tau = \tau_{base} \rhd \Delta', \Gamma}$$

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

$$\frac{\Delta, \Gamma \vdash x : \tau_{reg} \qquad \Delta, \Gamma \vdash e : \text{string}}{\Delta, \Gamma \vdash \text{let } x.\text{txt} = e \rhd \Delta, \Gamma}$$

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

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

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

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

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

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* $\rhd \Delta, \Gamma$. 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.)

Specs. For specifications we need two helper rules, 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 $\Delta, \Gamma \vdash$ *frame*. This lets us write the real rule, which has the form *machine* $\vdash$ *spec* and means that the spec 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.

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

(More Declaration Typing)

$$\overline{\Delta, \Gamma \vdash \epsilon \triangleright \Delta, \Gamma}$$

$$\frac{\Delta, \Gamma \vdash \textit{defop} \triangleright \Delta', \Gamma' \qquad \Delta', \Gamma' \vdash \textit{defops} \triangleright \Delta'', \Gamma''}{\Delta, \Gamma \vdash \textit{defop}; \textit{defops} \triangleright \Delta'', \Gamma''}$$

$$\frac{\begin{array}{c} x_{op} \notin \Delta, \Gamma \\ \Delta, \Gamma \vdash e : \texttt{string} \qquad \Delta, \Gamma \vdash S \qquad \Gamma' = \Gamma[x_{op} \mapsto () \to ()] \end{array}}{\Delta, \Gamma \vdash \texttt{defop}\ x_{op}\ \{\texttt{txt} = e, \texttt{sem} = S\} \triangleright \Delta, \Gamma'}$$

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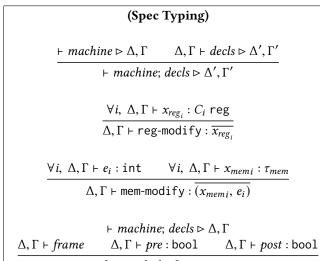tau_{base_i} \neq () \wedge \tau_{base_i} \neq \tau_{regs} \\ \Gamma' = \Gamma[\forall i, x_i \mapsto \tau_{base_i}] \qquad \Delta, \Gamma' \vdash e : \texttt{string} \\ \Delta, \Gamma' \vdash S \qquad \Gamma'' = \Gamma[x_{op} \mapsto (\overline{x_i : \tau_{base_i}} \to ())] \end{array}}{\Delta, \Gamma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

(Program Typing)

$$\frac{\Delta, \Gamma \vdash x_{op} : (() \to ())}{\Delta, \Gamma \vdash x_{op}}$$

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

$$\overline{\Delta, \Gamma \vdash \epsilon}$$

$$\frac{\Delta, \Gamma \vdash \textit{inst} \qquad \Delta, \Gamma \vdash \textit{insts}}{\Delta, \Gamma \vdash \textit{inst}; \textit{insts}}$$

$$\frac{\vdash \textit{machine} \triangleright \Delta, \Gamma \qquad \Delta, \Gamma \vdash \textit{program}}{\textit{machine} \vdash \textit{program}}$$

(Machine Typing)

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

(Spec Typing)

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

$$\frac{\forall i,\ \Delta, \Gamma \vdash x_{reg_i} : C_i\ \texttt{reg}}{\Delta, \Gamma \vdash \texttt{reg-modify} : \overline{x_{reg_i}}}$$

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

$$\frac{\begin{array}{c} \vdash \textit{machine}; \textit{decls} \triangleright \Delta, \Gamma \\ \Delta, \Gamma \vdash \textit{frame} \qquad \Delta, \Gamma \vdash \textit{pre} : \texttt{bool} \qquad \Delta, \Gamma \vdash \textit{post} : \texttt{bool} \end{array}}{\textit{machine} \vdash \textit{decls}; \textit{frame}; \textit{pre}; \textit{post}}$$

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* ⊢ *program* and means that the program is well typed relative to the machine.

Soundness. Note that even though we do not check certain things statically, the type system remains sound: we include the necessary checks and failure states in the semantics so that evaluation does not get stuck.

We have a partial but largely complete mechanized proof of soundness for an old version of Cassiopea. We intend to update and release it when and if time permits.

5 Cassiopea Semantics

This section defines the semantics of Cassiopea.

Environment. The execution environment Λ maps Cassiopea variables x to values v. However, we take advantage of/abuse the polymorphism and dynamic typing of paper rules to also store the following in the same environment:

- x_{label} (label names) map to values; specifically each label maps to a pointer that points to the base (offset 0) of the region associated with the label.
- 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.
- $r.\texttt{txt}$ (the form for the text version of a register) maps to a value.

(Expression Semantics)

$$\frac{\Lambda(x) = v}{\Lambda \vdash (\rho, \sigma, x) \Downarrow v}$$

$$\frac{\Lambda(x_{label}) = v}{\Lambda \vdash (\rho, \sigma, x_{label}) \Downarrow v}$$

$$\frac{\Lambda \vdash (\rho, \sigma, e) \Downarrow r \qquad \Lambda(r.\mathsf{txt}) = v}{\Lambda \vdash (\rho, \sigma, e.\mathsf{txt}) \Downarrow v}$$

$$\frac{\Lambda \vdash (\rho, \sigma, e) \Downarrow r \qquad r.\mathsf{txt} \notin \Lambda}{\Lambda \vdash (\rho, \sigma, e.\mathsf{txt}) \Downarrow \mathtt{fail}}$$

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

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

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

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

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

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

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

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

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

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

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

Note that x_{mem}, x_τ, and x_{module} do not appear in Λ as these require no translation 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 (this is the only way they can be 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.

Expressions. We describe expressions with a big-step operational semantics. 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 refer to 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.

<div style="text-align:center">(More Expression Semantics)</div>

$$\frac{\Lambda \vdash (\rho, \sigma, e) \Downarrow r \qquad \rho(r) = v}{\Lambda \vdash (\rho, \sigma, {}^{*}e) \Downarrow v}$$

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

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

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

$$\frac{\Lambda \vdash (\rho, \sigma, e) \Downarrow \mathtt{0bC}}{\Lambda \vdash (\rho, \sigma, \mathsf{fetch}(e, C_l)) \Downarrow \mathsf{fail}}$$

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

$$\frac{\Lambda(x_{label}) = (x_{mem}, 0)}{\Lambda \vdash (\rho, \sigma, x_{label}) \Downarrow (x_{mem}, 0)}$$

$$\frac{\forall i \in (1 \ldots k), \ \Lambda(x_i) = r_i}{\Lambda \vdash (\rho, \sigma, \{x_1, \ldots, x_k\}) \Downarrow \{r_1, \ldots, r_k\}}$$

$$\frac{\Lambda \vdash (\rho, \sigma, e) \Downarrow \{r_1, \ldots, r_C\}}{\Lambda \vdash (\rho, \sigma, \|e\|) \Downarrow C}$$

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

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

Statements. Unlike expressions, statements can change machine state. Thus, the form of a machine state semantics judgment (also large step) is $\Lambda \vdash (\rho, \sigma, S) \Downarrow (\rho', \sigma', S')$. 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'.

<div style="text-align:center">(Statement Semantics)</div>

$$\frac{\begin{array}{c}\Lambda \vdash (\rho, \sigma, S_1) \Downarrow (\rho', \sigma', \mathsf{skip}) \\ \Lambda \vdash (\rho', \sigma', S_2) \Downarrow (\rho'', \sigma'', S_2')\end{array}}{\Lambda \vdash (\rho, \sigma, S_1; S_2) \Downarrow (\rho'', \sigma'', S_2')}$$

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

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

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

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

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

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

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

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

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

$$\frac{\Lambda \vdash (\rho, \sigma, e_1) \Downarrow \mathtt{0bC}}{\Lambda \vdash (\rho, \sigma, \mathsf{store}(e_1, C_l) \leftarrow e_2) \Downarrow (\rho, \sigma, \mathsf{crash})}$$

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

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

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 $\Lambda, (\rho, \sigma) \vdash decl \triangleright \Lambda'$. 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 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 (which chain in the obvious way), then the rules for the various declarations, then the rules for a list of operation definitions (which chain in the same way) 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. (And for registers, only the mapping of the identifier to its underlying register r is entered; nothing for r is inserted.)

Machines. Like with the typing rules, the semantics rule for a whole machine description bakes in the initial environment and gives a judgment of the form $\vdash machine \triangleright \Lambda'$.

We also include a comparable form that includes additional declarations, as it will be used below.

Programs. Instructions update the machine state, and we chose to represent programs as lists of instructions (rather than having dummy instruction forms for skip and sequence) so the form of the judgments is slightly different: $\Lambda \vdash (\rho, \sigma, inst) \rightarrow (\rho', \sigma')$ means that the instruction executes and updates the machine state ρ, σ to ρ', σ'. The judgments for lists of programs has the same form.

There are two versions of the judgment for instructions because instructions with no arguments are declared as taking unit, but invoked with empty operands (not with unit) to correspond to the way assembly languages normally work.

We include a final judgment of the form $machine \vdash (\rho, \sigma, program) \rightarrow (\rho', \sigma')$ 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

---

(Declaration Semantics)

$$\frac{}{\Lambda, (\rho, \sigma) \vdash \epsilon \triangleright \Lambda}$$

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

$$\frac{}{\Lambda, (\rho, \sigma) \vdash \texttt{type } x_\tau = \tau_{base} \triangleright \Lambda}$$

$$\frac{\Lambda \vdash (\rho, \sigma, e) \Downarrow v}{\Lambda, (\rho, \sigma) \vdash \texttt{let } x : \tau_{base} = e \triangleright \Lambda[x \mapsto v]}$$

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

$$\frac{\Lambda' = \Lambda[x_{func} \mapsto \{\overline{x_i}, e\}]}{\Lambda, (\rho, \sigma) \vdash \texttt{def } x_{func} \ \overline{x_i : \tau_{base_i}} \rightarrow \tau_{base} = e \triangleright \Lambda'}$$

$$\frac{\Lambda' = \Lambda[x_{proc} \mapsto \{\overline{x_i}, S\}]}{\Lambda, (\rho, \sigma) \vdash \texttt{proc } x_{proc} \ \overline{x_i : \tau_{base_i}} \rightarrow () = S \triangleright \Lambda'}$$

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

$$\frac{}{\Lambda, (\rho, \sigma) \vdash \texttt{letstate } x_{mem} : \tau_{mem} \triangleright \Lambda}$$

$$\frac{\Lambda' = \Lambda[x_{label} \mapsto (x_{mem}, 0)]}{\Lambda, (\rho, \sigma) \vdash \texttt{letstate } x_{mem} : \tau_{mem} \texttt{ with } x_{label} \triangleright \Lambda'}$$

$$\frac{}{\Lambda, (\rho, \sigma) \vdash \epsilon \triangleright \Lambda}$$

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

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

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

(Machine Semantics)

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

<div style="border:1px solid">

(Program Semantics)

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

$$\frac{\begin{array}{c}\forall i, \Lambda \vdash (\rho, \sigma, e_i) \Downarrow v_i \qquad \Lambda(x_{op}) = \{\overline{x_i}, \_, S\} \\ \Lambda' = \Lambda[\forall i, x_i \mapsto v_i] \qquad \Lambda' \vdash (\rho, \sigma, S) \Downarrow (\rho', \sigma', \texttt{skip})\end{array}}{\Lambda \vdash (\rho, \sigma, x_{op}\ \overline{e_i}) \rightarrow (\rho', \sigma')}$$

$$\frac{}{\Lambda \vdash (\rho, \sigma, \epsilon) \rightarrow (\rho, \sigma)}$$

$$\frac{\begin{array}{c}\Lambda \vdash (\rho, \sigma, inst) \rightarrow (\rho', \sigma') \\ \Lambda \vdash (\rho', \sigma', insts) \rightarrow (\rho'', \sigma'')\end{array}}{\Lambda \vdash (\rho, \sigma, inst; insts) \rightarrow (\rho'', \sigma'')}$$

$$\frac{\vdash machine \rhd \Lambda \qquad \Lambda \vdash (\rho, \sigma, program) \rightarrow (\rho', \sigma')}{machine \vdash (\rho, \sigma, program) \rightarrow (\rho', \sigma')}$$

</div>

<div style="border:1px solid">

(Spec Semantics)

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

$$\frac{\begin{array}{c}\forall i, \Lambda \vdash (\rho, \sigma, (x_{mem_i}, e_i)) \Downarrow (x_{mem_i}, C_i) \\ \forall x_{mem}, C, (x_{mem}, C) \notin \{(x_{mem_i}, C_i)\}, \\ \sigma((x_{mem}, C)) = \sigma'((x_{mem}, C))\end{array}}{\Lambda, \sigma, \sigma' \vdash \texttt{mem-modify} : \overline{(x_{mem_i}, e_i)}}$$

$$\frac{\begin{array}{c}\vdash machine; decls \rhd \Delta, \Gamma \\ \vdash machine \rhd \Lambda \qquad \forall \rho, \sigma, (\Delta, \Gamma, \Lambda \vdash \rho, \sigma) \implies \\ \Lambda, (\rho, \sigma) \vdash decls \rhd \Lambda' \implies \\ \Lambda' \vdash (\rho, \sigma, pre) \Downarrow \texttt{true} \implies \\ \forall \rho', \sigma', (\Lambda' \vdash (\rho, \sigma, program) \rightarrow (\rho', \sigma')) \implies \\ (\Lambda' \vdash (\rho', \sigma', post) \Downarrow \texttt{true} \wedge \\ \Lambda', \rho, \rho' \vdash frame \wedge \Lambda', \sigma, \sigma' \vdash frame)\end{array}}{machine, (decls; frame; pre; post) \vdash program}$$

</div>

<div style="border:1px solid">

(Machine State Validity)

$$\frac{\begin{array}{c}(\forall x,\ \Delta, \Gamma \vdash x : C\ \texttt{reg} \wedge \Lambda(x) = r) \Leftrightarrow \\ (\exists v,\ \rho(r) = v \wedge \Delta, \Gamma \vdash v : C\ \texttt{bit})\end{array}}{\Delta, \Gamma, \Lambda \vdash \rho}$$

$$\frac{\begin{array}{c}\forall x_{mem},\ \Delta, \Gamma \vdash x_{mem} : C_1\ \texttt{bit}\ C_2\ \texttt{len}\ C_3\ \texttt{ref} \Leftrightarrow \\ (\forall i \in \{0, C_1/8, \ldots, (C_2 - 1) * C_1/8\}, \\ \exists v,\ \sigma(x_{mem}, i) = (v, C_1) \wedge \Delta, \Gamma \vdash v : C_1\ \texttt{bit})\end{array}}{\Delta, \Gamma \vdash \sigma}$$

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

</div>

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: 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 rule as shown is slightly misleading, because the register and pointer list actually written in the input file 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). In order to do this we must, however, have a predicate to reject machine states that do not match the machine description.

The validity judgment has the form $\Delta, \Gamma, \Lambda \vdash \rho$, and correspondingly for σ (except without Λ) and then for ρ, σ (both stores at once). It means that the given stores match the given environments.

We will use this with the typing environments that come from both the machine description and the additional declarations arising from a specification. (Recall that memory regions are normally defined by specifications and not by machines.)

In the case of registers we need access to Λ in order to handle the names of registers. We do *not* use the Λ generated from the additional declarations in a specification; this avoids circularity. We can get away with this 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.

(Alewife Symbolic Constants)
$N ::=$ $\qquad\qquad\qquad\qquad C \mid x$

(Alewife Types)
$\tau ::=$ $\qquad\qquad\qquad \tau_{base} \mid \tau_{mem} \mid \tau_{func}$
$\tau_{base} ::=$ $\qquad\qquad\qquad$ int $\mid$ bool $\mid x_\tau$
$\mid$ $\qquad N$ vec $\mid N$ ptr $\mid N$ reg $\mid N$ label $\mid \tau_{regs}$
$\tau_{regs} ::=$ $\qquad\qquad\qquad\qquad N$ reg set
$\tau_{mem} ::=$ $\qquad\qquad\qquad N_1$ bit N_2 len N_3 ref
$\tau_{func} ::=$ $\qquad\qquad\qquad\qquad \overline{\tau_{base_i}} \rightarrow \tau_{base}$

(Alewife Values)
$v ::=$ $\qquad$ true $\mid$ false $\mid C \mid$ 0b$C \mid (x_{mem}, C)$
$\mid$ $\qquad\qquad\qquad\qquad\qquad$ fail

6 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.

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)
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 six syntactic categories of identifiers:

Like in Cassiopea, x_{mem} name memory regions and x_{label} are assembler labels. x_{func} name functions, and x_τ are type aliases.

x_{block} are the names given to blocks of code to be synthesized, which are used to match up with the module names in Cassiopea mappings.

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 they do not change once bound.

Symbolic Constants. In some places in Alewife symbolic constants N are permitted to occur in 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 mapping, or by the Cassiopea machine description. This allows the concrete sizes of bitvectors to vary depending on the machine architecture.

Note: for compatibility with older versions of Alewife the symbolic constant word is recognized and replaced with wordsize. New specifications should not use word as a symbolic constant, as it is properly expected to be a type name.

Types. Like in Cassiopea, Alewife types are divided syntactically into base types and others. 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 before and after states)... but also currently pointless as we do not define a static type-check for Alewife.

Strings and unit are also absent, as they are not needed for specifications.

As syntactic sugar and for compatbility with an earlier version of Alewife, if no size is given wordsize is implicitly inserted. This applies to all the base types but not τ_{mem}, which worked differently then.

Values and expressions. The values in Alewife correspond directly to the values in Cassiopea. Likewise for operators and most expressions. Note that the width argument of fetch can be a symbolic size.

The notable change is the addition of bounded quantification. forall and exists are allowed to quantify over finite sets of whatever type; these are expanded to logical and/or expressions (respectively) in Cassiopea. This is useful when the set is abstract at the Alewife level and provided by the

<div style="border:1px solid">

(Alewife Operators)

$unop ::=$ $\qquad -\ |\ \mathsf{b-}\ |\ \neg\ |\ \mathsf{bnot}$

$binop ::=$ $\quad =\ |\ \neq\ |\ +\ |\ -\ |\ *\ |\ /\ |\ <\ |\ <=\ |\ >\ |\ >=$

$|$ $\qquad \vee\ |\ \wedge\ |\ \oplus$

$|$ $\qquad >>\ |\ >>_S\ |\ <<\ |\ \mathsf{band}\ |\ \mathsf{bor}\ |\ \mathsf{bxor}$

$|$ $\qquad \mathsf{b+}\ |\ \mathsf{b-}\ |\ \mathsf{b*}\ |\ \mathsf{b/}$

$|$ $\qquad \mathsf{b<}\ |\ \mathsf{b<=}\ |\ \mathsf{b>}\ |\ \mathsf{b>=}$

$|$ $\qquad \mathsf{bs<}\ |\ \mathsf{bs<=}\ |\ \mathsf{bs>}\ |\ \mathsf{bs>=}$

$|$ $\qquad \cup\ |\ \cap\ |\ \subseteq\ |\ \backslash$

(Alewife Expressions)

$e ::=$ $\qquad v\ |\ x$

$|$ $\qquad x_{func}\ (\overline{e})$

$|$ $\qquad unop\ e$

$|$ $\qquad e_1\ binop\ e_2$

$|$ $\qquad e[C]\ |\ e[C_1,\ C_2]$

$|$ $\qquad \mathsf{let}\ x : \tau_{base} = e_1\ \mathsf{in}\ e_2$

$|$ $\qquad \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$

$|$ $\qquad (x_{mem},\ e)\ |\ x_{label}$

$|$ $\qquad {}^*e\ |\ \mathsf{fetch}(e, N)$

$|$ $\qquad \{x_1, \ldots, x_k\}$

$|$ $\qquad \|e\|\ |\ e_1 \in e_2$

$|$ $\qquad \mathsf{forall}\ x \in e_1.e_2$

$|$ $\qquad \mathsf{exists}\ x \in e_2.e_2$

</div>

<div style="border:1px solid">

(Alewife Declarations)

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

$decl ::=$ $\qquad \mathsf{require\ type}\ x_\tau$

$|$ $\qquad \mathsf{require\ value}\ x : \tau_{base}$

$|$ $\qquad \mathsf{require\ func}\ x_{func} : \tau_{func}$

$|$ $\qquad \mathsf{provide\ type}\ x_\tau = \tau$

$|$ $\qquad \mathsf{provide\ value}\ x : \tau_{base} = e$

$|$ $\quad \mathsf{provide\ func}\ x_{func} : \overline{x_i : \tau_{base_i}} \rightarrow \tau_{base} = e$

$|$ $\qquad \mathsf{region}\ x_{mem} : \tau_{mem}$

$|$ $\qquad \mathsf{region}\ x_{mem} : \tau_{mem}\ \mathsf{with}\ x_{label}$

</div>

mapping definitions or the machine descriptions. (For example, the set of callee-save registers might often appear here.)

<div style="border:1px solid">

(Initial State Bindings)

$block\text{-}lets ::=$ $\qquad \epsilon\ |\ block\text{-}let;\ block\text{-}lets$

$block\text{-}let ::=$ $\qquad \mathsf{let}\ x : \tau_{base} = e$

</div>

<div style="border:1px solid">

(Alewife Blocks)

$frame ::=$ $\qquad \epsilon\ |\ \mathsf{reg\text{-}modify} : \overline{x_{reg_i}}$

$|$ $\qquad \mathsf{mem\text{-}modify} : \overline{(x_{mem_i}, e_i)}$

$pre ::=$ $\qquad e$

$post ::=$ $\qquad e$

$block ::=$ $\quad \mathsf{block}\ x_{block}\ \{block\text{-}lets;\ frame;\ pre;\ post\}$

(Alewife Specifications)

$spec ::=$ $\qquad decls;\ block$

</div>

Declarations. Alewife declarations come in two forms: `require` and `provide`. The second form declares elements in the ordinary way.

The first form declares an element that must be provided by the Cassiopea mappings (or the machine description). (So 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 mappings. However, we do not currently define or implement 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 `region` declarations declare memory regions, like the memory-typed `let state` declarations in Cassiopea. (These are implicitly always `provide`, because for memory regions the corresponding `require` declaration would be entirely equivalent but require cutpaste in the Cassiopea mapping.) Note that the parameters of the region can be symbolic constants if abstraction is needed.

Block-lets. While Alewife expressions include let-bindings, the scope of those let-bindings is conventional: it lasts until the end of the expression. In order 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, 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 before and after machine states with it) and the scope is the entire block.

$$\overline{\Sigma \vdash \epsilon \triangleright \Sigma}$$

$$\frac{\Sigma \vdash decl \triangleright \Sigma' \qquad \Sigma' \vdash decls \triangleright \Sigma''}{\Sigma \vdash decl; decls \triangleright \Sigma''}$$

$$\overline{\Sigma \vdash \mathtt{type}\ x_\tau = \tau_{base} \triangleright \Sigma}$$

$$\frac{\Sigma' = \Sigma[x \mapsto C]}{\Sigma \vdash \mathtt{let}\ x : \mathtt{int} = C \triangleright \Sigma'}$$

$$\frac{e \neq C}{\Sigma \vdash \mathtt{let}\ x : \tau_{base} = e \triangleright \Sigma}$$

$$\overline{\Sigma \vdash \mathtt{let}\ x.\mathtt{txt} = e \triangleright \Sigma}$$

$$\overline{\Sigma \vdash \mathtt{def}\ x_{func}\ \overline{x_i : \tau_{base_i}} \to \tau_{base} = e \triangleright \Sigma}$$

$$\overline{\Sigma \vdash \mathtt{proc}\ x_{proc}\ \overline{x_i : \tau_{base_i}} \to () = S \triangleright \Sigma}$$

$$\overline{\Sigma \vdash \mathtt{letstate}\ x : \tau_{reg} \triangleright \Sigma}$$

$$\overline{\Sigma \vdash \mathtt{letstate}\ x_{mem} : \tau_{mem} \triangleright \Sigma}$$

$$\overline{\Sigma \vdash \mathtt{letstate}\ x_{mem} : \tau_{mem}\ \mathtt{with}\ x_{label} \triangleright \Sigma}$$

$$\frac{\Sigma \vdash decls \triangleright \Sigma'}{\vdash decls; defops \triangleright \Sigma'}$$

Frames. Frame declarations in Alewife are exactly the same as in Cassiopea. Because Alewife files are machine-independent, the registers mentioned must be abstract and concretized via the Cassiopea mappings.

Blocks and specs. As just noted, a block is a single synthesis problem, and a full spec is that plus a preamble of declarations. (It used to be possible to place multiple related blocks together in a single Alewife specification, which would produce one Cassiopea specification for each block. This was removed because it caused unspecified problems in the implementation, as well as complications generating output because Cassiopea only allows one spec at a time. Common declarations can now be shared with `include`.)

As noted elsewhere, the block name is used to look up the Cassiopea mapping module to apply.

7 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. (Invalid results might be caused by invalid Alewife input, or by bad/mismatched Cassiopea mapping 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 in order 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. These populate a substitution environment Σ that we use for lowering Alewife types containing symbolic constants.

These rules are judgments of the form $\Sigma \vdash decl \triangleright \Sigma'$ or $\Sigma \vdash decls \triangleright \Sigma'$, plus one of the form $\vdash machine \triangleright \Sigma$ for a whole machine description.

Typing. The declaration typing rules are intended to accumulate types for all the declarations in an Alewife spec. They are applied concurrently with the Cassiopea declaration rules to the Alewife specification, the Cassiopea machine description, and the Cassiopea mapping. (How this is made to happen is described below.)

The declaration typing rules have judgments of the form $\Delta, \Gamma, \Sigma \vdash decl \triangleright \Delta', \Gamma', \Sigma'$ and $\Delta, \Gamma, \Sigma \vdash decls \triangleright \Delta', \Gamma', \Sigma'$. 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. 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

(Alewife Declaration Typing)

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

$$\frac{\Delta \vdash_{\mathsf{wf}} x_\tau}{\Delta, \Gamma, \Sigma \vdash \mathtt{require\ type}\ x_\tau \rhd \Delta, \Gamma, \Sigma}$$

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

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

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

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

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

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

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

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

(Alewife Block Typing)

$$\frac{}{\Delta, \Gamma, \Sigma \vdash \epsilon \rhd \Delta, \Gamma, \Sigma}$$

$$\frac{\begin{array}{c}\Delta, \Gamma, \Sigma \vdash block\text{-}let \rhd \Delta', \Gamma', \Sigma', \\ \Delta', \Gamma', \Sigma \vdash block\text{-}lets \rhd \Delta'', \Gamma'', \Sigma''\end{array}}{\Delta, \Gamma, \Sigma \vdash block\text{-}let; block\text{-}lets \rhd \Delta'', \Gamma'', \Sigma''}$$

$$\frac{\begin{array}{c}\Delta, \Gamma, \Sigma \vdash \mathcal{AC}[\![\tau_{base}]\!] = \tau \qquad \Delta \vdash_{\mathsf{wf}} \tau \\ \Delta, \Gamma, \Sigma \vdash \mathcal{AC}[\![e]\!] = e' \qquad \Delta, \Gamma \vdash e' : \tau \qquad \Gamma' = \Gamma[x \mapsto \tau]\end{array}}{\Delta, \Gamma, \Sigma \vdash \mathtt{let}\ x : \tau_{base} = e \rhd \Delta, \Gamma', \Sigma}$$

(Alewife Spec Semantics)

$$\frac{\begin{array}{c}module = \mathtt{module}\ x_{module}\ \{decls_{map}; frame_{map}\} \\ spec = decls_{ale}; \mathtt{block}\ x_{block}\ \{block\text{-}lets; frame_{ale}; pre; post\} \\ x_{module} = x_{block} \qquad frame = frame_{ale} \cup frame_{map} \\ spec_{new} = decls_{ale}; block\text{-}lets; frame; pre; post \\ \vdash machine \rhd \Delta_0, \Gamma_0 \\ \vdash machine \rhd \Sigma_0 \qquad (\Delta_0 \subseteq \Delta) \wedge (\Gamma_0 \subseteq \Gamma) \wedge (\Sigma_0 \subseteq \Sigma) \\ \Delta, \Gamma \vdash decls_{map} \rhd \Delta, \Gamma \qquad \Sigma \vdash decls_{map} \rhd \Sigma \\ \Delta, \Gamma, \Sigma \vdash decls_{ale} \rhd \Delta, \Gamma, \Sigma \qquad \Delta, \Gamma, \Sigma \vdash block\text{-}lets \rhd \Delta, \Gamma, \Sigma \\ \Delta, \Gamma, \Sigma \vdash \mathcal{AC}[\![spec_{new}]\!] = \Omega\end{array}}{machine, module, spec \rhd \Omega}$$

into Σ from block-lets. Integer constants used in types and defined in the Alewife spec should be defined with `provide value`; block-lets are intended to provide access to machine state.

Specs. The spec-level rule is enormous and difficult to read, so the next few paragraphs walk through it.

The conclusion is that a given machine, mapping module, and Alewife specification produce a final translation output Ω.

The first two premises expand the mapping module and Alewife specification as we'll need to work with the components. Then we require that these match (by name), combine the frame declarations, and generate a new spec updated with the additional frame declarations that we'll feed to the translation.

The next two premises generate initial environments: the Cassiopea typing environments induced by the machine description, and its integer constants.

We then nondeterministically/magically (by the power of paper rules) choose a final set of environments Γ, Δ, Σ that represents a fixpoint of collecting all the declarations. These contain at least everything in the initial environments, but also contain everything needed such that going through, respectively, the Cassiopea declarations from the mapping

file (both types and constants); the Alewife declarations; and the Alewife block-lets does not add anything additional.

The final environments can then be used to run the translation on the entire spec and get the final output.

Such an evaluation strategy for declarations is required because the Alewife declarations rely on the Cassiopea mapping file (most notably for resolving symbolic constants) but the Cassiopea mapping file is in turn also specifically allowed to refer to objects declared by the Alewife spec, 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 *complete* output file includes the declarations from the Cassiopea mapping module ($decls_{map}$) as well as the translated Alewife spec Ω. Apart from symbolic constants we do not substitute the definitions of the mapping elements, as that would make a rather large mess, 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 complete output the mapping declarations and translated Alewife declarations can be arbitrarily interleaved.

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

Alewife −Cassiopea Type Translation

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

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

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

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

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

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

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

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

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

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

---

8 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 messy (as can be seen from the material in the previous section, which does not even attempt to handle expression evaluation) and not necessarily very rewarding or illuminating.

So instead, as mentioned already, we write only enough typing rules to prepare material for writing a translation (lowering) to Cassiopea, and apply the Cassiopea typing (and, implicitly, semantics) to the lowered material. That 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, $\mathcal{AC}[\![e]\!]$ is the translation of an expression, etc. Some of the translation rules rely on the environments; these are written $\Delta, \Gamma, \Sigma \vdash \mathcal{AC}[\![a]\!]$.

Some of the translation rules produce $\bot$. If these are reached, the translation fails; this can happen if the Alewife spec was malformed and also 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.

<div style="border:1px solid">

Alewife −Cassiopea Expression Translation

$$\Delta, \Gamma, \Sigma \vdash \mathcal{AC}[\![x]\!] = \begin{cases} x & x \in \Gamma \\ \bot & x \notin \Gamma \end{cases}$$

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

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

$$\mathcal{AC}[\![C]\!] = C$$

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

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

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

$$\mathcal{AC}[\![unop\ e]\!] = unop\ \mathcal{AC}[\![e]\!]$$

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

$$\mathcal{AC}[\![e[C]]\!] = \mathcal{AC}[\![e]\!][C]$$

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

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

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

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

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

$$\mathcal{AC}[\![^{*}e]\!] = {}^{*}\mathcal{AC}[\![e]\!]$$

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

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

$$\mathcal{AC}[\![\|e\|]\!] = \|\mathcal{AC}[\![e]\!]\|$$

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

</div>

Notice that the translations for `require` declarations are empty; as mentioned in the previous section, this is because the declarations from the mapping module are output along with the translated Alewife specification.

9 Conclusion

In this technical report, we described two domain specific languages involved in the Aquarium kernel synthesis project. First, we presented a machine modeling language named Cassiopea that can be used used to describe the semantics of many different processor ISAs at the assembly language level. Then, we presented a specification language named Alewife that allows stating abstract specifications for blocks of assembly code, such that these abstract specifications can be lowered to concrete specifications and used for synthesis and verification against Cassiopea machine descriptions.

We note that this is work in progress, and does not yet present a final or complete view of either the Aquarium system or the languages presented.

References

[1] K. Rustan M. Leino and Greg Nelson. 2002. Data abstraction and information hiding. *ACM Trans. Program. Lang. Syst.* 24, 5 (2002), 491–553. https://doi.org/10.1145/570886.570888

<div style="border: 1px solid black; padding: 10px;">

Alewife −Cassiopea Block Translation

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

$$\mathcal{AC}[\![\text{reg-modify} : \overline{x_{reg_i}}]\!] = \text{reg-modify} : \overline{\mathcal{AC}[\![x_{reg_i}]\!]}$$

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

</div>

<div style="border: 1px solid black; padding: 10px;">

Alewife −Cassiopea Declaration Translation

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

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

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

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

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

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

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

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

</div>