Defining contextual refinement for capability machines

Internship Report - October 2022 - March 2023

Dorian Lesbre
Supervised by Lars Birkedal

March 31, 2023

Version 1 (06adee76)

Abstract

Contextual refinement is a useful notion to relate two open programs x and y by saying that for all contexts C, any observable behavior of $C[x]$ is also seen in $C[y]$. As such it offers a strong relation based only on the operational semantics of closed programs. I present here a definition of contextual refinement for capability machines, a type of CPU which uses special hardware checks to enforce safety constraints on memory accesses. I explain the challenges of porting refinement to low-level programs, show some results obtained by combining refinement with capability safety, and explore how refinement can be proven using a logical binary relation.

Contents

1 Introduction

Comparing different programs is quite common in software engineering. Is this new implementation better than the old one? Do they behave in the same way? Will this change break the rest of the code? Formally, these comparisons can be described by the notion of *contextual refinement* and equivalence.

Generally, contextual refinement between two programs x and y states that for all contexts C, if $C[x]$ has some observable behavior, then so does $C[y]$. Contextual equivalence of two programs is then just saying the x refines y and y refines x. These notions have many advantages. For one, they allow to reason about open programs x using only the behavior of closed programs $C[x]$. Thus, they impose no restrictions on the programming features x can use (e.g. higher order function, self-modifying code, mutual recursion...). They also benefit from vertical compositionality (transitivity) and horizontal compositionality (refinement of multiple modules implies refinement of their link), allowing refinement of large programs to be established by decomposing them into many smaller ones.

Such relations between programs offer multiple applications in computer science. For instance, they allow specifying the behavior of a program by another specification program. Not only can this be useful as programs can be easier to read then a mathematical specification; but it also enables relating complex and efficient data structures (like a set implemented as a balanced search tree) to their more naive, but easier to understand, versions (e.g. the same set represented as an unordered list).

Another application is to express *representation independence* [Mitchell, 1986]. Two equivalent modules with the same interface can internally use different data structures without the context being made aware of it. As a simple example, figure 1 presents two minimalistic counter modules which store a private integer and offer functions to read and increment it. The first stores the value as is, while the other actually stores its negation. Saying one refines the other means it can be substituted safely without changing the behavior of any external program using it. Thus, the program does not depend on the representation used.

```
let make_counter () =
  let x = ref 0 in
  let incr () = x := !x + 1 in
  let read () = !x in
  incr, read
```

```
let make_counter_neg () =
  let x = ref 0 in
  let incr () = x := !x - 1 in
  let read () = - !x in
  incr, read
```

Figure 1: Two counter modules with different internal data representation, written in OCaml

As useful as it can be, proving contextual refinement can be quite a challenge. It requires reasoning about any context that can run the given open programs. Consequently, significant efforts have been made to show refinement indirectly. One such method, and the one used in this report is to use logical relations [Timany et al., 2022, Frumin et al., 2020]. Such relations establish a logical refinement based on the shape and type of the terms, rather than a quantification over all contexts. The relations used here were defined using Iris [Jung et al., 2018, Birkedal and Bizjak, 2020], a higher-order concurrent separation logic. Iris supports impredicative invariants and user-defined ghost states; features which are quite handy in defining these relations. Furthermore, Iris' theory has been mechanized using the Coq proof assistant, and includes a proof-mode [Krebbers et al., 2017] to help establish Iris proofs interactively.

The research question

This report focuses on contextual refinement in the particular case of a capability machine. It is a special type of CPU with strong restrictions and hardware runtime checks on memory accesses. In a capability machine, programs manipulate two kinds of machine words: integers, used for arithmetic only - and special words called capabilities used to read, write and jump to memory. Each capability can be seen as a token granting access to a memory region. Such architectures permit fine-grained control over what memory each program accesses. Specifically, the model of capability machine used is Cerise [Georges et al., 2021], a formal model based on a simplified CHERI architecture [Watson et al., 2015]. Cerise has been formalized in Coq, using the Iris framework.

Since Cerise models processors directly, its programs are very low-level. They are simply sequences of integers in memory representing machine instructions. The first challenge of this work was to adapt the notion of contextual refinement, which is defined in terms of higher-level programming language features [He et al., 1986, Frumin et al., 2020, Timany et al., 2022, Song et al., 2023] such as inductive contexts and terms to these low-level programs.

Another challenge was to see how this notion of contextual refinement could be linked with a binary logical relation. Such a relation was already defined in Cerise, along with a rather useful fundamental theorem. It was also proven to hold on a few examples. Ideally, some form of contextual refinement would be implied by the current relation.

Contributions

Withing this context, my work was centered on defining a notion of contextual refinement for our model of capability machines. This required formalizing the notion of open programs, closed program and contexts and linking to be properly defined.

Cerise, the model of capability machines presented in section 2 is not new work. It is described here as context for the unfamiliar reader. For more details about it, refer to the work by Georges et al. [2021]. The definitions of components and linking in section 3 aren't completely new either. They were originally presented by Georges et al. [2022]. I did change the definitions a bit and prove the lemmas included here. The last two section are the core of this work. Section 4 presents the definition of contextual refinement and some easy results concerning it. Finally, section 5 introduces Cerise's binary logical relation (inspired by the unary relation of Georges et al. [2021]) and results linking it to my definition of contextual refinement.

Coq mechanization

Just like Cerise, this work and results were fully mechanized in Coq. The development can be found online at `https://github.com/logsem/cerise` and its documentation at `https://logsem.github.io/cerise/branch/dorian/contextual-refinement`. This report will present these results in a more mathematical fashion, and only give high-level arguments for the proof of the various results. The reader can refer to the Coq proofs if additional details are needed.

2 Presentation of capability machines

This section offers a brief presentation of capability machines and presents our model of capability machines and its operational semantics. It may be skipped by readers already familiar with the concept. This capability machine is based on Cerise by Georges et al. [2021] which is itself inspired by a simplified version of CHERI by Watson et al. [2015].

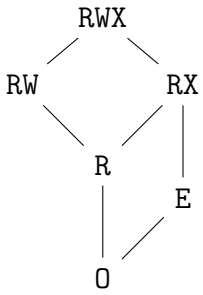

Figure 2: Lattice defining the partial order $\preccurlyeq$ on permissions

2.1 What is a capability machine?

A capability machine is a special type of CPU that has extra hardware checks for memory accesses. Where most processors, like x86, have a single machine word to represent both integers and memory addresses, capability machines have two separate ones. The first is standard integer that is used for arithmetic operations (addition, subtraction, comparison...) and the second is called a *capability* and used for memory operations (load, store and jump targets).

Capabilities contain more information than just a pointer address. They also store a memory region (range of addresses for which they are valid) and an access permission. These extra fields can be viewed as a token granting access to a set region with a set permission. A program's access to memory is limited to the capabilities it possesses. It can only read, write or jump to a section of memory if it has the relevant capability. This way, one can ensure strong properties like data confidentiality, program non-interference and modularity by restricting what capabilities a given program can access.

Capability machines fail if any memory access is attempted without the correct capabilities. This may seem problematic as it increases the number of way programs can trigger runtime faults. However, that is the point here. It is safer to fail in these cases, as failure does not break data integrity or confidentiality. These processors prefer a clean and early failure to potentially undefined memory-accesses which could break programs in much more subtle and unobvious ways. This is especially useful when working with unknown code, as one can rely on the machine failing before this code could change things outside the memory it has access to.

Definition 2.1 : (Capability)
Capabilities are represented as a tuple containing four values:

$$c \in \texttt{Cap} \;:=\; (p, b, e, a)$$

where $p \in \{\texttt{O}, \texttt{E}, \texttt{R}, \texttt{RW}, \texttt{RX}, \texttt{RWX}\}$ is a permission and b, e and a are addresses.

(p, b, e, a) points to a specific address a in a fixed memory-region $[b; e)$ with a given permission p. There are six different permissions: opaque (`O`), read-only (`R`), enter (`E`), read-write (`RW`), read-execute (`RX`), and read-write-execute (`RWX`). These permissions are ordered by a partial order given in figure 2. Any permission greater than `R` grants the right to read from $[b; e)$, any greater than `RW` grants the right to write to $[b; e)$ and any permission greater than `E` grants the right to jump to code in $[b; e)$.

In true capability machines, capabilities can be encoded in various ways to optimize space. In order to keep the model simple, Cerise abstract away from such details and simply assumes that capabilities are a tuple. Similarly, it uses unbounded integer values to avoid dealing with arithmetic overflows. This gives us the following definition of machine words:

$$\texttt{Word} := \mathbb{Z} \sqcup \texttt{Cap}$$

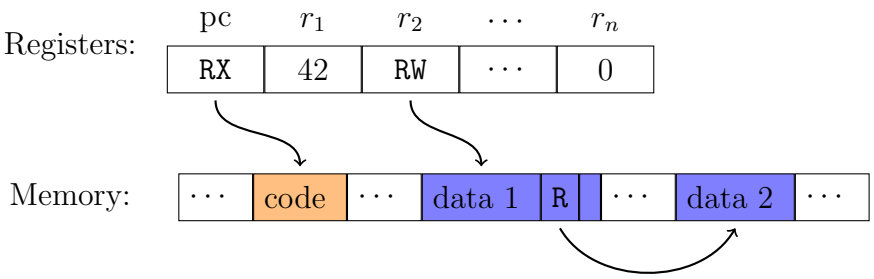

Figure 3: An example of reachable memory via capabilities

Capability manipulation is limited to avoid gaining permissions. There are only three instructions that change capabilities: "`lea` c z" allows changing a to another address $a + z$; "`subseg` c b' e'" allows changing (b, e) to (b', e') provided $[b'; e') \subseteq [b; e)$; and "`restrict` c p'" changes the permission p to p' provided $p' \preccurlyeq p$.

Of course capabilities can be copied using `move`, or read and written to memory using `load` and `store` like any other machine word. This means that a program can access any memory region transitively pointed to by the capabilities in the registers. An example is provided in figure 3. Here the program can access the `code` region and region `data 1` through registers, but also region `data 2` by reading the capability in `data 1`.

The enter permission is a bit special; it provides a form of encapsulation. Capabilities with permission `E` do not allow to read, write, or to change their addresses via `lea` or `subseg`. However, they are valid jump targets. When jumping to them, they are copied to `pc` and their permission is changed to `RX`. This in effect allows to call a some encapsulated code without allowing to read it (and any capability it may contain) or changing the entry point.

2.2 Operational semantics of Cerise

Cerise's model of a capability machine is fairly simple. It is a single core processor with no interruptions, privilege levels, virtual memory or memory alignment issues. Instructions can be encoded as a single machine word. The instruction set is also fairly small. And it assumes being able to store unbounded integers to abstract away from overflow issues. On the flip side, Cerise does model some low-level concepts accurately. It has finite memory, a fixed number of registers, and no distinction between code and data in memory.

Figure 4 regroups the definitions of our machine words, states and instructions. ρ denotes instruction parameters that can be either a register name or an immediate integer. `pc` is the program counter, the machine tries to read it each cycle to get the next instruction; so it should always contain a capability with permission at least `RX` (or the machine will fail). All other registers are general purpose and interchangeable.

The following encodings of instructions and permissions as integers are left as parameters, as their precise values matter little:

$$\texttt{encodeInstr} \in \texttt{Instr} \to \mathbb{Z} \qquad \texttt{encodePerm} \in \texttt{Perm} \to \mathbb{Z}$$
$$\texttt{decodeInstr} \in \mathbb{Z} \to \mathcal{O}(\texttt{Instr}) \qquad \texttt{decodePerm} \in \mathbb{Z} \to \mathcal{O}(\texttt{Perm})$$

The machine's state is simply a pair $\texttt{ExecMode} \times \texttt{ExecConf}$ which contains the current status of the machine (is it running, has it finished or did it encounter an error) as well as the current state of the memory and the registers. $(\texttt{Halted}, \_)$ and $(\texttt{Failed}, \_)$ are values and don't reduce any further. The $(\texttt{Running}, \_)$ state can be reduced according to the rule ExecStep given in figure 5. This rule reduces to $(\texttt{Failed}, \_)$ if `pc` does not contain a capability with executable permission pointing to a valid instruction. Otherwise, it uses the function `execInstr`

$$
\begin{array}{rcl}
a \in \texttt{Addr} & := & [0; \texttt{AddrMax}] \\
r \in \texttt{RegName} & := & \texttt{pc} \mid \texttt{r}_0 \mid \ldots \mid \texttt{r}_{\texttt{RegMax}} \\
p \in \texttt{Perm} & := & \texttt{O} \mid \texttt{E} \mid \texttt{R} \mid \texttt{RW} \mid \texttt{RX} \mid \texttt{RWX} \\
c \in \texttt{Cap} & := & \texttt{Perm} \times \texttt{Addr} \times \texttt{Addr} \times \texttt{Addr} \\
w \in \texttt{Word} & := & \mathbb{Z} \sqcup \texttt{Cap} \\
 & & \\
(\texttt{mem}, \texttt{regs}) \in \texttt{ExecConf} & := & (\texttt{Addr} \to \texttt{Word}) \times (\texttt{RegName} \to \texttt{Word}) \\
\delta \in \texttt{ExecMode} & := & \texttt{Halted} \mid \texttt{Failed} \mid \texttt{Running} \\
 & & \\
\rho \in \mathbb{Z} \sqcup \texttt{RegName} & & \\
i \in \texttt{Instr} & := & \texttt{fail} \mid \texttt{halt} \mid \texttt{jmp}\ r \mid \texttt{jnz}\ r\ r \mid \\
 & & \texttt{move}\ r\ \rho \mid \texttt{load}\ r\ r \mid \texttt{store}\ r\ \rho \mid \\
 & & \texttt{add}\ r\ \rho\ \rho \mid \texttt{sub}\ r\ \rho\ \rho \mid \texttt{lt}\ r\ \rho\ \rho \mid \\
 & & \texttt{restrict}\ r\ \rho \mid \texttt{subseg}\ r\ \rho\ \rho \mid \texttt{lea}\ r\ \rho \mid \texttt{isptr}\ r\ r \mid \\
 & & \texttt{getp}\ r\ r \mid \texttt{getb}\ r\ r \mid \texttt{gete}\ r\ r \mid \texttt{geta}\ r\ r
\end{array}
$$

Figure 4: Machine words, states and instructions

to determine the new state. This function is defined by case disjunction on the instruction in the table making up most of figure 5. Most instruction use `updPC` to increment the program counter after performing their specific update to memory and registers. The auxiliary function `getWord` returns either the register value (if its argument is a register) or the immediate value (if its argument is an integer).

3 Definition of components and contexts in Cerise

3.1 Components and programs

In order to define contextual refinement, one must first have a notion of open programs, closed programs and contexts. At the level of binary code, our executables are segments of memory containing instructions (encoded as integers) and data, coupled with initial values for the registers. For simplicity, we will define open programs only in using memory segments. As such, the initial register state will always be a part of our contexts.

An open program, which we call a memory component, is a bit more than just a section of memory. Intuitively, a component is a memory segment with holes (called imports) to be filled by values originating from other components (exports) when linking. This allows our components to be self-contained blocks of code and data. For instance, we can require that any capabilities appearing in a component only point to segments of memory within that component. Imports and exports are identified using a fixed set of symbols left as a parameter (typically, strings would be a good choice of symbols).

Definition 3.1 : (Component)
Given a set `Symbols`, we define a *component* to be record:

$$
\texttt{component} := \left\{ \begin{array}{lcl} \texttt{segment} & : & \texttt{Addr} \rightharpoonup \texttt{Word} \\ \texttt{imports} & : & \texttt{Addr} \rightharpoonup \texttt{Symbols} \\ \texttt{exports} & : & \texttt{Symbols} \rightharpoonup \texttt{Word} \end{array} \right\}
$$

Definition 3.2 : (Well-formed components)
We say that a component x is *well-formed* if it satisfies the following properties:

ExecStep

$$(\texttt{Running},\ (\texttt{mem}, \texttt{regs})) \longrightarrow \begin{cases} \texttt{execInstr mem regs}\ i & \text{if } \texttt{regs}(\texttt{pc}) = (p, b, e, a)\ \wedge\ \texttt{RX} \preccurlyeq p \wedge \\ & a \in [b; e)\ \wedge\ \texttt{decodeInstr}(\texttt{mem}(a)) = \texttt{Some}\ i \\ \texttt{Failed},\ (\texttt{mem}, \texttt{regs}) & \text{otherwise} \end{cases}$$

i	execInstr mem regs i	conditions
halt	(Halted, (mem, regs))	
fail	(Failed, (mem, regs))	
jmp r	(Running, (mem, regs[pc $\mapsto$ updPerm regs(r)]))	
jnz r_1 r_2	(Running, (mem, regs[pc $\mapsto$ updPerm regs(r)]))	regs(r_2) $\neq 0$
	updPC mem regs	regs(r_2) $= 0$
move r ρ	updPC mem regs[$r \mapsto$ getWord regs ρ]	
load r_1 r_2	updPC mem regs[$r_1 \mapsto$ mem(a)]	regs(r_2) $= (p, b, e, a) \wedge \texttt{R} \preccurlyeq p \wedge a \in [b; e)$
store r ρ	updPC mem[$a \mapsto$ getWord regs ρ] regs	regs(r) $= (p, b, e, a) \wedge \texttt{RW} \preccurlyeq p \wedge a \in [b; e)$
add r ρ_1 ρ_2	updPC mem regs[$r \mapsto z_1 + z_2$]	$z_1 = \texttt{getWord regs}\ \rho_1 \in \mathbb{Z} \wedge$ $z_1 = \texttt{getWord regs}\ \rho_2 \in \mathbb{Z}$
sub r ρ_1 ρ_2	updPC mem regs[$r \mapsto z_1 - z_2$]	
lt r ρ_1 ρ_2	updPC mem regs[$r \mapsto$ if $z_1 < z_2$ then 1 else 0]	
restrict r ρ	updPC mem regs[$r \mapsto (p', b, e, a)$]	$z = \texttt{getWord regs}\ \rho \in \mathbb{Z} \wedge$ $\texttt{decodePerm}(z) = \texttt{Some}\ p' \wedge$ $\texttt{regs}(r) = (p, b, e, a) \wedge p' \preccurlyeq p$
subseg r ρ_1 ρ_2	updPC mem regs[$r \mapsto (p, b', e', a)$]	$b' = \texttt{getWord regs}\ \rho_1 \in \mathbb{Z} \wedge$ $e' = \texttt{getWord regs}\ \rho_2 \in \mathbb{Z} \wedge$ $\texttt{regs}(r) = (p, b, e, a) \wedge p \neq \texttt{E} \wedge$ $b' < e' \wedge [b'; e') \subseteq [b; e)$
lea r ρ	updPC mem regs[$r \mapsto (p, b, e, a + z)$]	$z = \texttt{getWord regs}\ \rho_1 \in \mathbb{Z} \wedge$ $\texttt{regs}(r) = (p, b, e, a) \wedge p \neq \texttt{E} \wedge$ $a + z \in [0; \texttt{AddrMax})$
isptr r_1 r_2	updPC mem regs[$r_1 \mapsto 1$]	regs(r_2) $= (\_, \_, \_, \_)$
	updPC mem regs[$r_1 \mapsto 0$]	regs(r_2) $\in \mathbb{Z}$
getp r_1 r_2	updPC mem regs[$r_1 \mapsto$ encodePerm p]	regs(r_2) $= (p, b, e, a)$
getb r_1 r_2	updPC mem regs[$r_1 \mapsto b$]	
gete r_1 r_2	updPC mem regs[$r_1 \mapsto e$]	
geta r_1 r_2	updPC mem regs[$r_1 \mapsto a$]	
_	(Failed, (mem, regs))	otherwise

$$\begin{aligned} \texttt{getWord} &\in (\texttt{RegName} \to \texttt{Word}) \to (\mathbb{Z} \sqcup \texttt{RegName}) \to \texttt{Word} \\ \texttt{getWord regs}\ \rho &:= \begin{cases} \texttt{regs}(\rho) & \text{if } \rho \in \texttt{RegName} \\ \rho & \text{otherwise} \end{cases} \end{aligned}$$

$$\begin{aligned} \texttt{updPC} &\in (\texttt{Addr} \to \texttt{Word}) \to (\texttt{RegName} \to \texttt{Word}) \to \texttt{ExecMode} \times \texttt{ExecConf} \\ \texttt{updPC mem regs} &:= \begin{cases} \texttt{Running},\ (\texttt{mem}, \texttt{regs}[\texttt{pc} \mapsto (p, b, e, a + 1)]) & \text{if } \texttt{regs}(pc) = (p, b, e, a) \wedge \\ & a + 1 < \texttt{AddrMax} \\ \texttt{Failed},\ (\texttt{mem}, \texttt{regs}) & \text{otherwise} \end{cases} \end{aligned}$$

$$\begin{aligned} \texttt{updPerm} &\in \texttt{Word} \to \texttt{Word} \\ \texttt{updPerm}\ w &:= \begin{cases} (\texttt{RX}, b, e, a) & \text{if } w = (\texttt{E}, b, e, a) \\ w & \text{otherwise} \end{cases} \end{aligned}$$

Figure 5: Small step operational semantics of Cerise: execution of a single instruction

1. imports and exports symbols are disjoint: $\texttt{img}\ (x.\texttt{imports}) \cap \texttt{dom}\ (x.\texttt{exports}) = \emptyset$
2. import addresses are part of the component's memory: $\texttt{dom}\ (x.\texttt{imports}) \subseteq \texttt{dom}\ (x.\texttt{segment})$.

   That way, we can simply identify the component's memory region as $\texttt{dom}\ (x.\texttt{segment})$. However, such a restriction implies the segment must have placeholder values in the places imports will go.
3. capabilities that appear in $x.\texttt{segment}$ and $x.\texttt{exports}$ can only point to addresses in x's own memory region:

$$\forall\, (\_, b, e, \_) \in \texttt{img}\ x.\texttt{segment} \cup \texttt{img}\ x.\texttt{exports},\ \ [b; e) \subseteq \texttt{dom}\ x.\texttt{segment}$$

This representation of components is somewhat arbitrary, and other definitions can capture much the same objects. For instance, we considered merging the `segment` and `imports` fields into a single field $\texttt{Addr} \rightharpoonup (\texttt{Word} \sqcup \texttt{Symbols})$. This ensures the second point of well-formedness is always true and removes the need for dummy values. Unfortunately, it also makes some definitions more cumbersome, namely the first point of well-formedness and the definition of linking.

Closed programs can then be defined as well-formed components without any imports. They also need to be paired with initial register values to get a full machine configuration (Recall that `ExecConf` is just a memory map and a register map). The only difference is that a program's memory map can be partial while a configuration's one should be full. This isn't a problem when the extra memory is inaccessible anyway. Well-formedness already requires that the program's memory doesn't contain capabilities pointing to undefined memory, so if we also require this of the register values, then we know our program has no way of accessing memory outside its defined region.

Definition 3.3 : (Closed program)
A *closed program* is a pair $(x, \texttt{regs}) \in \texttt{component} \times (\texttt{RegName} \to \texttt{Word})$ such that:

1. the component x is well-formed;
2. the component x has no imports: $x.\texttt{imports} = \emptyset$;
3. register values are either 0 or exports: $\texttt{img regs} \subseteq \{0\} \cup \texttt{img}\ x.\texttt{exports}$

   Combined with well-formedness, this ensures any capabilities that appear in `regs` only grant access to x's memory region.

Programs can then be executed step by step, using the concrete semantics defined in figure 5. We can define a function

$$\texttt{machine\_run} : \mathbb{N} \to \texttt{program} \to \texttt{ExecMode}$$

which returns the state of the machine after taking a fixed number of steps starting from the configuration $(\texttt{Running}, x)$ where x is our program, and using ExecStep to reduce each step if possible. The `machine_run` function provides a simple way of expressing a reduction property, as formalized by the following lemma.

Lemma 3.1 : (`machine_run` correctness)
For all programs (`mem`, `regs`) and for all $v \in \{\texttt{Halted}, \texttt{Failed}\}$, the two following properties are equivalent:

1. $\exists\, n,\ \texttt{machine\_run}\ n\ (\texttt{mem}, \texttt{regs}) = v$
2. $\exists\, \texttt{mem}'\ \texttt{regs}',\ (\texttt{Running}, (\texttt{mem}, \texttt{regs})) \longrightarrow^* (v, (\texttt{mem}', \texttt{regs}'))$

where $\longrightarrow^*$ is the transitive reflexive closure of program reduction.

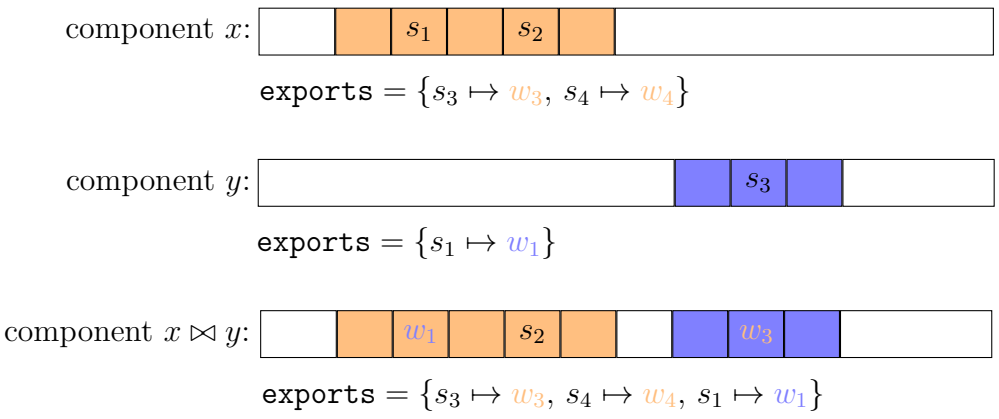

Figure 6: An example of linking two components

Proof : the semantics only has a single reduction rule, which `machine_run` attempts to apply. Therefore `machine_run` n (`mem`, `regs`) is the configuration of the system after n reduction steps (eventually stagnating on `Halted` or `Failed`). Quantifying over n then yields the transitive-reflexive closure of taking a reduction step. □

3.2 Linking

Definition 3.1 introduces open programs and definition 3.3 closed programs. In order to define contextual refinement we need a way to combine two open programs (the component being refined and a context) into a single closed program. This operation is called linking.

Intuitively, linking is a fairly simple operation. Take two components x and y; check which imports of x point to symbols defined in the exports of y, replace these imports with their new values and removes them from x.`imports`. Then do the same to y. Finally, return a component built by joining the resulting segments, remaining undefined imports and exports. Figure 6 illustrates this process. It represents components as a region of memory, with imports mixed in by having their identifying symbol s_i, paired with a map of exports.

Defining this formally is a matter of some technicality. For starters, we can't link any arbitrary pair of components together, as two components may define different values for the same memory addresses. As a result we introduce a notion of disjoint components.

Definition 3.4 : (Disjoint components)
Two components x and y are said to be *disjoint* when:

1. x and y are well-formed;
2. x.`segment` and y.`segment` are disjoint: `dom` $(x$.`segment`$) \cap$ `dom` $(y$.`segment`$) = \emptyset$;
3. x.`exports` and y.`exports` are disjoint: `dom` $(x$.`exports`$) \cap$ `dom` $(y$.`exports`$) = \emptyset$;

Since well-formedness requires import addresses to be segment addresses, disjoint components also have disjoint imports. Note that disjoint imports does not mean two components can't import the same symbols. It only implies they do not import symbols to the same addresses.

Definition 3.5 : (Linking)
Given two disjoint components x and y we define a new component, their link, as:

$$x \bowtie y \;:=\; \left\{ \begin{array}{lcl} \texttt{exports} & := & x.\texttt{exports} \;\uplus\; y.\texttt{exports} \\ \texttt{imports} & := & \left\{ a \mapsto s \;\middle|\; \begin{array}{l} a \mapsto s \in x.\texttt{imports} \;\uplus\; y.\texttt{imports} \;\wedge \\ s \mapsto \_ \notin x.\texttt{exports} \;\uplus\; y.\texttt{exports} \end{array} \right\} \\ \texttt{segment} & := & x.\texttt{segment}[y.\texttt{exports} \circ x.\texttt{imports}] \;\uplus \\ & & y.\texttt{segment}[x.\texttt{exports} \circ y.\texttt{imports}] \end{array} \right\}$$

Here $x.\texttt{segment}[y.\texttt{exports} \circ x.\texttt{imports}]$ is the map obtained by taking $x.\texttt{segment}$ and updating it with the values of $y.\texttt{exports} \circ x.\texttt{imports}$. That is to say, replacing the values at imported addresses with the values of the symbols defined in $y.\texttt{exports}$. This definition correspond well to the intuitive notion described above. Joining this with the same operation on $y.\texttt{segment}$ yields a union of both segments where all defined imports have been replaced. The definition of `imports` then just keeps the remaining undefined imports, and the definition of `exports` is a simple union.

This definition allows us to state and prove a few desirable results on linking, namely well-formedness, commutativity and associativity.

Lemma 3.2 : (Linking is well-formed)
Given two disjoint components x and y, their link $x \bowtie y$ is well-formed.

Proof :

1. The definition of $(x \bowtie y).\texttt{imports}$ clearly implies they are disjoint from $(x \bowtie y).\texttt{exports}$.
2. $(x \bowtie y).\texttt{imports}$ is included in $x.\texttt{imports} \uplus y.\texttt{imports}$. Since x and y are disjoint, they are well-formed and so their imports' domains are included in their segments' domains. Hence $\texttt{dom}\ (x \bowtie y).\texttt{imports} \subseteq \texttt{dom}\ x.\texttt{segment} \cup \texttt{dom}\ y.\texttt{segment}$. This right hand-side is included in $\texttt{dom}\ (x \bowtie y).\texttt{segment}$ since map updates can only increase the domain.
3. Words that appear in $(x \bowtie y).\texttt{segment}$ and $(x \bowtie y).\texttt{exports}$ all come from either the segments or exports of x (or y). Thus, any capability only points to a subsection of the region of x (or y), which is included in the region of $x \bowtie y$. □

Lemma 3.3 : (Properties of disjoint components)
Let us use $x \#_\ell y$ to denote the property "x and y are disjoint components". We then have the following for all x, y and z:

1. symmetry: $x \#_\ell y \;\Leftrightarrow\; y \#_\ell x$
2. compatibility with link: $x \#_\ell (y \bowtie z) \;\wedge\; y \#_\ell z \;\Leftrightarrow\; x \#_\ell y \;\wedge\; x \#_\ell z \;\wedge\; y \#_\ell z$.

Proof : Symmetry is trivial. The components with link is a consequence of lemma 3.2 and the inclusion $\texttt{dom}\ (x \bowtie y).\texttt{segment} \subseteq \texttt{dom}\ x.\texttt{segment} \cup \texttt{dom}\ y.\texttt{segment}$ □

Lemma 3.4 : (Properties of linking)
For all components x, y and z, we have:

1. commutativity: $x \#_\ell y \;\Rightarrow\; x \bowtie y = y \bowtie x$
2. associativity: $x \#_\ell y \;\wedge\; x \#_\ell z \;\wedge\; y \#_\ell z \;\Rightarrow\; x \bowtie (y \bowtie z) = (x \bowtie y) \bowtie z$

Proof : Commutativity is simply given by the commutativity of $\uplus$. Associativity involves some fairly technical details on partial functions. Namely:

- associativity and commutativity of $\uplus$
- $(f_1 \uplus f_2) \circ g = (f_1 \circ g) \uplus (f_2 \circ g)$
- if g_1 and g_2 are disjoint then $f \circ (g_1 \uplus g_2) = (f \circ g_1) \uplus (f \circ g_2)$
- $f \circ g = f \circ (g_{|\ \texttt{dom}\ f})$
- $f[g_1 \uplus g_2] = g_1 \uplus g_2 \uplus f_{|\ \texttt{dom}\ f \setminus (\texttt{dom}\ g_1 \cup \texttt{dom}\ g_2)}$ □

3.3 Contexts

We can now define a notion of context for our components. Essentially, a context is just what is missing from a component in order to form a closed program: another component to link it to defining its imports, and an initial register state.

Definition 3.6 : (Context)
Given a well-formed component x, a context of x is a pair $(z, \texttt{regs}) \in \texttt{component} \times (\texttt{RegName} \to \texttt{Word})$ such that:

1. z is a well-formed component disjoint from x;
2. $\texttt{img}\ x.\texttt{imports} \subseteq \texttt{dom}\ z.\texttt{exports}$ and $\texttt{img}\ z.\texttt{imports} \subseteq \texttt{dom}\ x.\texttt{exports}$;
3. register values are either 0 or exported: $\texttt{img regs} \subseteq \{0\} \cup \texttt{img}\ x.\texttt{exports} \cup \texttt{img}\ z.\texttt{exports}$

These conditions are sufficient to ensure that $(z \bowtie x, \texttt{regs})$ is a closed program. Typically, $\texttt{regs}(\texttt{pc})$ will be a capability with permission `RX`, pointing to a code subregion of z, and the other registers $\texttt{regs}(\texttt{r}_i)$ can be capabilities for data regions of z or 0.

A few technical properties of context can be inherited directly from those of linking.

Lemma 3.5 : (Context properties)
Let x, y and z be pairwise disjoint components, and let $\texttt{regs} \in \texttt{RegName} \to \texttt{Word}$, then:

1. if $(z, \texttt{regs})$ is a context of x then $(z \bowtie x, \texttt{regs})$ is a closed program;
2. symmetry: if $(z, \texttt{regs})$ is a context of x then $(x, \texttt{regs})$ is a context of z;
3. associativity: $(z, \texttt{regs})$ is a context of $x \bowtie y \quad \Leftrightarrow \quad (z \bowtie x, \texttt{regs})$ is a context of y;

Proof : Trivial by definition of contexts and associativity of links. □

Another interesting property is that the context z only has limited access to x's memory region, be it to read data, write data or execute code. Indeed, the only capabilities pointing to x that z's region and `regs` can initially contain must be from $x.\texttt{exports}$ (whose values are pasted in z when linking). As the program runs, z can obtain further capabilities to x in two ways. It can read them from segments of x it has a read access to. Or, it can obtain them by executing code in x that leaves new capabilities in a register or an accessible memory region, assuming it has a capability with permission `E` to that code.

While these notions of component, linking and contexts can be generalized to other processors, this isolation property is inherent to capability machines. The following lemma illustrates this by showing how portions of a component that don't export capabilities are essentially irrelevant to the context.

Lemma 3.6 : (Removing exportless components)
Let x and y be two disjoint components with $y.\texttt{exports} = \emptyset$. Then for all components z and register state `regs` :

$$(z, \texttt{regs}) \text{ is a context of } x \bowtie y \ \Rightarrow\ (z, \texttt{regs}) \text{ is a context of } y$$

Proof : $z \#_\ell\ x \bowtie y$ implies $z \#_\ell\ x$ by lemma 3.3. Since $y.\texttt{exports} = \emptyset$ and x is well-formed we have $(x \bowtie y).\texttt{exports} = x.\texttt{exports}$ and $x.\texttt{imports} \subseteq (x \bowtie y).\texttt{imports}$, which is sufficient to prove the second point. For the third point, reuse $(x \bowtie y).\texttt{exports} = x.\texttt{exports}$. □

Finally, the following lemma presents sufficient conditions on x and y for all contexts of x to be contexts of y. It is mainly a technical result, but nice to keep in mind as such an implication will show up in the definition of contextual refinement.

Lemma 3.7 : (Sufficient conditions for context of x to be context of y)
Let x and y be two components such that:

- x and y are well-formed;
- the memory region of y is included in x: $\texttt{dom}\ y.\texttt{segment} \subseteq \texttt{dom}\ x.\texttt{segment}$
- symbols imported by y are also imported by x: $\texttt{img}\ y.\texttt{imports} \subseteq \texttt{dom}\ x.\texttt{imports}$
- x and y exports are the same: $y.\texttt{exports} = x.\texttt{exports}$

Then for all $(z, \texttt{regs})$,

$$(z, \texttt{regs}) \text{ is a context of } x \quad \Rightarrow \quad (z, \texttt{regs}) \text{ is a context of } y$$

Proof : Use the inclusions to show that $x \ \#_\ell \ z \Rightarrow y \ \#_\ell \ z$, then use them again to show that the imports of y are included in the exports of z. □

4 Contextual refinement

With the definitions introduced in the previous section, we now have all the tools needed to formally defined contextual refinement for capability machines. Recall that the general way one would define contextual refinement between two open programs x and y is by saying x refines y if, for all contexts C, if $C[x]$ has some observable behavior then so does $C[y]$. In practice this is often simply stated as "if $C[x]$ terminates then so does $C[y]$". This captures all observable behavior since you can always choose C to be a program that branches on the behavior you want to observe, with one branch halting and the other looping indefinitely. An example of such a context is presented in figure 7. It is written in OCaml for simplicity, but one could easily write something behavior using Cerise's instructions.

```
if test <insert component here>
then ()
else let rec loop () = loop () in loop ()
```

Figure 7: Example of context that terminates if and only if `test x` succeeds

In this section, we will first explain how we adapted this high-level definition to our capability machine, and then prove some easy consequences of our definition.

4.1 Defining contextual refinement

Our notions of contexts and components present a few challenges when combined with the universal quantification we want to use. For starters, we cannot apply any context $(z, \texttt{regs})$ to any pair of components x and y, since the definition of context requires that z be disjoint from x (and y). Should we then quantify on all contexts of x *and* y; or quantify on all contexts of x and require that any context of x also be a context of y? The former presents limited usefulness as it allows x and y to be very different. For instance, if x and y have disjoint exports, then a context of both accesses none of them (since it would only import symbols from their intersection), and thus can't inspect x and y in any meaningful way. This can lead to x refining y even though they are totally unrelated. On the other hand, the latter imposes strong constraints on y, namely that its memory region is included in x's. We choose the latter as we would rather have a strong notion with few refinements than have weak refinements with limited usefulness. As an example, transitivity of contextual refinement may not be provable using the first notion.

Another common way to deal with this quantification problem (as seen in Timany et al. [2022] and Frumin et al. [2020]) is to add a type judgement to contextual refinement: $x \preccurlyeq_{\text{ctx}} y : \tau$ which asserts that x and y are components of type τ and only quantifies on contexts with a hole of type τ. We could define a type system on components to do this, but it would be fairly unwieldy. In order to deduce from "$x : \tau$" and "$(z, \texttt{regs})$ has a hole of type τ" that "$(z, \texttt{regs})$ is a context of x", τ would need to at least list all symbols in $x.\texttt{imports}$ and $x.\texttt{exports}$, and all addresses are in $\texttt{dom}\ x.\texttt{segment}$. This leads to a bulky type system which imposes constraints at least as strong our choice of requiring contexts of x to be contexts of y.

However, even this quantification over the context is not completely satisfying, as the set of possible context could be empty. Indeed, since our memory is finite, a component taking up the whole memory leaves no place for a context to exist. This is slightly problematic as it means any random component could be made to refine any other by just expanding it to fill the every address. Additionally, many of the properties of contextual refinement are proved by applying it on a well-designed explicit context which takes a set amount of space. Therefore, we choose to only define contextual refinement on components that leave a fixed amount of free memory. For ease of proof, this is defined by saying components cannot address the fixed region $[0; \text{min\_ctxt\_size})$, although other choices could also work here.

Finally, we also require that $x.\texttt{exports}$ defines at least all the symbols in $y.\texttt{exports}$. In practice, we are only interested in refinement between components with the same exports, as these are what the context can see about the components. Any extra export in y would just be ignored by a context of x. Since it is easy to remove exports from a component, this requirement doesn't change the notion much beyond ensuring the right-hand side has a minimal exports list.

Definition 4.1 : (Contextual refinement)
Let x, y be two components, we say that x *contextually refines* y and we write $x \preccurlyeq_{\text{ctx}} y$ when:

1. x and y are well-formed;
2. x has free space: $\texttt{dom}\ x.\texttt{segment} \cap [0; \text{min\_ctxt\_size}) = \emptyset$;
3. all exports of y are also defined in x: $\texttt{dom}\ y.\texttt{exports} \subseteq \texttt{dom}\ x.\texttt{exports}$;
4. for all $(z, \texttt{regs}) \in \texttt{component} \times (\texttt{RegName} \to \texttt{Word})$ and for all values $v \in \{\texttt{Halted}, \texttt{Failed}\}$,
   if $(z, \texttt{regs})$ is a context of x and $\exists\, n,\ \texttt{machine\_run}\ n\ (z \bowtie x, \texttt{regs}) = v$
   then $(z, \texttt{regs})$ is a context of y and $\exists\, n,\ \texttt{machine\_run}\ n\ (z \bowtie y, \texttt{regs}) = v$

With this definition, the observable behaviors of x also occurring in y are simply terminating in a value $v \in \{\texttt{Halted}, \texttt{Failed}\}$. It may seem a weak notion at first, but the quantification on all context actually means it can capture a lot of behaviors. For instance, suppose x and y export a simple capability with permission $\texttt{R}$ pointing to a value v_x and v_y. Then $x \preccurlyeq_{\text{ctx}} y$ will imply $v_x = v_y$, since we can create a simple context that contains code which reads from the imported capability, then halts if the value read is equal to v_x and fails otherwise. This code will always terminate (it only runs known code from the context, not from x or y), halt when executed linked with x, and thus must also halt when executed linked with y.

The same reasoning can apply to more complex examples. Suppose x and y contain some code accessible through a capability with permission $\texttt{E}$, then we can assert if the code of x halts, or fails when called on some fixed arguments (register values passed during the jump), then the code of y must also halt or fail. Furthermore, if the code of x returns control the context z then so does y, and we can assert that the returned values left in the registers must also be the same. However, if the code of x loops indefinitely, then the code of y can exhibit any behavior, since the condition $\exists\, n, \texttt{machine\_run}\ n\ (z \bowtie x, \texttt{regs}) \in \{\texttt{Halted}, \texttt{Failed}\}$ will be false.

4.2 Properties of contextual refinement

One of the key properties of contextual refinement is compositionality. This comes in two forms: vertical compositionality (also called transitivity) $x \preccurlyeq_{\text{ctx}} y \Rightarrow y \preccurlyeq_{\text{ctx}} z \Rightarrow x \preccurlyeq_{\text{ctx}} z$; and horizontal compositionality ($x \preccurlyeq_{\text{ctx}} y \Rightarrow x' \preccurlyeq_{\text{ctx}} y' \Rightarrow x \bowtie x' \preccurlyeq_{\text{ctx}} y \bowtie y'$). These compositionality lemmas allow us to derive a large contextual refinement from multiple smaller ones, and thus allows for modular and incremental verification. Both of these properties are fairly easy to establish with our definition and are shown in lemma 4.1 and lemma 4.3 respectively.

Lemma 4.1 : (Contextual refinement is a preorder)
Contextual refinement verifies the following properties:

1. reflexivity: for all well-formed components x with free space, we have $x \preccurlyeq_{\text{ctx}} x$;
2. transitivity: for all components x, y, z, if $x \preccurlyeq_{\text{ctx}} y$ and $y \preccurlyeq_{\text{ctx}} z$ then $x \preccurlyeq_{\text{ctx}} z$.

Proof : for both points, the first two conditions of contextual refinement are immediate with the given hypotheses. Then we can simply use reflexivity and transitivity of the subset relation for the third condition and of logical implication for the fourth. □

The following lemma looks into consequences of refinement. More specifically, it results from the implication that all contexts of x must also be contexts of y. This imposes a few conditions on y. The condition on y.`imports` and y.`exports` are actually nice to have, since we only want refinement between components that look the same, meaning they define the same symbols and have the same dependencies. Here x can have more imports than y, but these extra imports can only be of limited use. They can't change the behavior of x, since the behavior of y (which doesn't depend on these extra imports) would then also have to change.

The inclusion of memory segments is a more unfortunate side condition. We can always add dummy values to a small x to artificially grow it to be larger than y if needed. Lemma 4.4 illustrates how adding dummy values in this way does not change contextual refinement. Still this method is in many ways clumsy and unsatisfying.

Upon closer inspection though, this restriction is weaker than it may seem. The context will always be able to inspect the memory space occupied by x and y to some degree. For instance, it will be able to read the addresses from all capabilities into x and y it can obtain using the instructions `getb`, `gete` and `geta`. Therefore, all capabilities of x and y the context can obtain (even `E` capabilities), must point to exactly the same addresses. The only capabilities that can point to different addresses are private capabilities: capabilities only accessible by the code of x and y and never leaked to the context. As a result, this restriction really only says that y's private memory must be included in x's.

In fact, this readily available inspection of memory also leads to many more questionable behaviors. For example, a component will hardly ever refine a copy of itself that has been shifted in memory. Ideally, we would like to have a relation where these programs refine each other. However, since all exported capabilities will have different addresses, a context can just call `geta` and change its behavior according to the returned value.

This is an unavoidable consequence of contextual refinement. Any reasonable definition of such a refinement requires all observable behaviors to be the same. Thus, since addresses can be inspected and compared to integers, they must match. Notice how changing the observable behaviors on our machines also changes refinement. If cerise had a `time` instruction returning the number of cycles since startup, execution time would become an observable behavior and refinement would imply all functions of x and y that return take the exact same amount of steps.

The following lemma presents necessary conditions on the shapes of x and y when $x \preccurlyeq_{\text{ctx}} y$. These results essentially derive from the "all contexts of x are contexts of y" part of the

definition. It is worth comparing them to the sufficient condition presented in lemma 3.7. The shows the first two points, relating imports and segments are both necessary and sufficient. The third point relating exports requires full equality in lemma 3.7 whereas we only show domain equality here. In practice though, we are only interested in refinements between components that have the same exports.

Lemma 4.2 : (Necessary conditions of contextual refinement)
Let x and y be two components such that $x \preccurlyeq_{\text{ctx}} y$, then:

- all symbols imported by y are also imported by x : $\texttt{img}\ y.\texttt{imports} \subseteq \texttt{img}\ x.\texttt{imports}$
- the memory region of y is included in x's : $\texttt{dom}\ y.\texttt{segment} \subseteq \texttt{dom}\ x.\texttt{segment}$
- x and y export the same symbols : $\texttt{dom}\ x.\texttt{exports} = \texttt{dom}\ y.\texttt{exports}$
- y has free space: $\texttt{dom}\ y.\texttt{segment} \cap [0; \text{min\_ctxt\_size}) = \emptyset$;
- for all z, if x and z are disjoint, then so are y and z

Proof : For the first point, consider the following context:

$$z \ := \ \left\{ \begin{array}{lcl} \texttt{segment} & := & \{0 \mapsto \texttt{encodeInstr}(\texttt{halt})\} \\ \texttt{imports} & := & \emptyset \\ \texttt{exports} & := & \{s \mapsto 0,\ s \in \texttt{img}\ (x.\texttt{imports})\} \ \uplus \ \{s' \mapsto (\texttt{RX}, 0, 1, 0)\} \end{array} \right\}$$

$$\texttt{regs} \ := \ \{pc \mapsto (\texttt{RX}, 0, 1, 0),\ \texttt{r}_0 \mapsto 0,\ \ldots,\ \texttt{r}_{\texttt{RegMax}} \mapsto 0\}$$

where s' is a fresh symbol not in $x.\texttt{imports}$ or $x.\texttt{exports}$.

It is a valid context of x and clearly halts in a single step. Therefore, it must be a valid context of y. This implies that $\texttt{img}\ y.\texttt{imports} \subseteq \texttt{dom}\ z.\texttt{exports}$ which is $\texttt{img}\ (x.\texttt{imports})$ by definition.

For the second point, take an address a in $\texttt{dom}\ y.\texttt{segment}$ but not in $\texttt{dom}\ x.\texttt{segment}$ Then taking the same context z but adding $\{a \mapsto 0\}$ to the segment is still valid context of x. So it must be a valid context of y which leads to a contradiction.

The third point is show by double inclusion. One inclusion is found in $x \preccurlyeq_{\text{ctx}} y$, for the other, take $s \in \texttt{dom}\ x.\texttt{exports}$ then use the same context as point 1 but add $\{1 \mapsto 0\}$ to the segment and $\{1 \mapsto s\}$ to the imports. This is still a context of x, so it is a context of y. Hence, s must also be exported by y.

The last two points are easy consequences of the first two. □

Lemma 4.3 : (Horizontal compositionality)
Contextual refinement can also be composed horizontally via linking:

1. Let x, y and z be three components such that x and z are disjoint, then:

$$x \preccurlyeq_{\text{ctx}} y \quad \Rightarrow \quad x \bowtie z \preccurlyeq_{\text{ctx}} y \bowtie z$$

2. Let x, x', y, y' be four components with x and x' disjoint, then:

$$x \preccurlyeq_{\text{ctx}} y \ \wedge \ x' \preccurlyeq_{\text{ctx}} y' \quad \Rightarrow \quad x \bowtie x' \preccurlyeq_{\text{ctx}} y \bowtie y'$$

Proof : For the first compositionality, let us take a context $(t, \texttt{regs})$ of $x \bowtie z$ and a value v such that $\exists n$, $\texttt{machine\_run}\ n\ (t \bowtie x \bowtie z, \texttt{regs}) = v$, then we use lemma 3.5 and link associativity/-commutativity to show that $(t \bowtie z, \texttt{regs})$ is a context of x. We can then use the $x \preccurlyeq_{\text{ctx}} y$ to show the $(t \bowtie z, \texttt{regs})$ is a context of y and $\exists n$, $\texttt{machine\_run}\ n\ (t \bowtie y \bowtie z, \texttt{regs}) = v$, finally we reuse lemma 3.5 to move z out of the context.

The second compositionality is obtained from the first. Specifically, we use it twice to show $x \bowtie x' \preccurlyeq_{\text{ctx}} y \bowtie x'$ and $y \bowtie x' \preccurlyeq_{\text{ctx}} y \bowtie y'$. We conclude by transitivity.

Note that using lemma 3.5 and link associativity/commutativity (lemma 3.4) require proving that the different components involved are disjoint. This can fairly easily be established from the hypotheses, lemma 3.3 and the last point of lemma 4.2. □

Lemma 4.4 : (Refinement with exportless component)
Let x, y and z be three components where z has no exports, then we can

1. add z to the left-hand side: if $x \preccurlyeq_{\text{ctx}} y$ and $x \#_\ell z$ then $x \bowtie z \preccurlyeq_{\text{ctx}} y$
2. remove z from the right-hand side: if $x \preccurlyeq_{\text{ctx}} y \bowtie z$ and $y \#_\ell z$ then $x \preccurlyeq_{\text{ctx}} y$

Proof : For the first point let us take a context $(t, \texttt{regs})$ of $x \bowtie z$ such that `machine_run` converges to a value v. Lemma 3.6 implies $(t, \texttt{regs})$ is also a context of x. Now we only need to show that $\exists\, n$, `machine_run` n $(t \bowtie x, \texttt{regs}) = v$. In fact, using the restriction on capabilities of well-formed components, we can show that `machine_run` doesn't depend on the added memory.

This can be done by induction, using a case disjunction of the current instruction to show that the memory after one step is still well-formed.

For the second point, take a context $(t, \texttt{regs})$ of $x \bowtie z$ such that `machine_run` converges to a value v. Then use refinement followed by lemma 3.6 to show that it is a context of y. Use the sema reasoning as before to that `machine_run` hasn't changed. □

5 Validity Relation

While our definition of contextual refinement can be pretty powerful, it is also hard to prove as is. It requires reasoning about all possible contexts that can interact with our components in multiple different ways. Ideally we would like a simple condition on our components to establish refinement. More specifically, we want to define a binary logical relation on components that is simpler to prove and manipulate than refinement, and show that it implies refinement.

Fortunately, Cerise already has logical binary relation that relates program. It is inspired by the unary validity relation established by Georges et al. [2021]. A similar binary relation can be found in Georges et al. [2022], where it is used to show confidentiality properties. Although the one we use here is simpler as we don't have local or uninitialized capabilities in our model. We will present this relation and its fundamental theorem; then explore how to establish a link between it and contextual refinement via Iris's adequacy theorem.

5.1 The binary validity relation

The binary validity relation $\mathcal{V}$ and its paired binary expression relation $\mathcal{E}$ are presented in figure 8. Initially this relation was unary and meant to capture the notion of values which are safe to share with unknown code. That is to say, values that can be shared without risk of breaking memory invariants. Similarly, the expression relation is designed to capture the notion of values that are safe to execute without breaking the memory.

This definition is defined within Iris's logic. The separating conjunction $*$ (similar to $\wedge$ but also asserts ownership of resources), separating implication $-\!\!*$ (similar to $\Rightarrow$) and points-to predicates ($a \mapsto_\ell v$ denotes that address a contains value v and $r \rightarrowtail_\ell w$ says register r contains value w) are usual separation logic predicates. Since we work on pairs of programs, we also have two more points-to predicate ($a \mapsto_r v$ and $r \rightarrowtail_r w$) for the resources of the right-hand side program. The predicate `WP` (s_ℓ, s_r) $\{(v_\ell, v_r), P\ v_\ell\ v_r\}$ is a weakest-precondition. If it holds then running the left and right programs from configurations s_ℓ, $s_r \in$ `ExecMode` results in two

$$
\begin{array}{lcl}
\mathcal{V}\,(z,\,z) & := & \texttt{True} \\
\mathcal{V}\,((\texttt{O},b,e,a),\,(\texttt{O},b,e,a)) & := & \texttt{True} \\
\mathcal{V}\,((\texttt{E},b,e,a),\,(\texttt{E},b,e,a)) & := & \triangleright\,\square\,\mathcal{E}\,((\texttt{RX},b,e,a),\,(\texttt{RX},b,e,a)) \\
\mathcal{V}\,((\texttt{R/RX},b,e,a),\,(\texttt{R/RX},b,e,a)) & := & \ast_{a\in[b;e)}\ \exists\,P,\ \left\{ \begin{array}{l} \boxed{\exists\,w\,w',\ a\mapsto_\ell w * a\mapsto_r w' * P(w,w')} * \\ \triangleright\,\square\,\forall\,w\,w',\ P(w,w') \mathrel{-\!\!*} \mathcal{V}\,(w,\,w') \end{array} \right. \\
\mathcal{V}\,((\texttt{RW/RWX},b,e,a),\,(\texttt{RW/RWX},b,e,a)) & := & \ast_{a\in[b;e)}\ \boxed{\exists\,w\,w',\ a\mapsto_\ell w * a\mapsto_r w' * \mathcal{V}\,(w,\,w')}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{E}\,(w_\ell,\,w_r) & := & \forall\,\texttt{regs}_\ell,\,\texttt{regs}_r \in \texttt{RegName} \to \texttt{Addr},\ \texttt{regs}_\ell(\texttt{pc}) = w_\ell \ \wedge\ \texttt{regs}_r(\texttt{pc}) = w_r \ \Rightarrow \\
 & & \left(\ast_{r\in\texttt{RegName}}\ r \rightarrowtail_\ell \texttt{regs}_\ell(r) * r \rightarrowtail_r \texttt{regs}_r(r) * \mathcal{V}\,(\texttt{regs}_\ell(r),\,\texttt{regs}_r(r)) \right) \mathrel{-\!\!*} \\
 & & \texttt{WP}\ (\texttt{Running},\texttt{Running})\ \{(v_\ell,v_r),\ v_\ell = \texttt{Halted} \Rightarrow v_r = \texttt{Halted}\}
\end{array}
$$

Figure 8: Definition of the binary validity relation

values v_ℓ, $v_r \in \texttt{ExecMode}$ verifying P. Typically, P and WP contain assertions the memory and register states via points-to predicates or invariants.

The symbols $\triangleright\,P$, $\square\,P$ and $\boxed{P}$ are more Iris-specific. The first denotes the later modality, saying P holds after one step of computation. The second the persistently modality, saying P only depends on persistent resources, typically no specific memory values. Finally, $\boxed{P}$ denotes that P holds as an invariant; it must hold at every step of the program.

The definitions of $\mathcal{V}$ and $\mathcal{E}$ are circular, but they be well-defined in Iris thanks to the later modality ($\triangleright\,P$). Iris provides a fixpoint operator on recursive definition where all recursive calls are guarded under a later (or in an invariant $\boxed{P}$, which adds a later when opened), as is the case here.

Let us walk through and explain the definition of figure 8. Binary validity is defined on pair of equal values. Words and opaque capabilities (O) are always safe to share, as they grant no access to memory. Enter capabilities (E) are safe to share when they are safe to execute. Since they can be called multiple times, it must be persistently safe to execute. This is expressed by the persistent modality $\square$, which can only be proven if we can establish $\mathcal{E}$ using only persistent resources (typically, invariants).

Read-write(-execute) capabilities are safe to share when they point to a memory region whose values are also safe to share. This is also true for read(-execute) capabilities. However, since memory cannot be modified with them, they can depend on any invariant P that implies $\mathcal{V}$. This allows for stronger predicates in read-only sections. Notice that there is no distinction between capabilities with and without the execute permission. They have the same expressive power since unknown code could always copy their instructions to an executable section of memory and run them there.

Finally, a pair of values are safe to execute when, for any valid register state pair whose pcs are the given values, if running the machine with those register states terminates in Halted to the left, then it also terminates in Halted to the right.

The fundamental theorem on this logical relation essentially states that if a value is safe to share then it is safe to execute.

Theorem 5.1 : (Fundamental theorem on value interpretation)
For all capabilities (p,b,e,a) :

$$\texttt{spec\_ctx} \ \Rightarrow\ \mathcal{V}\,((p,b,e,a),\,(p,b,e,a)) \ \Rightarrow\ \mathcal{E}\,((p,b,e,a),\,(p,b,e,a))$$

Here the `spec_ctx` hypothesis is a technical requirement due to our encoding of the right-hand side resources into Iris, and can be ignored by the reader.

The proof of this theorem is quite involved and beyond the scope of this report. It requires using the semantics of each instruction individually. The paper by Georges et al. [2021] presents a proof of a similar theorem for the unary relation.

5.2 The valid exports relation

The binary expression relation $\mathcal{E}$ seems to capture a similar notion to contextual refinement, as we once again have an implication of the form "if the left-hand side halts then so does the right". The question then arises, can we relate both concepts? Ideally we would like to be able to deduce contextual refinement from some form of validity. Mainly because the latter is easier to prove on known programs, given the fundamental theorem and the lack of necessity to reason about arbitrary contexts.

They are also both designed for interacting with unknown code in some way. Contextual refinement requires components to be called by an unknown context, while validity is designed to show which values a known program can safely share with unknown code.

Of course, contextual refinement is defined on components, whereas validity is only defined on words. However, validity does make use of a full memory as well, it is just implicit in our program logic instead of explicitly given with a map. As argued in the contextual refinement section, all that really matters about components from the context's point of view is a components's exports. This leads to the following definition.

Definition 5.1 : (Valid exports relation)
Two *components* x and y verify the valid export relation when:

$$\mathcal{V}_{\text{exp}}\,(x,\,y) \;:=\; \mathop{\Large *}_{s \mapsto w_r \,\in\, y.\texttt{exports}} \exists\, w_\ell,\; s \mapsto w_\ell \in x.\texttt{exports} * \mathcal{V}\,(w_\ell,\,w_r)$$

Formulated as such, it implies that the left-hand side exports at least all the symbols of the right-hand side ($\texttt{dom}\; y.\texttt{exports} \subseteq \texttt{dom}\; x.\texttt{exports}$). This is one of the requirements of contextual refinement, as it ensures our components have common symbols which common contexts can use.

Additionally, since validity is duplicable, so is this exports validity relation.

Using this definition we can already present the shape of the theorem we want to prove. For a given pair of components x and y, if one can show $\mathcal{V}_{\text{exp}}\,(x,\,y)$ using the left-memory points-to of x and the right-memory points-to of y, then $x \preccurlyeq_{\text{ctx}} y$. The general plan for the proof is to first show that for all contexts $(z, \texttt{regs})$, $\mathcal{V}_{\text{exp}}\,(x,\,y) \Rrightarrow \mathcal{V}_{\text{exp}}\,(x \bowtie z,\, y \bowtie z)$ (lemma 5.2). Then we use the FTLR (lemma 5.1) to obtain a `WP` from the validity of the registers. To conclude we use Iris' adequacy theorem (lemma 5.3) to transform this `WP` into a statement about the concrete semantics that matches our definition of refinement.

The final theorem is presented in lemma 5.5. It is weaker than the presentation given above as we need extra hypotheses on the components and the context. We will introduce these extra restrictions as they show up.

Notation 5.2 :
Since writing resource predicates can be quite verbose, let us introduce a few notations to make our statements more legible:

- $x \bowtie_\ell y = x.\texttt{segment}[y.\texttt{exports} \circ x.\texttt{imports}]$

  This is the part of $(x \bowtie y).\texttt{segment}$ that comes from x, with y's exports mixed in. Note that when $x.\texttt{imports} = \emptyset$, then $x \bowtie_\ell y = x.\texttt{segment}$

- $x \bowtie_r y = y.\texttt{segment}[x.\texttt{exports} \circ y.\texttt{imports}]$ similarly.

  By definition 3.5 we have $(x \bowtie y).\texttt{segment} = (x \bowtie_\ell y) \uplus (x \bowtie_r y)$
- $\texttt{mem\_map}_\ell(x) = \ast_{a \mapsto w \in x} a \mapsto_\ell w$ and $\texttt{mem\_map}_r(x) = \ast_{a \mapsto w \in x} a \mapsto_r w$

  The left-memory (resp. right-memory) points-tos for a partial function $x \in \texttt{Addr} \rightharpoonup \texttt{Word}$.
- $\texttt{reg\_map}_\ell(\texttt{regs}) = \ast_{r \mapsto w \in \texttt{regs}} r \rightarrowtail_\ell w$ and $\texttt{reg\_map}_r(\texttt{regs}) = \ast_{r \mapsto w \in \texttt{regs}} r \rightarrowtail_r w$

  The left-register (resp. right-register) points-tos for a function $\texttt{regs} \in \texttt{RegName} \to \texttt{Word}$.

Compatibility of valid exports and linking

The first tool we will need in order to prove our theorem relating $\mathcal{V}_{\text{exp}}$ and contextual refinement is to show a compatibility result between $\mathcal{V}_{\text{exp}}$ and linking. The following lemma proves that, given a few hypotheses, we can deduce $\mathcal{V}_{\text{exp}}\,(x \bowtie z,\, y \bowtie z)$ from $\mathcal{V}_{\text{exp}}\,(x,\, y)$ and the points-tos of z.

Lemma 5.2 : (Valid exports and linking)
Let x, y, z be components such that:

- x and z are disjoint; y and z are disjoint;
- z's segment only contains integers: $\texttt{img}\,(z.\texttt{segment}) \subseteq \mathbb{Z}$;
- x's export symbols are also exported by y: $\texttt{dom}\,x.\texttt{exports} \subseteq \texttt{dom}\,y.\texttt{exports}$;

Then we can deduce $\mathcal{V}_{\text{exp}}\,(x \bowtie z,\, y \bowtie z)$ from $\mathcal{V}_{\text{exp}}\,(x,\, y)$ and the memory points-to for the z part of the link:

$$\texttt{spec\_ctx} * \mathcal{V}_{\text{exp}}\,(x,\, y) * \texttt{mem\_map}_\ell(x \bowtie_r z) * \texttt{mem\_map}_r(y \bowtie_r z) \;\Rrightarrow\; \mathcal{V}_{\text{exp}}\,(x \bowtie z,\, y \bowtie z)$$

Here $\texttt{mem\_map}_\ell(x \bowtie_r z) * \texttt{mem\_map}_r(y \bowtie_r z)$ essentially contains left and right memory points-tos of z, but after linking with x (and y). This is because they must include the true values of any imports of z, not the dummy values present in $z.\texttt{segment}$ before the link. We could show this using $\texttt{mem\_map}_\ell((x \bowtie z).\texttt{segment}) * \texttt{mem\_map}_r((y \bowtie z).\texttt{segment})$ directly, but we would then also consume the points-tos of the x (and y) part of the link. Points-tos which aren't needed here and are most likely going to be needed to prove $\mathcal{V}_{\text{exp}}\,(x,\, y)$.

Proof : This proof is done in two separate steps. First we consume the memory points-to to allocate invariants on all of z's memory:

$$\ast_{a \in \texttt{dom}\,z.\texttt{segment}} \boxed{\exists\, w_\ell\, w_r,\, a \mapsto_\ell w_\ell * a \mapsto_r w_r * \mathcal{V}\,(w_\ell,\, w_r)}$$

This can be established because words in $z.\texttt{segment}$ (linked with x to the left and y to the right) are either integers (thanks to the hypothesis) which always verify $\mathcal{V}$, or they imported values and are therefore valid since $\mathcal{V}_{\text{exp}}\,(x,\, y)$ holds and $\texttt{dom}\,x.\texttt{exports} \subseteq \texttt{dom}\,y.\texttt{exports}$ ensures the exports define the same symbols, so there are no cases where only one value is imported.

Second we show that $\mathcal{V}_{\text{exp}}\,(x \bowtie z,\, y \bowtie z)$ holds because exports of $(x \bowtie z, y \bowtie z)$ are either exports of (x, y) (which are valid by $\mathcal{V}_{\text{exp}}\,(x,\, y)$) or exports of (z, z). The exports of z are valid they are equal and can only address a z's memory region (thanks to well-formedness), which we proved to be valid in the first step. We need to use the fundamental theorem here for the case of $\texttt{E}$ capabilities. □

While this lemma does not explicitly require x and y to not import anything from the context z, proving $\mathcal{V}_{\text{exp}}\,(x,\,y)$ while x and y contain undefined imports could be rather challenging. We could prove $\mathcal{V}_{\text{exp}}\,(x,\,y)$ assuming the imports are valid, but since those imports (exports of z) could then depend on the exports of x and y, we would be stuck with a circular entailment.

Another point worth noting is the requirement that $z.\texttt{segment}$ not contain any capabilities. This is for ease of proof and could theoretically be relaxed. It avoids circular capabilities in the context, whose validity is harder to establish. Thus, the only capabilities the context will have access to will be in the registers. Since it's always possible to have the program start with a few instructions to write capabilities from the registers to memory as needed, proving the stronger version was skipped. It is also worth noting that this restriction is quite common and also present in previous work [Georges et al., 2022].

5.3 Adequacy and link to contextual refinement

Passing from Iris' program logic, in which $\mathcal{V}_{\text{exp}}$ is defined, into the more traditional logic in which we defined contextual refinement is possible thanks to an adequacy theorem. The general form of such a theorem is that for any program e such that $\texttt{WP}\ e\ \{\Phi\}$, if $e \longrightarrow^* e'$ and e' is a value v then $\Phi(v)$ holds. Here we will use the following, more specific to our pairs of programs.

Lemma 5.3 : (Simplified binary adequacy)
Let $\texttt{mem}_\ell$ and $\texttt{mem}_r \in \texttt{Addr} \rightharpoonup \texttt{Word}$ be memory states and let $\texttt{regs} \in \texttt{RegName} \to \texttt{Word}$ be a register state, if:

- the memory-points to imply a $\texttt{WP}$:

$$\begin{aligned}\texttt{spec\_ctx} * \texttt{mem\_map}_\ell(\texttt{mem}_\ell) * \texttt{mem\_map}_r(\texttt{mem}_r) * \texttt{reg\_map}_\ell(\texttt{regs}) * \texttt{reg\_map}_r(\texttt{regs}) \\ \Rrightarrow\ \texttt{WP}\ (\texttt{Running}, \texttt{Running})\ \{(v_\ell,\, v_r),\ v_\ell = \texttt{Halted}\ \Rightarrow\ v_r = \texttt{Halted}\}\end{aligned}$$

- the left program reduces to $\texttt{Halted}$: $(\texttt{Running}, (\texttt{mem}_\ell, \texttt{regs}))\ \longrightarrow^*\ (\texttt{Halted}, \_)$

Then the right program also reduces to $\texttt{Halted}$: $(\texttt{Running}, (\texttt{mem}_r, \texttt{regs}))\ \longrightarrow^*\ (\texttt{Halted}, \_)$

This theorem can be derived from Iris' own adequacy theorem [Jung et al., 2018]. It is a slightly weaker version that is sufficient for our follow-up proofs. I have also hidden a few Iris technicalities that appear in the statement for legibility reasons.

Expressed as such, adequacy has no notion of components or links, and requires proving a $\texttt{WP}$ where we would rather prove validity of exports. The next lemma will come in handy to simplify and rephrase adequacy's first hypothesis in terms of our logical relations.

Lemma 5.4 : (From valid exports to a weakest precondition)
Let x, y and z be three well-formed components and $\texttt{regs}$ be a register state such that:

1. $(z, \texttt{regs})$ is a context of x and $(z, \texttt{regs})$ is a context of y;
2. $\texttt{regs}(\texttt{pc})$ is a capability;
3. z's segment does not contain capabilities: $\texttt{img}\ z.\texttt{segment} \subseteq \mathbb{Z}$
4. x's export symbols are also exported by y: $\texttt{dom}\ x.\texttt{exports} \subseteq \texttt{dom}\ y.\texttt{exports}$;
5. we can show x and y exports are valid:

$$\texttt{spec\_ctx} * \texttt{mem\_map}_\ell(x \bowtie_\ell z) * \texttt{mem\_map}_r(x \bowtie_\ell z)\ \Rrightarrow\ \mathcal{V}_{\text{exp}}\,(x,\,y)$$

Then $x \bowtie z$ and $y \bowtie z$ satisfy the first hypothesis of adequacy:

$$\begin{aligned}\texttt{spec\_ctx} * \texttt{mem\_map}_\ell((x \bowtie z).\texttt{segment}) * \texttt{mem\_map}_r((y \bowtie z).\texttt{segment}) * \\ \texttt{reg\_map}_\ell(\texttt{regs}) * \texttt{reg\_map}_r(\texttt{regs}) \\ \Rrightarrow\ \texttt{WP}\ (\texttt{Running}, \texttt{Running})\ \{(v_\ell,\, v_r),\ v_\ell = \texttt{Halted}\ \Rightarrow\ v_r = \texttt{Halted}\}\end{aligned}$$

Proof : Split $\texttt{mem\_map}_\ell((x \bowtie z).\texttt{segment})$ into $\texttt{mem\_map}_\ell(x \bowtie_\ell z) * \texttt{mem\_map}_\ell(x \bowtie_r z)$ and do the same for $\texttt{mem\_map}_r$. Give the first parts to the hypothesis 5 to obtain $\mathcal{V}_{\text{exp}}\,(x,\, y)$. Then give the second parts and $\mathcal{V}_{\text{exp}}\,(x,\, y)$ to lemma 5.2 in order to obtain $\mathcal{V}_{\text{exp}}\,(x \bowtie z,\, y \bowtie z)$ (the other hypotheses of lemma 5.2 are easily shown, and since `spec_ctx` is duplicable it can be given twice).

Next we show that all register values are valid. This is true since by definition of contexts, they are either 0 (so valid), or an export of $(x \bowtie z, y \bowtie z)$ which we have just shown to be valid. This implies `regs(pc)` is valid, so we have $\mathcal{E}\,(\texttt{regs(pc)}, \texttt{regs(pc)})$ via the FTLR (lemma 5.1) since `regs(pc)` is a capability.

All we need to conclude is to unfold the definition of $\mathcal{E}$ and feed it the register points to as well as our proof that the registers are valid. This yields the desired `WP`. □

With adequacy, compatibility between valid exports and linking, and this last lemma, we finally have all the tools to prove our main theorem. Lemma 5.5 shows how one can almost deduce contextual refinement from a proof of valid exports. Unfortunately, its results doesn't fully match our definition of refinement, as some extra hypotheses on the contexts are necessary.

Lemma 5.5 : (From valid exports to weak contextual refinement)
Let x and y be two well-formed components such that

1. x and y have no imports: $x.\texttt{imports} = y.\texttt{imports} = \emptyset$
2. x and y have the same exports: $x.\texttt{exports} = y.\texttt{exports}$
3. y's memory segment is included in x's: $\texttt{dom}\ y.\texttt{segment} \subseteq \texttt{dom}\ x.\texttt{segment}$
4. We can prove x and y's exports are valid:

$$\texttt{spec\_ctx} * \texttt{mem\_map}_\ell(x.\texttt{segment}) * \texttt{mem\_map}_r(y.\texttt{segment}) \;\Rrightarrow\; \mathcal{V}_{\text{exp}}\,(x,\, y)$$

Then, for all $(z,\, \texttt{regs})$, assuming $z.\texttt{segment}$ only contains integers,
if $(z, \texttt{regs})$ is a context of x and $\exists\, n,\ \texttt{machine\_run}\ n\ (z \bowtie x, \texttt{regs}) = \texttt{Halted}$
then $(z, \texttt{regs})$ is a context of y and $\exists\, n,\ \texttt{machine\_run}\ n\ (z \bowtie y, \texttt{regs}) = \texttt{Halted}$

Proof : First prove the result assuming `regs(pc)` is a capability.

The first three hypotheses allow us to use lemma 3.7 to show that $(z, \texttt{regs})$ is a context of y. We can use lemma 3.1 to replace the $\exists\, n,\ \texttt{machine\_run}\ n\ \bullet = \texttt{Halted}$ hypothesis and goal with $(\texttt{Running}, \bullet) \longrightarrow^* (\texttt{Halted}, \_)$.

We then use lemma 5.4 to convert hypothesis 4 into the first hypothesis of adequacy. We can do this since x and y have no exports, so $x \bowtie_\ell z = x.\texttt{segment}$ and same for y. Finally, we apply adequacy (lemma 5.3) with $\texttt{mem}_\ell := (x \bowtie z).\texttt{segment}$ and $\texttt{mem}_r := (y \bowtie z).\texttt{segment}$ to prove the remaining point.

For the case where `regs(pc)` is an integer, ExecStep dictates that the program reduces to `Failed` in a single step. Therefore, the condition $\exists\, n,\ \texttt{machine\_run}\ n\ (z \bowtie x, \texttt{regs}) = \texttt{Halted}$ cannot hold, and thus the implication is true. □

This is the main final result of this report and worth commenting on. To start with, we can point out how why this lemma is interesting. Recall that the reason we introduced the validity logical relation was to find a way to prove contextual refinement without having to reason about arbitrary contexts. Looking back on the lemma, we can notice it is close to achieving that. The hypotheses only focus on x and y and don't require reasoning about contexts. The first three hypotheses just concern the shape of our components, so really the only thing one needs to prove in order to apply this result is the fourth hypothesis. That is to say, show a logical relation on components x, y using the resources of these components.

This relation is indeed simpler to prove then contextual refinement. As a matter of fact, it was successfully proven on a Cerise encoding of the two counters from figure 1. We managed to show that $\mathcal{V}_{\text{exp}}$ (`counter`, `counter_neg`) and the symmetric hold given the relevant points-tos, and thus we could successfully apply lemma 5.5 on both to show they are (almost) contextually equivalent. One thing worth noting is that in order to achieve this, we had to pad one of the counters to ensure their code took the same amount of memory space.

Of course, the result of this lemma isn't $x \preccurlyeq_{\text{ctx}} y$; but a slightly weaker property in two ways: we only quantify on contexts which don't contain capabilities, and we only show the result for the final value of `Halted`, not `Failed`. The first restriction results from the proof of lemma 5.2. It should be possible to lift, but that would make the proof significantly harder, as it would require using some sort of fixed-point operator to prove validity of capabilities pointing to themselves. This restriction isn't really a problem either: we can replace a context z containing capabilities with a z' which has all the capabilities it needs in `regs`, and contains extra code just after the entry point that writes these capabilities to memory. We therefore decided to keep it as is and may add this restriction to the main definition of contextual refinement.

The restriction on the final value being fixed to `Halted` is a result of how we defined the validity relation. Trying to handle the `Failed` case would require redefining it and thus reproving the FTLR. As it stands refinement with only `Halted` is already powerful enough to capture most behaviors, since we are mostly interested in programs that succeed. The main limitation with redefining contextual refinement to only care about the `Halted` state is that programs that fail consistently would then refine anything, in the same way programs that don't terminate currently do.

The hypotheses on x and y are also worth discussing here. Hypotheses 2 and 3 aren't problematic as they are to be expected of components that refine each other (as seen in lemma 4.2). However, the requirement that they have no imports is the biggest limitation of this theorem. Ideally we would like to use this on modules with similar imports. For example, our `counter` example uses a `malloc` macro to allocate memory space. This macro is hard-coded in both counters. It would be nice to have refinement between the counter modules with `malloc` as an import rather than included code. Once again lifting this could be quite tedious as we could have to show circular validities if two components import from each other.

6 Conclusion

This work was in many ways a proof of concept. Its aim was to see if Cerise's defined binary validity relation could be used to show a form of refinement. I believe it to be rather successful at defining contextual refinement for capability-machines (in section 4). The definition is a reasonable port of the high-level notion and its limitations are mostly expected consequences of refinement and not of this adaptation.

The results in section 5 show how this notion can be related to validity. They prove that our goal was fairly reasonable and that validity does indeed imply some sort of refinement. The binary counter example shows this can be used on a real, albeit simple, pair of programs. They fall a bit short of proving the full implication however.

Future work

One could expand improve this work in a few ways. The results in section 5 can probably be strengthened with some additional proof effort. In particular, finding a way to prove validity of capabilities that transitively point to themselves could lift both the restrictions on import-less components and capability-free contexts. We could also look into applying this to some additional, more complex examples.

Studying this has also revealed that this notion is too strong for most practical purposes. We cannot prove refinement for many programs that nonetheless exhibit very similar behaviors. It would therefore be interesting to see what weaker notions of refinement could be defined, and how to limit the behaviors they can observe. For instance, we could look at relations based on comparing program states [Benton, 2004] or traces [Antonopoulos et al., 2019].

Internship retrospective

When I started looking for this internship I had two objectives in mind. Find an internship abroad since all previous internships I have done were confined to France by COVID; and if possible work on a large Coq-related project as I'd just discovered the proof assistant and wanted to see more of what could be done with it. I think it's safe to say it delivered on both these points. I enjoyed living in and visiting Denmark, even in the winter. It's a beautiful country and one I hope to be able to return to in the future.

The subject was very interesting. I'd worked with program verification before but mostly on automated checkers. Working directly on manual proofs was a new approach filled with wonderful discoveries and the occasional unspeakable frustration, but overall I really enjoyed the experience.

I also got to see more of academia. This isn't my first research internship, but comparing work in different labs helped me grasp what working in this field would be like. I'm planning on starting a thesis next semester and this internship gave me a better idea of what that involves.

Acknowledgments

I would like to thank the wonderful people of the LogSem and PL teams of Aarhus University for their welcome. In particular, many thanks to my supervisor Lars Birkedal for his guidance during the internship, and to Aïna Linn Georges for her helping me understand Cerise and establish the results presented above. I would also like to thank Bastien Rousseau for proofreading this report and offering useful suggestions to improve it.

A Notations

This short section gives a quick rundown of the notations used in this report. If you have this document in PDF format, most notations will link to their definition. Asides from the standard mathematical notations ($\in$, $\subseteq$, $\cap$, $\cup$, $\setminus$, $\emptyset$, $\mathbb{N}$, $\mathbb{Z}$...), we use some special notations for partial functions and separation logic.

Partial functions: for X and Y two sets, we denote by $X \rightharpoonup Y$ the set of *partial functions* (also called maps) from X to Y. On partial function, we define the following notations:

- $\{x \mapsto y\}$ for $x \in X$ and $y \in Y$ a singleton map.

  $\{x_0 \mapsto y_0, \ldots, x_n \mapsto y_n\}$ for disjoint x_i a map with the given explicit bindings.
- $x \mapsto y \in f$ if the map $f \in X \rightharpoonup Y$ is defined on $x \in X$ with value $y \in Y$.
- $x \mapsto \_ \notin f$ if the map $f \in X \rightharpoonup Y$ is undefined on $x \in X$.
- $f[g]$ the map f updated with the bindings of g:

  $\forall\, x \in X,\ y \in Y, \qquad x \mapsto y \in f[g] \quad \Leftrightarrow \quad x \mapsto y \in g \ \vee\ \big(x \mapsto \_ \notin g \ \wedge\ x \mapsto y \in f\big)$
- for $f \in X \rightharpoonup Y$, $\texttt{dom}\ f$ the domain of f (subset of X) and $\texttt{img}\ f$ its image (subset of Y).
- $f \uplus g$ for $f, g \in X \rightharpoonup Y$ with disjoint domains, their union:

  $\forall\, x \in X,\ y \in Y, \qquad x \mapsto y \in f \uplus g \quad \Leftrightarrow \quad x \mapsto y \in f \ \vee\ x \mapsto y \in g$
- $f \circ g$ for $f \in Y \rightharpoonup Z$ and $g \in X \rightharpoonup Y$ their composition:

  $\forall\, x \in X,\ z \in Z, \qquad x \mapsto z \in f \circ g \quad \Leftrightarrow \quad \exists y \in Y,\ y \mapsto z \in f \ \wedge\ x \mapsto y \in g$
- $f_{|\,S}$ for $f \in X \rightharpoonup Y$ and $S \subseteq X$ the restriction of f to S:

  $\forall\, x \in X,\ y \in Y, \qquad x \mapsto y \in f_{|\,S} \quad \Leftrightarrow \quad x \mapsto y \in f \ \wedge\ x \in S$

Iris separation logic: we use the following notation from separation logic and Iris:

- $P * Q$ is the separating conjunction of P and Q. It is essentially an "and" connector that also asserts its arguments are separate. So $P * P$ is false in general, unless P is duplicable.

  It is associative and commutative, allowing us to use the big operator notation: $\ast_{i \in S}\, s_i$.
- $P \mathrel{-\!\!*} Q$ is the separating implication (read as a standard implication).
- $\triangleright\, P$ is the Iris later modality. It asserts P holds "after one logical step" and is used to break circular definitions.
- $\square\ P$ is the Iris persistently modality. It asserts P ownership of the duplicable resources of P. It is thus duplicable.
- $\boxed{P}$ asserts P holds as an invariant. Invariant can be opened to access the resources they contain, but only for one step, after which one must give those resources back. This allows the invariant to be duplicable.
- To assert ownership of resources, we have two point two predicates, one for memory addresses ($a \mapsto_\ell w$) and one for registers ($r \rightarrowtail_\ell w$). Since we work with pairs of programs, we have a second copy of each point to ($a \mapsto_r w$ and $r \rightarrowtail_r w$) for the memory of the second (rightmost) program.
- $\texttt{WP}\ (s_\ell, s_r)\ \{(v_\ell, v_r), P\ v_\ell\ v_r\}$ is a weakest-precondition such that running the left and right programs from configurations s_ℓ, $s_r \in \texttt{ExecMode}$ results in two values v_ℓ, $v_r \in \texttt{ExecMode}$ verifying P

- $P \Rrightarrow Q$ is a Iris' frame-preserving update, essentially a separating implication that can also allocate invariants.

Miscellaneous: Finally, we define the following notations:

- $X \sqcup Y$ the disjoint union of X and Y: $(\{0\} \times X) \cup (\{1\} \times Y)$. We will leave the bidirectional mappings $x \leftrightarrow (0, x)$ and $y \leftrightarrow (1, y)$ implicit when unambiguous ($X \cap Y = \emptyset$).
- $\mathcal{O}(X) = X \sqcup \{\texttt{None}\}$ the option set (X with a single extra element `None`), and `Some` the projection $X \to \mathcal{O}(X)$
- $[b; e)$ the integer interval $\{b, b+1, \ldots, e-1\}$

References

Timos Antonopoulos, Eric Koskinen, and Ton Chanh Le. Specification and inference of trace refinement relations. volume 3, pages 1–30. Association for Computing Machinery (ACM), October 2019. doi: 10.1145/3360604. URL `http://arxiv.org/abs/1903.07213`.

Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In Neil D. Jones and Xavier Leroy, editors, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 14–25, New York, NY, USA, January 2004. Association for Computing Machinery. ISBN 158113729X. doi: 10.1145/964001.964003. URL `https://doi.org/10.1145/964001.964003`.

Lars Birkedal and Aleš Bizjak. Lecture notes on iris: Higher-order concurrent separation logic. 2020. URL `https://iris-project.org/tutorial-pdfs/iris-lecture-notes.pdf`.

Dan Frumin, Robert Krebbers, and Lars Birkedal. ReLoC reloaded: A mechanized relational logic for fine-grained concurrency and logical atomicity. *CoRR*, abs/2006.13635, July 2020. ISSN 1860-5974. URL `https://arxiv.org/abs/2006.13635`.

Aïna Linn Georges, Armaël Guéneau, Thomas van Strydonck, Amin Timany, Alix Trieu, Dominique Devriese, and Lars Birkedal. Cerise: Program verification on a capability machine in the presence of untrusted code. 2021. URL `https://cs.au.dk/~birke/papers/cerise.pdf`.

Aïna Linn Georges, Alix Trieu, and Lars Birkedal. Le temps des cerises: Efficient temporal stack safety on capability machines using directed capabilities. *Proc. ACM Program. Lang.*, 6(OOPSLA1):1–30, April 2022. ISSN 2475-1421. doi: 10.1145/3527318. URL `https://cs.au.dk/~ageorges/publications_pdfs/monotone-technical.pdf`.

Jifeng He, C. A. R. Hoare, and Jeff W. Sanders. Data refinement refined. *ESOP 86, European Symposium on Programming*, 213:187–196, 1986. doi: 10.1007/3-540-16442-1_14.

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. volume 28, page e20. Cambridge University Press (CUP), 2018. doi: 10.1017/S0956796818000151. URL `https://people.mpi-sws.org/~dreyer/papers/iris-ground-up/paper.pdf`.

Robbert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order concurrent separation logic. *ACM SIGPLAN Notices*, 52:205–217, May 2017. ISSN 0362-1340. doi: 10.1145/3093333.3009855.

John C. Mitchell. Representation independence and data abstraction. In *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '86, pages 263–276, New York, NY, USA, 1986. Association for Computing Machinery. ISBN 9781450373470. doi: 10.1145/512644.512669. URL https://doi.org/10.1145/512644.512669.

Youngju Song, Minki Cho, Dongjae Lee, Chung-Kil Hur, Michael Sammler, and Derek Dreyer. Conditional contextual refinement. *Proc. ACM Program. Lang.*, 7(POPL):1121–1151, January 2023. ISSN 2475-1421. doi: 10.1145/3571232. URL https://doi.org/10.1145/3571232.

Amin Timany, Robert Krebbers, Derek Dreyer, and Lars Birkedal. A logical approach to type soundness. 2022. URL https://cs.au.dk/~birke/papers/2022-submitted-logical-type-soundness.pdf.

Robert N.M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy*, pages 20–37. IEEE, May 2015. doi: 10.1109/SP.2015.9.