— *Homework* —

Compilation and Program Analysis

Hugo SALOU

1 AssembLISP: an imperative lambda calculus with asynchronous computations.

1.1 Local semantics.

Question 1. *Explain the difference between the store we used in the* while *language during the course and this one.*

Stores $\sigma : \textbf{Vars} \rightharpoonup \mathbb{Z} = \textbf{Values}$ for the while language were used to store the values associated with variables. Here, we do not use these as we rely on substitution in the expression, as variables are immutable. Instead, stores $\sigma : \mathbb{L} \rightharpoonup \mathbb{V}$ are used for references as they are needed to handle the mutability of the heap.

Question 2. *Add a* `while` *construct to the language: Extend the syntax and define its small step semantics with a dedicated rule. You may need to extend other intermediate constructs, explain.*

We update the AssembLISP syntax by adding:

$$e \in \text{Expr} ::= \cdots \mid \texttt{while}[\![b, e, e']\!]\ e'',$$

with $b \in \{0, 1\}$, and we define the "regular" `while` loop like the `let` as:

$$\texttt{while}\ e\ \{\ e'\ \} \triangleq \texttt{while}[\![0, e, e']\!]\ e,$$

and we enforce that we only write the "regular" while construct when writing programs. The idea is that the $\texttt{while}[\![\ldots]\!]$ construct is used in the small-steps as, if we do not save the original condition and loop body, they may get reduced during evaluation. The b parameter tells us if we are currently evaluating the condition or the body of the loop, and the right part (e'' in the grammar) is the current expression being evaluated (either the condition or the body depending on b).

Then we update the contexts to add

$$C ::= \cdots \mid \texttt{while}[\![b, e, e']\!]\ C,$$

as the only part we can update during the reduction is the rightmost part, except for b which is handled by the following inference rules

$$\textsc{WhileTrue}\quad \frac{}{(\texttt{while}[\![0, e, e']\!]\ tt, \sigma) \to (\texttt{while}[\![1, e, e']\!]\ e', \sigma)}$$

$$\textsc{WhileFalse}\quad \frac{}{(\texttt{while}[\![0, e, e']\!]\ ff, \sigma) \to ((), \sigma)}$$

$$\textsc{WhileLoop}\quad \frac{}{(\texttt{while}[\![1, e, e']\!]\ v, \sigma) \to (\texttt{while}[\![0, e, e']\!]\ e, \sigma)}$$

Instead of relying on a "shortcut" for the regular while loop, we could have added it directly to the syntax and use one step of reduction to transform from the "regular" to the other one. We can also add that this "immediate transformation" does one step of reduction too, but this requires complex rules (as we have to consider the case $\texttt{while}\ ff\ \{\cdots\}$ for example). And, in the end, we get an equivalent semantics to the one given above.

Question 3. *Add the sequence* `;` *to the language: extend the syntax. Give a translation rule that exploit the call-by-value semantics to transform the sequence into a term of the original language*

We can easily define a predicate "$\mathbf{x}$ does not freely occur in e", written $\mathbf{x} \notin e$ by induction on expression e:

$$\frac{\mathbf{x} \notin e_1 \qquad \mathbf{x} \notin e_2}{\mathbf{x} \notin e_1\ e_2} \qquad \frac{\mathbf{x} \notin e_1 \qquad \mathbf{x} \notin e_2 \qquad \mathbf{x} \notin e_3}{\mathbf{x} \notin \texttt{if}\ e_1\ \{\ e_2\ \}\ \texttt{else}\ \{\ e_3\ \}}$$

$$\frac{}{\mathbf{x} \notin \lambda \mathbf{x}.e} \qquad \frac{\mathbf{x} \notin e \qquad \mathbf{x} \neq \mathbf{y}}{\mathbf{x} \notin \lambda \mathbf{y}.e} \qquad \frac{}{\mathbf{x} \notin ()} \qquad \frac{}{\mathbf{x} \notin \ell}$$

and the rest of the induction rules follow this pattern.

Then, we can define the translation rule of the "**;**" operation:

$$\textsc{SeqTranslation}\ \frac{\mathbf{x} \notin e_2}{\mathcal{T}(e_1\ ;\ e_2) = \underbrace{\texttt{let}\ \mathbf{x} = e_1\ \texttt{in}\ e_2}_{(\lambda \mathbf{x}.e_2)\ e_1}}.$$

The rest of the translation function is done in the expected way, where we translate every sub expression, *e.g.*

$$\frac{}{\mathcal{T}(\texttt{if}\ e\ \{\ e_1\ \}\ \texttt{else}\ \{\ e_2\ \}) = \texttt{if}\ \mathcal{T}(e)\ \{\ \mathcal{T}(e_1)\ \}\ \texttt{else}\ \{\ \mathcal{T}(e_2)\ \}}.$$

Question 4. *Write a lambda expression* SUM *that takes an integer n as parameter and sums the n first integers with a while loop. You will need to use an accumulator inside the loop. What construct do you use for it?*

We can define:

$$\texttt{SUM}(n) \triangleq \left|\begin{array}{l}\texttt{let}\ \mathbf{s} = \texttt{new}\ 0\ \texttt{in}\\ \texttt{let}\ \mathbf{i} = \texttt{new}\ 1\ \texttt{in}\\ \texttt{while}\ \mathbf{!i} \leq n\ \{\\ \quad \mathbf{s} := \mathbf{!s} + \mathbf{!i}\ ;\\ \quad \mathbf{i} := \mathbf{!i} + 1\\ \}\ ;\\ \mathbf{!s}\end{array}\right.$$

Here, I assumed that we wanted the first n non-zero integers (if we want the first n integers including the zero, we can change the while condition to a strict inequality). Here, I use the reference-related constructs (creation, dereference and affectation), as well as the while loop, sequence and binary operations.

Question 5. *We wish to add tuples to our language. We add the syntax $(e_0, \ldots, e_{n-1})$ to create a tuple of size n and $e.k$ to access the k^{th} field. Give a definition by translation for*

tuples of three elements (construction and access).

When considering the "introduction rule of the triple", we start by evaluating the elements, and then we create a function that'll allow us to access the values of each element:

$$\textsc{TripleIntroTranslation}\ \frac{\mathbf{x} \notin b \qquad \mathbf{x} \notin a \qquad \mathbf{y} \notin a}{\mathscr{T}((a,b,c)) = \left|\begin{array}{l}\texttt{let } \mathbf{x} = \mathscr{T}(c) \texttt{ in}\\ \texttt{let } \mathbf{y} = \mathscr{T}(b) \texttt{ in}\\ \texttt{let } \mathbf{z} = \mathscr{T}(a) \texttt{ in}\\ (\lambda \mathbf{k}.\\ \quad \texttt{if } \mathbf{k} = 1 \texttt{ then } \{\ \mathbf{z}\ \} \texttt{ else } \{\\ \quad\quad \texttt{if } \mathbf{k} = 2 \texttt{ then } \{\ \mathbf{y}\ \} \texttt{ else } \{\\ \quad\quad\quad \texttt{if } \mathbf{k} = 3 \texttt{ then } \{\ \mathbf{x}\ \} \texttt{ else } \{\ (\ )\ \}\\ \quad\quad \}\\ \quad \}\\ )\end{array}\right.}.$$

We have to use `let`s as we only need to compute once the value of a, b and c, and then we simply get it directly (with no recomputation needed). (The order used to evaluate is OCaml's, but we may want to reorder them if we want a different semantics for **AssembLISP**.)

To access the k^{th} element of the triple, we simply evaluate it at k:

$$\textsc{TripleElimTranslation}\ \frac{k \in \{0,1,2\}}{\mathscr{T}(e.k) = \mathscr{T}(e)\ k}.$$

For this rule, it was ambiguous if k could be allowed to be *any* expression (like in Python, for example), or it'd necessarily be either $0, 1, 2$ ("like" in OCaml, for example).

For the rest of the translation function $\mathscr{T}$ we extend as expected: translating every subexpression not covered by the two previous cases.

1.2 Concurrent AssembLISP.

Question 6. *What is a final configuration if all computations are finished?*

The final configuration is (Θ, σ, Φ) with:

- $\mathrm{dom}(\Theta)$ is non-empty (and can contain more than one thread if other threads were spawned) ;
- for all $t \in \mathrm{dom}(\Theta)$, we have $\Theta(t) = [\,]$ (all tasks have been completed) ;
- for all $f \in \mathrm{dom}(\Phi)$, we have $\Phi(f) \neq \emptyset$ (all tasks have been completed) ;
- $\mathrm{dom}(\Phi)$ is non-empty (and can contain more than one future if other tasks were launched).

> **Note.** I do not consider that a program execution has finished where there is an irreducible task. This "all computations are finished" is ambiguous so I emailed Gabriel Radanne on 11/14 about this (and about question 11), but did not get any reply.

Question 7. *We define a program* `CONC` *as follows:*

$$\texttt{CONC} \triangleq \left|\begin{array}{l} \texttt{let}\,\mathbf{x} = \texttt{new}\,0\,\texttt{in} \\ \texttt{let}\,\mathbf{t_1} = \texttt{spawn}\,\texttt{in} \\ \texttt{let}\,\mathbf{t_2} = \texttt{spawn}\,\texttt{in} \\ \texttt{launch}\,\mathbf{t_1}\,\{\,\mathbf{x} := !\mathbf{x} + 1\,\}\,; \\ \texttt{launch}\,\mathbf{t_1}\,\{\,\mathbf{x} := !\mathbf{x} + 2\,\}\,; \\ \texttt{launch}\,\mathbf{t_2}\,\{\,\mathbf{x} := !\mathbf{x} + 1\,\}. \end{array}\right.$$

What are the possible outcomes of this computation? Sketch the derivation of the semantics for one of these outcomes. Explain what can happen in a couple of lines.

The possible outcomes are 4, 3, 2 or 1 depending on the ordering of the steps done in each thread. Let t_1 and t_2 be the thread identifiers "contained" in variables $\mathbf{t_1}$ and $\mathbf{t_2}$ respectively. Let also ℓ be the location "contained" in variable x. Then, one execution is shown in figure 1 (page 6).

To get a value of $1, 2, 3$ in $\sigma(\ell)$, we need to execute tasks in a way that splits the dereferencing and the affecting, as updating the value of $\sigma(\ell)$ can (and in that case, will) be based on an "outdated" value of $\sigma(\ell)$.

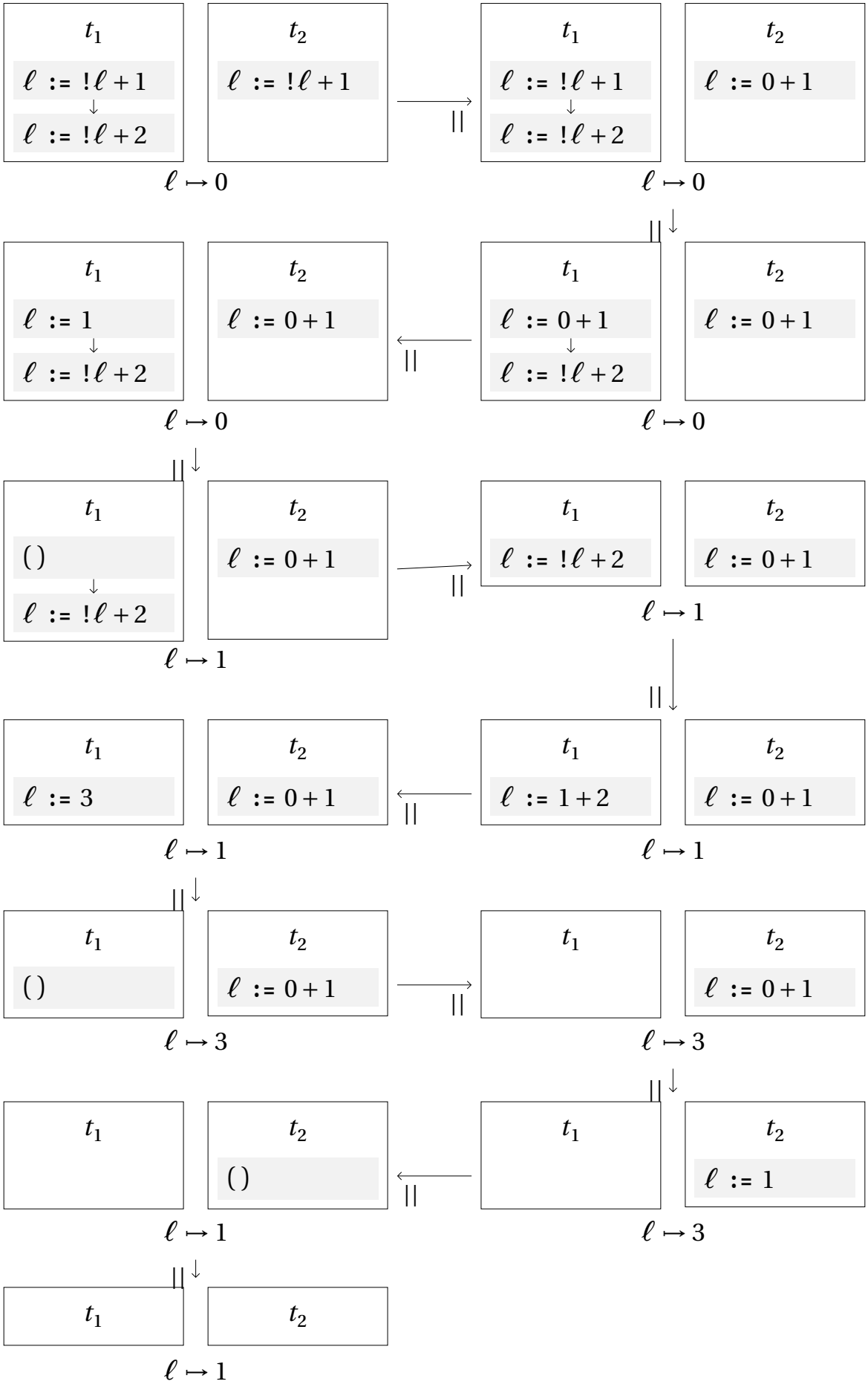

Figure 1 | *One execution of* CONC

Question 8. *Here is a simple program that also uses futures:*

$$\mathsf{FUTEX} \triangleq (\mathtt{get}\ (\mathtt{launch}\ \mathtt{spawn}\ \{\ 1+1\ \}))+1.$$

Derive the semantics for this simple example (you can omit trivial steps).

We have the following reductions

$$\begin{array}{c}
([t_0 \mapsto (f, (\mathtt{get}\ (\mathtt{launch}\ \mathtt{spawn}\ \{\ 1+1\ \}))+1) :: [\,]], [f \mapsto \emptyset], \emptyset) \\
\| \downarrow \\
([t_0 \mapsto (f, (\mathtt{get}\ (\mathtt{launch}\ t_1\ \{\ 1+1\ \}))+1) :: [\,]] \uplus [t_1 \mapsto [\,]], [f \mapsto \emptyset], \emptyset) \\
\| \downarrow \\
([t_0 \mapsto (f, (\mathtt{get}\ f')+1) :: [\,]] \uplus [t_1 \mapsto (f', 1+1) :: [\,]], [f \mapsto \emptyset] \uplus [f' \mapsto \emptyset], \emptyset) \\
\| \downarrow \\
([t_0 \mapsto (f, (\mathtt{get}\ f')+1) :: [\,]] \uplus [t_1 \mapsto (f', 2) :: [\,]], [f \mapsto \emptyset] \uplus [f' \mapsto \emptyset], \emptyset) \\
\| \downarrow \\
([t_0 \mapsto (f, (\mathtt{get}\ f')+1) :: [\,]] \uplus [t_1 \mapsto [\,]], [f \mapsto \emptyset] \uplus [f' \mapsto 2], \emptyset) \\
\| \downarrow \\
([t_0 \mapsto (f, 2+1) :: [\,]] \uplus [t_1 \mapsto [\,]], [f \mapsto \emptyset] \uplus [f' \mapsto 2], \emptyset) \\
\| \downarrow \\
([t_0 \mapsto (f, 3) :: [\,]] \uplus [t_1 \mapsto [\,]], [f \mapsto \emptyset] \uplus [f' \mapsto 2], \emptyset) \\
\| \downarrow \\
([t_0 \mapsto [\,]] \uplus [t_1 \mapsto [\,]], [f \mapsto 3] \uplus [f' \mapsto 2], \emptyset).
\end{array}$$

Question 9. *Write a term that spawns two threads, launches two tasks that compute* $\mathsf{SUM}(5)$ *and* $\mathsf{SUM}(4)$ *on the two threads, and retrieves the two results to compute and return* $\mathsf{SUM}(5) + \mathsf{SUM}(4)$

One such term is the following.

```
let f1 = launch spawn { SUM(5) } in
let f2 = launch spawn { SUM(4) } in
(get f1) + (get f2)
```

Question 10. *Write an invariant relating the value of a future in* Φ *and its occurrences in the rest of the configuration. There are in fact be three invariants: one corresponding to the thread that computes the value of the future, another one for the expressions that*

can refer to this future, for this one we use $e \in e'$ to state that the expression e is a sub-expression of e', finally there is one for future identifiers appearing in the store, or in other future values. We give the structure of the invariant, fill the blanks ? *as precisely as possible:*

for all runtime configurations (Θ, σ, Φ)*:*

$$(f, e) \in \Theta(t) \implies \Phi(f) = \text{?}$$
$$(f, e) \in \Theta(t) \land f' \in e \implies \text{?}$$
$$\sigma(\ell) = f \lor \Phi(f') = f \implies \text{?}.$$

We have the following invariants:

$$(f, e) \in \Theta(t) \implies \Phi(f) = \emptyset \tag{inv$_1$}$$
$$(f, e) \in \Theta(t) \land f' \in e \implies f' \in \mathrm{dom}(\Phi) \land (\Phi(f') = \emptyset \implies \exists t', e', (f', e') \in \Theta(t')) \tag{inv$_2$}$$
$$\sigma(\ell) = f \lor \Phi(f') = f \implies f \in \mathrm{dom}(\Phi) \land (\Phi(f) = \emptyset \implies \exists t, e, (f, e) \in \Theta(t)). \tag{inv$_3$}$$

The first one says that a thread is always associated with an element in the future map Φ. The second one says that if there is f' occurs inside of an expression e in a task of some thread, then this future corresponds to some value in Φ (or maybe $\emptyset$). Furthermore, if it is $\emptyset$ then there is some task associated with f' waiting to be computed. The last one is exactly the same as the one before, except that the future occurs in some reference or in some other future's result.

To prove this, we have to check the $cc \to_{||} cc'$ rules as they are the one dealing *directly* with the futures.

Local in t. We only need to check (inv$_2$) and (inv$_3$). No matter the "local" reduction applied, the invariant remains true (for RefWrite and RefNew, we apply the (inv$_2$), and for RefRead we apply (inv$_3$)).

Spawn in t. For (inv$_1$), (inv$_2$) and (inv$_3$), nothing changes as $\Theta(t') = []$ and σ, Φ are not changed.

Launch in t. For (inv_1), we add the new task (f', e) to $\Theta(t')$ and we have that $\Phi(t') = \emptyset$. For (inv_2) for t', we use (inv_2) for t and get the required result (some future $f' \in e$ can be passed to the new task). For the last one, nothing changes.

Resolve in t. For (inv_1) and (inv_2), we have $(f, v) \notin \Theta(t)$ so not much to prove here. For (inv_3), we apply (inv_2) to task (f, v).

Get in t. For (inv_1), nothing changes. For (inv_2), we apply (inv_3) to f'. For (inv_3), nothing changes here.

We also need to check that (inv_1), (inv_2) and (inv_3) hold initially. They do hold as futures cannot appear in the "source code", but only at execution.

Question 11 (Difficult). *We define a program* LOOP *as follows:*

$$\text{LOOP} \triangleq \left|\begin{array}{l} \texttt{let } \mathbf{n} = \texttt{new } 1 \texttt{ in} \\ \texttt{let } \mathbf{t} = \texttt{spawn in} \\ \texttt{let } \mathbf{run} = \lambda \mathbf{f}.(\,!\mathbf{n} := 2 \times !\mathbf{n}\,;\, \texttt{launch } \mathbf{t}\,\{\,\mathbf{f}\,\mathbf{t}\,\})\,\texttt{in} \\ \texttt{launch } \mathbf{t}\,\{\,\mathbf{run}\,\mathbf{run}\,\}\,; \\ !\mathbf{n} \end{array}\right.$$

During the execution of this program, the thread $\mathbf{t}$ launches *a task to itself. Why is this not possible given the current reduction rules?*

Propose a new reduction rule which allows this program to run fully.

What does the program do with your rule?

Recall the inference rule responsible for handling launches:

$$\textsc{Launch in } t \quad \frac{t \neq t' \qquad f' \notin \mathrm{dom}(\Phi)}{\begin{array}{c}\Big(\big[t \mapsto (f, C[\texttt{launch } t' \{\, e \,\}]) :: e\ell\big] \uplus \big[t' \mapsto e\ell'\big] \uplus \Theta, \quad \sigma, \quad \Phi\Big) \\ \Big\| \downarrow \\ \Big(\big[t \mapsto (f, C[f']) :: e\ell\big] \uplus \big[t' \mapsto e\ell' :: (f', e)\big] \uplus \Theta, \quad \sigma, \quad \Phi[f' \mapsto \emptyset]\Big)\end{array}}.$$

If we have, in a thread t (in the first task of t), an expression launch t $\{\, e \,\}$, then we cannot apply this rule, and thus it is irreducible. Thus, we add a new rule:

LAUNCH SELF IN t

$$\frac{f' \notin \mathrm{dom}(\Phi)}{\begin{array}{c}\Big(\big[t \mapsto (f, C[\texttt{launch}\ t\ \{\ e\ \}]) :: e\ell\big] \uplus \Theta,\quad \sigma,\quad \Phi\Big) \\ \Big\| \downarrow \\ \Big(\big[t \mapsto (f, C[f']) :: e\ell :: (f', e)\big] \uplus \Theta,\quad \sigma,\quad \Phi[f' \mapsto \emptyset]\Big)\end{array}}.$$

This rules allows that, when a thread t executes `launch` t `{` e `}`, then it adds this task at the end of the list of tasks to execute.

> **Note.** I do not consider that a program execution has ran fully where there is an irreducible task like $t\ t$. This is even more ambiguous as "runs fully" could be interpreted as "run and terminates" (that's what I imagined at first). So, I emailed Gabriel Radanne on 11/14 about this (and about question 6), but did not get any reply.
>
> I'm thus updating the program `LOOP` to be the following.
>
> ```
> LOOP ≜ let n = new 1 in
>        let t = spawn in
>        let run = λf.(n := 2 × !n ; launch t { f t f }) in
>        launch t { run run } ;
>        !n
> ```
>
> It runs fully (at each step there is a reduction) but does not terminate.

With the new `LOOP` program, and the added rule, this program runs fully and will return 2^{k+1} where k is the number of tasks completed in the thread contained in variable **t** (sometimes 2^{k+2} if the thread already executed **n** := 2 × !**n** (modulo replacing the variables with the locations)). Indeed, this program starts by launching a thread t, and then, for each task, it'll doubles **n**'s content and spawn one of itself (*i.e.* another task **run run**), and repeating the doubling step over and over.

2 Erlisp: An object oriented language for distributed computing.

Question 12. *Complete the rules* Let *and* AttrWrite.

We can write:

$$\textsc{Let}\ \frac{\mathbf{x} \neq \mathbf{a}}{\mathscr{E}_{\mathbf{a}}(\texttt{let}\ \mathbf{x} = e\ \texttt{in}\ e') = \texttt{let}\ \mathbf{x} = \mathscr{E}_{\mathbf{a}}(e)\ \texttt{in}\ \mathscr{E}_{\mathbf{a}}(e')},$$

and

$$\textsc{AttrWrite}\ \frac{}{\mathscr{E}_{\mathbf{a}}(\mathbf{z} \leftarrow e) = \mathbf{z}\ \mathbf{:=}\ \mathscr{E}_{\mathbf{a}}(e)}.$$

Personally, I'd also require that $\mathbf{z} = \mathbf{a}$ (by directly substituting in the rule) as, that way, referencing an undefined attribute would lead to a compilation error, and not an evaluation error.

Question 13. *Consider the following program.*

```
Counter = {
    state = 0
    add(x) = state ← state + x ; state
} ;
let x = Counter.add(10) in
let y = Counter.add(15) in
(get x) + (get y)
```

Compile this program and execute it (you can skip trivial steps).

After compiling, we end up with the following **AssembLISP** program:

```
let Counter =
  let state = new 0 in
  (spawn, λself.λx.state := !state + x ; !state)
in
let x =
  let x_self = Counter in
  let x_arg = 10 in
  launch x_self.0 { x_self.1 x_self x_arg }
in
let y =
  let x_self = Counter in
  let x_arg = 15 in
  launch x_self.0 { x_self.1 x_self x_arg }
in
(get x) + (get y).
```

After some steps of concurrent reduction (starting with $\sigma = \emptyset$ and $\Phi = [f \mapsto \emptyset]$), we get

$t_0 : f,$

```
let Counter =                          σ = [ℓ ↦ 0]
  (t, λself.λx.ℓ := !ℓ + x ; !ℓ)       Φ = [f ↦ ∅]
in
let x =
  let x_self = Counter in
  let x_arg = 10 in
  launch x_self.0 { x_self.1 x_self x_arg }
in
let y =
  let x_self = Counter in
  let x_arg = 15 in
  launch x_self.0 { x_self.1 x_self x_arg }
in
(get x) + (get y).
```

then

$$
\begin{array}{rl}
 & \left|\begin{array}{l}
\texttt{let}\ \mathbf{x} = \\
\quad \texttt{let}\ \mathbf{x_{self}} = (t, \lambda\mathbf{self}.\lambda\mathbf{x}.\ell\ \texttt{:=}\ !\ell + \mathbf{x}\ \texttt{;}\ !\ell)\ \texttt{in} \\
\quad \texttt{let}\ \mathbf{x_{arg}} = 10\ \texttt{in} \\
\quad \texttt{launch}\ \mathbf{x_{self}}.0\ \{\ \mathbf{x_{self}}.1\ \mathbf{x_{self}}\ \mathbf{x_{arg}}\ \} \\
\texttt{in} \\
\texttt{let}\ \mathbf{y} = \\
\quad \texttt{let}\ \mathbf{x_{self}} = (t, \lambda\mathbf{self}.\lambda\mathbf{x}.\ell\ \texttt{:=}\ !\ell + \mathbf{x}\ \texttt{;}\ !\ell)\ \texttt{in} \\
\quad \texttt{let}\ \mathbf{x_{arg}} = 15\ \texttt{in} \\
\quad \texttt{launch}\ \mathbf{x_{self}}.0\ \{\ \mathbf{x_{self}}.1\ \mathbf{x_{self}}\ \mathbf{x_{arg}}\ \} \\
\texttt{in} \\
(\texttt{get}\ \mathbf{x}) + (\texttt{get}\ \mathbf{y}).
\end{array}\right. \\
t_0 : f, & \qquad \sigma = [\ell \mapsto 0] \\
 & \qquad \Phi = [f \mapsto \emptyset]
\end{array}
$$

and then

$$
\begin{array}{rl}
 & \left|\begin{array}{l}
\texttt{let}\ \mathbf{x} = f'\ \texttt{in} \\
\texttt{let}\ \mathbf{y} = \\
\quad \texttt{let}\ \mathbf{x_{self}} = (t, \lambda\mathbf{self}.\lambda\mathbf{x}.\ell\ \texttt{:=}\ !\ell + \mathbf{x}\ \texttt{;}\ !\ell)\ \texttt{in} \\
\quad \texttt{let}\ \mathbf{x_{arg}} = 15\ \texttt{in} \\
\quad \texttt{launch}\ \mathbf{x_{self}}.0\ \{\ \mathbf{x_{self}}.1\ \mathbf{x_{self}}\ \mathbf{x_{arg}}\ \} \\
\texttt{in} \\
(\texttt{get}\ \mathbf{x}) + (\texttt{get}\ \mathbf{y}),
\end{array}\right. \\
t_0 : f, & \qquad \sigma = [\ell \mapsto 0] \\
 & \qquad \Phi = [f \mapsto \emptyset, f' \mapsto \emptyset] \\
t : f', & \left|\ \ell\ \texttt{:=}\ !\ell + 10\ \texttt{;}\ !\ell\right.
\end{array}
$$

and then

$$
\begin{array}{rl}
t_0 : f, & \left|\begin{array}{l}
\texttt{let}\ \mathbf{y} = f''\ \texttt{in} \\
(\texttt{get}\ f') + (\texttt{get}\ \mathbf{y})
\end{array}\right. \\
 & \qquad \sigma = [\ell \mapsto 0] \\
 & \qquad \Phi = [f \mapsto \emptyset, f' \mapsto \emptyset, f'' \mapsto \emptyset] \\
t : f', & \left|\ \ell\ \texttt{:=}\ !\ell + 10\ \texttt{;}\ !\ell\right. \quad :: \quad f'', \left|\ \ell\ \texttt{:=}\ !\ell + 15\ \texttt{;}\ !\ell\right.
\end{array}
$$

after that we can reduce the two “add” tasks

$$
\begin{array}{rl}
t_0 : f, & \left|\ (\texttt{get}\ f') + (\texttt{get}\ f'')\right. \\
 & \qquad \sigma = [\ell \mapsto 10] \\
 & \qquad \Phi = [f \mapsto \emptyset, f' \mapsto 10, f'' \mapsto \emptyset] \\
t : f'', & \left|\ \ell\ \texttt{:=}\ !\ell + 15\ \texttt{;}\ !\ell\right.
\end{array}
$$

then

$$\begin{array}{l} t_0 : f, \\ t : [\,] \end{array} \left| \begin{array}{l} (\texttt{get}\ f') + (\texttt{get}\ f'') \end{array} \right. \qquad \begin{array}{r} \sigma = [\ell \mapsto 25] \\ \Phi = [f \mapsto \emptyset, f' \mapsto 10, f'' \mapsto 25] \end{array}$$

and we can finally conclude with the last four reductions

$$\begin{array}{l} t_0 : f, \\ t : [\,] \end{array} \left| \begin{array}{l} 10 + 25 \end{array} \right. \qquad \begin{array}{r} \sigma = [\ell \mapsto 25] \\ \Phi = [f \mapsto \emptyset, f' \mapsto 10, f'' \mapsto 25] \end{array}$$

and get to the last configuration

$$\begin{array}{ll} t_0 : [\,] & \sigma = [\ell \mapsto 25] \\ t : [\,] & \Phi = [f \mapsto 35, f' \mapsto 10, f'' \mapsto 25] \end{array} .$$

Question 14. *Here is an alternative rule for* METH. *Why is it incorrect? Illustrate with examples.*

$$\textsc{meth-incorrect}\ \frac{e_0' = \mathscr{E}_{\mathbf{a}}(e_0) \qquad e_1' = \mathscr{E}_{\mathbf{a}}(e_1)}{\mathscr{E}_{\mathbf{a}}(e_0.\mathrm{meth}(e_1)) = \texttt{launch}\ e_0'.0\ \{\ e_0'.1\ e_0'\ e_1'\ \}}$$

The rule METH-INCORRECT is incorrect as, during a reduction, the expression e_0' will be evaluated three times instead of once. For example, the result of this **Erlisp** program

```
Counter = (* seen in question 13 *) ;
test = {
  a = 0
  meth(x) = ( )
} ;
(Counter.add(10) ; test).meth(1) ;
get (Counter.add(0))
```

.

We would expect to have a result of 10 but we end up with a result of 30 as the **Counter**.add(10) method evaluated three times, once in the original thread, and then twice in **test**'s thread.

Question 15. *We now wish to support an arbitrary number of attributes. Expand the rules* NewObj, AttrRead, AttrWrite *for this purpose.*

We will write $\mathscr{E}_{\mathbf{a};\mathbf{a_1},\ldots,\mathbf{a_n}}(-)$ for this extended function. The idea is to say that variable $\mathbf{a}$ (in **AssembLISP**) will contain the data of **Erlisp** attributes $\mathbf{a_1},\ldots,\mathbf{a_n}$. With this in mind, we edit the translation rules:

NewObj

$$\frac{\mathbf{a} \notin e_m}{\mathscr{P}\left(\begin{array}{l}\mathbf{o} = \{\\ \quad \mathbf{a_1} = e_1\\ \quad \mathbf{a_2} = e_2\\ \quad\quad \vdots\\ \quad \mathbf{a_n} = e_n\\ \quad \mathrm{meth}(\mathbf{x}) = e_m\\ \};\\ p\end{array}\right) = \left|\begin{array}{l}\texttt{let } \mathbf{o} =\\ \quad \texttt{let } \mathbf{a} = (\\ \qquad \texttt{new } \mathscr{E}_{\emptyset}(e_1),\\ \qquad \texttt{new } \mathscr{E}_{\emptyset}(e_2),\\ \qquad\quad \vdots\\ \qquad \texttt{new } \mathscr{E}_{\emptyset}(e_n)\\ \quad) \texttt{ in}\\ \quad (\texttt{spawn}, \lambda\mathbf{self}.\lambda\mathbf{x}.\mathscr{E}_{\mathbf{a};\mathbf{a_1},\ldots,\mathbf{a_n}}(e_m))\\ \texttt{in } \mathscr{P}(p)\end{array}\right.$$

AttrRead

$$\frac{}{\mathscr{E}_{\mathbf{a};\mathbf{a_1},\ldots,\mathbf{a_n}}(\mathbf{a_i}) = !(\mathbf{a}.i)}$$

and finally

AttrWrite

$$\frac{}{\mathscr{E}_{\mathbf{a};\mathbf{a_1},\ldots,\mathbf{a_n}}(\mathbf{a_i} \leftarrow e) = (\mathbf{a}.i) := \mathscr{E}_{\mathbf{a};\mathbf{a_1},\ldots,\mathbf{a_n}}(e)}.$$

For all the other rules, we extend them in the obvious way: writing $\mathscr{E}_{\mathbf{a};\mathbf{a_1},\ldots,\mathbf{a_n}}$ instead of $\mathscr{E}_{\mathbf{a}}$. (For rule Var, we use **AssembLISP**'s variable $\mathbf{a}$ and not **Erlisp**'s variable $\mathbf{a_1},\ldots,\mathbf{a_n}$ in the precondition $\mathbf{a} \neq \mathbf{x}$.

2.1 Multiple methods with dynamic dispatch.

Question 16. *Write an* ***Erlisp*** *object with a counter attribute and three methods with unit arguments:* **incr**, *which increments the counter and returns the new value,* **decr**, *which decrements the counter and returns the new value, and* **reset**, *which resets the counter to zero and returns unit.*

We use the following **Erlisp** object:

```
Counter = {
  counter = 0
  incr(x) = (counter ← counter + 1 ; state)
  decr(x) = (counter ← counter − 1 ; state)
  reset(x) = (counter ← 0 ; ())
}.
```

Question 17. *Write an **AssembLISP** program with a function* **counter_method** *taking as argument a label among* **'incr**, **'decr** *and* **'reset** *and implementing the same counter semantics as the previous question.*

We use the following **AssembLISP** partial program:

```
let counter_method =
  let t = spawn in
  let counter = new 0 in
  λl.(
    if l = 'incr then spawn t { counter := counter + 1 ; !counter }
    else if l = 'decr then spawn t { counter := counter − 1 ; !counter }
    else if l = 'reset then spawn t { counter := 0 ; () }
    else ()
  )
```

In this program, we omitted curly braces around the content's of the `if` and `else` expressions, to simplify the syntax (where there is no ambiguity).

Question 18. *Extend* NewObj *and* Meth *to support multiple methods in an object. You can use "…" for part of the rules that are identical to previous answers.*

For this question, we suppose having a "predicable" way of transforming a method's (*e.g.* incr) name into a symbol (*e.g.* **'incr**). We also suppose that no two different methods (inside the same object) lead to the same symbol. Here, we will only change the font's weight to bold with the added uptick.

METH

$$\frac{\mathbf{x_{self}} \notin e \qquad \mathbf{x_{arg}} \notin e \qquad \mathbf{x_{self}} \neq \mathbf{a} \qquad \mathbf{x_{arg}} \neq \mathbf{a}}{\mathscr{E}_{\mathbf{a};\mathbf{a_1},\ldots,\mathbf{a_n}}(e.\mathrm{meth}(e')) = \left|\begin{array}{l} \texttt{let}\ \mathbf{x_{self}} = \mathscr{E}_{\mathbf{a};\mathbf{a_1},\ldots,\mathbf{a_n}}(e)\ \texttt{in} \\ \texttt{let}\ \mathbf{x_{arg}} = \mathscr{E}_{\mathbf{a};\mathbf{a_1},\ldots,\mathbf{a_n}}(e')\ \texttt{in} \\ \texttt{launch}\ \mathbf{x_{self}}.0\ \{ \\ \quad \mathbf{x_{self}}.1\ \mathbf{x_{self}}\ {}'\mathbf{meth}\ \mathbf{x_{arg}} \\ \} \end{array}\right.}$$

NEWOBJ

$$\frac{\mathbf{a} \notin m_1 \qquad \mathbf{a} \notin m_2 \qquad \ldots \qquad \mathbf{a} \notin m_k}{\mathscr{P}\left(\begin{array}{l} \mathbf{o} = \{ \\ \quad \mathbf{a_1} = e_1 \\ \quad \mathbf{a_2} = e_2 \\ \qquad \vdots \\ \quad \mathbf{a_n} = e_n \\ \quad \mathrm{meth}_1(\mathbf{x_1}) = m_1 \\ \quad \mathrm{meth}_2(\mathbf{x_2}) = m_2 \\ \qquad \vdots \\ \quad \mathrm{meth}_k(\mathbf{x_k}) = m_k \\ \}\ ; \\ p \end{array}\right) = \left|\begin{array}{l} \texttt{let}\ \mathbf{o} = \\ \quad \texttt{let}\ \mathbf{a} = (\\ \qquad \texttt{new}\ \mathscr{E}_{\emptyset}(e_1), \\ \qquad \texttt{new}\ \mathscr{E}_{\emptyset}(e_2), \\ \qquad\ \vdots \\ \qquad \texttt{new}\ \mathscr{E}_{\emptyset}(e_n) \\ \quad)\ \texttt{in} \\ \quad (\texttt{spawn}, \lambda\mathbf{self}.\lambda\mathbf{l}. \\ \qquad \texttt{if}\ \mathbf{l} = {}'\mathbf{meth_1}\ \texttt{then} \\ \qquad\quad \lambda\mathbf{x_1}.\, \mathscr{E}_{\mathbf{a};\mathbf{a_1},\ldots,\mathbf{a_n}}(m_1) \\ \qquad \texttt{else if}\ \mathbf{l} = {}'\mathbf{meth_2}\ \texttt{then} \\ \qquad\quad \lambda\mathbf{x_2}.\, \mathscr{E}_{\mathbf{a};\mathbf{a_1},\ldots,\mathbf{a_n}}(m_2) \\ \qquad\ \vdots \\ \qquad \texttt{else if}\ \mathbf{l} = {}'\mathbf{meth_k}\ \texttt{then} \\ \qquad\quad \lambda\mathbf{x_k}.\, \mathscr{E}_{\mathbf{a};\mathbf{a_1},\ldots,\mathbf{a_n}}(m_k) \\ \qquad \texttt{else}\ \lambda\mathbf{x}.\ () \\ \quad) \\ \texttt{in}\ \mathscr{P}(p) \end{array}\right.}$$

2.2 Encapsulation.

Question 19. *Let us consider the following object definition:*

```
pass_checker = {
    secret = 42
    validate(s) = (s = secret)
}.
```

Can the **secret** *attribute be written or read by any other object? What part of the translation ensures that?*

No, it cannot. In deed, this is ensured by declaring the **secret** *inside* of the `let`-binding: a partial translation (using the "one attribute/one method" translation, to remove all the added clutter with multiple attributes and methods) of the **Erlang** program would be

```
let pass_checker =
    let secret = new 42 in
    (spawn, λself.λs.(s = !secret)),
```

and after the "`new 42`" is evaluated into $\ell \in \mathbb{L}$, the substitution only occurs in

$$\text{“(spawn, } \lambda \mathbf{self}.\lambda \mathbf{s}.(\mathbf{s} = \,!\,\underbrace{\mathbf{secret}}_{\downarrow\ \ell})\text{)”}$$

and nowhere else, so this is the only part of the code that knows the location where the secret key is stored.

Question 20. *We are interested in formalizing this encapsulation property. For this purpose, we consider what happens if the memory is filled with a new inert value, $\bot$, except for the object being executed.*

Given a program ***Erlisp*** p*, We consider an execution of its compiled program* $s_0 = \mathscr{P}(p)$*. Let* $\mathbf{o}$ *an object definition in* p*, We denote* $t_{\mathbf{o}}$ *its thread (resulting from the* `spawn` *instruction) and* $A_{\mathbf{o}}$ *the set of memory location containing its attributes during the execution of* s_0*.*

Let $\Theta, \Theta'_1, \sigma_1, \sigma'_1, \Phi, \Phi'_1, e\ell, e\ell'_1$ *such that*

$$\left([t_{\mathbf{o}} \mapsto e\ell] \uplus \Theta, \sigma_1, \Phi\right) \to_{||} \left([t_{\mathbf{o}} \mapsto e\ell'_1] \uplus \Theta'_1, \sigma'_1, \Phi'_1\right)$$

where $t_{\mathbf{o}}$ *is the thread in which the reduction is performed,* i.e. *the rule applied is of the form* ? IN $t_{\mathbf{o}}$ (e.g. LOCAL IN $t_{\mathbf{o}}$).

Let σ_2 *such that* $\sigma_2(\ell) = \begin{cases} \sigma_1(\ell) & \text{if } \ell \in A_{\mathbf{o}} \\ \bot & \text{otherwise.} \end{cases}$

Then, there exists $\sigma'_2, \Phi'_2, e\ell'_2$ *such that*

$$\left([t_{\mathbf{o}} \mapsto e\ell] \uplus \Theta, \sigma_2, \Phi\right) \to_{||} \left([t_{\mathbf{o}} \mapsto e\ell'_2] \uplus \Theta'_2, \sigma'_2, \Phi'_2\right).$$

We try to relate $\Theta'_2, \sigma'_2, \Phi'_2, e\ell'_2$ *with that happens in the first reduction that leads to* $e\ell'_1$.

1. *In each of the following three cases, relate* $\Theta'_2, \sigma'_2, \Phi'_2, e\ell'_2$ *with the other variables and explain informally what happens:*
   a) *the reduction corresponds to a method call;*
   b) *the reduction corresponds to a* `get`*;*
   c) *the reduction corresponds to a read or write to an attribute.*
2. *Give the complete encapsulation property that characterizes* $\Theta'_2, \sigma'_2, \Phi'_2, e\ell'_2$ *independently of the reduction applied.*

In the following, for a store σ, we will write $\sigma[\bot]$ for a store defined by

$$\sigma[\bot](\ell) := \begin{cases} \sigma(\ell) & \text{if } \ell \in A_{\mathbf{o}} \\ \bot & \text{otherwise.} \end{cases}$$

With this notation, $\sigma_2 = \sigma_1[\bot]$.

1. We consider three cases.
   a) For a method call, we do a `launch` operation, starting the method call on some (potentially different) thread. If the method call is

on object $\mathbf{o}' \neq \mathbf{o}$, then

$$\begin{cases} \Phi'_2 = \Phi[f \mapsto \emptyset] = \Phi'_1 \\ \Theta'_2 = [t_{\mathbf{o}'} \mapsto e\ell' :: (f, e)] \uplus \Theta' = \Theta'_1 \\ e\ell'_2 = e\ell' = e\ell'_1 \\ \sigma'_2 = \sigma_2 = \sigma_1[\bot] = \sigma'_1[\bot] \end{cases}$$

where $\Theta =: [t_{\mathbf{o}'} \mapsto e\ell'] \uplus \Theta'$ and $f \notin \mathrm{dom}(\Phi)$.[a] If the method call is on object $\mathbf{o}$, then

$$\begin{cases} \Phi'_2 = \Phi[f \mapsto \emptyset] = \Phi'_1 \\ \Theta'_2 = \Theta = \Theta'_1 \\ e\ell'_2 = e\ell' :: (e, f) = e\ell'_1 \\ \sigma'_2 = \sigma_2 = \sigma_1[\bot] = \sigma'_1[\bot], \end{cases}$$

where $f \notin \mathrm{dom}(\Phi)$.

b) For a `get`, it amounts to an **AssembLISP** `get`, so we have:

$$\begin{cases} \Phi'_2 = \Phi = \Phi'_1 \\ \Theta'_2 = \Theta = \Theta'_1 \\ e\ell'_2 = (f_0, C[\Phi(f)]) :: e\ell' = e\ell'_1 \\ \sigma'_2 = \sigma_2 = \sigma'_1[\bot], \end{cases}$$

where $e\ell = (f_0, C[\texttt{get}\ f]) :: e\ell'$.

c) For reading an attribute, we have that

$$\begin{cases} \Phi'_2 = \Phi = \Phi'_1 \\ \Theta'_2 = \Theta = \Theta'_1 \\ e\ell'_2 = (f_0, C[\sigma_2(\ell)]) :: e\ell' = e\ell'_1 \\ \sigma'_2 = \sigma_2 = \sigma'_1[\bot], \end{cases}$$

where $e\ell = (f_{\mathbf{o}}, C[!\ell]) :: e\ell'$. Note that, because the compi-

[a] The equality "$\Phi'_2 = \Phi'_1$" is set assuming we will choose the same f in similar settings (*e.g.* with some kind of deterministic choice). In the following, we will always assume that.

lation only uses references to handle attributes and doesn't let **Erlisp** programs use them directly, we already know $\ell \in A_{\mathbf{o}}$ and so $\sigma_2(\ell) = \sigma_1(\ell)$ (technically, this is part of the encapsulation property).

For writing an attribute, we have that

$$\begin{cases} \Phi'_2 = \Phi = \Phi'_1 \\ \Theta'_2 = \Theta = \Theta'_1 \\ e\ell'_2 = (f_0, C[(\,)]) :: e\ell' = e\ell'_1 \\ \sigma'_2 = \sigma_2[\ell \mapsto v] = \sigma'_1[\bot], \end{cases}$$

where $e\ell = (f_{\mathbf{o}}, C[\ell \texttt{ := } v]) :: e\ell'$ and again $\ell \in A_{\mathbf{o}}$ (it is again part of the encapsulation property).

2. After analyzing those cases, we generalize to the following property: For every object $\mathbf{o}$ in p,

   for any reduction

   $$(\Theta, \sigma, \Phi) \to_{||} (\Theta', \sigma', \Phi')$$

   where the reduction is done in $t_{\mathbf{o}}$, then we have a reduction

   $$(\Theta, \sigma[\bot], \Phi) \to_{||} (\Theta, \sigma'[\bot], \Phi')$$

   where the reduction is also done in $t_{\mathbf{o}}$.

To prove this property, we will require that the choice for a new future $f \notin \mathrm{dom}(\Phi)$ is done leads to the same result in both reductions.

Intuitively, this property expressed this way tells us that, if we execute something in a thread $t_{\mathbf{o}}$, then we can only use/update the memory allocated to $\mathbf{o}$ (for the attributes), and that the other values do not matter to this reduction.

Question 21 (Difficult). *Where is evaluated the code computing the initial value of attributes? To perfect our encapsulation, we wish to compute this code in the object's thread as well. Adjust the compilation and state explicitly which instructions are now executed in the object thread.*

The initial value is computed by the "original" thread (the one containing the whole program at the beginning of the execution). We can change the NewObj with the following. We only need to make sure that variable **t** is not captured inside any e_i or m_i as it'd override some object **t** defined before.

NewObj

$$\frac{\forall i, \mathbf{a} \notin m_i \qquad \forall i, \mathbf{t} \notin m_i \qquad \forall j, \mathbf{t} \notin e_j}{\mathscr{P}\left(\begin{array}{l} \mathbf{o} = \{ \\ \quad \mathbf{a_1} = e_1 \\ \quad \mathbf{a_2} = e_2 \\ \quad\quad \vdots \\ \quad \mathbf{a_n} = e_n \\ \quad \mathrm{meth}_1(\mathbf{x_1}) = m_1 \\ \quad \mathrm{meth}_2(\mathbf{x_2}) = m_2 \\ \quad\quad \vdots \\ \quad \mathrm{meth}_k(\mathbf{x_k}) = m_k \\ \}\ ; \\ p \end{array}\right) = \left|\begin{array}{l} \texttt{let } \mathbf{o} = \\ \quad \texttt{let } \mathbf{t} = \texttt{spawn in} \\ \quad \texttt{let } \mathbf{a} = \texttt{get launch } \mathbf{t} \texttt{ \{ (} \\ \qquad \texttt{new } \mathscr{E}_{\emptyset}(e_1), \\ \qquad \texttt{new } \mathscr{E}_{\emptyset}(e_2), \\ \qquad\quad \vdots \\ \qquad \texttt{new } \mathscr{E}_{\emptyset}(e_n) \\ \quad \texttt{) \} in} \\ \quad (\mathbf{t}, \lambda \mathbf{self}.\lambda \mathbf{l}. \\ \qquad \texttt{if } \mathbf{l} = \texttt{'}\mathbf{meth_1} \texttt{ then} \\ \qquad\quad \lambda \mathbf{x_1}.\, \mathscr{E}_{\mathbf{a};\mathbf{a_1},\ldots,\mathbf{a_n}}(m_1) \\ \qquad \texttt{else if } \mathbf{l} = \texttt{'}\mathbf{meth_2} \texttt{ then} \\ \qquad\quad \lambda \mathbf{x_2}.\, \mathscr{E}_{\mathbf{a};\mathbf{a_1},\ldots,\mathbf{a_n}}(m_2) \\ \qquad\quad \vdots \\ \qquad \texttt{else if } \mathbf{l} = \texttt{'}\mathbf{meth_k} \texttt{ then} \\ \qquad\quad \lambda \mathbf{x_k}.\, \mathscr{E}_{\mathbf{a};\mathbf{a_1},\ldots,\mathbf{a_n}}(m_k) \\ \qquad \texttt{else } \lambda \mathbf{x}.\, () \\ \quad) \\ \texttt{in } \mathscr{P}(p) \end{array}\right.}$$