15-791 ATPL
Week 7 Notes

Sonya Simkin

March 11, 2024

1 Motivation

Suppose we have some type `seq` representing a sequence of elements. We might expect the following to be true:

$$\texttt{msort} \doteq \texttt{isort} \in \texttt{seq} \to \texttt{seq}$$

where `msort` is merge sort and `isort` is insertion sort. With our running definition of exact equality, this should mean that `msort` and `isort` produce equal sequences on equal inputs. While this is true, there is a notable difference between the two functions: `msort` is polylogarithmic in the length of the input, whereas `isort` is quadratic.

We would want to be able to express this difference in cost between the functions. However, if these functions are exactly equal by our definition, how could one have a property that the other doesn't? The key to making this distinction will be found in the splitting of cost and behavior into two separate categories, where cost is considered to be an *intensional* property and behavior is considered to be an *extensional* property. This categorization will lend itself to the idea of distinct *phases* under which we can formalize the notion of exact equality, which we can then use to express the extensional likeness and intensional difference between `msort` and `isort`.

2 Cost

To come up with a way to formalize a notion of cost in our system, we should begin by considering different notions of cost.

One of the "textbook" notions of cost is the RAM (random-access machine) cost model, where cost is measured by the number of instruction steps to completion. Since this cost model evaluates memory accesses and instruction calls, it is highly dependent on how the code is compiled rather than the properties of the program itself. This causes a rift between code and data, which shouldn't be the case if cost is something we consider to be intrinsic to a particular piece of code.

Instead, we can try to express the cost through the dynamics of a language. For example, we could state that

$$\texttt{isort}(s) \longmapsto^{|s|^2} s_0$$

which would indicate that the expression $\texttt{isort}(s)$ steps to a sorted permutation of s (namely s_0) with cost $|s|^2$. Unlike the RAM model, the cost is now independent of compilation, and is instead determined solely by the code.

The dynamics approach is closer to what we would want from a cost model, in that we are lifting cost away from the machine level and up to the level of types and expressions. Bringing cost to the language level facilitates two important properties:

1. *Abstraction*

   There is no unilateral definition of cost for every algorithm – cost may be determined by the number of comparisons, the number of edges inserted into a graph, or bounded by the number of recursive calls. To account for this, we would want our notion of cost to be abstract enough to be adapted to each of these definitions.

2. *Composition*

   Types mediate composition of programs – given a program, we are able to write a specification for it, which then allows us to substitute that program in wherever specified. If we lift cost to the level of our language, we then are able to compose programs *and* their costs.

With this in mind, we can start to come up with some way to account for cost at the language level.

2.1 Profiling

Our first attempt at cost accounting is through *profiling*. For example, if we wanted to evaluate the cost of our code by counting the number of comparisons made, we could adjust our comparison function in the following way:

```
fun compare (x, y) =
  count++;
  ...
```

Now, upon calling `compare`, some global counter called `count` is updated. By the end of the program, the value stored in `count` should give us our program cost.

While this does achieve our goal of cost accounting, there are a few issues with this approach. One issue is that the existence of some global cost counter could influence behavior – there is nothing stopping the programmer from arbitrarily updating the counter or using the counter value for computations, which

is problematic if the counter is only meant to keep track of cost. Additionally, cost accounting is a piece of data that is internal to the developer of the code – when delivering the code to a client, a developer would have to go through and remove each of these cost counters, which is a tedious task (especially if the behavior depends on the counter, as in the previous issue). We would like to do profiling in a way that does not interfere with the code we write, and can be easily "switched off" if we do not need to consider cost.

2.2 Profiling with Erasure

To counter the issues faced by regular profiling, we introduce the idea of *profiling with erasure*, where we utilize *phase propositions* to indicate whether or not the cost of steps are taken into consideration.

To continue with our sorting example, we can introduce a proposition EXT to represent whether the intensional properties of the functions are being taken into account. If the proposition EXT holds, then we can consider ourselves to be in the "extensional" phase, which means we suppress any intensional properties (e.g. cost) in our considerations for equality. That is, we would say that

$$\mathsf{EXT} \vdash \texttt{msort} \doteq \texttt{isort} \in \texttt{seq} \to \texttt{seq}$$

Otherwise, if EXT does not hold, then we would no longer suppress any intensional properties, which would mean that `msort` and `isort` would no longer be considered equal on all accounts. Instead, we would have something more akin to the following (for some $s, s_0 : \texttt{seq}$, where s_0 is a sorted permutation of s):

$$\texttt{isort}(s) \doteq \texttt{step}^{|s|^2}(\texttt{ret}(s_0)) \mathrel{\widetilde{\in}} \texttt{seq}$$

$$\texttt{msort}(s) \doteq \texttt{step}^{|s| \log |s|}(\texttt{ret}(s_0)) \mathrel{\widetilde{\in}} \texttt{seq}$$

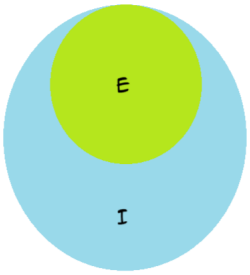
Here, the `step` construct will "record" the specified step cost. The introduction of this construct lifts the idea of profiling to the language level, and will allow us to easily switch "on" or "off" cost considerations via phases.

With these ideas in mind, we can now start formalizing a general system for cost accounting and phase distinctions.

3 Formalizing

3.1 Phase Distinctions

In our formalization, we will distinguish between the extensional phase (EXT), where any considerations of cost are suppressed, and the intensional phase ($\top$), where nothing is suppressed. Although we are able to isolate EXT from $\top$, we should not consider them to be independent – instead, we should consider EXT as being contained within $\top$, as in the following picture:



When considering judgments on equality in $\top$, we need to both consider behavior and cost, which is consistent with $\top$ being dependent on judgments in EXT. In this way, we have this relationship between the phases where they are not mutually exclusive, but instead where one phase can be considered independently, while the other must be considered within the context of the first.

Similar analogies exist in other parts of language design – for example, in a module system, we can consider solely the signature and treat the implementation as a "black box," however the implementation itself is dependent on the specifications in the signature.

3.2 Grammar

We will formulate this system in the lax type setting, where we have terms ($\Gamma \vdash M : A$) and computations ($\Gamma \vdash E \div A$).

$$\begin{array}{rrl} \text{Computations} & E ::= & \mathtt{ret}(M) \mid \mathtt{bind}(M\ ;\ x\ .\ E) \mid \mathtt{step}^c(E) \\ \text{Terms} & M ::= & \ldots \mid \mathtt{comp}(E) \\ \text{Types} & A ::= & \ldots \mid \mathtt{comp}(A) \end{array}$$

There are a few things to note about the grammar:

- The language now has a $\mathtt{step}^c(E)$ computation, which is the "profiling" step which records a computation of cost c. In the extensional phase, the `step` computation is treated as a no-op.
- The c in the $\mathtt{step}^c(E)$ computation is an element of a cost monoid, $\mathbb{C}$. A monoid is a set which contains an identity element and has an associative binary operation.
- You can include any other types/terms in the language as long as they are total.

3.3 Statics

With the new computation construct, we have a new static rule:

$$\frac{c \in \mathbb{C} \qquad \Gamma \vdash E \mathrel{\dot{\sim}} A}{\Gamma \vdash \mathtt{step}^c(E) \mathrel{\dot{\sim}} A} \text{[T-Step]}$$

The judgment $c \in \mathbb{C}$ means that c is an element of the cost monoid $\mathbb{C}$.

3.4 Dynamics

For the dynamics, we have the following judgments:

1. $M \Downarrow V$, to say that term M evaluates to a value V
2. $E \overset{c}{\longmapsto} E'$, to say that a computation E steps to another computation E' with cost c (with $c \in \mathbb{C}$). If a step has no cost (i.e. the cost is the identity element of $\mathbb{C}$), you can simply say $E \longmapsto E'$.

We then define the following dynamics:

$$\frac{M \Downarrow V}{\mathtt{ret}(M) \longmapsto \mathtt{ret}(V)} \text{[D-Ret]} \qquad \frac{}{\mathtt{step}^c(E) \overset{c}{\longmapsto} E} \text{[D-Step]}$$

$$\frac{M \Downarrow \mathtt{comp}(E_1) \qquad E_1 \overset{c}{\longmapsto} E_1'}{\mathtt{bind}(M \mathbin{;} x \mathbin{.} E_2) \overset{c}{\longmapsto} \mathtt{bind}(\mathtt{comp}(E_1') \mathbin{;} x \mathbin{.} E_2)} \text{[D-Bind]}$$

$$\frac{M \Downarrow \mathtt{comp}(\mathtt{ret}(V))}{\mathtt{bind}(M \mathbin{;} x \mathbin{.} E_2) \longmapsto [V/x]E_2} \text{[D-Bind-Ret]}$$

We also have the judgment $E \Downarrow^c V$ to express that, in the complete execution of computation E, we reach the value V (with cost $c \in \mathbb{C}$). With this judgment, we have the following dynamics:

$$\frac{M \Downarrow V}{\mathtt{ret}(M) \Downarrow^0 V} \text{[D-Val]} \qquad \frac{E \overset{c}{\longmapsto} E' \qquad E' \Downarrow^{c'} V}{E \Downarrow^{c+c'} V} \text{[D-Step]}$$

In the second rule, $+$ is the associative binary operator that the monoid $\mathbb{C}$ is equipped with.

4 Equality and Equivalence

When defining our equational rules for this system, we will need to now take the phase into consideration, since different phases may entail different conditions for equivalence. We first define the judgment $\phi \vdash \psi$ to mean that "being in phase ϕ entails being in phase ψ". For example, we have that $\mathsf{EXT} \vdash \top$. If we consider our pictorial representation of the phases from Section 3.1, then this should make sense – even though EXT can be isolated from $\top$, it is still contained within $\top$.

For the phase entailment judgment, we have the following rules, which defines a preorder on the phases:

$$\frac{}{\phi \vdash \phi}\ [\textsc{refl}] \qquad \frac{\phi \vdash \psi \qquad \psi \vdash \gamma}{\phi \vdash \gamma}\ [\textsc{Trans}]$$

4.1 Equality

As one would expect, our exact equality judgments now depend on the current phase, which gives us the judgments $M \doteq M' \in A[\phi]$ for terms and $E \doteq E' \mathrel{\widetilde{\in}} A[\phi]$ for computations.

For terms, we define the judgment for $\mathtt{comp}(A)$ and $A_1 \to A_2$ as representative cases:

$$M \doteq M' \in \mathtt{comp}(A)[\phi] \text{ iff } M \Downarrow \mathtt{comp}(E),\ M' \Downarrow \mathtt{comp}(E') \text{ and } E \doteq E' \mathrel{\widetilde{\in}} A[\phi]$$

$$M \doteq M' \in A_1 \to A_2[\phi] \text{ iff } \begin{cases} M \Downarrow \lambda(x\,.\,M_2),\ M' \Downarrow \lambda(x\,.\,M_2') \\ \forall\ \psi \vdash \phi, \text{if } M_1 \doteq M_1' \in A_1[\psi], \text{then } [M_1/x]M_2 \doteq [M_1'/x]M_2' \in A_2[\psi] \end{cases}$$

For computations, we define exact equality as follows:

$$E \doteq E' \mathrel{\widetilde{\in}} A[\phi] \text{ iff } \begin{cases} E \Downarrow^c V,\ E' \Downarrow^{c'} V' \\ V \doteq V' \in A[\phi] \text{ and either } c = c' \text{ or } \phi \vdash \mathsf{EXT} \end{cases}$$

More explicitly, we say that two computations are equal if they execute to the same final value with the same cost, *or* we are in the extensional phase and do not need to consider cost.

For these judgments, we have the following anti-monotonicity lemmas:

Lemma 4.1.

1. *If $M \doteq M' \in A[\phi]$ and $\psi \vdash \phi$, then $M \doteq M' \in A[\psi]$*
2. *If $E \doteq E' \mathrel{\widetilde{\in}} A[\phi]$ and $\psi \vdash \phi$, then $E \doteq E' \mathrel{\widetilde{\in}} A[\psi]$*

If we constrain this lemma to $\mathsf{EXT} \vdash \top$, this would mean that any terms or computations found equal in $\top$ should also be found equal in EXT, whereas things found equal in EXT may not necessarily be equal in $\top$ (as with `msort` and `isort`).

We also have head expansion for terms:

Lemma 4.2. *If* $M \doteq M' \in A[\phi]$ *and* $N \longmapsto M$*, then* $N \doteq M' \in A[\phi]$*, and if* $N \longmapsto M'$*, then* $M \doteq N \in A[\phi]$

Finally, we introduce new entailment judgments $\Gamma \vdash^{\phi} M : A$ and $\Gamma \vdash^{\phi} E \mathrel{\dot{\sim}} A$ to represent entailment under a particular phase ϕ. We also introduce the judgment $\Gamma \gg^{\phi} M \doteq M' \in A$ (and analogously for computations), which means "If $\gamma \doteq \gamma' \in \Gamma[\phi]$, then $\hat{\gamma}(M) \doteq \hat{\gamma}'(M') \in A[\phi]$." With these, we can define the reflexivity lemmas:

Lemma 4.3.

1. *If* $\Gamma \vdash^{\phi} M : A$*, then* $\Gamma \gg^{\phi} M \doteq M \in A$
2. *If* $\Gamma \vdash^{\phi} E \mathrel{\dot{\sim}} A$*, then* $\Gamma \gg^{\phi} E \doteq E \mathrel{\widetilde{\in}} A$

4.2 Equivalence

The equivalence judgments are also dependent on the current phase, which gives us judgments of the form $\Gamma \vdash^{\phi} M \equiv M' : A$ for terms and $\Gamma \vdash^{\phi} E \equiv E' \mathrel{\dot{\sim}} A$ for computations.

Since the execution steps of `ret` and `bind` do not incur cost, any equational rules not involving `step` will be the same as in the lax setting. We do, however, have new equational rules for `step` computations:

[Step-Zero]
$$\frac{\Gamma \vdash^{\phi} E \mathrel{\dot{\sim}} A}{\Gamma \vdash^{\phi} \mathtt{step}^0(E) \equiv E \mathrel{\dot{\sim}} A}$$

[Step-Step]
$$\frac{\Gamma \vdash^{\phi} E \mathrel{\dot{\sim}} A}{\Gamma \vdash^{\phi} \mathtt{step}^c(\mathtt{step}^{c'}(E)) \equiv \mathtt{step}^{c+c'}(E) \mathrel{\dot{\sim}} A}$$

[Step-Ext]
$$\frac{\Gamma \vdash^{\mathsf{EXT}} E \mathrel{\dot{\sim}} A}{\Gamma \vdash^{\mathsf{EXT}} \mathtt{step}^c(E) \equiv E \mathrel{\dot{\sim}} A}$$

[Step-Bind]
$$\frac{\Gamma \vdash^{\phi} E_1 \mathrel{\dot{\sim}} A_1 \qquad \Gamma, x : A_1 \vdash^{\phi} E_2 \mathrel{\dot{\sim}} A_2}{\Gamma \vdash^{\phi} \mathtt{bind}(\mathtt{comp}(\mathtt{step}^c(E_1)) \mathbin{;} x \,.\, E_2) \equiv \mathtt{step}^c(\mathtt{bind}(\mathtt{comp}(E_1) \mathbin{;} x \,.\, E_2)) \mathrel{\dot{\sim}} A_2}$$

We might consider adding the following rule similar to the last one:

[Step-Bind?]
$$\frac{\Gamma \vdash^{\phi} E_1 \mathrel{\dot{\sim}} A_1 \qquad \Gamma, x : A_1 \vdash^{\phi} E_2 \mathrel{\dot{\sim}} A_2}{\Gamma \vdash^{\phi} \mathtt{bind}(\mathtt{comp}(E_1) \mathbin{;} x \,.\, \mathtt{step}^c(E_2)) \equiv \mathtt{step}^c(\mathtt{bind}(\mathtt{comp}(E_1) \mathbin{;} x \,.\, E_2)) \mathrel{\dot{\sim}} A_2}$$

This rule would be permissible in the current language, however if E_1 diverged, then the step cost of E_2 would never be incurred.

Finally, we can state the fundamental theorems:

Theorem 4.4.

1. *If* $\Gamma \vdash^{\phi} M \equiv M' : A$, *then* $\Gamma \gg^{\phi} M \doteq M' \in A$
2. *If* $\Gamma \vdash^{\phi} E \equiv E' \dot{\sim} A$, *then* $\Gamma \gg^{\phi} E \doteq E' \mathrel{\widetilde{\in}} A$

To prove the theorem(s), we proceed by induction on the equivalence judgment. We will prove the STEP-BIND case of the theorem as a representative case:

Proof.

Fix $\gamma \doteq \gamma' \in \Gamma[\phi]$. We would like to show that $\texttt{bind}(\texttt{comp}(\texttt{step}^c(\hat{\gamma}(E_1)))\,;x.\hat{\gamma}(E_2)) \doteq \texttt{step}^c(\texttt{bind}(\texttt{comp}(\hat{\gamma}'(E_1))\,;x.\hat{\gamma}'(E_2))) \mathrel{\widetilde{\in}} A_2[\phi]$.

From the first premise, we have that $\Gamma \vdash^{\phi} E_1 \dot{\sim} A_1$. By Lemma 4.3, we have $\hat{\gamma}(E_1) \doteq \hat{\gamma}'(E_1) \mathrel{\widetilde{\in}} A_1[\phi]$. By definition, this means that $\hat{\gamma}(E_1) \Downarrow^{c_1} V_1$, $\hat{\gamma}'(E_1) \Downarrow^{c_1'} V_1'$ such that $V_1 \doteq V_1' \in A_1[\phi]$ and either $c_1 = c_1'$ or $\phi \vdash \mathsf{EXT}$.

Consider the following stepping for both computations:

$$\begin{aligned}
&\texttt{bind}(\texttt{comp}(\texttt{step}^c(\hat{\gamma}(E_1))) \;;\; x \,.\, \hat{\gamma}(E_2)) \\
&\overset{\mathrm{c}}{\longmapsto} \texttt{bind}(\texttt{comp}(\hat{\gamma}(E_1)) \;;\; x \,.\, \hat{\gamma}(E_2)) \\
&\Downarrow^{c_1} [V_1/x]\hat{\gamma}(E_2)
\end{aligned}$$

$$\begin{aligned}
&\texttt{step}^c(\texttt{bind}(\texttt{comp}(\hat{\gamma}'(E_1)) \;;\; x \,.\, \hat{\gamma}'(E_2))) \\
&\overset{\mathrm{c}}{\longmapsto} \texttt{bind}(\texttt{comp}(\hat{\gamma}'(E_1)) \;;\; x \,.\, \hat{\gamma}'(E_2)) \\
&\Downarrow^{c_1'} [V_1'/x]\hat{\gamma}'(E_2)
\end{aligned}$$

In other words, we have $\texttt{bind}(\texttt{comp}(\texttt{step}^c(\hat{\gamma}(E_1)))\,;x.\hat{\gamma}(E_2)) \Downarrow^{c+c_1} [V_1/x]\hat{\gamma}(E_2)$ and $\texttt{step}^c(\texttt{bind}(\texttt{comp}(\hat{\gamma}'(E_1)) \;;\; x \,.\, \hat{\gamma}'(E_2))) \Downarrow^{c+c_1'} [V_1'/x]\hat{\gamma}'(E_2)$.

From the second premise, we have $\Gamma, x : A_1 \vdash^{\phi} E_2 \dot{\sim} A_2$. By Lemma 4.3, we would have $\hat{\gamma}^*(E_2) \doteq \hat{\gamma}'^*(E_2) \mathrel{\widetilde{\in}} A_2[\phi]$ for $\gamma^* \doteq \gamma'^* \in \Gamma, x : A_1[\phi]$. Note that since $V_1 \doteq V_1' \in A_1[\phi]$ and $\gamma \doteq \gamma' \in \Gamma[\phi]$, we have that $\gamma[x \mapsto V_1] \doteq \gamma'[x \mapsto V_1'] \mathrel{\widetilde{\in}} \Gamma, x : A_1[\phi]$. As such, by the result of reflexivity, we have $\gamma\widehat{[x \mapsto V_1]}(E_2) \doteq \gamma'\widehat{[x \mapsto V_1']}(E_2) \mathrel{\widetilde{\in}} A_2[\phi]$, i.e. $[V_1/x]\hat{\gamma}(E_2) \doteq [V_1'/x]\hat{\gamma}'(E_2) \mathrel{\widetilde{\in}} A_2[\phi]$ by definition of substitution.

Since $[V_1/x]\hat{\gamma}(E_2) \doteq [V_1'/x]\hat{\gamma}'(E_2) \mathrel{\widetilde{\in}} A_2[\phi]$, by definition, $[V_1/x]\hat{\gamma}(E_2) \Downarrow^{c_2} V_2$, $[V_1'/x]\hat{\gamma}'(E_2) \Downarrow^{c_2'} V_2'$, and either $c_2 = c_2'$ or $\phi \vdash \mathsf{EXT}$. By combining these results with the previous stepping, we have $\texttt{bind}(\texttt{comp}(\texttt{step}^c(\hat{\gamma}(E_1)))\,;x.\hat{\gamma}(E_2)) \Downarrow^{c+c_1+c_2} V_2$ and $\texttt{step}^c(\texttt{bind}(\texttt{comp}(\hat{\gamma}'(E_1)) \;;\; x \,.\, \hat{\gamma}'(E_2))) \Downarrow^{c+c_1'+c_2'} V_2'$.

If $\phi \vdash \mathsf{EXT}$, then we immediately have that $\texttt{bind}(\texttt{comp}(\texttt{step}^c(\hat{\gamma}(E_1))) \;;\; x \,.$

$\hat{\gamma}(E_2)) \doteq \mathtt{step}^c(\mathtt{bind}(\mathtt{comp}(\hat{\gamma}'(E_1)) \ ; \ x \ . \ \hat{\gamma}'(E_2))) \mathrel{\widetilde{\in}} A_2[\phi]$, since we do not need to take cost into consideration. Otherwise, from our previous invocations of reflexivity, we have that $c_1 = c_1'$ and $c_2 = c_2'$. Then, it follows that $c + c_1 + c_2 = c + c_1' + c_2'$.

Thus, in either case, we have $\mathtt{bind}(\mathtt{comp}(\mathtt{step}^c(\hat{\gamma}(E_1)));x.\hat{\gamma}(E_2)) \doteq \mathtt{step}^c(\mathtt{bind}(\mathtt{comp}(\hat{\gamma}'(E_1)); x \ . \ \hat{\gamma}'(E_2))) \mathrel{\widetilde{\in}} A_2[\phi]$.

□

5 Non-Interference

In the context of extensional and intensional phases, the concept of *non-interference* refers to the idea that any extensional behavior should not be affected by our intensional considerations, i.e. profiling.

The statement of the non-interference theorem is as follows:

Theorem 5.1. *If* $\cdot \vdash^{\top} F : \mathtt{comp}(\mathtt{unit}) \to \mathtt{comp}(\mathtt{ans})$*, then* F *is extensionally a constant function*

In other words, F should be extensionally equal to $\lambda(\_ \, . \, \mathtt{comp}(\mathtt{ret}(V)))$ for $V : \mathtt{ans}$.

Proof.

To prove this, suppose we have $\cdot \vdash^{\top} F : \mathtt{comp}(\mathtt{unit}) \to \mathtt{comp}(\mathtt{ans})$. By Lemma 4.3, we have that $F \doteq F \in \mathtt{comp}(\mathtt{unit}) \to \mathtt{comp}(\mathtt{ans})[\top]$. By Lemma 4.1, since $\mathsf{EXT} \vdash \top$, we have that $F \doteq F \in \mathtt{comp}(\mathtt{unit}) \to \mathtt{comp}(\mathtt{ans})[\mathsf{EXT}]$.

By definition of exact equality, for $M \doteq M' \in \mathtt{comp}(\mathtt{unit})[\mathsf{EXT}]$, we have that $\mathtt{ap}(F \ ; \ M) \doteq \mathtt{ap}(F \ ; \ M') \in \mathtt{comp}(\mathtt{ans})[\phi]$. Using that result, we have that $\mathtt{ap}(F \ ; \ M) \Downarrow \mathtt{comp}(E)$, $\mathtt{ap}(F \ ; \ M') \Downarrow \mathtt{comp}(E')$ and $E \doteq E' \mathrel{\widetilde{\in}} \mathtt{ans}[\mathsf{EXT}]$. Additionally, since $E \doteq E' \mathrel{\widetilde{\in}} \mathtt{ans}[\mathsf{EXT}]$, we have that $E \Downarrow^c V$, $E' \Downarrow^{c'} V'$ such that $V \doteq V' \in \mathtt{ans}[\mathsf{EXT}]$. Since $\mathsf{EXT} \vdash \mathsf{EXT}$, we need not consider the cost, and we simply observe that both functions return equal answers on equal inputs, i.e. F returns either `yes` or `no`. Thus, we have that F is extensionally equal to $\lambda(\_ \, . \, \mathtt{comp}(\mathtt{ret}(V)))$ for $V : \mathtt{ans}$.

□