Assignment 1:
Statics and Dynamics, Safety, and Finite Data Types

15-312: Principles of Programming Languages (Fall 2023)

On this assignment, you will implement the statics and dynamics of a simple programming language, **PSF**, which has product, sum, and function types. You will also prove that the language is safe, showing some cases of progress and preservation. Finally, you will explore some curious connections to algebra and logic.

1 Introduction to PSF

On this assignment, we will be working with **PSF**[1], a programming language with products, sums, and functions.

Typ	τ	$::=$	$\texttt{arr}(\tau_1\,;\tau_2)$	$\tau_1 \to \tau_2$	function
			$\texttt{unit}$	$\texttt{unit}$	nullary product
			$\texttt{prod}(\tau_1\,;\tau_2)$	$\tau_1 \times \tau_2$	binary product
			$\texttt{void}$	$\texttt{void}$	nullary sum
			$\texttt{sum}(\tau_1\,;\tau_2)$	$\tau_1 + \tau_2$	binary sum
Exp	e	$::=$	x	x	variable
			$\lambda[\tau_1](x\,.\,e_2)$	$\lambda\,(x:\tau_1)\,e_2$	abstraction
			$\texttt{ap}(e\,;e_1)$	$e(e_1)$	application
			$\texttt{triv}$	$\langle\rangle$	nullary tuple
			$\texttt{pair}(e_1\,;e_2)$	$\langle e_1, e_2\rangle$	binary tuple
			$\texttt{pr}[\texttt{l}](e)$	$e\cdot\texttt{l}$	left projection
			$\texttt{pr}[\texttt{r}](e)$	$e\cdot\texttt{r}$	right projection
			$\texttt{absurd}[\tau](e)$	$\texttt{absurd}(e)$	nullary case analysis
			$\texttt{in}[\texttt{l}][\tau_1\,;\tau_2](e)$	$\texttt{l}\cdot e$	left injection
			$\texttt{in}[\texttt{r}][\tau_1\,;\tau_2](e)$	$\texttt{r}\cdot e$	right injection
			$\texttt{case}(e\,;x_1\,.\,e_1\,;x_2\,.\,e_2)$	$\texttt{case}\,e\,\{\texttt{l}\cdot x_1 \hookrightarrow e_1 \mid \texttt{r}\cdot x_2 \hookrightarrow e_2\}$	binary case analysis

The statics and dynamics of **PSF** are given in Appendices A and B, respectively.

[1] Creatively standing for "products, sums, and functions".

2 Statics

Just like **E**, we define the static semantics of **PSF** via judgment $\Gamma \vdash e : \tau$, where Γ is the *context*, e is an *expression*, and τ is a *type*. The rules defining this judgment are given in Appendix A.

2.1 Implementation

On the previous assignment, you implemented an ABT structure for working with α-equivalence classes of terms. On this and all subsequent assignments, we will provide you with the analogous structures for the languages we will study, as promised. Importantly, you need only worry about the signature `PSF` in `lang-psf/psf.abt.sig`, not the implementation of structure `PSF` itself.

Observe that there are two sorts in **PSF**, `Typ` and `Exp`, each with their own sub-structure. As on the previous assignment, you will find the "primed" auxiliaries (such as `Exp.Pair'`) useful for constructing elements of `Exp.t` and `Exp.out : view -> exp` useful for going by cases on an element of `Exp.t`.

In this section, we'll implement the statics; in other words, we'll write a typechecker.

2.1.1 `core/` Utilities

Before we get to implementing, we'll introduce some common signatures that will be used throughout the semester.

We'll often use `signature SHOW` denoting a type that can be converted to a string:

`core/common/show.sig`


```
signature SHOW =
sig
  type t
  val toString: t -> string
end
```

The `toString` function will be used to provide a convenient REPL interface.

Now, we will consider the signature for implementing the statics of a programming language.

First, consider `signature CONTEXT`, given in Fig. 2.1. In general, there are relatively few requirements on a context: there should be an `empty` context and the ability to `append` two contexts. In **PSF** (and in many languages we'll work with), though, we will desire additional operations. This is expressed in `signature CONTEXT_PSF` in Fig. 2.2: after including all operations from `CONTEXT`, we add the ability to `insert` a variable of a particular type and `lookup` the type of a variable.

We provide a structure `ContextPSF :> CONTEXT_PSF` in `lang-psf/context-psf.sml`, implemented as a dictionary.

core/statics/context.sig

```
signature CONTEXT =
sig
  type context
  type t = context
  val empty: context
  val append: context * context -> context
end
```

core/statics/statics.sig

```
signature STATICS =
sig
  structure Typ: SHOW (* parameter *)
  structure Term: SHOW (* parameter *)

  structure Context: CONTEXT (* parameter *)

  structure Error: SHOW (* abstract *)
  exception TypeError of Error.t
  val inferType: Context.t -> Term.t -> Typ.t
  and checkType: Context.t -> Term.t -> Typ.t -> unit
end
```


Figure 2.1: The `STATICS` signature.

lang-psf/context-psf.sig

```
signature CONTEXT_PSF =
sig
  include CONTEXT

  type var = PSF.Exp.Var.t
  type typ = PSF.Typ.t

  val singleton: var -> typ -> context
  val insert: context -> var -> typ -> context

  val lookup: context -> var -> typ
end
```

Figure 2.2: The `CONTEXT_PSF` signature, extending the `CONTEXT` signature from Fig. 2.1.

Now, consider `signature STATICS`, also defined in Fig. 2.1. Observe that structures `Typ`, `Term`, and `Context` are marked as "parameters". For example, the typechecker for **PSF** will ascribe to the following signature:

`lang-psf/statics-psf.sml`

```
structure StaticsPSF :>
  STATICS
    where type Typ.t  = PSF.Typ.t
      and type Term.t = PSF.Exp.t
      and Context = ContextPSF
```

On the other hand, the type `Error.t` is left abstract. We will not grade the quality of your error messages; thus, throughout the semester, *you may choose any type you wish* for `Error.t`! Here are some common choices:

- `structure UnitError`, defined in `core/common/error-unit.sml`.

  It sets `type t = unit`, allowing you to `raise TypeError ()`. This is is easy to use, but it may make debugging type errors more challenging.

- `structure StringError`, defined in `core/common/error-string.sml`

  It sets `type t = string`, allowing you to `raise TypeError "error message"`. This is fairly easy to use and quite flexible, but it forces you to do string processing in-line.

- `structure StaticsErrorPSF`, defined in `lang-psf/statics-error-psf.sml`.

  It uses a custom datatype for `type t`, providing errors for scenarios where an elimination form is used incorrectly and for when a type assertion fails. This allows you you to factor out all string processing logic.

- Your own custom implementation, defined directly after `structure Error =`.

The starter code sets `structure Error = StaticsErrorPSF`, but you are welcome to change this as you wish.

Every typechecker should implement:

- `inferType`, such that `inferType context term` infers the type of `term` in `context`. If no type exists, it should `raise TypeError error`, for some `error : Error.t`.
- `checkType`, such that `checkType context term typ` checks that the type of `term` in `context` is `typ`, returning `()`. If this is not the case, it should `raise TypeError error`, for some `error : Error.t`.

Given `inferType`, it is easy to implement `checkType`; we provide its implementation for you. You may find it productive to use `checkType` (mutually recursively) from within `inferType`.

2.1.2 The First of Many

Task 2.1 (40 pts). In `lang-psf/statics-psf.sml`, implement the statics of **PSF** given in Appendix A.

Hint. The rules in Appendix A precisely determine how `inferType` should behave. If you wish to determine the type of some expression based on the conclusion of an inference rule, you can make recursive calls to `inferType` (and `checkType`) to determine (or assert) the types of subexpressions as given in the premises.

2.1.3 Testing

See Appendix C; note that for the time being, the dynamics will not be implemented, but your typechecker will be used to infer the types of expressions in test files and in the REPL.

3 Dynamics

Now that we have determined which programs are well-formed, we will consider how to run these programs. We give the dynamics of **PSF** in Appendix B, defined by the judgments $\boxed{e \ \mathsf{val}}$ and $\boxed{e \longmapsto e'}$.

For the dynamics of **PSF**, we must prove *safety*, which consists of two properties:

1. **Progress**, which states that every well-typed program is either a value or can step.
2. **Preservation**, which states that steps preserve types.

Progress will tell us how to implement the dynamics, and preservation will help us check that the dynamics are sensible.

3.1 Progress

First, we will prove progress for **PSF**; then, we will use our proof to implement the language. For simplicity, you will only focus on the cases for (nullary and binary) sums, `void` and $\tau_1 + \tau_2$.

Before we can prove the theorem itself, we will need an important lemma.

Lemma 3.1 (Canonical Forms). *If e val, then:*

1. *If $\cdot \vdash e : \texttt{void}$, then we have reached a contradiction.*
2. *If $\cdot \vdash e : \tau_1 + \tau_2$, then either:*
   - *$e = \texttt{l} \cdot e_1$, for some e_1 with e_1 val, or*
   - *$e = \texttt{r} \cdot e_2$, for some e_2 with e_2 val.*

Here, we only enumerate the cases for (nullary and binary) sums.

Task 3.1 (10 pts). State the Canonical Forms Lemma for the remaining types in **PSF**: nullary products, binary products, and arrow types.

Task 3.2 (10 pts). Prove the cases of Lemma 3.1 for (nullary and binary) sums only.

Hint. In each case, you assume that e val and that e is well-typed. Which assumption should you go by induction on first?

Hint. Feel free to state that all remaining are vacuous, as long as this claim is true.

Theorem 3.2 (Progress). *If $e : \tau$, then either:*

- *there exists some e' such that $e \longmapsto e'$, or*
- *e val.*

Task 3.3 (25 pts). Prove Theorem 3.2 for the introduction and elimination forms for (nullary and binary) sums: cases `absurd`(e), `l` $\cdot e$, `r` $\cdot e$, and `case` $e\,\{\texttt{l} \cdot x_1 \hookrightarrow e_1 \mid \texttt{r} \cdot x_2 \hookrightarrow e_2\}$.

You *may omit symmetric cases* (e.g., left vs. right branches of a case); simply say "this case is symmetric". You can assume that progress is proved for all other forms (i.e., the IH holds). Moreover, you may use Lemma 3.1, as long as the citations are correct.

3.2 Implementation

Now that we know that every expression form makes progress, we are able to implement **PSF**!

3.2.1 `core/` Utilities

In Section 2.1.1, we discussed core signatures for implementing a statics. Now, we will introduce the analogous utilities for impementing a dynamics.

First, consider `signature STATE`, partially reproduced in Fig. 3.1. When implementing dynamics, we will have two important types: the type of user programs and the type of "machine states". The type `'a t` in `signature STATE` represents a machine state, where we will choose `'a` to be the type of programs. We include `val initial : 'a -> 'a t` to create a machine state containing a program.

Often, the two machine states will be "taking a step" and "finished with a value", corresponding to judgments $e \longmapsto e'$ and `e val`. For this purpose, we will extend the signature as shown in Fig. 3.2, where for **PSF**, type `Value.t` will be `Exp.t`. We provide this structure as `StatePSF`.

The key signature is `signature DYNAMICS`, shown in Fig. 3.1. Every dynamics will come equipped with a state parameter and a term parameter; for example, the dynamics for **PSF** will ascribe to the following signature:

`lang-psf/dynamics-psf.sml`

```
structure DynamicsPSF :>
  DYNAMICS
    where State = StatePSF
      and type term = PSF.Exp.t
```

Just like with the typechecker, you may choose a suitable `structure Error`; the data in a `Malformed` exception is for your debugging. We include `structure DynamicsErrorPSF` in `lang-psf/dynamics-error-psf.sml`, although you are welcome to use an error structure described in Section 2.1.1.

core/dynamics/state.sig

```
signature STATE =
sig
  type 'a t
  val initial: 'a -> 'a t
end
```

core/dynamics/dynamics.sig

```
signature DYNAMICS =
sig
  structure State: STATE (* parameter *)

  type term (* parameter *)

  structure Error: SHOW (* abstract *)
  exception Malformed of Error.t

  (** `progress term`
   *
   * Step `term` to a machine state.
   * If `term` is malformed, raise `Malformed error`, for some `error : Error.t`.
   *)
  val progress: term -> term State.t
end
```


Figure 3.1: The `DYNAMICS` signature.

core/dynamics/state-transition.fun

```
functor TransitionState(Value: SHOW) :>
sig
  datatype ('a, 'v) state = Step of 'a | Val of 'v
  val map2: ('a -> 'b) * ('v -> 'w) -> ('a, 'v) state -> ('b, 'w) state
end
```

Figure 3.2: The result signature for the `TransitionState` functor.

Task 3.4 (40 pts). In `lang-psf/dynamics-psf.sml`, implement the dynamics of **PSF** given in Appendix B. You should assume that your input is well-typed.

Hint. The progress theorem precisely determines how `progress` should be implemented.

- When the proof of progress says that $e \longmapsto e'$, your code should produce `State.Step e'`.
- When the proof of progress says that e val, your code should produce `State.Val e`.
- When the proof of progress appeals to the IH and cases on the result, you should recursively call `progress` and case on `State.Step e` and `State.Val v`.
- When the proof of progress uses the Canonical Forms Lemma on e, you should case on `Exp.out e`, raising `Malformed` in cases guaranteed to be impossible.

3.2.2 Testing

See Appendix C; you should be able to replicate the given test outputs. Feel free to build your own tests (in a test file or in `InterpreterPSF.repl ()`), as well!

3.3 Preservation

Now, it remains to show that the dynamics for **PSF** are sensible, i.e., that transitions preserve typing.

Once again, we will first need some lemmas.

Lemma 3.3 (Inversion). *The following inversions hold:*

1. *If* $\Gamma \vdash$ `absurd(`e`)` $: \tau$*, then:*
   - $\Gamma \vdash e :$ `void`
2. *If* $\Gamma \vdash$ `l` $\cdot\, e : \tau$*, then:*
   - $\tau = \tau_1 + \tau_2$
   - $\Gamma \vdash e : \tau_1$
3. *If* $\Gamma \vdash$ `case` $e\,\{$`l` $\cdot\, x_1 \hookrightarrow e_1 \mid$ `r` $\cdot\, x_2 \hookrightarrow e_2\} : \tau$*, then for some* τ_1, τ_2*:*
   - $\Gamma \vdash e : \tau_1 + \tau_2$
   - $\Gamma, x_1 : \tau_1 \vdash e_1 : \tau$
   - $\Gamma, x_2 : \tau_2 \vdash e_2 : \tau$

Like Lemma 3.1, we only enumerate the cases for (nullary and binary) sums here.

Task 3.5 (10 pts). Prove the `case` $e\,\{$`l` $\cdot\, x_1 \hookrightarrow e_1 \mid$ `r` $\cdot\, x_2 \hookrightarrow e_2\}$ case of Lemma 3.3.

Hint. There is only one possibility to induct on. Furthermore, many cases will be vacuous!

Additionally, we have the following lemma:

Lemma 3.4 (Substitution). *If* $\Gamma \vdash e_1 : \tau_1$ *and* $\Gamma, x : \tau_1 \vdash e : \tau$, *then* $\Gamma \vdash \{e_1/x\}e : \tau$.

Now, on to the main theorem:

Theorem 3.5 (Preservation). *If* $e \longmapsto e'$, *then if* $\cdot \vdash e : \tau$, *then* $\cdot \vdash e' : \tau$.

Task 3.6 (25 pts). Prove Theorem 3.5 for the introduction and elimination forms for (nullary and binary) sums: cases $\texttt{absurd}(\, e\,)$, $\texttt{l} \cdot e$, $\texttt{r} \cdot e$, and $\texttt{case}\, e\, \{\texttt{l} \cdot x_1 \hookrightarrow e_1 \mid \texttt{r} \cdot x_2 \hookrightarrow e_2\}$.

You *may omit symmetric cases* (e.g., left vs. right branches of a case); simply say "this case is symmetric". You can assume that preservation is proved for all other forms (i.e., the IH holds). Moreover, you may use Lemmas 3.3 and 3.4, as long as their citations are correct.

4 Computational Trinitarianism

In this section, we will explore how *types*, *algebra*, and *logic* are all fundamentally related.

4.1 Algebra

Consider the types $\alpha \times \beta$ and $\beta \times \alpha$, for some fixed types α, β. While elements of the former cannot be used directly in place of elements of the latter, they seem to contain equivalent information. To prove this, we can exhibit an *isomorphism* between the types.

Definition 4.1 (Isomorphic Types)**.** An *isomorphism* between types τ_1 and τ_2 in **PSF** consists of a **PSF** expression $f_1 : \tau_1 \to \tau_2$ and a **PSF** expression $f_2 : \tau_2 \to \tau_1$ with following two properties.

1. For all values $v_1 : \tau_1$, we have $f_2(f_1(v_1)) = v_1$.
2. For all values $v_2 : \tau_2$, we have $f_1(f_2(v_2)) = v_2$.

If there is an isomorphism between τ_1 and τ_2 in **PSF** then we say τ_1 and τ_2 are *isomorphic* in **PSF** and write $\tau_1 \simeq \tau_2$.

Notice that this definition is similar to the notion of "bijection" from set theory.

Note: for simplicity, we will not treat the notion of equality $=$ used in the proofs rigorously, since it's beyond the scope of this course. Thus, for our purposes, both proofs need only be argued *informally*.

Now, let's consider a proof that $\alpha \times \beta \simeq \beta \times \alpha$:

Consider the following functions:

$$f_1 = \lambda\,(x : \alpha \times \beta)\,\langle x \cdot \mathtt{r}, x \cdot \mathtt{l}\rangle$$
$$f_2 = \lambda\,(x : \beta \times \alpha)\,\langle x \cdot \mathtt{r}, x \cdot \mathtt{l}\rangle$$

Clearly, $f_2(f_1(v_1)) = v_1$ for each value $v_1 : \alpha \times \beta$, since the elements of the pair are swapped and swapped back. The other direction is similar.

This isomorphism serves as a "data migration" scheme: it tells us precisely how to migrate data stored as $\alpha \times \beta$ to a new format, $\beta \times \alpha$. Intuitively, this process roughly corresponds to swapping two columns in a database.

The isomorphism $\alpha \times \beta \simeq \beta \times \alpha$ is an example of a broader pattern: types form an *algebraic structure*! Amazingly enough, this isomorphism is much like a fact you learned in elementary school: $a \times b = b \times a$, where $a, b \in \mathbb{N}$.

4.1.1 Notation

For the duration of this problem, we will use:

- α, β, γ to represent arbitrary types,
- a, b, c to represent numbers, and
- `A`, `B`, `C` to represent types in the **PSF** implementation.

Using our implementation of **PSF**, we can implement type isomorphisms. For example, we implement the previous isomorphism in `lang-psf/tests/iso0.psf`.

Task 4.1 (16 pts). Prove each of the following isomorphisms in `lang-psf/tests/`, giving two functions of the appropriate types (first the left-to-right direction, then the right-to-left direction). In a comment, provide a brief informal justification (1-3 sentences) about why your functions are mutually inverse. The arithmetic expressions in the "Arithmetic Correspondent" column are just for reference/intuition.

	Type Isomorphism	Arithmetic Correspondent	File
1.	$\alpha \times \texttt{unit} \simeq \alpha$	$a \times 1 = a$	`iso1.psf`
2.	$(\alpha \times \beta) \times \gamma \simeq \alpha \times (\beta \times \gamma)$	$(a \times b) \times c = a \times (b \times c)$	`iso2.psf`
3.	$\alpha + \texttt{void} \simeq \alpha$	$a + 0 = a$	`iso3.psf`
4.	$\alpha \times (\beta + \gamma) \simeq (\alpha \times \beta) + (\alpha \times \gamma)$	$a \times (b + c) = (a \times b) + (a \times c)$	`iso4.psf`
5.	$\texttt{void} \to \texttt{void} \simeq \texttt{unit}$	$0^0 = 1$	`iso5.psf`
6.	$\alpha \to (\beta \to \gamma) \simeq (\alpha \times \beta) \to \gamma$	$(c^b)^a = c^{a \times b}$	`iso6.psf`
7.	$\alpha \to (\beta \times \gamma) \simeq (\alpha \to \beta) \times (\alpha \to \gamma)$	$(b \times c)^a = b^a \times c^a$	`iso7.psf`
8.	$(\alpha + \beta) \to \gamma \simeq (\alpha \to \gamma) \times (\beta \to \gamma)$	$c^{a+b} = c^a \times c^b$	`iso8.psf`

By proving these isomorphisms, you have provided witnesses to the fact that programmers may go freely between the data representations on each side of the equation. You may even find it useful to reorganize your data structures with simple arithmetic facts in mind!

4.2 Logic

There is a remarkable—and influential—correspondence between principles of reasoning (logic) and principles of programming (type theory) called the *propositions-as-types* correspondence. Informally, the rules of logic determining when a propositional formula φ is a tautology (i.e., always true) correspond to the typing rules for the corresponding type. In school, the rules of logic are usually taught in terms of "truth tables" for connectives, defining when propositions are *true* or *false*. And yet when using logic to show that a proposition is true, we *give a proof* of it. When we have a proof of φ, the truth table will say that it is always true.

What is a proof, then? The propositions-as-types principle states that a proof of a proposition φ is a *program* of the type corresponding to φ (and vice versa), according to the following chart:

Connective	Proposition φ	Type $\overline{\varphi}$
trivial truth	$\top$	`unit`
conjunction	$\varphi_1 \wedge \varphi_2$	$\overline{\varphi_1} \times \overline{\varphi_2}$
trivial falsehood	$\bot$	`void`
disjunction	$\varphi_1 \vee \varphi_2$	$\overline{\varphi_1} + \overline{\varphi_2}$
implication	$\varphi_1 \supset \varphi_2$	$\overline{\varphi_1} \to \overline{\varphi_2}$

Thus, proving $\varphi_1 \supset \varphi_2$ amounts to writing a program that takes an element of type $\overline{\varphi_1}$ (an assumed proof of φ_1) and produces an element of type $\overline{\varphi_2}$ (a proof of φ_2). This suggests that the proofs that corresponds to programs are constructive. Indeed, the propositions that we can prove with programs in **PSF** correspond to the tautologies of *constructive logic*. Be careful: we cannot prove all "classical" tautologies (i.e., tautologies that can be proved via truth tables). All tautologies

that are constructively true also hold classically, but not all tautologies that are true classically hold constructively.

Let A, B, C be arbitrary propositions, with:

$$\overline{A} \triangleq \texttt{A}$$
$$\overline{B} \triangleq \texttt{B}$$
$$\overline{C} \triangleq \texttt{C}$$

In your solutions, you need not be careful about the fonts; conflate at your convenience.

Example Consider the following formula:

$$\varphi \triangleq (A \supset (A \vee B)) \wedge (B \supset (C \supset B))$$

The corresponding type $\overline{\varphi}$ is:

$$(\,\texttt{A} \to \texttt{A} + \texttt{B}\,) \times (\,\texttt{B} \to (\,\texttt{C} \to \texttt{B}\,))$$

Thus, we can prove the theorem via the following term:

$$\langle \lambda\,(a : \texttt{A}\,)\,\texttt{1} \cdot a, \lambda\,(b : \texttt{B}\,)\,\lambda\,(c : \texttt{C}\,)\,b \rangle$$

Example Consider the following formula:

$$\varphi \triangleq (A \vee B) \supset A$$

The corresponding type $\overline{\varphi}$ is:

$$\texttt{A} + \texttt{B} \to \texttt{A}$$

No program of this type exists, since if the input is a right injection, we have no way of transforming a $\texttt{B}$ into a $\texttt{A}$.

Task 4.2 (24 pts). In this exercise, you are to explore the propositoins-as-types correspondence by exhibiting proofs or arguing that none exists. For each proposition φ below, show that φ is (constructively) true by exhibiting a program $e : \overline{\varphi}$, or argue informally that no such e can exist.

1. $\bot$
2. $\top \vee \bot$
3. $(A \supset (B \vee C)) \supset ((A \supset B) \vee (A \supset C))$
4. $((A \supset B) \vee (A \supset C)) \supset (A \supset (B \vee C))$

Define $\neg\varphi \triangleq \varphi \supset \bot$.

5. $A \supset (\neg\neg A)$
6. $(\neg\neg A) \supset A$

7. $A \vee \neg A$
8. $\neg\neg(A \vee \neg A)$
9. $\neg(A \vee B) \supset (\neg A \wedge \neg B)$
10. $(\neg A \wedge \neg B) \supset \neg(A \vee B)$
11. $\neg(A \wedge B) \supset (\neg A \vee \neg B)$
12. $(\neg A \vee \neg B) \supset \neg(A \wedge B)$

Hint. If you believe you have constructed an expression of the correct type, you can check your solution via `InterpreterPSF.repl ()`. You may also find it helpful to submit your solutions in a `\begin{codeblock}` environment, perhaps in **PSF** concrete syntax (or something close to it).

Remark. A theorem φ is true constructively if there is *at least* one proof of it. However, it's possible for there to be *multiple* proofs of a given proposition. For example, consider:

$$\varphi \triangleq \top \vee \top$$

Then, $\overline{\varphi}$ is `unit + unit`, which can be proven in both `l . ()` and `r . ()`.

Remark. If $\overline{\varphi_1} \simeq \overline{\varphi_2}$, then $\varphi_1 \supset\subset \varphi_2$. Consider what the type isomorphisms from the previous question mean logically.

A Statics

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\ \textsc{Var}$$

A.1 Functions

$$\frac{\Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \lambda[\tau_1](x\,.\,e_2) : \tau_1 \to \tau_2}\ \to\text{-I} \qquad \frac{\Gamma \vdash e : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_1 : \tau_1}{\Gamma \vdash \mathtt{ap}(e\,;e_1) : \tau_2}\ \to\text{-E}$$

A.2 Products

$$\frac{}{\Gamma \vdash \mathtt{triv} : \mathtt{unit}}\ \mathtt{unit}\text{-I}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathtt{pair}(e_1;e_2) : \tau_1 \times \tau_2}\ \times\text{-I} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathtt{pr}[\mathtt{l}](e) : \tau_1}\ \times\text{-E}_1 \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathtt{pr}[\mathtt{r}](e) : \tau_2}\ \times\text{-E}_2$$

A.3 Sums

$$\frac{\Gamma \vdash e : \mathtt{void}}{\Gamma \vdash \mathtt{absurd}[\tau](e) : \tau}\ \mathtt{void}\text{-E}$$

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathtt{in}[\mathtt{l}][\tau_1\,;\tau_2](e) : \tau_1 + \tau_2}\ +\text{-I}_1 \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathtt{in}[\mathtt{r}][\tau_1\,;\tau_2](e) : \tau_1 + \tau_2}\ +\text{-I}_2$$

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \qquad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \qquad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \mathtt{case}(e\,;x_1\,.\,e_1\,;x_2\,.\,e_2) : \tau}\ +\text{-E}$$

B Dynamics (Eager, Left-to-Right)

B.1 Functions

$$\frac{}{\lambda[\tau_1](x\,.\,e_2)\ \mathsf{val}}$$

$$\frac{e \longmapsto e'}{\mathtt{ap}(e\,;e_1) \longmapsto \mathtt{ap}(e'\,;e_1)} \qquad \frac{e\ \mathsf{val} \qquad e_1 \longmapsto e_1'}{\mathtt{ap}(e\,;e_1) \longmapsto \mathtt{ap}(e\,;e_1')} \qquad \frac{e_1\ \mathsf{val}}{\mathtt{ap}((\lambda[\tau_1](x\,.\,e_2))\,;e_1) \longmapsto \{e_1/x\}e_2}$$

B.2 Products

$$\frac{}{\mathtt{triv}\ \mathsf{val}}$$

$$\frac{e_1 \longmapsto e_1'}{\mathtt{pair}(e_1\,;e_2) \longmapsto \mathtt{pair}(e_1'\,;e_2)} \qquad \frac{e_1\ \mathsf{val} \qquad e_2 \longmapsto e_2'}{\mathtt{pair}(e_1\,;e_2) \longmapsto \mathtt{pair}(e_1\,;e_2')} \qquad \frac{e_1\ \mathsf{val} \qquad e_2\ \mathsf{val}}{\mathtt{pair}(e_1\,;e_2)\ \mathsf{val}}$$

$$\frac{e \longmapsto e'}{\mathtt{pr[l]}(e) \longmapsto \mathtt{pr[l]}(e')} \qquad \frac{e_1\ \mathsf{val} \qquad e_2\ \mathsf{val}}{\mathtt{pr[l]}(\mathtt{pair}(e_1\,;e_2)) \longmapsto e_1} \qquad \frac{e \longmapsto e'}{\mathtt{pr[r]}(e) \longmapsto \mathtt{pr[r]}(e')}$$

$$\frac{e_1\ \mathsf{val} \qquad e_2\ \mathsf{val}}{\mathtt{pr[r]}(\mathtt{pair}(e_1\,;e_2)) \longmapsto e_2}$$

B.3 Sums

$$\frac{e \longmapsto e'}{\mathtt{absurd}[\tau](e) \longmapsto \mathtt{absurd}[\tau](e')}$$

$$\frac{e \longmapsto e'}{\texttt{in}[\texttt{l}][\tau_1\,;\tau_2](\,e\,) \longmapsto \texttt{in}[\texttt{l}][\tau_1\,;\tau_2](\,e'\,)} \qquad \frac{e\ \mathsf{val}}{\texttt{in}[\texttt{l}][\tau_1\,;\tau_2](\,e\,)\ \mathsf{val}}$$

$$\frac{e \longmapsto e'}{\texttt{in}[\texttt{r}][\tau_1\,;\tau_2](\,e\,) \longmapsto \texttt{in}[\texttt{r}][\tau_1\,;\tau_2](\,e'\,)} \qquad \frac{e\ \mathsf{val}}{\texttt{in}[\texttt{r}][\tau_1\,;\tau_2](\,e\,)\ \mathsf{val}}$$

$$\frac{e \longmapsto e'}{\texttt{case}(\,e\,;x_1\,.\,e_1\,;x_2\,.\,e_2\,) \longmapsto \texttt{case}(\,e'\,;x_1\,.\,e_1\,;x_2\,.\,e_2\,)}$$

$$\frac{e\ \mathsf{val}}{\texttt{case}(\,\texttt{in}[\texttt{l}][\tau_1\,;\tau_2](\,e\,)\,;x_1\,.\,e_1\,;x_2\,.\,e_2\,) \longmapsto \{e/x_1\}e_1}$$

$$\frac{e\ \mathsf{val}}{\texttt{case}(\,\texttt{in}[\texttt{r}][\tau_1\,;\tau_2](\,e\,)\,;x_1\,.\,e_1\,;x_2\,.\,e_2\,) \longmapsto \{e/x_2\}e_2}$$

C PSF Interpreter

We combine `StaticsPSF` and `DynamicsPSF` into `InterpreterPSF`, which ascribes to the following signature:

`core/interpreter/interpreter.sig`

```
signature INTERPRETER =
sig
  val repl: unit -> unit
  val evalFile: string -> unit
end
```

In `lang-psf/tests/`, some small **PSF** examples are demonstrated in concrete syntax:

`lang-psf/tests/tests.psf`

```
fn (x : unit) => x;

fn (f : unit -> unit + unit) => f <>;

<<>, <<>, <>>>;

<(fn (x : unit) => <x, x>) <> : in[r]{void, unit}(<>)>.r;

(* boolean negation function *)
fn (bool : unit + unit) =>
  case[unit + unit] bool {
  | l.x => in[r]{unit, unit}(<>)
  | r.x => in[l]{unit, unit}(<>)
  }
;

fn (x : unit * void) => case[unit + unit] x.r {};
```


To run the tests or enter the **PSF** REPL, change your directory to `lang-psf/` and load `sources.cm`:

```
smlnj -m sources.cm

- InterpreterPSF.evalFile "tests/tests.psf";
(Lam (Unit, (x2 . x2)))
Type: (Arrow (Unit, Unit))
Evaluating... val (Lam (Unit, (x8 . x8)))

(Lam ((Arrow (Unit, (Sum (Unit, Unit)))), (f11 . (Ap (f11,
   Triv)))))
```

```
Type: (Arrow ((Arrow (Unit, (Sum (Unit, Unit)))), (Sum (Unit,
   Unit))))
Evaluating... val (Lam ((Arrow (Unit, (Sum (Unit, Unit)))), (f17
   . (Ap (f17, Triv)))))

(* ... *)

val it = () : unit
- InterpreterPSF.repl ();
-> <<>, in[r]{unit,unit}(<>)>.r;
(PrR (Pair (Triv, (InR ((Unit, Unit), Triv)))))
Type: (Sum (Unit, Unit))
Evaluating... val (InR ((Unit, Unit), Triv))

->
```