Command-Line Flags as Types

Harrison Grodin, advised by Robert Harper

Background

- Software imposes requirements on the world:
  - effects/primitives
  - system/hardware requirements
  - library dependencies
- Usually, **expressivity and usability** are at odds.
  - Want to specify requirements in the program, but don't want to manually propagate resources.
- **Each requirement can be a command-line flag identifying a language.**
- Sterling and Harper recently developed a synthetic generalization of the ML phase distinction via topos theory. We will adapt this approach, **representing each command-line flag via a phase**.

Goal

Develop a framework for specifying programming languages involving compiler flags via phases, and use it to **understand and consolidate** accounts of features and extensions common in ML-style programming languages.

Flags, Phases, and Types

- Define a poset of **flags**.
- Then, define **phases** as lower sets of flags.

Flag	φ, ψ	::=	$\ldots$		
Phase	Φ, Ψ	::=	$\downarrow \varphi$	principal phase	$[\![\downarrow \varphi]\!] = \{\psi \mid \psi \leq \varphi\}$
			$\bot$	falsity	$[\![\bot]\!] = \varnothing$
			$\Phi_1 \vee \Phi_2$	disjunction	$[\![\Phi_1 \vee \Phi_2]\!] = [\![\Phi_1]\!] \cup [\![\Phi_2]\!]$
			$\top$	truth	$[\![\top]\!] = P$
			$\Phi_1 \wedge \Phi_2$	conjunction	$[\![\Phi_1 \wedge \Phi_2]\!] = [\![\Phi_1]\!] \cap [\![\Phi_2]\!]$

- Define type system **relative to a phase**. $\boxed{\Gamma \vdash_\Phi e : \tau}$

Typ	τ	::=	$\ldots$	
			$\bigcirc_\Phi(\tau)$	open modality
Exp	e	::=	$\ldots$	
			$\lambda_\Phi(e)$	phase assumption
			$e()$	phase usage

$$\frac{\Gamma \vdash_{\Phi \wedge \Psi} e : \tau}{\Gamma \vdash_\Phi \lambda_\Psi(e) : \bigcirc_\Psi(\tau)} \qquad \frac{\Gamma \vdash_\Phi e : \bigcirc_\Psi(\tau) \qquad \Phi \subseteq \Psi}{\Gamma \vdash_\Phi e() : \tau}$$

Operational Semantics

- For each flag, define an operational semantics.
- Programmers will ultimately select one flag relative to which code will be typechecked and run.

$$\Downarrow_\varphi \subseteq \mathsf{Exp} \times S_\varphi \qquad t_{\varphi_1 \leq \varphi_2} : S_{\varphi_2} \longrightarrow S_{\varphi_1}$$

Theorem (Progress). *If $\vdash_{\downarrow\varphi} e : \tau$, then there exists some s such that $e \Downarrow_\varphi s$.*

Theorem (Consistency). *For all $\varphi_1 \leq \varphi_2$, if $e \Downarrow_{\varphi_2} s$, then $e \Downarrow_{\varphi_1} t_{\varphi_1 \leq \varphi_2}(s)$.*

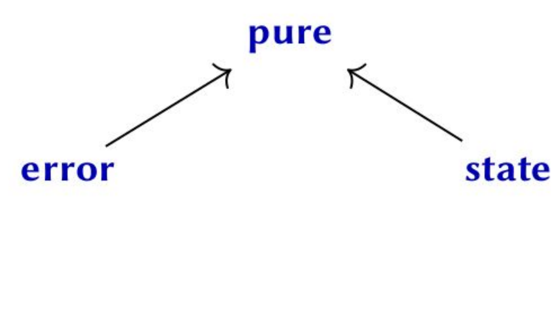
Example 1: Simple Effects

Pure, Error, and State

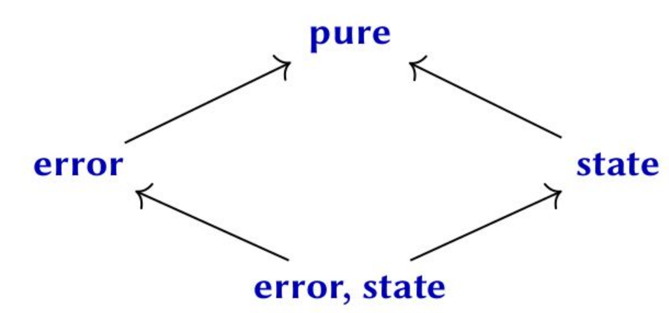
$$\frac{\Phi \ni \mathbf{error}}{\Gamma \vdash_\Phi \underline{\mathrm{error}} : \tau}$$

$$\frac{\Phi \ni \mathbf{state}}{\Gamma \vdash_\Phi \underline{\mathrm{get}} : \underline{\mathrm{nat}}} \qquad \frac{\Phi \ni \mathbf{state} \qquad \Gamma \vdash_\Phi e : \underline{\mathrm{nat}}}{\Gamma \vdash_\Phi \underline{\mathrm{set}}(e) : \underline{\mathrm{unit}}}$$

Combining Effects

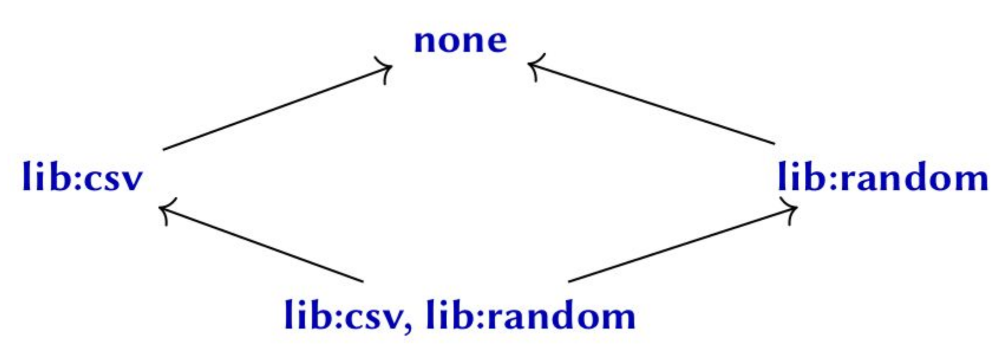


$\mathbf{error} \wedge \mathbf{state} \equiv \bot$

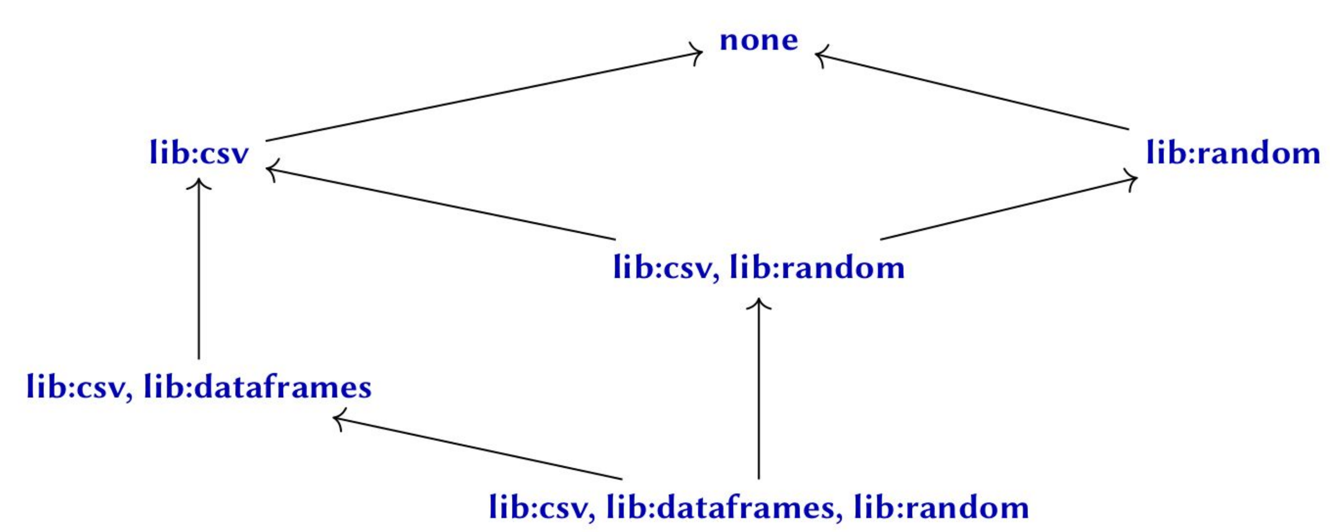


$\mathbf{error} \wedge \mathbf{state} \equiv \mathbf{error, state}$

Library Imports



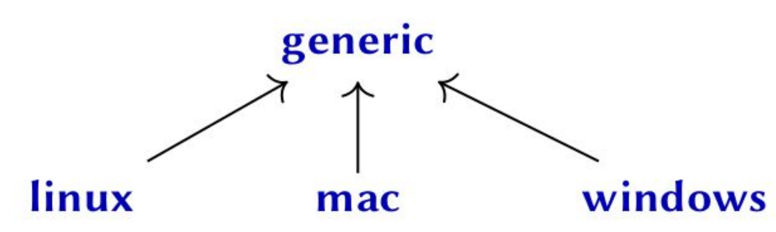
$$\frac{\Phi \ni \mathbf{lib}{:}l}{\Gamma \vdash_\Phi \underline{\mathrm{import}}[l] : \tau_l}$$




```
(* --pure *)

- (fn (x : int) => x + 1) 2;
Type: Int
Evaluating... (Int 3)

- GET + 1;
Type Error: using GET without --state

- fn (--state) => GET + 1;
Type: (Open ({state,ALWAYS}, Int))
Evaluating... (Assume ({state,ALWAYS}, (Plus (Get, (Int 1)))))

- (fn (--state) => 0) ();
Type Error: attempted to apply unavailable phase: {state,ALWAYS}

(* --state *)

- GET;
Type: Int
Evaluating... starting from 0, (Int 0) with store 0

- SET 1;
Type: Unit
Evaluating... starting from 0, Triv with store 1

- (fn (u : unit) => GET) (SET (GET + 1));
Type: Int
Evaluating... starting from 0, (Int 1) with store 1
```

Example 2: Algebraic Effects

Hardware Resources

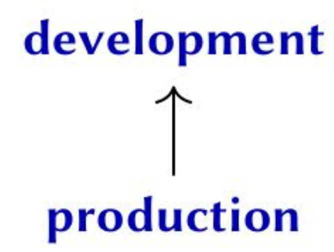


$\mathbf{unix} \triangleq \mathbf{linux} \vee \mathbf{mac}$

$$\frac{\Phi \ni \mathbf{linux} \qquad \Gamma \vdash_\Phi e : \underline{\mathrm{pid}}}{\Gamma \vdash_\Phi \underline{\mathrm{pidfd\text{-}open}}(e) : \underline{\mathrm{fd}}} \qquad \frac{\Phi \ni \mathbf{unix} \qquad \Gamma \vdash_\Phi e : \underline{\mathrm{pid}}}{\Gamma \vdash_\Phi \underline{\mathrm{getpgid}}(e) : \underline{\mathrm{pgid}}}$$

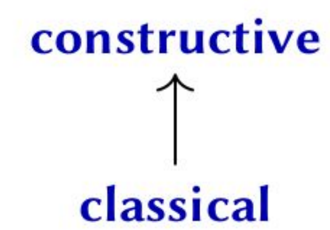
- Linux-only algorithm has type $\boxed{\bigcirc_{\downarrow \mathbf{linux}}(\underline{\mathrm{nat}})}$.

Debugging Tools



$$\Sigma_{\mathbf{development}} = \{\underline{\mathrm{log}} : \underline{\mathrm{string}} \Rightarrow \underline{\mathrm{unit}}\}$$
$$\mathcal{E}_{\mathbf{development}} = \varnothing$$
$$\Sigma_{\mathbf{production}} = \{\underline{\mathrm{log}} : \underline{\mathrm{string}} \Rightarrow \underline{\mathrm{unit}}\}$$
$$\mathcal{E}_{\mathbf{production}} = \{\underline{\mathrm{log}}(s; x.e) \equiv e\}$$

Example 3: Classical Logic



$$\frac{\Phi \ni \mathbf{classical} \qquad \Gamma,\ x : \neg\tau \vdash_\Phi e : \tau}{\Gamma \vdash_\Phi \underline{\mathrm{afsoc}}(x.\ e) : \tau}$$

$$\vdash_{\downarrow \mathbf{constructive}} \lambda_{\mathbf{classical}}(\lambda(k.\ \underline{\mathrm{afsoc}}(k'.\ k(k')))) : \bigcirc_{\mathbf{classical}}(\neg\neg\alpha \to \alpha)$$

Future Work

- Allow types to be induced by flags
- Flag poset evolution
- Dynamic flag allocation?
- Flag handlers?
- Multi-language optimization (e.g., GPU)
- Coeffects
- Closed modality/equality implementation

References

1. J. Sterling and R. Harper. "Logical Relations as Types: Proof-Relevant Parametricity for Program Modules". (2021)
2. J. Sterling and R. Harper. "A metalanguage for multi-phase modularity". (2021)
3. Y. Niu, J. Sterling, H. Grodin, and R. Harper. "A cost-aware logical framework". (2021)