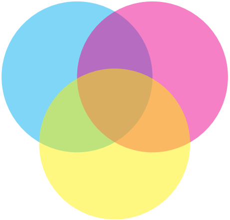
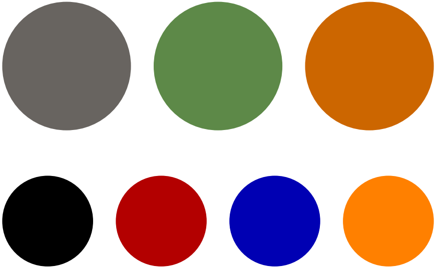




Substructural logics provide a framework for designing resource-aware type systems. While several substructural type systems have been proposed and implemented, these either have been developed for a special purpose or have been too unwieldy for practical use.

Practical Affine Types

Jesse A. Tov and Riccardo Pucella

Northeastern University

January 27, 2010


OpenGL

Example: OpenGL on Android

All you have to do to initialize a GLSurfaceView is call `setRenderer()`. However, if desired, you can modify the default behavior of GLSurfaceView by calling one or more of these methods before `setRenderer`:

- `setDebug()`
- `setChooser()`
- `setWrapper()`

(Android 2.2 API Reference)

Example: OpenGL on Android

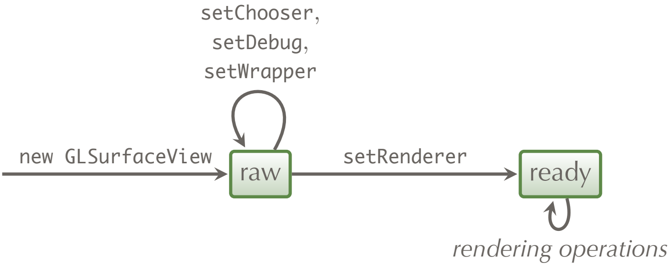
All you have to do to initialize a GLSurfaceView is call `setRenderer()`. However, if desired, you can modify the default behavior of GLSurfaceView by calling one or more of these methods before `setRenderer`:

- `setDebug()`
- `setChooser()`
- `setWrapper()`

(Android 2.2 API Reference)

Example: OpenGL on Android

> All you have to do to initialize a GLSurfaceView is call `setRenderer()`. However, if desired, you can modify the default behavior of GLSurfaceView by calling one or more of these methods before `setRenderer`:
>
> - `setDebug()`
> - `setChooser()`
> - `setWrapper()`
>
> (Android 2.2 API Reference)



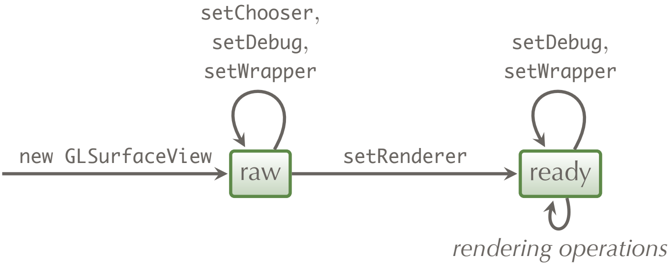
Example: OpenGL on Android

All you have to do to initialize a GLSurfaceView is call `setRenderer()`.
Howev … View
by call …

- se…
- se…
- `setWrapper()`

> You can optionally modify the behavior of GLSurfaceView by calling one or more of the debugging methods `setDebug()`, and `setWrapper()`. These methods may be called before and/or after `setRenderer`, …

(Android 2.2 API Reference)



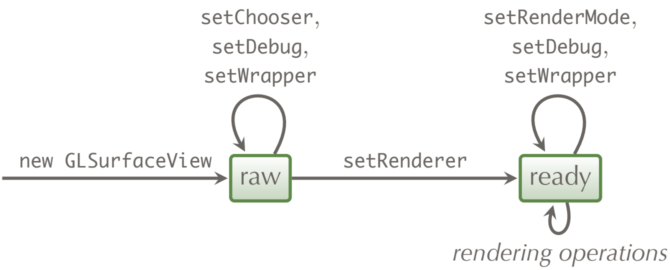
Example: OpenGL on Android

All you have to do to initialize a GLSurfaceView is call setRenderer().
Howev You can optionally modify the behavior of GLSurfaceView View
by

Once the renderer is set, you can control whether the renderer draws continuously or on-demand by calling `setRenderMode()`.

- se fore and/or after `setRenderer`, …
- `setWrapper()`

(Android 2.2 API Reference)



Example: OpenGL on Android

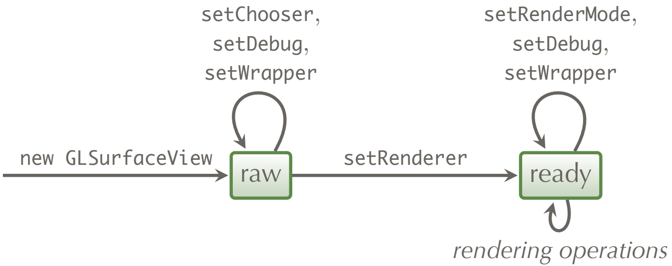
All you have to do to initialize a GLSurfaceView is call `setRenderer()`.
Howev… You can optionally modify the behavior of GLSurfaceView …View
by … Once the renderer is set, you can control whether the renderer draws continuously or on-demand by calling `setRenderMode()`.

- se… fore and/or after `setRenderer`, …
- `setWrapper()`

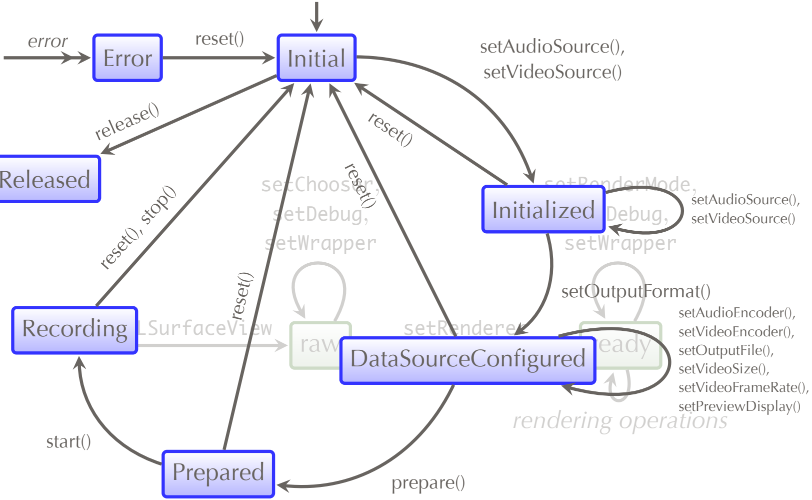
(Android 2.2 API Reference)

Typestate

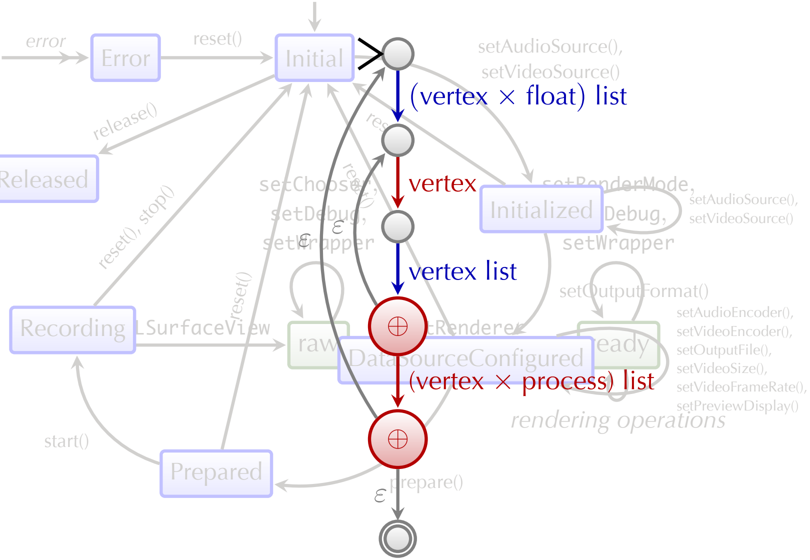
Stateful Programming



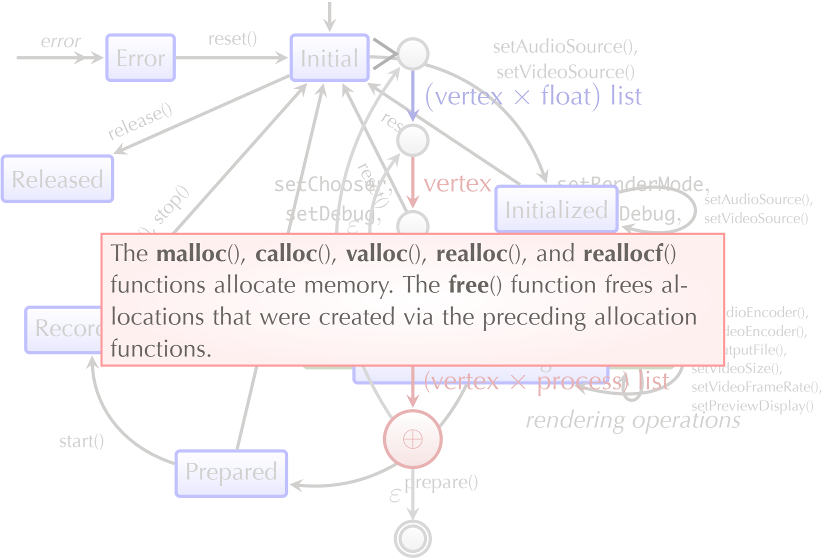
Stateful Programming



Stateful Programming

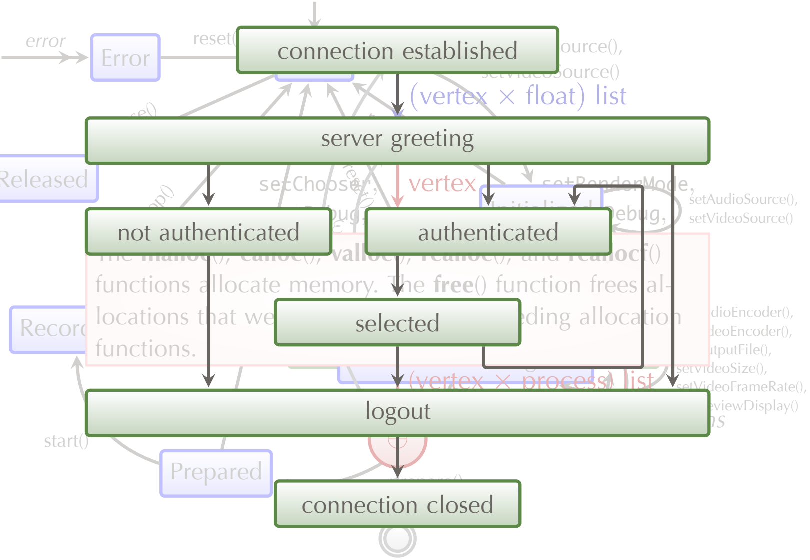


Stateful Programming

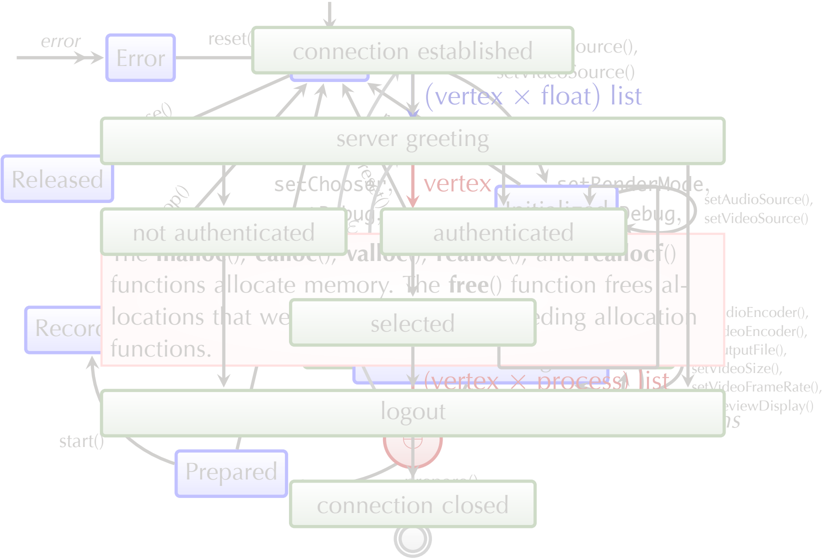


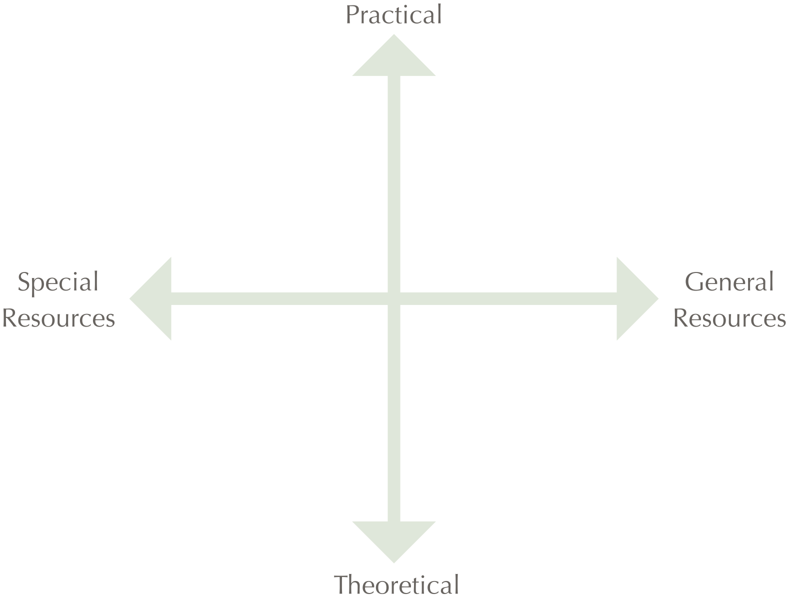
The **malloc**(), **calloc**(), **valloc**(), **realloc**(), and **reallocf**() functions allocate memory. The **free**() function frees allocations that were created via the preceding allocation functions.

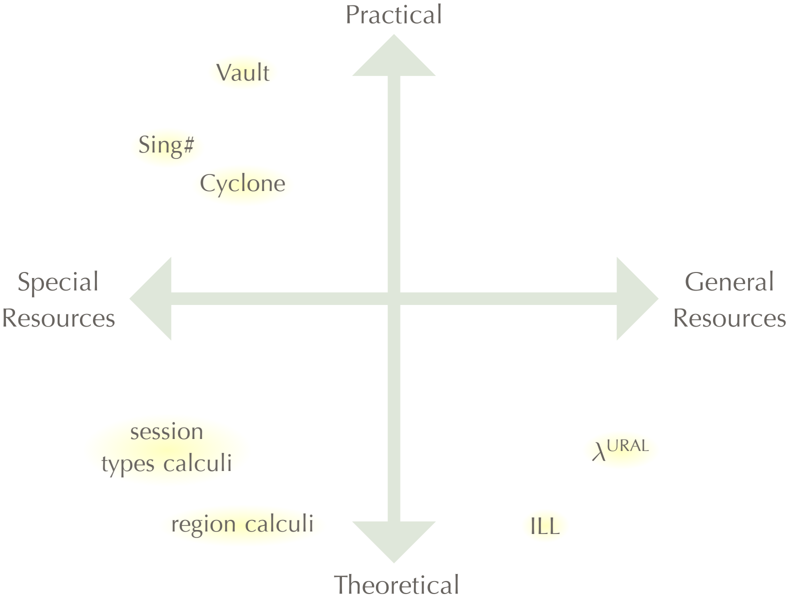
Stateful Programming

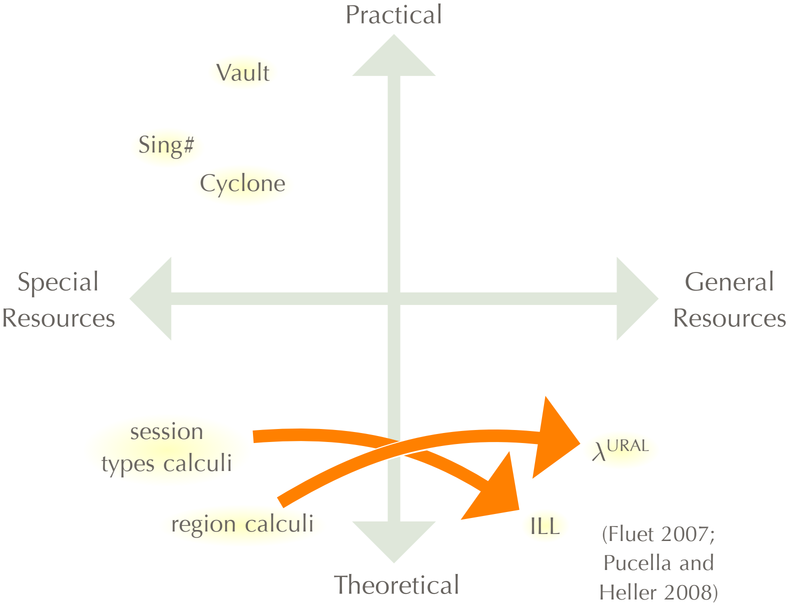


Stateful Programming

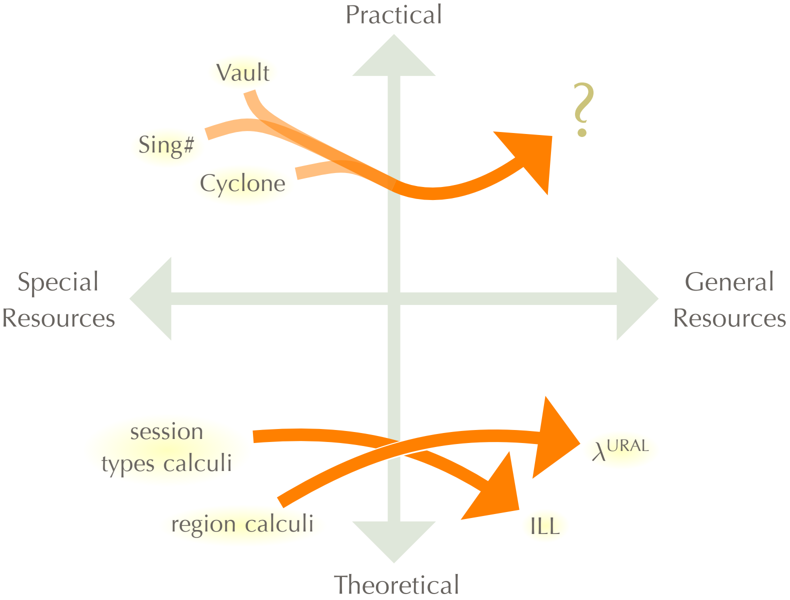


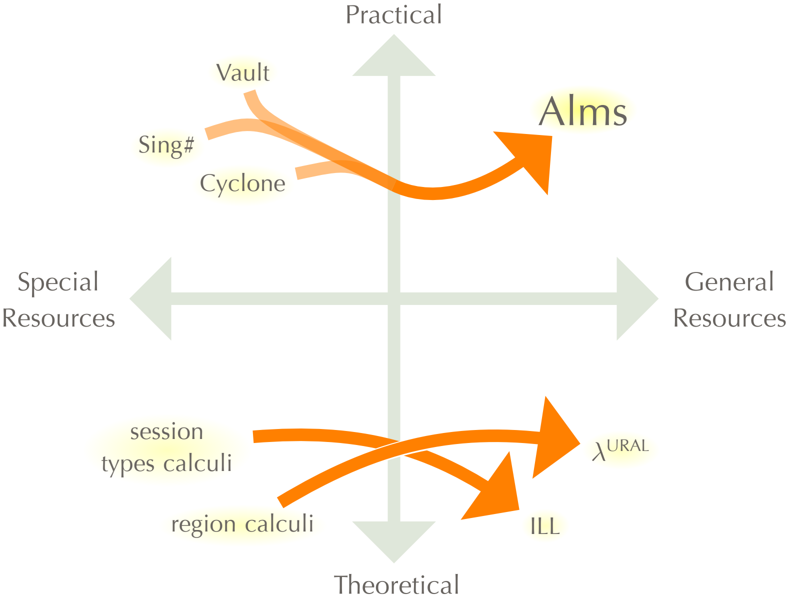

Practical
Special Resources
General Resources
Theoretical


Practical
Vault
Sing#
Cyclone
Special Resources
General Resources
session types calculi
λ^{URAL}
region calculi
ILL
Theoretical



Practical
Vault
Sing#
Cyclone
Special Resources
General Resources
session types calculi
region calculi
λ^{URAL}
ILL
(Fluet 2007; Pucella and Heller 2008)
Theoretical


Practical
Vault
?
Sing#
Cyclone
Special Resources
General Resources
session types calculi
λ^{URAL}
region calculi
ILL
Theoretical


Practical
Vault
Alms
Sing#
Cyclone
Special Resources
General Resources
session types calculi
λ^URAL
region calculi
ILL
Theoretical

What We've Done

A language design (like Ocaml, but with affine types)

A prototype implementation (with libraries and examples)

A core model (with nice theorems)

What We've Done

A language design (like Ocaml, but with affine types)

A prototype implementation (with libraries and examples)

A core model (with nice theorems)

Alms by Example

OpenGL in Alms

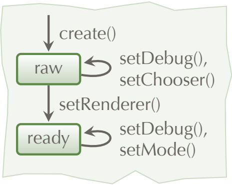
```
module PrimGLSurface : sig
  type glSurface

  val create      : unit → glSurface
  val setChooser  : glSurface → unit
  val setRenderer : glSurface → unit
  val setMode     : glSurface → unit
  val setDebug    : glSurface → unit
end
```

An OpenGL Client

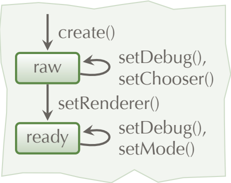
```
let newSurface () =
  let surface = create () in
    setChooser surface;
    setRenderer surface;
    setMode surface;
    setDebug surface;
    surface
```

An OpenGL Client



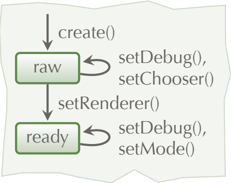

```
let newSurface () =
  let surface = create () in      (* →raw *)
    setChooser surface;           (* raw *)
    setRenderer surface;          (* raw→ready *)
    setMode surface;              (* ready *)
    setDebug surface;             (* ready *)
    surface
```

An OpenGL Client




```
let newSurface () =
  let surface = create () in        (* →raw *)
    setChooser surface;             (* raw *)
    setMode surface;                (* ready? *)
    setRenderer surface;            (* raw→ready *)
    setDebug surface;               (* ready *)
    surface
```

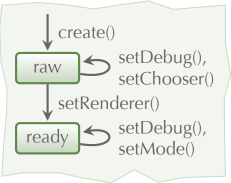
OpenGL in Alms: Take 2



module type GL_SURFACE = sig
  type raw
  type ready
  type β glSurface

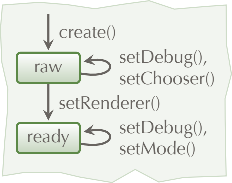
  val *create* : unit $\rightarrow$ raw glSurface
  val *setChooser* : raw glSurface $\rightarrow$ unit
  val *setRenderer* : raw glSurface $\rightarrow$ ready glSurface
  val *setMode* : ready glSurface $\rightarrow$ unit
  val *setDebug* : $\forall\beta.\ \beta$ glSurface $\rightarrow$ unit
end

An OpenGL Client: Take 2




```
let newSurface () =
  let surface = create () in                  (* →raw *)
    setChooser surface;                       (* ready *)
    let surface = setRenderer surface in      (* raw→ready *)
    setMode surface;                          (* ready *)
    setDebug surface;                         (* ready *)
    surface
```

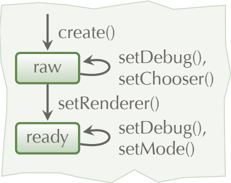
An OpenGL Client: Take 2




```
let newSurface () =
  let surface = create () in              (* →raw *)
    setChooser surface;                   (* ready *)
    setMode surface in                    (* ready? *)
    let surface = setRenderer surface;    (* raw→ready *)
    setDebug surface;                     (* ready *)
    surface
```

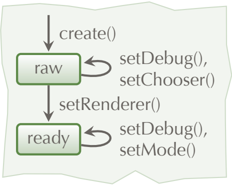
```
Type error at <opengl.alms> (line 4, columns 13-20):
    In application, operand type not in operator's domain:
        actual:   raw glSurface
        expected: ready glSurface
```

An OpenGL Client: Take 2




```
let newSurface () =
  let surface = create () in                (* →raw *)
    setChooser surface;                     (* raw *)
    let surface = setRenderer surface in    (* raw→ready *)
    setMode surface;                        (* ready *)
    setDebug surface;                       (* ready *)
    surface
```

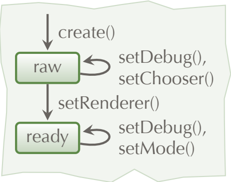
An OpenGL Client: Take 2




```
let newSurface () =
  let surface = create () in                  (* →raw *)
    let surface = setRenderer surface;        (* raw→ready *)
    setChooser surface in                     (* raw? *)
    setMode surface;                          (* ready *)
    setDebug surface;                         (* ready *)
    surface
```

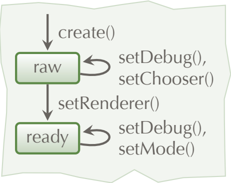
```
Type error at <opengl.alms> (line 4, columns 16-23):
    In application, operand type not in operator’s domain:
        actual:   ready glSurface
        expected: raw glSurface
```

An OpenGL Client: Take 2




```
let newSurface () =
  let surface = create () in                    (* →raw *)
    let surface′ = setRenderer surface;         (* raw→ready′ *)
    setChooser surface in                       (* raw *)
    setMode surface′;                           (* ready′ *)
    setDebug surface′;                          (* ready′ *)
    surface′
```

OpenGL in Alms: Take 3



module type GL_SURFACE = sig

  type raw

  type ready

  type β glSurface : A

  val *create* : unit → raw glSurface

  val *setChooser* : raw glSurface → raw glSurface

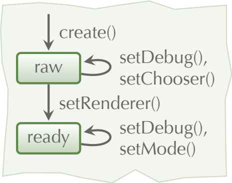
  val *setRenderer* : raw glSurface → ready glSurface

  val *setMode* : ready glSurface → ready glSurface

  val *setDebug* : $\forall\beta.\ \beta$ glSurface → β glSurface

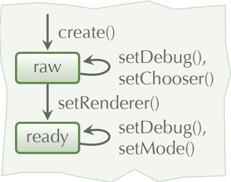
end

An OpenGL Client: Take 3



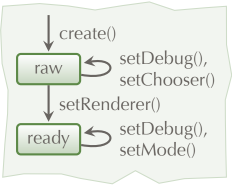

```
let newSurface () =
  let surface = create () in                 (* →raw *)
  let surface = setChooser surface in        (* raw *)
  let surface = setRenderer surface in       (* raw→ready *)
  let surface = setMode surface in           (* ready *)
  let surface = setDebug surface in          (* ready *)
    surface
```

An OpenGL Client: Take 3




```
let newSurface () =
  let surface  = create () in                (* →raw *)
  let surface′ = setRenderer surface in      (* raw→ready′ *)
  let _        = setChooser surface in       (* raw *)
  let surface  = setMode surface′ in         (* ready′→ready *)
  let surface  = setDebug surface in         (* ready *)
    surface
```

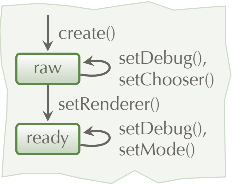
An OpenGL Client: Take 3




```
let newSurface () =
  let surface  = create () in               (* →raw *)
  let surface' = setRenderer surface in     (* raw→ready' *)
  let _        = setChooser surface in      (* raw *)
  let surface  = setMode surface' in        (* ready'→ready *)
  let surface  = setDebug surface in        (* ready *)
    surface
```

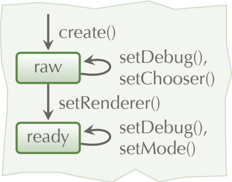
```
Type error at <opengl.alms> (line 2, col. 7 to line 7, col. 12):
    Affine variable ‘surface’ of type ‘raw glSurface’
    duplicated in match or let.
```

An OpenGL Client: Take 3




```
let newSurface () =
  let surface = create () in                  (* →raw *)
  let surface = setChooser surface in         (* raw *)
  let surface = setRenderer surface in        (* raw→ready *)
  let surface = setMode surface in            (* ready *)
  let surface = setDebug surface in           (* ready *)
    surface
```

An OpenGL Client: Take 3




```
let newSurface () =
  let ↯surface = create () in        (* →raw *)
    setChooser surface;               (* raw *)
    setRenderer surface;              (* raw→ready *)
    setMode surface;                  (* ready *)
    setDebug surface                  (* ready *)
```

OpenGL Implementation

```
module GLSurface : GL_SURFACE = struct
   type raw            = unit
   type ready          = unit
   type β glSurface = PrimGLSurface.glSurface
   . . .
end
```

```
module type GL_SURFACE = sig
   type raw     type ready
   type β glSurface : A
   val create        : unit → raw glSurface
   val setChooser  : raw glSurface → raw glSurface
   val setRenderer : raw glSurface → ready glSurface
   val setMode      : ready glSurface → ready glSurface
   val setDebug     : β glSurface → β glSurface
end
```

OpenGL Implementation

$$
\begin{array}{l}
\text{module } \mathit{GLSurface} : \text{GL\_SURFACE} = \text{struct} \\
\quad \text{type raw} \qquad\quad = \text{unit} \\
\quad \text{type ready} \qquad = \text{unit} \\
\quad \text{type } \beta \text{ glSurface} = \text{PrimGLSurface.glSurface} \\
\quad \ldots \\
\text{end}
\end{array}
$$

$$
\begin{array}{ll}
\text{PrimGLSurface.glSurface} & : \mathrm{U} \\
\beta \text{ glSurface} & : \mathrm{A}
\end{array}
$$

$$\mathrm{U} \sqsubseteq \mathrm{A}$$

$$
\begin{array}{l}
\text{module type GL\_SURFACE} = \text{sig} \\
\quad \text{type raw} \quad \text{type ready} \\
\quad \text{type } \beta \text{ glSurface} : \mathrm{A} \\
\quad \text{val } \mathit{create} \qquad\ : \text{unit} \to \text{raw glSurface} \\
\quad \text{val } \mathit{setChooser} \ : \text{raw glSurface} \to \text{raw glSurface} \\
\quad \text{val } \mathit{setRenderer} : \text{raw glSurface} \to \text{ready glSurface} \\
\quad \text{val } \mathit{setMode} \quad\ : \text{ready glSurface} \to \text{ready glSurface} \\
\quad \text{val } \mathit{setDebug} \quad : \beta \text{ glSurface} \to \beta \text{ glSurface} \\
\text{end}
\end{array}
$$

OpenGL Implementation

```
module GLSurface : GL_SURFACE = struct
  type raw          = unit
  type ready        = unit
  type β glSurface = PrimGLSurface.glSurface

  let create = PrimGLSurface.create

  let setRenderer (surface: raw glSurface) =
    PrimGLSurface.setRenderer surface;
    surface
  ...
end
```

```
module type GL_SURFACE = sig
  type raw    type ready
  type β glSurface : A
  val create      : unit → raw glSurface
  val setChooser  : raw glSurface → raw glSurface
  val setRenderer : raw glSurface → ready glSurface
  val setMode     : ready glSurface → ready glSurface
  val setDebug    : β glSurface → β glSurface
end
```

More Examples

Typestate

$$\mathit{Socket.accept} : \alpha \text{ socket} \to \alpha \text{ listening} \to (\exists\beta.\ \beta \text{ socket} \times \beta \text{ ready}) \times \alpha \text{ listening}$$

Session types

$$\mathit{Session.send} : (!\hat{\alpha};\ \beta) \text{ channel} \to \hat{\alpha} \xrightarrow{A} \beta \text{ channel}$$

Regions (with adoption/focus)

$$\mathit{Rgn.adopt} : (\gamma,\hat{\alpha}) \text{ rgn} \to (\delta,\hat{\alpha}) \text{ rgn1} \xrightarrow{A} \delta \text{ ptr} \xrightarrow{A} \gamma \text{ ptr} \times (\gamma,\hat{\alpha}) \text{ rgn}$$

Strong updates

$$\mathit{Ref.swap} : \hat{\alpha} \text{ aref} \to \hat{\beta} \xrightarrow{A} \hat{\beta} \text{ aref} \times \hat{\alpha}$$

Fractional capabilities

$$\mathit{Fractional.split} : (\beta,\gamma) \text{ cap} \to (\beta,\gamma/2) \text{ cap} \times (\beta,\gamma/2) \text{ cap}$$

Design Rationale

The Exponential

Linear Logic (Girard 1987):

$$\frac{\Gamma, !B, !B \vdash \Delta}{\Gamma, !B \vdash \Delta} \text{ (Contraction)}$$

The Problem

Ocaml:

$$\lambda f\ (x,y) \rightarrow f\ x\ y$$
$$: (\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow \alpha \times \beta \rightarrow \gamma$$

The Problem

Ocaml:

$$\lambda f\ (x, y) \to f\ x\ y$$
$$: (\alpha \to \beta \to \gamma) \to \alpha \times \beta \to \gamma$$

ILL (Bierman 1993):

$$\begin{aligned}
&\lambda f \to \text{promote } f \text{ for } g \text{ in} \\
&\quad \lambda p \to \text{let derelict } p \text{ be } x \otimes y \\
&\qquad\qquad \text{in derelict } (\text{derelict } g\ x)\ y \\
&\qquad : \ !(\alpha \multimap !(\beta \multimap \gamma)) \multimap !(!(\alpha \otimes \beta) \multimap \gamma)
\end{aligned}$$

$$\begin{aligned}
&\lambda f\ p \to \text{let derelict } p \text{ be } x \otimes y \\
&\qquad\qquad \text{in derelict } (f\ x)\ y \\
&\qquad : \ (\alpha \multimap !(\beta \multimap \gamma)) \multimap !(\alpha \otimes \beta) \multimap \gamma
\end{aligned}$$

The Problem

Ocaml:

$$\lambda f\ (x,y) \to f\ x\ y$$
$$:\ (\alpha \to \beta \to \gamma) \to \alpha \times \beta \to \gamma$$

ILL to Alms:

$$\begin{aligned}
&\lambda f \to \text{promote } f \text{ for } g \text{ in} \\
&\quad \lambda p \to \text{let derelict } p \text{ be } x \otimes y \\
&\qquad\qquad \text{in derelict (derelict } g\ x)\ y \\
&\qquad :\ (\alpha \to \beta \to \gamma) \xrightarrow{\mathrm{A}} \alpha \times \beta \to \gamma
\end{aligned}$$

$$\begin{aligned}
&\lambda f\ p \to \text{let derelict } p \text{ be } x \otimes y \\
&\qquad\qquad \text{in derelict } (f\ x)\ y \\
&\qquad :\ (\alpha \xrightarrow{\mathrm{A}} \beta \to \gamma) \xrightarrow{\mathrm{A}} \alpha \times \beta \xrightarrow{\mathrm{A}} \gamma
\end{aligned}$$

The Problem

Ocaml:

$$\lambda f\ (x, y) \to f\ x\ y$$
$$: (\alpha \to \beta \to \gamma) \to \alpha \times \beta \to \gamma$$

ILL to Alms:

$$\lambda f\ (x, y) \to f\ x\ y$$
$$: \ (\alpha \to \beta \to \gamma) \to \alpha \times \beta \to \gamma$$

$$\lambda f\ (x, y) \to f\ x\ y$$
$$: \ (\alpha \xrightarrow{\mathsf{A}} \beta \to \gamma) \to \alpha \times \beta \xrightarrow{\mathsf{A}} \gamma$$

The Problem

Ocaml:

$$\lambda f\ (x,y) \to f\ x\ y$$
$$: (\alpha \to \beta \to \gamma) \to \alpha \times \beta \to \gamma$$

Alms:

$$\lambda f\ (x,y) \to f\ x\ y$$
$$: (\alpha \xrightarrow{\delta} \beta \to \gamma) \to \alpha \times \beta \xrightarrow{\delta} \gamma$$

Dereliction Subtyping

$$\begin{aligned} &\mathit{workerThread} : \text{unit} \xrightarrow{\text{U}} \text{unit} \\ &\mathit{Thread.fork} \quad : (\text{unit} \xrightarrow{\text{A}} \text{unit}) \xrightarrow{\text{U}} \text{thread} \end{aligned}$$

Dereliction Subtyping

$\mathit{workerThread} : \mathrm{unit} \xrightarrow{\mathrm{U}} \mathrm{unit}$
$\mathit{Thread.fork} \quad : (\mathrm{unit} \xrightarrow{\mathrm{A}} \mathrm{unit}) \xrightarrow{\mathrm{U}} \mathrm{thread}$

$$\mathrm{unit} \xrightarrow{\mathrm{U}} \mathrm{unit} \quad \leq \quad \mathrm{unit} \xrightarrow{\mathrm{A}} \mathrm{unit} \qquad (\mathrm{U} \sqsubseteq \mathrm{A})$$

$\mathit{workerThread} : \mathrm{unit} \xrightarrow{\mathrm{A}} \mathrm{unit}$

$\mathit{Thread.fork}\ \mathit{workerThread} : \mathrm{thread}$

Principal Promotion

$\lambda x \to x$

$: \alpha \xrightarrow{\cup} \alpha$

$\lambda f\ x \to f\ x$

$: (\alpha_1 \xrightarrow{\gamma} \alpha_2) \xrightarrow{\cup} \alpha_1 \xrightarrow{\gamma} \alpha_2$

$\lambda f\ g\ x \to f\ (g\ x)$

$: (\alpha_2 \xrightarrow{\gamma} \alpha_3) \xrightarrow{\cup} (\alpha_1 \xrightarrow{\delta} \alpha_2) \xrightarrow{\gamma} \alpha_1 \xrightarrow{\gamma \sqcup \delta} \alpha_3$

Principal Promotion

$\lambda x \to x$

$$: \alpha \xrightarrow{\mathsf{U}} \alpha$$

$\lambda f\, x \to f\, x$

$$: (\alpha_1 \xrightarrow{\gamma} \alpha_2) \xrightarrow{\mathsf{U}} \alpha_1 \xrightarrow{\gamma} \alpha_2$$

$\lambda f\, g\, x \to f\, (g\, x)$

$$: (\alpha_2 \xrightarrow{\gamma} \alpha_3) \xrightarrow{\mathsf{U}} (\alpha_1 \xrightarrow{\delta} \alpha_2) \xrightarrow{\gamma} \alpha_1 \xrightarrow{\gamma \sqcup \delta} \alpha_3$$

Theorem. Alms's type system finds the type with least kind for every typable function.

Usage Kinds

$\text{type } \alpha \text{ list} = \text{Nil} \mid \text{Cons of } \alpha \times \alpha \text{ list}$

$$
\begin{array}{l}
\text{let rec } \mathit{foldr}\ f\ z\ \mathit{xs} = \text{match } \mathit{xs} \text{ with} \\
\quad \mid \mathit{Cons}(x, \mathit{xs}) \rightarrow f\ x\ (\mathit{foldr}\ f\ z\ \mathit{xs}) \\
\quad \mid \mathit{Nil} \qquad\qquad\ \ \rightarrow z
\end{array}
$$

Usage Kinds

type α list = Nil | Cons of $\alpha \times \alpha$ list

$$
\begin{array}{l}
\text{let rec } \mathit{foldr}\ f\ z\ \mathit{xs} = \text{match } \mathit{xs} \text{ with} \\
\quad \mid \mathit{Cons}(x, \mathit{xs}) \rightarrow f\ x\ (\mathit{foldr}\ f\ z\ \mathit{xs}) \\
\quad \mid \mathit{Nil} \qquad\qquad\ \ \rightarrow z
\end{array}
$$

$$
\begin{array}{ll}
\text{int list} & : \mathsf{U} \\
\text{raw glSurface list} & : \mathsf{A}
\end{array}
$$

Usage Kinds

type α list = Nil | Cons of $\alpha \times \alpha$ list

let rec *foldr f z xs* = match *xs* with
  | *Cons*(*x*, *xs*) $\rightarrow$ *f x* (*foldr f z xs*)
  | *Nil* $\rightarrow$ *z*

int list : U
raw glSurface list : A
raw glSurface ref : ?

Usage Kinds

type α list = Nil | Cons of $\alpha \times \alpha$ list

let rec *foldr f z xs* = match *xs* with
  | *Cons*(*x*, *xs*) $\rightarrow$ *f x* (*foldr f z xs*)
  | *Nil* $\rightarrow$ *z*

int list : U
raw glSurface list : A
raw glSurface ref : ?

Usage Kinds

$\text{type } (\alpha{:}\text{U}) \text{ list} = \text{Nil} \mid \text{Cons of } \alpha \times \alpha \text{ list}$

$$
\begin{array}{l}
\text{let rec } \mathit{foldr}\ f\ z\ \mathit{xs} = \text{match } \mathit{xs} \text{ with} \\
\quad \mid \mathit{Cons}(x, \mathit{xs}) \rightarrow f\ x\ (\mathit{foldr}\ f\ z\ \mathit{xs}) \\
\quad \mid \mathit{Nil} \qquad\qquad\ \ \rightarrow z
\end{array}
$$

Usage Kinds

$\text{type } (\alpha{:}\mathrm{U})\ \mathrm{listU} = \mathrm{NilU} \mid \mathrm{ConsU} \text{ of } \alpha \times \alpha \text{ list} \qquad (*\ \mathrm{listU} : \mathrm{U} \Rightarrow \mathrm{U}\ *)$

$$
\begin{array}{l}
\text{let rec } \mathit{foldrU}\ f\ z\ xs = \text{match } xs \text{ with} \\
\quad \mid \mathit{ConsU}(x, xs) \rightarrow f\ x\ (\mathit{foldrU}\ f\ z\ xs) \\
\quad \mid \mathit{NilU} \qquad\qquad\ \ \rightarrow z
\end{array}
$$

Usage Kinds

$\text{type } (\alpha{:}\mathrm{U})\ \mathrm{listU} = \mathrm{NilU} \mid \mathrm{ConsU} \text{ of } \alpha \times \alpha \text{ list} \qquad (*\ \mathrm{listU} : \mathrm{U} \Rightarrow \mathrm{U}\ *)$

$$\begin{array}{l}
\text{let rec } \mathit{foldrU}\ f\ z\ \mathit{xs} = \text{match } \mathit{xs} \text{ with} \\
\quad \mid \mathit{ConsU}(x, \mathit{xs}) \rightarrow f\ x\ (\mathit{foldrU}\ f\ z\ \mathit{xs}) \\
\quad \mid \mathit{NilU} \qquad\qquad\quad \rightarrow z
\end{array}$$

$\text{type } (\alpha{:}\mathrm{A})\ \mathrm{listA} = \mathrm{NilA} \mid \mathrm{ConsA} \text{ of } \alpha \times \alpha \text{ listA} \qquad (*\ \mathrm{listA} : \mathrm{A} \Rightarrow \mathrm{A}\ *)$

$$\begin{array}{l}
\text{let rec } \mathit{foldrA}\ f\ z\ \mathit{xs} = \text{match } \mathit{xs} \text{ with} \\
\quad \mid \mathit{ConsA}(x, \mathit{xs}) \rightarrow f\ x\ (\mathit{foldrA}\ f\ z\ \mathit{xs}) \\
\quad \mid \mathit{NilA} \qquad\qquad\quad \rightarrow z
\end{array}$$

Dependent Usage Kinds

type α list = Nil | Cons of $\alpha \times \alpha$ list (* list : $\Pi\alpha.\,\langle\alpha\rangle$ *)

```
let rec foldr f z xs = match xs with
  | Cons(x,xs) → f x (foldr f z xs)
  | Nil        → z
```

Dependent Usage Kinds

$\text{type } \alpha \text{ list} = \text{Nil} \mid \text{Cons of } \alpha \times \alpha \text{ list}$ $\quad (* \text{ list} : \Pi\alpha.\, \langle\alpha\rangle *)$

$$
\begin{array}{l}
\text{let rec } \mathit{foldr}\ f\ z\ \mathit{xs} = \text{match } \mathit{xs} \text{ with} \\
\quad \mid \mathit{Cons}(x,\mathit{xs}) \rightarrow f\ x\ (\mathit{foldr}\ f\ z\ \mathit{xs}) \\
\quad \mid \mathit{Nil} \qquad\qquad\ \ \rightarrow z
\end{array}
$$

$$
\begin{array}{lcl}
(\times) & : & \Pi\alpha.\, \Pi\beta.\, \langle\alpha\rangle \sqcup \langle\beta\rangle \\
(+) & : & \Pi\alpha.\, \Pi\beta.\, \langle\alpha\rangle \sqcup \langle\beta\rangle \\
\text{ref} & : & \Pi\alpha.\, \mathsf{U} \\
\text{glSurface} & : & \Pi\alpha.\, \mathsf{A}
\end{array}
$$

Conclusion

Related Work

λ^{URAL} (Ahmed et al. 2005)
"Uniqueness Typing Simplified" (de Vries et al. 2008)

Related Work

λ^{URAL} (Ahmed et al. 2005)
"Uniqueness Typing Simplified" (de Vries et al. 2008)

System F° (Mazurak et al. 2010)
Fine (Swamy et al. 2010)
Plaid (Aldrich et al. 2009)

Alms: Practical Affine Types

Affine types:

- are for revocation
- generalize other resource-aware type systems
- don't have to be weird or difficult

Thank You

Affine types:

- are for revocation
- generalize other resource-aware type systems
- don't have to be weird or difficult

Paper: more examples and our model

Online: prototype implementation and extended paper

This frame intentionally left blank.

Why Affine Types?

Control.

$$E[\text{callcc } v] \longmapsto E[v\ (\lambda x.\text{abort } E[x])]$$

Why Affine Types?

Control.

$$E[\text{callcc } v] \longmapsto E[v\ (\lambda x.\text{abort } E[x])]$$

Why Affine Types?

Control.

$$E[\mathcal{C}\ v] \longmapsto v\ (\lambda x.\,\mathrm{abort}\ E[x])$$

Why Affine Types?

Control.

$$
\begin{aligned}
E[\mathcal{C}\ v] &\longmapsto v\ (\lambda x.\ \text{abort}\ E[x]) \\
E[\text{abort}\ e] &\longmapsto e
\end{aligned}
$$