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
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)
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)

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

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`, ...

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

However, you can optionally modify the behavior of GLSurfaceView by calling one or more of these methods before `setRenderer`:

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

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

- `setRenderMode()`
- `setDebug()`
- `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`, …

Once the renderer is set, you can control whether the renderer draws continuously or on-demand by calling `setRenderMode()`. Rendering operations are performed within the `render` method of the `GLSurfaceView`. The `setRenderMode()` method sets the rendering mode to continuous or on-demand.

![Diagram]

The diagram illustrates the process of initializing a GLSurfaceView and controlling its rendering mode. The `new GLSurfaceView` object is created, followed by calling `setRenderer()`. The rendering operations are performed within the `render` method of the `GLSurfaceView`.
Example: OpenGL on Android

All you have to do to initialize a GLSurfaceView is call `setRenderer()`.

However, you can optionally modify the behavior of GLSurfaceView by calling one or more of these methods before `setRenderer`:

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

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

Typestate

```
new GLSurfaceView
raw
setChooser, setDebug, setWrapper
setRenderer
ready
setRenderMode, setDebug, setWrapper

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

Connection established.

Server greeting.

Not authenticated.

Authenticated.

Selected.

Logout.

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

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

Practical

Special Resources

General Resources

Theoretical

(Fluet 2007; Pucella and Heller 2008)
Theoretical

Practical

Vault

Sing#

Cyclone

Special Resources

General Resources

session types calculi

region calculi

\( \lambda^{URAL} \)

ILL

(Fluet 2007; Pucella and Heller 2008)
Fluet 2007; Pucella and Heller 2008
Alms

Special Resources

Vault

Sing#

Cyclone

Theoretical

General Resources

session types calculi

region calculi

\( \chi_{\text{URAL}} \)

\( \text{ILL} \)
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
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
let newSurface () =
  let surface = create () in
  setChooser surface;
  setRenderer surface;
  setMode surface;
  setDebug surface;
  surface
let newSurface () =
  let surface = create () in
  setChooser surface; (* raw *)
  setRenderer surface; (* raw ready *)
  setMode surface; (* ready *)
  setDebug surface; (* ready *)
  surface
let newSurface () =
  let surface = create () in
  setChooser surface;
  setMode surface;
  setRenderer surface;
  setDebug surface;
  surface

(* ➡️ raw *)

(* raw *)

(* ready? *)

(* raw ➡️ ready *)

(* ready *)
module type GL_SURFACE = sig
  type raw
  type ready
  type β glSurface

  val create : unit → raw glSurface
  val setChooser : raw glSurface → unit
  val setRenderer : raw glSurface → ready glSurface
  val setMode : ready glSurface → unit
  val setDebug : ∀β. β glSurface → unit
end
let newSurface () =
    let surface = create () in
    setChooser surface;
    let surface = setRenderer surface in
    setMode surface;
    setDebug surface;
    surface
let newSurface () =
    let surface = create () in
    setChooser surface;
    setMode surface in
    let surface = setRenderer surface;
    setDebug surface;
    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
let newSurface () =
    let surface = create () in
    setChooser surface;
    let surface = setRenderer surface in
    setMode surface;
    setDebug surface;
    surface
let newSurface () =
  let surface = create () in
  let surface = setRenderer surface;
  setChooser surface in
  setMode surface;
  setDebug surface;
  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
let newSurface () =
  let surface = create () in
  let surface' = setRenderer surface;
  setChooser surface in
  setMode surface';
  setDebug surface';
  surface'
module type GL_SURFACE = sig
  type raw
  type ready
  type $\beta$ glSurface : A

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

end
An OpenGL Client: Take 3

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

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.
let newSurface () =
  let surface = create () in
  let surface′ = setRenderer surface in
  let _ = setChooser surface in
  let surface = setMode surface′ in
  let surface = setDebug surface in
  surface

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

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

An OpenGL Client: Take 3

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

(∗ ➝ raw ∗)
(∗ raw ∗)
(∗ raw ➝ ready ∗)
(∗ ready ∗)
(∗ ready ∗)
module GLSurface : GL_SURFACE = struct
  type raw = unit
  type ready = unit
  type β glSurface = PrimGLSurface.glSurface

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

    ...
end

PrimGLSurface.glSurface : U ⊑ A

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
module GLSurface : GL_SURFACE = struct
  type raw = unit
  type ready = unit
  type \( \beta \) glSurface = PrimGLSurface.glSurface

  let create = PrimGLSurface.create

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

  ...
end
More Examples

Typestate

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

Session types

\( \text{Session.send} : (!\alpha; \beta) \text{ channel} \rightarrow \alpha \Rightarrow \beta \text{ channel} \)

Regions (with adoption/focus)

\( \text{Rgn.adopt} : (\gamma,\hat{\alpha}) \text{ rgn} \rightarrow (\delta,\hat{\alpha}) \text{ rgn1} \Rightarrow \delta \text{ ptr} \Rightarrow \gamma \text{ ptr} \times (\gamma,\hat{\alpha}) \text{ rgn} \)

Strong updates

\( \text{Ref.swap} : \hat{\alpha} \text{ aref} \rightarrow \hat{\beta} \Rightarrow \hat{\beta} \text{ aref} \times \hat{\alpha} \)

Fractional capabilities

\( \text{Fractional.split} : (\beta,\gamma) \text{ cap} \rightarrow (\beta,\gamma/2) \text{ cap} \times (\beta,\gamma/2) \text{ cap} \)
Design Rationale
Linear Logic (Girard 1987):

\[
\frac{\Gamma, !B, !B \vdash \Delta}{\Gamma, !B \vdash \Delta} \quad \text{(Contraction)}
\]
The Problem

Ocaml:

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

$$\lambda f \to \text{promote } f \text{ for } g \text{ in}$$

$$\lambda p \to \text{let derelict } p \text{ be } x \otimes y$$

$$\text{in derelict } \ (\text{derelict } g \ x) \ y$$

$$: \ ! (\alpha \to ! (\beta \to \gamma)) \to ! (\! (\alpha \otimes \beta) \to \gamma)$$

$$\lambda f \ p \to \text{let derelict } p \text{ be } x \otimes y$$

$$\text{in derelict } \ (f \ x) \ y$$

$$: \ (\alpha \to ! (\beta \to \gamma)) \to ! (\alpha \otimes \beta) \to \gamma$$
The Problem

Ocaml:

\[
\lambda f \ (x,y) \rightarrow f \ x \ y
\]

: \((\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow \alpha \times \beta \rightarrow \gamma\)

ILL to Alms:

\[
\lambda f \rightarrow \text{promote } f \text{ for } g \text{ in }
\lambda p \rightarrow \text{let derelict } p \text{ be } x \otimes y
\]

in derelict \((\text{derelict } g \ x) \ y\)

: \((\alpha \rightarrow \beta \rightarrow \gamma) \xrightarrow{A} \alpha \times \beta \rightarrow \gamma\)

\[
\lambda f \ p \rightarrow \text{let derelict } p \text{ be } x \otimes y
\]

in derelict \((f \ x) \ y\)

: \((\alpha \xrightarrow{A} \beta \rightarrow \gamma) \xrightarrow{A} \alpha \times \beta \xrightarrow{A} \gamma\)
The Problem

Ocaml:

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

ILL to Alms:

\[ \lambda f (x,y) \rightarrow f \times y \]
\[ : (\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow \alpha \times \beta \rightarrow \gamma \]
\[ \lambda f (x,y) \rightarrow f \times y \]
\[ : (\alpha \uparrow \beta \rightarrow \gamma) \rightarrow \alpha \times \beta \uparrow \gamma \]
The Problem

Ocaml:

\[ \lambda f \left( x, y \right) \rightarrow f \; x \; y \]
\[ : \left( \alpha \rightarrow \beta \rightarrow \gamma \right) \rightarrow \alpha \times \beta \rightarrow \gamma \]

Alms:

\[ \lambda f \left( x, y \right) \rightarrow f \; x \; y \]
\[ : \left( \alpha \xrightarrow{\delta} \beta \rightarrow \gamma \right) \rightarrow \alpha \times \beta \xrightarrow{\delta} \gamma \]
Dereliction Subtyping

\[
\begin{align*}
\text{workerThread} &: \text{unit } \stackrel{U}{\rightarrow} \text{unit} \\
\text{Thread.fork} &: (\text{unit } \stackrel{A}{\rightarrow} \text{unit}) \stackrel{U}{\rightarrow} \text{thread}
\end{align*}
\]
Dereliction Subtyping

\[ \text{workerThread} : \text{unit} \rightarrow^U \text{unit} \]

\[ \text{Thread.fork} : (\text{unit} \rightarrow^A \text{unit}) \rightarrow^U \text{thread} \]

\[ \text{unit} \rightarrow^U \text{unit} \leq \text{unit} \rightarrow^A \text{unit} \quad (\text{U} \sqsubseteq \text{A}) \]

\[ \text{workerThread} : \text{unit} \rightarrow^A \text{unit} \]

\[ \text{Thread.fork} \text{workerThread} : \text{thread} \]
Principal Promotion

\[ \lambda x \rightarrow x \]
\[ : \alpha \xrightarrow{u} \alpha \]

\[ \lambda f \ x \rightarrow f \ x \]
\[ : (\alpha_1 \xrightarrow{\gamma} \alpha_2) \xrightarrow{u} \alpha_1 \xrightarrow{\gamma} \alpha_2 \]

\[ \lambda f \ g \ x \rightarrow f \ (g \ x) \]
\[ : (\alpha_2 \xrightarrow{\gamma} \alpha_3) \xrightarrow{u} (\alpha_1 \xrightarrow{\delta} \alpha_2) \xrightarrow{\gamma} \alpha_1 \xrightarrow{\gamma \perp \delta} \alpha_3 \]
Principal Promotion

\[ \lambda x \to x \quad : \quad \alpha \xrightarrow{u} \alpha \]

\[ \lambda f \ x \to f \ x \quad : \quad (\alpha_1 \rightarrow \alpha_2) \xrightarrow{u} \alpha_1 \rightarrow \alpha_2 \]

\[ \lambda f \ g \ x \to f \ (g \ x) \quad : \quad (\alpha_2 \rightarrow \alpha_3) \xrightarrow{u} (\alpha_1 \rightarrow \delta \alpha_2) \rightarrow \alpha_1 \xrightarrow{\gamma \sqcup \delta} \alpha_3 \]

**Theorem.** Alms’s type system finds the type with least kind for every typable function.
Usage Kinds

type $\alpha$ 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 $\rightarrow$ $z$
Usage Kinds

type $\alpha$ 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
Usage Kinds

type $\alpha$ 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 $\alpha$ 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 \((\alpha:U)\) 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 \(\rightarrow z\)
type \( (\alpha : U) \) list\(U \) = Nil\(U \) | Cons\(U \) of \( \alpha \times \alpha \) list  

\[ \text{let rec } \text{foldr}\(U \) f z xs = match xs with \]
\[ | \text{Cons} U(x, xs) \rightarrow f x (\text{foldr}\(U \) f z xs) \]
\[ | \text{Nil} U \rightarrow z \]
Usage Kinds

type \((\alpha:U)\) listU = NilU | ConsU of \(\alpha \times \alpha\) list \quad (*) \ listU : U \Rightarrow U (*)

let rec foldrU f z xs = match xs with
  | ConsU\((x, xs)\) \rightarrow f x (foldrU f z xs)
  | NilU \quad \rightarrow z

let rec foldrA f z xs = match xs with
  | ConsA\((x, xs)\) \rightarrow f x (foldrA f z xs)
  | NilA \quad \rightarrow z
Dependent Usage Kinds

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

let rec foldr $f\ z\ xs$ = match $xs$ with
  | Cons($x\ xs$) $\to f\ x\ (foldr\ f\ z\ xs)$
  | Nil $\to z$
Dependent Usage Kinds

\[
\text{let rec } \text{foldr } f \ z \ x s = \text{match } x s \text{ with } \\
\mid \text{Cons}(x, x s) \to f x (\text{foldr } f \ z \ x s) \\
\mid \text{Nil} \to z
\]

\[
\begin{align*}
\times & : \Pi \alpha. \Pi \beta. \langle \alpha \rangle \sqcup \langle \beta \rangle \\
(+) & : \Pi \alpha. \Pi \beta. \langle \alpha \rangle \sqcup \langle \beta \rangle \\
\text{ref} & : \Pi \alpha. \mathbf{U} \\
\text{glSurface} & : \Pi \alpha. \mathbf{A}
\end{align*}
\]
Conclusion
Related Work

\[ \lambda_{\text{URAL}} \text{ (Ahmed et al. 2005)} \]
“Uniqueness Typing Simplified” (de Vries et al. 2008)
Related Work

$\lambda^{\text{URAL}}$ (Ahmed et al. 2005)
"Uniqueness Typing Simplified" (de Vries et al. 2008)

System F$^\circ$ (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
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.

\[ E[\text{callcc } v] \rightarrow E[v (\lambda x. \text{abort } E[x])] \]
Why Affine Types?

Control.

\[ E[\text{callcc } v] \mapsto E[v \ (\lambda x.\text{abort } E[x])] \]
Why Affine Types?

Control.

\[ E[C \; v] \rightarrow v \; (\lambda x. \text{abort } E[x]) \]
Why Affine Types?

Control.

\[
E[C \ v] \longrightarrow v \ (\lambda x. \ abort \ E[x])
\]

\[
E[\text{abort } e] \longrightarrow e
\]