Substructural Types with Class

J. Garrett Morris
Why substructural types?

Short version: state!

• Session types: protocol captured in types
  send :: (Ch !t.s, s) → IO (Ch s)

• Type-changing update
  put :: Ref t → u → IO (Ref u)

• Destructive update
  update :: Ix n → t → Array n t → Array n t
Existing approaches

Explicit unlimited modality
- E.g., Wadler, “Linear types can change the world!”
- Syntactic overhead for unlimited types (let !)

Type modifiers (qualifiers)
- E.g., Walker, “Substructural type systems”
- Unexpected types (lin bool, unl (!t.s))
- Code multiplication

Kinds and subkinding
- E.g., Mazurak et al., “Lightweight Linear Types…”
- Code multiplication
This work

Objectives:

• Integrate substructural and unlimited types
  – No syntactic overhead (e.g., `let!`)
  – Partition types (no substructural Booleans or unlimited `!t.s`)

• Avoid code duplication
  – Particularly relevant for higher-order functions
This work

SOL: a functional substructural language
• With principal types and type inference
• Supporting existing functional idioms
• Reduction provably respects substructurality

Hurdles: implicit overloading of
• Duplication/discardng
• Application
• Abstraction
The first hurdle

Here’s an innocuous piece of code:

```
twice x = x + x
```

What do we know about the type of x:

- It must be numeric (i.e., support +)
- It must be unlimited (i.e., support duplication)
The first hurdle

Characterize unlimited types with a type class

class Unl t where
  drop :: t -> ()
  dup   :: t -> (t, t)
The first hurdle

Rewrite twice to use Unl:

\[
\text{twice} :: (\text{Num } t, \text{Unl } t) \Rightarrow t \rightarrow t
\]
\[
\text{twice } x = y + z
\]
where \((y, z) = \text{dup } x\)

Really want use of \text{dup} and \text{drop} to be implicit:

\[
\text{twice} :: (\text{Num } t, \text{Unl } t) \Rightarrow t \rightarrow t
\]
\[
\text{twice } x = x + x
\]
Qualified types

System of qualified types in one slide

\[ \begin{align*}
\tau, \nu, \varphi & : = t \mid K \mid \tau \rightarrow \nu \\
\pi & : = \ldots \\
\rho & : = \tau \mid \pi \Rightarrow \rho \\
\sigma & : = \rho \mid \forall t. \sigma
\end{align*} \]

\[ \begin{align*}
\frac{P \supseteq Q \quad \land \{ \vdash P \Rightarrow \pi \mid \pi \in Q \}}{\vdash P \Rightarrow Q} \end{align*} \]

\[ \begin{align*}
P, \pi \mid \Gamma \vdash M : \rho & \quad P \mid \Gamma \vdash M : \pi \Rightarrow \rho \\
\frac{}{P \mid \Gamma \vdash M : \pi \Rightarrow \rho} & \quad P \mid \Gamma \vdash M : \rho \\
\frac{x : \sigma \in \Gamma}{P \mid \Gamma \vdash x : \sigma} & \quad P \mid \Gamma \vdash M : \tau \rightarrow \nu \\
\frac{}{P \mid \Gamma \vdash N : \tau} & \quad P \mid \Gamma \vdash MN : \nu
\end{align*} \]
Substructural qualified types

Substructural variant:

\[ \begin{align*}
\tau, \nu, \varphi & ::= t \mid K \mid \tau \rightarrow \nu \mid \tau \circlearrowright \nu \mid \tau \bigoplus \nu \\
\pi & ::= \text{Unl} \tau \mid (\leadsto) \tau \mid \tau \geq \nu \\
\rho & ::= \tau \mid \pi \Rightarrow \rho \\
\sigma & ::= \rho \mid \forall t. \sigma
\end{align*} \]

\[ \begin{align*}
P \supseteq Q & \quad \frac{P \Rightarrow Q}{\vdash P \Rightarrow Q} \\
\land \{ \vdash P \Rightarrow \pi \mid \pi \in Q \} & \quad \frac{\vdash P \Rightarrow Q}{\vdash P \Rightarrow Q} \\
& \quad \vdots
\end{align*} \]

\[ \begin{align*}
P, \pi \mid H \vdash M: \rho & \quad P \mid H \vdash M: \pi \Rightarrow \rho \\
\hline
P \mid H \vdash M: \pi : \rho & \quad P \mid H \vdash M: \rho
\end{align*} \]

\[ \begin{align*}
P, x: \sigma \vdash x: \sigma & \quad P \mid H \vdash M: \tau \rightarrow \nu \\
\hline
P \mid H \vdash M: \nu & \quad P \mid H' \vdash N: \tau \\
\hline
P \mid H, H' \vdash MN: \nu
\end{align*} \]
The first hurdle

Contraction and weakening in terms of Unl:

\[
\begin{align*}
  \vdash P | H, x: \sigma, x: \sigma \vdash M: \sigma' & \quad P \vdash \sigma \text{ unl} \\
  \vdash P | H, x: \sigma \vdash M: \sigma' & \qquad \quad \vdash P | H \vdash M: \sigma' \quad P \vdash \sigma \text{ unl} \\
  \vdash P | H, x: \sigma \vdash M: \sigma' & \qquad \quad P \vdash \sigma \text{ unl}
\end{align*}
\]

Lifting Unl:

\[
\begin{align*}
  \vdash P \Rightarrow \text{Unl } \tau & \quad \vdash P, \pi \vdash \rho \text{ unl} \\
  \vdash P \vdash \tau \text{ unl} & \quad \vdash P \vdash \pi \Rightarrow \rho \text{ unl} \\
  \vdash P, \text{Unl } t \vdash \sigma \text{ unl} & \quad \vdash P \vdash \forall t. \sigma \text{ unl}
\end{align*}
\]
Unlimited types

When is a pair \((t, u)\) unlimited?

\[
\text{instance } \text{Unl} \ t, \text{Unl} \ u \Rightarrow \text{Unl} \ t, \text{Unl} \ u \text{ where}
\]
\[
\text{dup} \ (x, y) = ((x, x'), (y, y'))
\]
\[
\text{where} \ (x, x') = \text{dup} \ x
\]
\[
(y, y') = \text{dup} \ y
\]
\[
\text{drop} \ (x, y) = \text{drop} \ y
\]
\[
\text{where} \ () = \text{drop} \ x
\]
Unlimited types

When is a sum (Either t u) unlimited?

instance (Unl t, Unl u) ⇒ Unl (Either t u) where
dup (Left x) = (Left x, Left y)
    where (x,y) = dup x
dup (Right x) = (Right x, Right y)
    where (x,y) = dup x
drop …
Unlimited types

Rules for Unl predicates

\[ \vdash P \Rightarrow Unl \tau \quad \vdash P \Rightarrow Unl \nu \]

\[ \vdash P \Rightarrow Unl (\tau, \nu) \]

\[ \vdash P \Rightarrow Unl (\tau \oplus \nu) \]
What about functions?

When is a function from \( t \) to \( u \) unlimited?

- Intuition: when the captured environment contains only unlimited values
- Not observable from \( t \) and \( u \)

Consequence: need distinct function types

\[
\vdash P \Rightarrow \text{Unl}(\tau \rightarrow \nu) \quad \not\vdash \text{Unl}(\tau \rightarrow \nu)
\]
The second hurdle

Multiple function types make everything worse

\[
\text{app} \ (f, \ x) = f \ x
\]

can have either of the types

\[
\text{app} :: (t \to u, \ t) \to u
\]
\[
\text{app} :: (t \circlearrowright u, \ t) \to u
\]

Problem: overloading of juxtaposition
The second hurdle

Characterize function-like things:

```haskell
class (∼) f where
  app :: (f a b, a) → b

instance (∼) (↝) where ...
instance (∼) (⊸) where ...
- no other instances of (∼)
```
The second hurdle

Characterize function-like things:

```haskell
class (~) f where
    app :: (f a b, a) -> b
```

Syntactic sugar:

```haskell
\( t \sim u \equiv t \overset{\sim}{f} u \equiv (~) f \Rightarrow f \; t \; u \)
```
The second hurdle:

Application overloaded for function types:

\[
\frac{P \mid H \vdash M : \varphi \tau \nu \quad P \mid H' \vdash N : \tau \quad \vdash P \Rightarrow (\sim)\varphi}{P \mid H, H' \vdash M \, N : \nu}
\]

\[
\vdash P \Rightarrow (\sim)(\rightarrow) \quad \vdash P \Rightarrow (\sim)(\neg)\]
The third hurdle

Why was `app` uncurried?

\[ \text{app'} \ f \ x = f \ x \]

can have any of the types

\[ \text{app'} :: (t \rightarrow u) \rightarrow t \rightarrow u \]
\[ \text{app'} :: (t \rightarrow u) \rightarrow t \rightarrow u \]
\[ \text{app'} :: (t \rightarrow u) \rightarrow t \rightarrow u \]
\[ \text{app'} :: (t \rightarrow u) \rightarrow t \rightarrow u \]
The third hurdle

Why was `app` uncurried?

\[ \text{app'} \ f \ x = f \ x \]

can use \( \rightsquigarrow \) to abstract argument type

\[ \text{app'} :: (t \rightsquigarrow u) \rightarrow t \rightarrow u \]

but result type is still too linear
The third hurdle

Express result linearity as function of argument linearity:

\[
\text{app}' :: (t \nrightsquigarrow u) \rightarrow t \nrightsquigarrow u
\]
\[
\text{app}' = \forall f \rightarrow \forall x \rightarrow f \, x
\]

- Linearity of a λ-term depends on its captured environment
- Doesn't capture \((t \rightarrow u) \rightarrow t \rightarrow u\)
The third hurdle

Express result linearity in relation to argument linearity:

\[
\text{app'} :: f \geq g \Rightarrow (t \triangleleft u) \rightarrow t \triangleright u
\]

\[
\text{app'} f x = f x
\]

• Unlimited \( \geq \) linear
The third hurdle

Linearity relationships need not be among function types

\[ p :: (t \geq f) \Rightarrow t \rightarrow (u \sim f (t,u)) \]

\[ p \ x \ y = (x,y) \]
The third hurdle

Linearity of a function depends on its environment:

\[
P | H, x: \tau \vdash M: \nu \quad \vdash P \Rightarrow (\sim)\phi \quad P \vdash H \geq \phi
\]

\[
P | H_x \vdash \lambda x. M: \phi\tau\nu
\]
Defining \( \geq \)

Characterize “more unlimited than” relationship:

\[
\text{class } t \geq u
\]

Cunning observation: only needs to be defined for function types

\[
\begin{align*}
\text{instance } & (u \to v) \geq t \\
\text{instance } & t \geq (u \to v) \\
\text{no other instances of } & \geq
\end{align*}
\]
Defining $\geq$

Ordering for functions:

$$\vdash (\tau \rightarrow \tau') \geq \nu \quad \vdash \tau \geq (\nu \circ \nu')$$

Lifting:

$$\vdash P \Rightarrow \tau \geq \phi$$

$$\frac{P, \pi \vdash \rho \geq \phi}{P \vdash (\pi \Rightarrow \rho) \geq \phi}$$

$$\vdash \sigma \geq \phi$$

$$\vdash \forall t. \sigma \geq \phi$$

$$\Lambda\{P \vdash \sigma \geq \phi | x : \sigma \in H\}$$

$$\vdash H \geq \phi$$
SOL syntax-directed typing

Representative rules:

\[
\begin{align*}
P \vdash^S \Gamma \text{ un} \quad (Q \Rightarrow \tau) \sqsubseteq \sigma & \quad \vdash P \Rightarrow Q \\
\hline
P \mid \Gamma, x: \sigma \vdash^S x: \tau
\end{align*}
\]

\[
\begin{align*}
P \mid \Gamma, \Delta \vdash^S M: \varphi \tau \nu & \quad \vdash P \Rightarrow (\neg) \varphi \\
\hline
P \mid \Gamma, \Delta' \vdash^S N: \tau \\
\hline
P \mid \Gamma, \Delta, \Delta' \vdash^S MN: \nu
\end{align*}
\]

Representative results:

- If \( P \mid \Gamma \vdash^S M: \tau \) and \( P \vdash H \approx \Gamma \), then \( P \mid H \vdash M: \tau \)
- If \( P \mid H \vdash M: \sigma \) and \( P \vdash H \approx \Gamma \), then \( Q \mid \Gamma \vdash^S M: \tau \) where \((P|\sigma) \sqsubseteq Gen(\Gamma, Q \Rightarrow \tau)\).
SOL type inference

Representative rules:

\[ M(S; \Gamma; x; \tau) = UP; U \circ S; \{x\} \]
where \((x: \forall \tilde{t}. P \Rightarrow \nu) \in S\Gamma\)
and \(U = \text{Unify}(\nu[t_i := u_i], \tau)\)

\[ M(S; \Gamma; \lambda x. M; \tau) = TQ; T; \Sigma \setminus x \]
where \(P; T; \Sigma = M(\text{Unify}(\tau, u_1 u_2 u_3) \circ S; \Gamma, x: u_2; M; u_3)\)
and \(Q = \{ (\sim)u_1 \} \cup \text{Leq}(u_1, \Gamma|\Sigma) \cup \text{Weaken}(x, u_2, \Sigma)\)

\[ \text{Leq}(\phi, \Gamma) = \{ \tau \geq \phi: (x: \tau) \in \Gamma\Sigma\} \]
\[ \text{Weaken}(x, \tau, \Sigma) = \{ \text{Unl} \tau \} \text{ if } x \notin \Sigma \]
**SOL type inference**

Representative results:

- If \( M(S; \Gamma; M; \tau) = P; T; \Sigma, \)
  
  then \( TP \mid T(S\Gamma) \vdash^S M : T(S\tau). \)

- If \( SP \mid S\Gamma \vdash^S M : \tau, \)
  
  then \( M(\varepsilon; \Gamma; M; t) = Q; T; \Sigma \) such that:
  
  - \( S = S' \circ T \)
  
  - \( \vdash SP \Rightarrow S'(TQ) \)
  
  - \( \tau = S'(Tt) \)
Principal types for SOL

If $P_0|H \vdash M: \sigma_0$ and $P_1|H \vdash M: \sigma_1$, then there is some $\sigma$ such that

1. $\emptyset|H \vdash M: \sigma$
2. $(P_0|\sigma_0) \sqsubseteq \sigma$
3. $(P_1|\sigma_1) \sqsubseteq \sigma$. 
Continuing work

• Relevant/affine type systems
  – Extended definition of $\geq$, more arrows
  – Otherwise, should extend directly

• Bounded linearity
  – Related: fractional permissions
  – Problem: arrows
  – Subset where inference is possible?

• Implementation