Clamp

Type Classes for Substructural Types

Edward Gan

Advisors: Greg Morrisett and Jesse Tov

April 16, 2013

Statically Tracking State

- A common bug:

Incorrect File Handle Usage

```
let filetest () =
     let fhd = open "testfile" in
     write "initial output" fhd;
     close fhd;
     write "final output" fhd
```

- File Handles are state-ful resources, not substitutable values.
- How to track the fact that the handle is "consumed"?

Substructural Types

- Lambda Calculus with Substructural Rules

$$\text{Var}\ \frac{}{x : \tau \vdash x : \tau} \qquad \text{Lam}\ \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}$$

$$\text{App}\ \frac{\Gamma_1 \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma_2 \vdash e_2 : \tau_1}{\Gamma_1, \Gamma_2 \vdash e_1\ e_2 : \tau_2}$$

$$\text{Weakening}\ \frac{\Gamma \vdash e : \tau}{\Gamma, x : \tau' \vdash e : \tau} \qquad \text{Contraction}\ \frac{\Gamma, x : \tau' \vdash e : \tau}{\Gamma, x : \tau', x : \tau' \vdash e : \tau}$$

- Restricting Substructural Operations

Unlimited	Weakening and Contraction, arbitrary usage
Affine	Weakening, used at most once
Relevant	Contraction, used at least once
Linear	Neither Weakening nor Contraction, used exactly once

A Stateful File-I/O Library

- Suppose we have a type system with linear types

File I/O Library Interface

```
type filehandle : linear
val open : string -> filehandle
val write : string -> filehandle -> filehandle
val close : filehandle -> unit
```

- File Handle Misuse $\mapsto$ File Handle Reuse

Statically Incorrect File Usage

```
let filetest () =
     let fhd = open "testfile" in
     let fhd2 = write "initial output" fhd;
     close fhd2;
     write "final output" fhd2
```

Existing Substructural Languages

- Qualifier Based: λ^{URAL}, ATAPL
  - Break types τ into qualifier ξ and pretype $\overline{\tau}$, $\tau ::=^{\xi} \overline{\tau}$.
  - ξ determines substructural properties.
  - Verbose Polymorphism

$$\mathit{pair} : \forall \xi_1 : \mathsf{Q}.\ \forall \tau_1 : \overline{\star}, \tau_2 : \overline{\star}.\ \left({}^{\xi_1}\tau_1\right)^{\mathbf{U}} \multimap \left({}^{\xi_1}\tau_2\right)^{\mathbf{U}} \multimap^{\xi_1} \left(\left({}^{\xi_1}\tau_1\right) \otimes \left({}^{\xi_1}\tau_2\right)\right)$$

- Kind Based: Alms, F°, Clean
  - Assign types τ a kind κ that determines substructural properties, e.g. $\vdash \mathsf{int} : \mathbf{U}$
  - Polymorphism through subkinding, dependent kinds

Alms-K-Prod

$$\frac{\vdash \Gamma}{\Gamma \vdash (\otimes) : \Pi\alpha^{+}.\Pi\beta^{+}.\ \langle\alpha\rangle \sqcup \langle\beta\rangle}$$

$$\mathit{pair} : \forall \alpha : \mathbf{L}, \beta : \mathbf{L}.\ \alpha \to \beta \to \alpha \otimes \beta$$

The Clamp Programming Language

- Encode the different "kinds" of substructural types in terms of the supported substructural operations

Substructural Type Classes

```
class Dup a where
     dup :: a -> (a,a)

class Drop a where
     drop :: (a,b) -> b
```

- Benefits
  - Uniform Meta-theory
  - Cheap Polymorphism over U,R,A,L
  - Easy to add-on stateful built-ins (s/w references)
  - Orthogonal Implementation
  - Type Classes!

Clamp Examples

- dup and drop operations implicit
- Annotated arrows $\alpha \xrightarrow{x} \beta$ for $x = \mathbf{U}, \mathbf{R}, \mathbf{A}, \mathbf{L}$

Substructural Restrictions

```
let mygold = @minegold unit in
(fun a -L> (a,a)) (1,mygold) //Invalid
```

- fst : $\forall \alpha, \beta\, [\text{Drop } \beta] . \alpha \times \beta \xrightarrow{\mathbf{U}} \alpha$

Polymorphism and Datatypes

```
let fst = fun p -U>
     letp (p1, p2) = p in
     p1
```

Strong and Weak references

- Weak update: update contents of mutable reference with another of same type
  - Always type safe
- Strong update: update contents to value with different type
  - Can be unsound if aliased
- Key operations
  - swap: $\mathsf{ref}^{\mathsf{rq}}\alpha \times \alpha \xrightarrow{\mathbf{U}} \mathsf{ref}^{\mathsf{rq}}\alpha \times \alpha$
  - sswap: $\mathsf{ref}^{\mathsf{s}}\alpha \times \beta \xrightarrow{\mathbf{U}} \mathsf{ref}^{\mathsf{s}}\beta \times \alpha$
  - release: $\mathsf{ref}^{\mathsf{rq}}\alpha \xrightarrow{\mathbf{U}} \mathsf{unit} + \alpha$
  - srelease: $\mathsf{ref}^{\mathsf{s}}\alpha \xrightarrow{\mathbf{U}} \alpha$
- Need U,R,A,L: weak reference to linear data can aliased but not arbitrarily disposed

λ_{cl} Syntax

$$
\begin{aligned}
e ::= {} & x \mid \lambda^{aq}(x : \tau).e \mid e_1\ e_2 \mid \Lambda\overline{\alpha_i}\,[P].v \mid e\,[\overline{\tau_i}] \\
& \mid (e_1, e_2) \mid \textbf{inl}\ e \mid \textbf{inr}\ e \mid () \\
& \mid \textbf{letp}\ (x_1, x_2) = e\ \textbf{in}\ e_2 \\
& \mid \textbf{match}\ e\ \textbf{with}\ \textbf{inl}\ x_1 \rightarrow e_1; \textbf{inr}\ x_2 \rightarrow e_2 \\
& \mid \ell \mid \textbf{new}^{\textbf{rq}}\ e \mid \textbf{release}^{\textbf{rq}}\ e \mid \textbf{swap}^{\textbf{rq}}\ e_1\ \textbf{with}\ e_2 \\
& \mid \textbf{dup}\ e_1\ \textbf{as}\ x_1, x_2\ \textbf{in}\ e_2 \mid \textbf{drop}\ e_1\ \textbf{in}\ e_2 \\
\mathrm{rq} ::= {} & \textbf{s}\ (\text{strong}) \mid \textbf{w}\ (\text{weak}) \\
\mathrm{aq} ::= {} & \textbf{U}\ (\text{unlimited}) \mid \textbf{R}\ (\text{relevant}) \mid \textbf{A}\ (\text{affine}) \mid \textbf{L}\ (\text{linear})
\end{aligned}
$$

$$
\begin{aligned}
\tau ::= {} & \alpha \mid \tau_1 \xrightarrow{aq} \tau_2 \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid \mathrm{ref}^{\mathrm{rq}}\tau \mid \forall\overline{\alpha_i}\,[P].\tau \\
P ::= {} & \mathrm{Pred}_1, \ldots, \mathrm{Pred}_n \\
\mathrm{Pred} ::= {} & K\tau \\
K ::= {} & \mathrm{Dup} \mid \mathrm{Drop}
\end{aligned}
$$

λ_{cl} Type System

- Core

$$\text{Lam}\ \frac{P;\Gamma,x:\tau_1;\Sigma\vdash e:\tau_2 \qquad P\Vdash \mathrm{Constrain}^{aq}(\Gamma,\Sigma)}{P;\Gamma;\Sigma\vdash \lambda^{aq}(x:\tau_1).e:\tau_1\xrightarrow{aq}\tau_2}$$

$$\text{App}\ \frac{P;\Gamma_1;\Sigma_1\vdash e_1:\tau_2\xrightarrow{aq}\tau \qquad P;\Gamma_2;\Sigma_2\vdash e_2:\tau_2}{P;\Gamma_1\circ\Gamma_2;\Sigma_1+\Sigma_2\vdash e_1\ e_2:\tau}$$

- Type Class Constraints

$$\text{TAbs}\ \frac{P_1,P_2;\Gamma;\Sigma\vdash v:\tau \qquad \mathrm{Dom}(P_2)\subset\overline{\alpha_i}}{P_1;\Gamma;\Sigma\vdash \Lambda\overline{\alpha_i}[P_2].v:\forall\overline{\alpha_i}[P_2].\tau}$$

$$\text{TApp}\ \frac{P_1;\Gamma;\Sigma\vdash e:\forall\overline{\alpha_i}[P_2].\tau \qquad P_1\Vdash P_2\overline{\{\tau_i/\alpha_i\}}}{P_1;\Gamma;\Sigma\vdash e[\overline{\tau_i}]:\tau\overline{\{\tau_i/\alpha_i\}}}$$

λ_{cl} Type System continued

- Substructural

$$\text{Dup}\ \frac{\begin{array}{c} P; \Gamma_1; \Sigma_1 \vdash e_1 : \tau_1 \\ P; \Gamma_2, x_1 : \tau_1, x_2 : \tau_1; \Sigma_2 \vdash e_2 : \tau_2 \qquad P \Vdash \text{Dup } \tau_1 \end{array}}{P; \Gamma_1 \circ \Gamma_2; \Sigma_1 + \Sigma_2 \vdash \textbf{dup } e_1 \textbf{ as } x_1, x_2 \textbf{ in } e_2 : \tau_2}$$

$$\text{Drop}\ \frac{P; \Gamma_1; \Sigma_1 \vdash e_1 : \tau_1 \qquad P; \Gamma_2; \Sigma_2 \vdash e_2 : \tau_2 \qquad P \Vdash \text{Drop } \tau_1}{P; \Gamma_1 \circ \Gamma_2; \Sigma_1 + \Sigma_2 \vdash \textbf{drop } e_1 \textbf{ in } e_2 : \tau_2}$$

- Linear Variable environments Γ, Reference counted location environments Σ

$$\begin{aligned} \Gamma &::= x_1 : \tau_1, \ldots, x_n : \tau_n \\ \Sigma^s &::= \ell_1 \mapsto_{\mathbf{s}} \tau_1, \ldots, \ell_n \mapsto_{\mathbf{s}} \tau_n \\ \Sigma^w &::= \ell_1 \mapsto_{\mathbf{w}}^{j_1} \tau_1, \ldots, \ell_n \mapsto_{\mathbf{w}}^{j_n} \tau_n \qquad j_i > 0 \\ \Sigma &::= \Sigma^s, \Sigma^w \qquad \text{Dom}\left(\Sigma^s\right) \cap \text{Dom}\left(\Sigma^w\right) = \emptyset \end{aligned}$$

Type Class Instances

$$
\begin{aligned}
\mathrm{Dup}\ a, \mathrm{Dup}\ b &\implies \mathrm{Dup}\ \ (a \times b) \\
\mathrm{Dup}\ a, \mathrm{Dup}\ b &\implies \mathrm{Dup}\ \ (a + b) \\
nil &\implies \mathrm{Dup}\ \ \left(a \xrightarrow{U} b\right) \\
nil &\implies \mathrm{Dup}\ \ \left(a \xrightarrow{R} b\right) \\
nil &\implies \mathrm{Dup\ unit} \\
nil &\implies \mathrm{Dup}\ \ (\mathrm{ref}^{\mathrm{w}} a)
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{Drop}\ a, \mathrm{Drop}\ b &\implies \mathrm{Drop}\ \ (a \times b) \\
\mathrm{Drop}\ a, \mathrm{Drop}\ b &\implies \mathrm{Drop}\ \ (a + b) \\
nil &\implies \mathrm{Drop}\ \ \left(a \xrightarrow{U} b\right) \\
nil &\implies \mathrm{Drop}\ \ \left(a \xrightarrow{A} b\right) \\
nil &\implies \mathrm{Drop\ unit} \\
\mathrm{Drop}\ a &\implies \mathrm{Drop}\ \ (\mathrm{ref}^{\mathrm{rq}} a)
\end{aligned}
$$

- Very compact representation of kinding rules, reference qualifier restrictions, etc...

Type Soundness

- Two Key Lemmas to prove Preservation

Theorem

Constraints Capture Locations:
Consider $P; \Gamma; \Sigma \vdash v : \tau$. *If* $P \Vdash Dup\ \tau$ *then* $P \Vdash Dup\ \Sigma, Dup\ \Gamma$. *Similarly if* $P \Vdash Drop\ \tau$ *then* $P \Vdash Drop\ \Sigma, Drop\ \Gamma$.

Theorem

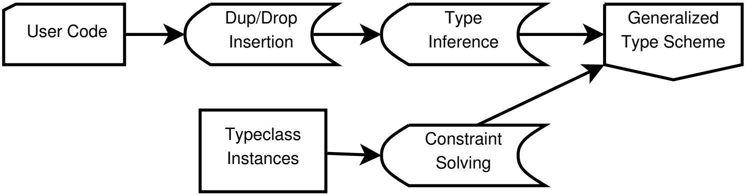
Substitution:
If $P; \Gamma, x : \tau_x; \Sigma_1 \vdash e : \tau$ *and* $P; \cdot; \Sigma_2 \vdash v : \tau_x$ *and* $\Sigma_1 \smile \Sigma_2$ *then*
$P; \Gamma; \Sigma_1 + \Sigma_2 \vdash e\{v/x\} : \tau$

Dup/Drop Insertion

- Writing dup and drop operations by hand a pain
- What would we like an automated insertion algorithm to do?
  - Use memory efficiently
  - Assume minimum number of Dup/Drop constraints
- Optimal Algorithm
  - Bottom up recursive
  - Annotate to minimize number of assumptions required at each level
  - Can prove: global memory usage minimized, no extraneous constraints

Implementation

- Overall Design



- Based off of a Haskell Type-checker with a few additions:
  - A dup/drop insertion pass
  - Substructural type class instances
  - Constraints to closure environments in the type inference step

Summary

- Why Clamp is interesting
  - Simple theory and metatheory built on established tools
  - Rich enough to encode URAL and strong/weak references easily
  - Implementation piggybacks off Haskell
- Other Aspects of Research
  - Substructural inference algorithm independently interesting
  - Type Classes are fun
- Future work
  - Custom dup/drop
  - Arrow Polymorphism
  - Runtime Considerations