Abstraction Elimination for Kappa Calculus

Adam Megacz
megacz@cs.berkeley.edu

9.Nov.2011

Outline

- **Overview**: the problem and solution in broad strokes
- **Detail**: kappa calculus and abstraction elimination
- **Applications**: some initial results

Part I: Overview

Lambda calculus is great

- It has first-class functions
- Unfortunately: first-class functions don't work everywhere
  - Circuits (Lava, Hawk, ...)
  - Nested Data Parallelism (NESL, Nepal, DPH)
  - Dataflow Concurrency (Kahn-McQueen)
  - GPU Shader Programs
  - Cryptocurrency contracts

Lambda calculus without first-class functions?

- Usual strategy: the "grammar hack":

$$\sigma ::= \texttt{Int} \mid \texttt{Bool} \mid \ldots \qquad \text{(ground types)}$$
$$\tau ::= \sigma \mid \sigma\texttt{->}\tau \qquad \text{(first-order types)}$$

  - Complicates (or ruins) principal types
  - Poor interaction with polymorphism
  - Two sorts of types means lots of things (type variables, etc) have to come in two flavors

Lambda calculus without first-class functions?

- Alternative: *Kappa calculus* (Lambek, Hasegawa)
- In a nutshell:
  - Each term has a *pair* of types: its *source type* and *target type*
  - *Compose* functions instead of *applying* them.
  - The source type of free variables is always the unit type.
- Unfortunately, programming without first-class functions is very repetitive
- I propose: use a *heterogeneous* two-level language.
  - Kappa calculus based object language
  - Lambda calculus based meta language
- In *homogeneous* two-level languages, you build an object program in order to *run* it.
  - In a *heterogeneous* two-level language you build object programs for other reasons.
  - I propose: to realize object language programs as *combinator expressions*.

Part II: Details

Kappa Calculus: Principles

Simply-typed lambda calculus is a notation for natural deduction proof terms in which:

- Propositions are closed under implication (->)
- *Modus ponens* as the basic inference rule

Simply-typed *kappa* calculus is a notation for natural deduction proof terms in which:

- Propositions are *always* implications
- *Cut* is the basic inference rule
- The *deduction theorem* holds for only some propositions.

Lambda Calculus

$$\tau := \texttt{Int} \mid \texttt{Bool} \mid \tau\texttt{->}\tau \mid \ldots$$
$$\texttt{e} := \texttt{x} \mid \texttt{e}\ \ \texttt{e} \mid \lambda\texttt{x.e}$$

$$[\text{Var}]\ \frac{}{\texttt{x}:\tau \qquad \vdash \texttt{x}:\tau}$$

$$[\text{App}]\ \frac{\Gamma_1 \vdash \texttt{e}_1:\tau_1 \qquad\qquad \Gamma_2 \vdash \texttt{e}_2:\tau_1\texttt{->}\tau_3}{\Gamma_1,\Gamma_2 \vdash \texttt{e}_2\ \ \texttt{e}_1:\tau_3}$$

$$[\text{Abs}]\ \frac{\Gamma,\texttt{x}:\tau_1 \qquad \vdash \texttt{e}:\tau_3}{\Gamma \vdash (\lambda\texttt{x.e}):\tau_1\texttt{->}\tau_3}$$

Lambda Calculus vs. Kappa Calculus

$$\tau := \texttt{Int} \mid \texttt{Bool} \mid 1 \mid \tau \times \tau \mid \ldots$$

$$\mathtt{e} := \mathtt{x} \mid \mathtt{e} \circ \mathtt{e} \mid \kappa\mathtt{x}.\mathtt{e} \mid \texttt{raise}_\tau(e)$$

Note: `->` is no longer part of the grammar for types; instead, $e : \tau_1 \texttt{->} \tau_2$ is a ternary relation asserting e has source type τ_1 and target type τ_2.

$$[\text{Var}]\ \frac{}{\mathtt{x}{:}\tau\texttt{->}\tau' \vdash \mathtt{x}{:}\tau\texttt{->}\tau'}$$

$$[\text{App}]\ \frac{\Gamma_1 \vdash \mathtt{e}_1 : \tau_1\texttt{->}\tau_2 \qquad \Gamma_2 \vdash \mathtt{e}_2 : \tau_2\texttt{->}\tau_3}{\Gamma_1, \Gamma_2 \vdash \mathtt{e}_2 \circ \mathtt{e}_1 : \tau_1\texttt{->}\tau_3}$$

$$[\text{Abs}]\ \frac{\Gamma, \mathtt{x}{:}1\texttt{->}\tau_1 \vdash \mathtt{e} : \tau_2\texttt{->}\tau_3}{\Gamma \vdash (\kappa\mathtt{x}.\mathtt{e}) : (\tau_1 \times \tau_2)\texttt{->}\tau_3}$$

$$[\text{Raise}]\ \frac{\Gamma \vdash e : \tau_1\texttt{->}\tau_2}{\Gamma \vdash \texttt{raise}_{\tau_3}(e) : (\tau_1 \times \tau_3)\texttt{->}(\tau_2 \times \tau_3)}$$

Type Equality vs. Isomorphism

In Hasegawa's work,

$$1 \times A = A = A \times 1$$

$$(A \times B) \times C = A \times (B \times C)$$

Later I will weaken these from *equalities* to *isomorphisms*, which will clutter the presentation. For now, I will invoke a rule "[Cheat]" when I appeal to these equalities.

Useful Definitions

$$\texttt{id} = \kappa x.x \qquad : \tau\texttt{->}\tau$$

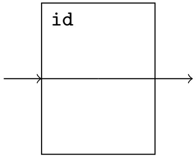
$$\begin{aligned}
\texttt{drop} &= \kappa x.id_1 && : \tau\texttt{->}1 \\
\texttt{copy} &= \kappa x.raise(x) \circ x && : \tau\texttt{->}(\tau \times \tau) \\
\texttt{swap} &= \kappa x.\kappa y.raise(y) \circ x && : (\tau_1 \times \tau_2)\texttt{->}(\tau_2 \times \tau_1)
\end{aligned}$$

$$\texttt{lower}_\tau(e) \overset{def}{=} \texttt{swap} \circ raise_\tau(e) \circ \texttt{swap} \qquad : (\tau_1 \times \tau_2)\texttt{->}(\tau_2 \times \tau_1)$$
$$(\text{if } e : \tau_1\texttt{->}\tau_2)$$

$$e_1 e_2 \overset{def}{=} e_1 \circ raise(e_2)$$

Useful Definitions: `id`

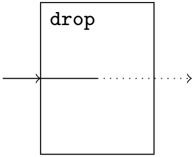
$$\mathtt{id}_\tau = \kappa x.x \qquad : \tau\texttt{->}\tau$$



$$\dfrac{\dfrac{}{x{:}1\texttt{->}\tau \vdash x \;:\; 1\texttt{->}\tau}\,[\mathrm{Var}]}{\dfrac{\vdash \kappa x.x \;:\; \tau \times 1 \;\texttt{->}\; \tau}{\vdash \kappa x.x \;:\; \tau \;\texttt{->}\; \tau}\,[\mathrm{Cheat}]}\,[\mathrm{Abs}]$$

Useful Definitions: drop

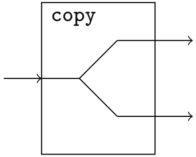
$\texttt{drop} = \kappa x.id_1 \qquad : \tau\texttt{->}1$



$$
\cfrac{
  \cfrac{
    \cfrac{\vdots}{\vdash id \ : \ 1\texttt{->}1}
  }{x{:}1\texttt{->}\tau \vdash id \ : \ 1\texttt{->}1}\,[\text{Weak}]
}{
  \cfrac{\vdash \kappa x.id \ : \ \tau \times 1 \texttt{ -> } 1}{\vdash \kappa x.id \ : \ \tau \texttt{ -> } 1}\,[\text{Cheat}]
}\,[\text{Abs}]
$$

Useful Definitions: copy

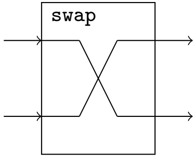
$$\mathtt{copy} = \kappa x.\mathit{raise}(x) \circ x \qquad : \tau \texttt{ -> } \tau \times \tau$$



$$\dfrac{\dfrac{\dfrac{\dfrac{}{x{:}1\texttt{->}\tau \vdash x \;:\; 1\texttt{->}\tau}\,[\text{Var}]}{x{:}1\texttt{->}\tau \vdash \mathit{raise}(x) \;:\; 1 \times \tau\texttt{->}\tau \times \tau}\,[\text{Raise}]}{x{:}1\texttt{->}\tau \vdash \mathit{raise}(x) \;:\; \tau\texttt{->}\tau \times \tau}\,[\text{Cheat}] \qquad \dfrac{}{x{:}1\texttt{->}\tau \vdash x \;:\; 1 \texttt{ -> } \tau}\,[\text{Var}]}{\dfrac{x{:}1\texttt{->}\tau, x{:}1\texttt{->}\tau \vdash \mathit{raise}(x) \circ x \;:\; 1 \texttt{ -> } \tau \times \tau}{\dfrac{x{:}1\texttt{->}\tau \vdash \mathit{raise}(x) \circ x \;:\; 1 \texttt{ -> } \tau \times \tau}{\dfrac{\vdash \kappa x.\mathit{raise}(x) \circ x \;:\; \tau \times 1 \texttt{ -> } \tau \times \tau}{\vdash \kappa x.\mathit{raise}(x) \circ x \;:\; \tau \texttt{ -> } \tau \times \tau}\,[\text{Cheat}]}\,[\text{Abs}]}\,[\text{Contr}]}\,[\text{Comp}]$$

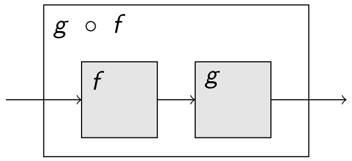
Useful Definitions: `swap`

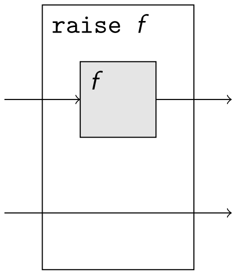
$$\texttt{swap} = \kappa x.\kappa y.\mathit{raise}(y) \circ x \qquad : \tau_1 \times \tau_2 \texttt{ -> } \tau_2 \times \tau_1$$

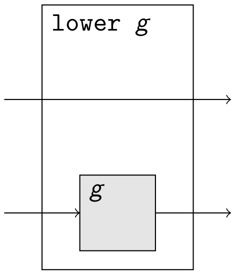


$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{y{:}1\texttt{->}\tau_2 \vdash y \;:\; 1\texttt{->}\tau_2}\,[\text{Var}]}{y{:}1\texttt{->}\tau_2 \vdash \mathit{raise}(y) \;:\; 1 \times \tau_1\texttt{->}\tau_2 \times \tau_1}\,[\text{Raise}]}{y{:}1\texttt{->}\tau_2 \vdash \mathit{raise}(y) \;:\; \tau_1\texttt{->}\tau_2 \times \tau_1}\,[\text{Cheat}] \qquad \dfrac{}{x{:}1\texttt{->}\tau_1 \vdash x{:}1\texttt{->}\tau_1}\,[\text{Var}]}{y{:}1\texttt{->}\tau_2, x{:}1\texttt{->}\tau_1 \vdash \mathit{raise}(y) \circ x \;:\; 1 \texttt{ -> } \tau_2 \times \tau_1}\,[\text{Comp}]}{x{:}1\texttt{->}\tau_1, y{:}1\texttt{->}\tau_2 \vdash \mathit{raise}(y) \circ x \;:\; 1 \texttt{ -> } \tau_2 \times \tau_1}\,[\text{Exch}]}{x{:}1\texttt{->}\tau_1 \vdash \kappa y.\mathit{raise}(y) \circ x \;:\; \tau_2 \times 1 \texttt{ -> } \tau_2 \times \tau_1}\,[\text{Abs}]}{x{:}1\texttt{->}\tau_1 \vdash \kappa y.\mathit{raise}(y) \circ x \;:\; \tau_2 \texttt{ -> } \tau_2 \times \tau_1}\,[\text{Cheat}]}{\vdash \kappa x.\kappa y.\mathit{raise}(y) \circ x \;:\; \tau_1 \times \tau_2\texttt{->}\tau_2 \times \tau_1}\,[\text{Abs}]$$

Penrose Diagrams



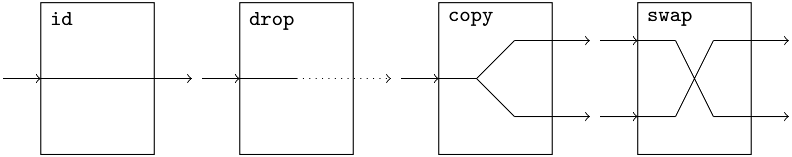




Penrose Diagrams

$$\texttt{id} = \kappa x.x \qquad : \alpha\texttt{->}\alpha$$

$$\texttt{drop} = \kappa x.id_1 \qquad : \alpha\texttt{->}1$$
$$\texttt{copy} = \kappa x.raise(x) \circ x \qquad : \alpha\texttt{->}(\alpha \times \alpha)$$
$$\texttt{swap} = \kappa x.\kappa y.raise(y) \circ x \qquad : (\alpha \times \beta)\texttt{->}(\beta \times \alpha)$$



The Combinator Basis

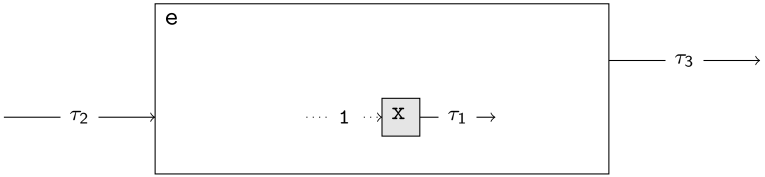
$$e ::= x \mid e \circ e \mid \kappa x.e \mid \mathtt{raise}_\tau(e) \qquad \underline{\text{(kappa calculus exps)}}$$

$$c ::= id_\tau \mid c \circ c \mid \mathtt{raise}_\tau(c) \mid \mathtt{lower}_\tau(c) \qquad \underline{\text{(combinator exps)}}$$
$$\mid \mathtt{drop}_\tau \mid \mathtt{copy}_\tau \mid \mathtt{swap}_{\tau_1,\tau_2}$$

- Abstraction elimination is the translation $\mathcal{E} : e \to c$; this eliminates all binding forms and free variables.
- On previous slides we saw how to define each of these combinators using κ-abstraction and free variables; substituting these provides the reverse translation $\mathcal{S} : c \to e$.

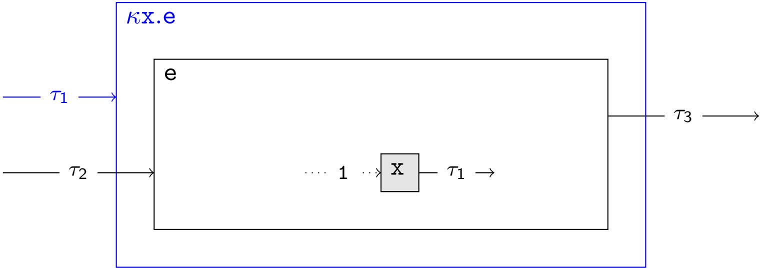
Visualizing Kappa Abstraction as a Penrose Diagram

$$[\mathrm{Abs}]\ \frac{\texttt{x:1->}\tau_1 \vdash \texttt{e:}\tau_2\texttt{->}\tau_3}{}$$



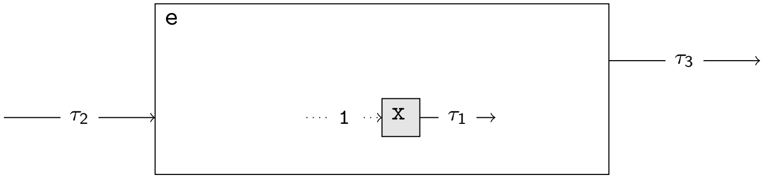
Visualizing Kappa Abstraction as a Penrose Diagram

$$[\text{Abs}]\ \frac{\texttt{x:1->}\tau_1 \vdash \texttt{e:}\tau_2\texttt{->}\tau_3}{\vdash (\kappa\texttt{x.e}) \quad \texttt{:}(\tau_1 \times \tau_2)\texttt{->}\tau_3}$$



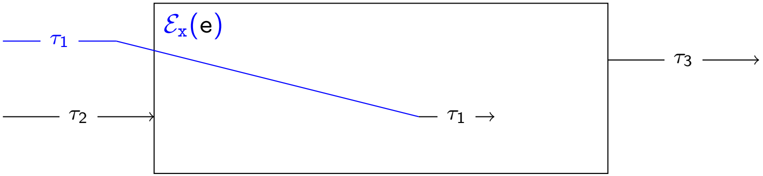
Visualizing Kappa Abstraction *Elimination*

$$\frac{\texttt{x}:\texttt{1->}\tau_1 \vdash \texttt{e}:\tau_2\texttt{->}\tau_3}{\vdash (\kappa\texttt{x.e}) \quad :(\tau_1 \times \tau_2)\texttt{->}\tau_3}$$



Visualizing Kappa Abstraction *Elimination*

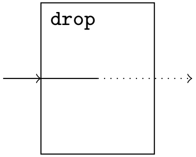
$$\frac{\vdash}{\vdash \mathcal{S}(\mathcal{E}_{\mathtt{x}}(\mathtt{e})) : (\tau_1 \times \tau_2)\texttt{->}\tau_3}$$

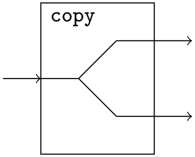


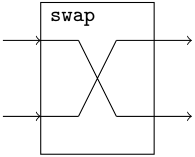
Abstraction Elimination Algorithm

- Most of the abstraction elimination process consists of re-arranging inputs and outputs.
- Note the similarity between structural rules for natural deduction and the `drop`/`copy`/`swap` combinators:

$$[\text{Weak}]\ \frac{\vdash e : \tau}{\Gamma \vdash e : \tau} \qquad [\text{Cont}]\ \frac{\Gamma, \Gamma \vdash e : \tau}{\Gamma \vdash e : \tau} \qquad [\text{Exch}]\ \frac{\Gamma_1, \Gamma_2 \vdash e : \tau}{\Gamma_2, \Gamma_1 \vdash e : \tau}$$



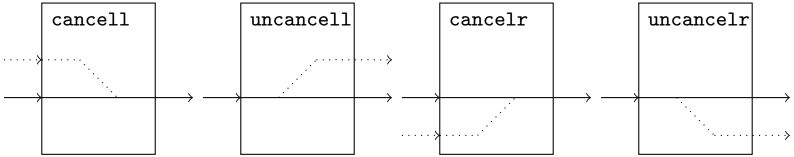




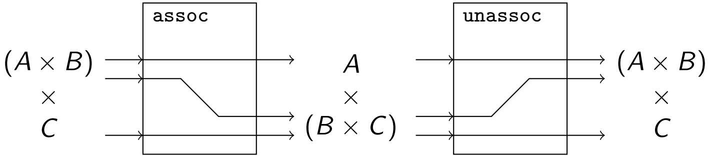
- This is how the abstraction elimination algorithm is defined: by walking the proof tree, replacing each inference rule with the corresponding combinator.

Loose Ends

To get rid of $1 \times A = A = A \times 1$ we need to introduce `cancell`, `cancelr`, `uncancell`, and `uncancelr`.



To get rid of $(A \times B) \times C = A \times (B \times C)$ we need to introduce `assoc` and `unassoc`.



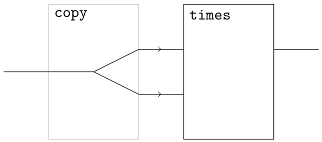
Implementation

- Modified version of GHC
- Expressions extended with code brackets `<[e]>` and escape `~~e`
- Types extended with code types `<[`τ`]>@`α
- *Flattening* pass:
  - Builds the typing derivation for expressions inside code brackets.
  - Walks the derivation, eliminating abstractions.
  - Passes the resulting combinator expression (ordinary Haskell) to the rest of the compiler.

Part III: Applications

Examples

```
square =
  \times -> <[ \y -> ~~times y y ]>
```

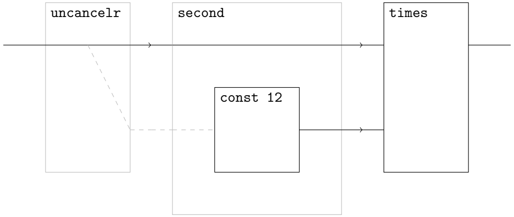


This diagram, and all the rest on future slides, were produced by running the flattener and instantiating the resulting term with `GArrowTikZ`

```
square :: <[ (a,a)~~> b ]> -> <[ a ~~> b ]>
```

Two-Level Syntax

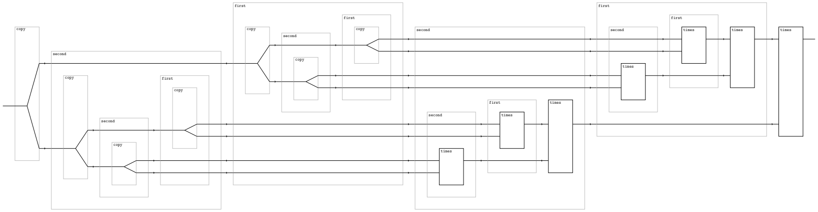
```
timesTwelve =
 \const ->
   \times ->
     <[ \y -> ~~times y ~~(const 12) ]>
```



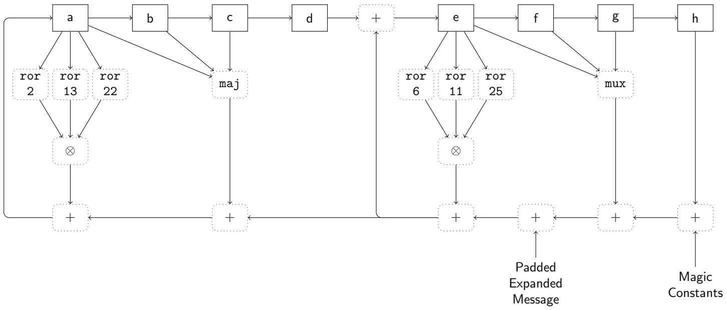
```
timesTwelve :: (Int -> <[ ()~~>a ]>) ->
               <[ (b,a)~~> c ]> ->
               <[    b ~~> c ]>
```

Unrolling

```
toEigthPower times = pow 8
 where
  pow 0 x = const 1
  pow 1 x = x
  pow n x = <[ ~~times
                   ~~(pow (n/2) x)
                   ~~(pow (if n ‘mod‘ 2 == 0
                           then n/2
                           else n/2+1) x) ]>
```



The SHA-256 Algorithm

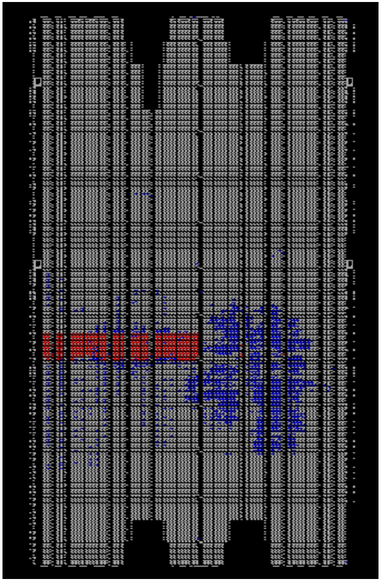


The SHA-256 Algorithm. Each solid rectangle is a 32-bit state variable; the path into each rectangle computes its value in the next round based on the values of the state variables in the previous round. The standard specifies initialization values for the state variables prior to the first message block.

One Round of SHA-256

```
sha256round =
  <[ \load input k_plus_w ->
     let a    = ~~(fifo 32) (mux2 load a_in input)
         b    = ~~(fifo 32) a
         c    = ~~(fifo 32) b
         d    = ~~(fifo 32) c
         e    = ~~(fifo 32) (mux2 load e_in d)
         f    = ~~(fifo 32) e
         g    = ~~(fifo 32) f
         h    = ~~(fifo 32) g
         s0   = xor3 (~~(rotRight  2) a_in)
                     (~~(rotRight 13) a_in)
                     (~~(rotRight 22) a_in)
         s1   = xor3 (~~(rotRight  6) e_in)
                     (~~(rotRight 11) e_in)
                     (~~(rotRight 25) e_in)
         a_in = adder t1 t2
         e_in = adder t1 d
         t1   = adder
                    (adder h s1)
                    (adder (mux2 e g f) k_plus_w)
         t2   = adder s0 (maj3 a b c)
     in h ]>
```

32 Copies of the Circuit



The region in red holds 32 copies of the SHA-256 circuit. Slices shown in blue are the "overhead" shared among all copies (address generators, etc).

Questions?

More information:

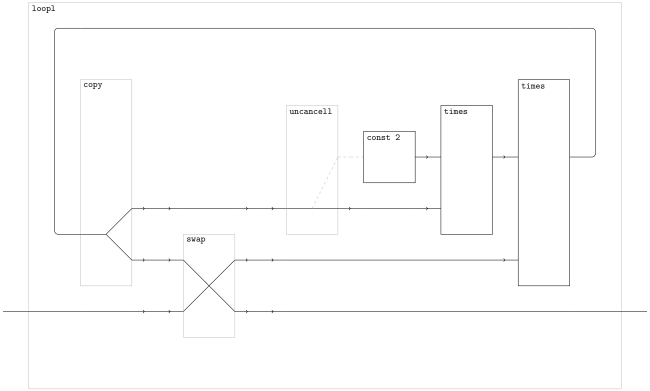
`http://www.cs.berkeley.edu/~megacz/garrows/`

Can't you Just Erase the Brackets?

- No; many reasons. Easy ones:
  - Expressions: brackets distinguish *repetitive structure* from feedback.
  - Types: a stream of pairs is not the same thing as a pair of streams.

Expression Brackets distinguish repetition from feedback

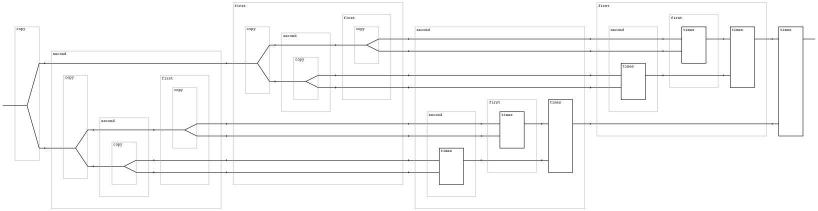
```
demo const times =
   <[ \x ->
      let out   = ~~times (~~times ~~(const 2) out) x
       in out
    ]>
```



Slogan: "Recursion (letrec) *inside* the brackets means feedback."

Expression Brackets distinguish repetition from feedback

```
toEigthPower times = pow 8
 where
  pow 0 x = const 1
  pow 1 x = x
  pow n x = <[ ~~times
                ~~(pow (n/2) x)
                ~~(pow (if n ‘mod‘ 2 == 0
                        then n/2
                        else n/2+1) x) ]>
```



Slogan: “Recursion *outside* the brackets means repetitive structure.”

Type Brackets matter too (1/2)

Phase distinction:

```
Int -> <[      ()   ~~> Int ]>
       <[ (Int,()) ~~> Int ]>
```

Type Brackets matter too (2/2)

Imagine realizing a combinator expression as some sort of *asynchronous* stream-processing network:

- The type `( <[Int]> , <[Int]> )` is for *pairs of streams* of `Ints`
- The type `<[ ( Int , Int ) ]>` is for *streams of pairs* of `Ints`

Type Brackets matter too (2/2)

Imagine realizing a combinator expression as some sort of *asynchronous* stream-processing network:

- The type `( <[Int]> , <[Int]> )` is for *pairs of streams* of `Ints`
- The type `<[ ( Int , Int ) ]>` is for *streams of pairs* of `Ints`
- These types are not equal, *nor are they even isomorphic!*

Type Brackets matter too (2/2)

Imagine realizing a combinator expression as some sort of *asynchronous* stream-processing network:

- The type `( <[Int]> , <[Int]> )` is for *pairs of streams* of `Int`s
  - Can read a single `Int` even if the other stream is empty.
- The type `<[ ( Int , Int ) ]>` is for *streams of pairs* of `Int`s
  - Will always get either a pair of `Int`s or none at all.
- These types are not equal, *nor are they even isomorphic!*

Type Brackets matter too (2/2)

Imagine realizing a combinator expression as some sort of *asynchronous* stream-processing network:

- The type `( <[Int]> , <[Int]> )` is for *pairs of streams* of `Int`s
  - Can read a single `Int` even if the other stream is empty.
- The type `<[ ( Int , Int ) ]>` is for *streams of pairs* of `Int`s
  - Will always get either a pair of `Int`s or none at all.
- These types are not equal, *nor are they even isomorphic!*
- Neither of the above is isomorphic to a stream of *co*products (hint: think about buffering and backpressure).

Generalized Arrows

```
class Category g => GArrow g (**) u where
--id             :: g x x
--(>>>)          :: g x y -> g y z -> g x z
  ga_first       :: g x y -> g (x ** z) (y ** z)
  ga_second      :: g x y -> g (z ** x) (z ** y)

  ga_cancell     :: g (u**x)          x
  ga_cancelr     :: g    (x**u)       x
  ga_uncancell   :: g     x        (u**x)
  ga_uncancelr   :: g     x           (x**u)

  ga_assoc       :: g ((x** y)**z ) ( x**(y **z))
  ga_unassoc     :: g ( x**(y **z)) ((x** y)**z )

  ga_copy        :: g x (x**x)
  ga_drop        :: g x u
  ga_swap        :: g (x**y) (y**x)

  ga_loop        :: g (x**z) (y**z) -> g x y
```

Every Arrow is a GArrow

```
instance Arrow a => GArrow a (,) () where
  ga_first      =  first
  ga_second     =  second
  ga_cancell    =  arr (\((),x) -> x)
  ga_cancelr    =  arr (\(x,()) -> x)
  ga_uncancell  =  arr (\x -> ((),x))
  ga_uncancelr  =  arr (\x -> (x,()))
  ga_assoc      =  arr (\((x,y),z) -> (x,(y,z)))
  ga_unassoc    =  arr (\(x,(y,z)) -> ((x,y),z))

instance Arrow a => GArrowDrop a (,) () where
  ga_drop       =  arr (\x -> ())

instance Arrow a => GArrowCopy a (,) () where
  ga_copy       =  arr (\x -> (x,x))

instance Arrow a => GArrowSwap a (,) () where
  ga_swap       =  arr (\(x,y) -> (y,x))

instance ArrowLoop a => GArrowLoop a (,) () where
  ga_loop       =  loop
```

… but `GArrow` does not let *arbitrary Haskell functions "leak" in* since there is no `arr`.