Perceus: garbage free reference counting with reuse

Kirsten Hagenaars & Astrid van der Jagt

In this presentation

Perceus: *garbage free* reference counting with *reuse*

- Reference counting vs Garbage collection
- Koka
- Perceus (optimizations of reference counting with Koka)
- FBIP
- Linear resource calculus λ^1
- Correctness of Perceus

Memory management

Reference counting

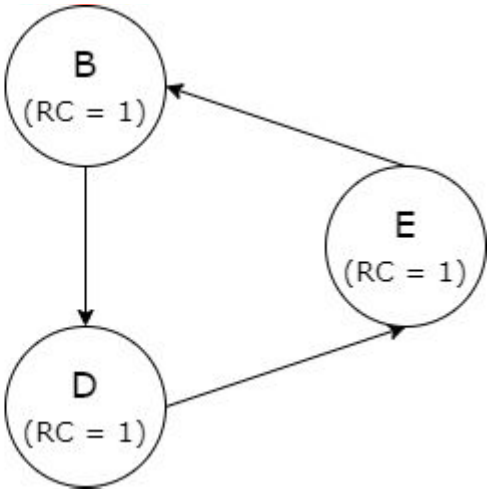
- Compilation
  Insert reference counting operations

Garbage collection

- Run-time
  Check whether objects are reachable from root nodes

Reference counting

Keep track of the number of pointers for each allocated object



Garbage collection

Check for each object whether it is reachable

Reference counting

- Low memory overhead
- Easy to implement

- Precision
- Concurrency
- Cycles

vs

Garbage collection

- Cycles are no problem
- Concurrency is no problem

- Precision
- Time
- Less manageable

Koka (1/2)

Strongly typed, functional language that tracks all (side) effects

```
fun square( x : int ) : total int { x * x }
```

effect type

```
fun println( s : string ) : console ()
```

```
fun divide( x : int, y : int ) : exn int
```

Koka (2/2)

Strongly typed, functional language that tracks all (side) effects

```
type list<a> {
    Cons( head : a, tail : list<a> )
    Nil
}

fun map( xs : list<a>, f : a -> e b ) : e list<b> {
    match(xs) {
        Cons(x,xx) -> Cons(f(x), map(xx, f))
        Nil        -> Nil
    }
}
```

Perceus

Optimizations of reference counting with Koka

- Code generated by the compiler has a grey background

- `dup` (duplication) operation increases the reference count
- `drop` operation decreases the reference count

Precise reference counting (1/2)

Free objects immediately

```
1           fun foo {
2               val xs = list(1, 1000000)  // create large list
3               val ys = map(xs, inc)      // increment elements
4               print(ys)
5           }   drop(xs)
6               drop(ys)
7           }
```

→ Perceus: the ownership of references is passed down into each function

Precise reference counting (2/2)

Free objects immediately

```
1           fun map( xs, f ) {
2               match(xs) {
3                   Cons(x,xx) {
4                       dup(x); dup(xx); drop(xs)
5                       Cons( dup(f)(x), map(xx, f) )
6                   }
7                   Nil { drop(xs); drop(f); Nil }
8               }
9           }
```

Drop specialization (1/3)

Inline the `drop` operation

```
fun drop( x ) {
    if (is-unique(x))
        then drop children of x; free(x)
        else decref(x)
}
```

Drop specialization (2/3)

Inline the `drop` operation

```
1           fun map( xs, f ) {
2               match(xs) {
3                   Cons(x,xx) {
4                       dup(x); dup(xx)
5                       if (is-unique(xs))
6                           then drop(x); drop(xx); free(xs)
7                           else decref(xs)
8                       Cons( dup(f)(x), map(xx, f) )
9                   }
10                  Nil { drop(xs); drop(f); Nil }
11              }
12          }
```

Drop specialization (3/3)

Inline the `drop` operation

```
1           fun map( xs, f ) {
2               match(xs) {
3                   Cons(x,xx) {
4                       if (is-unique(xs))
5                           then free(xs)
6                           else dup(x); dup(xx); decref(xs)
7                       Cons( dup(f)(x), map(xx, f) )
8                   }
9                   Nil { drop(xs); drop(f); Nil }
10              }
11          }
```

Reuse analysis (1/5)

Match pairs with the same size for reuse

```
1           fun map( xs, f ) {
2               match(xs) {
3                   Cons(x,xx) {
4                       dup(x); dup(xx); drop(xs)
5                       Cons( dup(f)(x), map(xx, f) )
6                   }
7                   Nil { drop(xs); drop(f); Nil }
8               }
9           }
```

Reuse analysis (2/5)

Match pairs with the same size for reuse

```
fun drop( x ) {
    if (is-unique(x))
      then drop children of x;
           free(x)
      else decref(x)
}
```

```
fun drop-reuse( x ) {
    if (is-unique(x))
      then drop children of x;
           &x
      else decref(x); NULL
}
```

Reuse analysis (3/5)

Match pairs with the same size for reuse

```
1          fun map( xs, f ) {
2              match(xs) {
3                  Cons(x,xx) {
4                      dup(x); dup(xx)
5                      val ru = drop-reuse(xs)
6                      Cons@ru( dup(f)(x), map(xx, f) )
7                  }
8                  Nil { drop(xs); drop(f); Nil }
9              }
10         }
```

Reuse analysis (4/5)

Match pairs with the same size for reuse

```
1           fun map( xs, f ) {
2               match(xs) {
3                   Cons(x,xx) {
4                       dup(x); dup(xx);
5                       val ru = if (is-unique(xs))
6                                  then   drop(x); drop(xx); &xs
7                                  else   decref(xs); NULL
8                       Cons@ru( dup(f)(x), map(xx, f) )
9                   }
10                  Nil { drop(xs); drop(f); Nil }
11              }
12          }
```

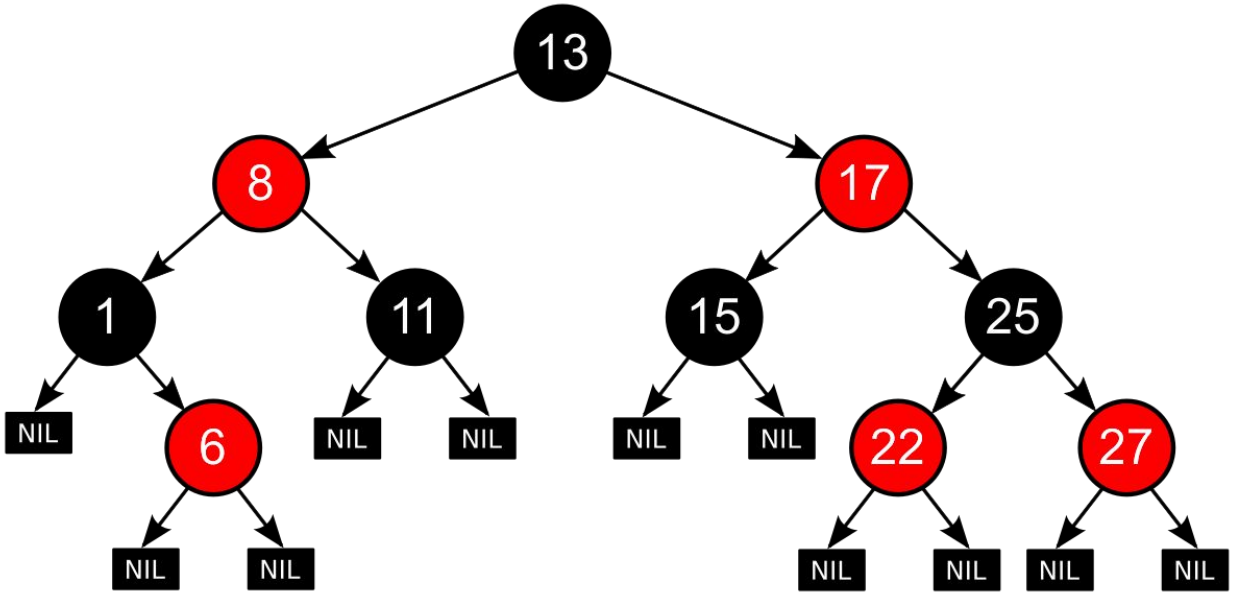
Reuse analysis (5/5)

Match pairs with the same size for reuse

```
1           fun map( xs, f ) {
2               match(xs) {
3                   Cons(x,xx) {
4                       val ru = if (is-unique(xs))
5                                   then    &xs
6                                   else    dup(x); dup(xx);
7                                           decref(xs); NULL
8                       Cons@ru( dup(f)(x), map(xx, f) )
9                   }
10                  Nil { drop(xs); drop(f); Nil }
11              }
12          }
```

Reuse specialization (1/3)

Further reuse the unchanged fields of a constructor



Reuse specialization (2/3)

```
fun ins( t : tree, k : int, v : bool ) : tree {
    match(t) {
        …
        Node(Red, l, kx, vx, r)
            -> if (k < kx) then
                Node(Red, ins(l, k, v), kx, vx, r)
        …
    }
}
```

Reuse specialization (3/3)

```
fun ins( t : tree, k : int, v : bool ) : tree {
    match(t) {
        ...
        Node(Red, l, kx, vx, r) {
            val ru = &t if t is unique

            if (dup(k)<dup(kx)) {
                if (ru!=NULL) then {ru->left := ins(l,k,v); ru}
                else Node(Red, ins(l,k,v), kx, vx, r)
        ...
    }
}
```

New paradigm: Functional But In-Place (FBIP) (1/3)

Describe in-place mutating algorithms in a functional way (and get persistence as well)

```
fun tmap( t : tree, f : int -> int ) : tree {
    match(t) {
        Node(l,x,r)    -> Node( tmap(l,f), f(x), tmap(r,f) )
        Leaf           -> Leaf
    }
}
```

New paradigm: Functional But In-Place (FBIP) (2/3)

New paradigm: Functional But In-Place (FBIP) (3/3)

```
fun tmap( f : int -> int, t : tree, visit : visitor, d : direction) :
tree {
    match(d) {
        Down -> match(t) {
            Node(l,x,r)    ->  tmap(f,l,NodeR(r,x,visit),Down)
            Leaf           ->  tmap(f,Leaf,visit,Up)
        }
        Up -> match(visit) {
            Done -> t
            NodeR(r,x,v)   ->  tmap(f,r,NodeL(t,f(x),v),Down)
            NodeL(l,x,v)   ->  tmap(f,Node(l,x,t),v,Up)
}}}
```

Static guarantees and language features

- Non-linear control flow
- Atomic reference counting for concurrent execution
- Mutable references and race conditions
- Cycles

Linear resource calculus λ^1 (1/4)

- Based on linear logic:
  Untyped lambda calculus extended with explicit binding and pattern matching

- Operational semantics in an explicit heap with reference counting

- Perceus is the syntax directed algorithm of λ^1

Linear resource calculus λ^1 (2/4)

Syntax

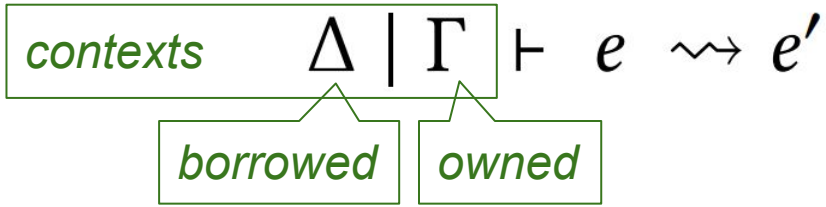
Expressions

e	::=	$v \mid e\ e$	(value, application)
	|	val $x = e; e$	(bind)
	|	match $x \{ \overline{p_i \rightarrow e_i} \}$	(match)
	|	dup $x; e$	(duplicate)
	|	drop $x; e$	(drop)
	|	match $e \{ \overline{p_i \rightarrow e_i} \}$	(match expr)

generated

Linear resource calculus λ^1 (3/4)

Judgement



Each owned variable is consumed exactly once

$$\frac{}{\Delta \mid x \vdash x \rightsquigarrow x}\ [\textsc{var}]$$

$$\frac{\varnothing \mid \Gamma, x \vdash e \rightsquigarrow e' \qquad \Gamma = \mathrm{fv}(\lambda x.\, e)}{\Delta \mid \Gamma \vdash \lambda x.\, e \rightsquigarrow \lambda^{\Gamma} x.\, e'}\ [\textsc{lam}]$$

Linear resource calculus λ^1 (4/4)

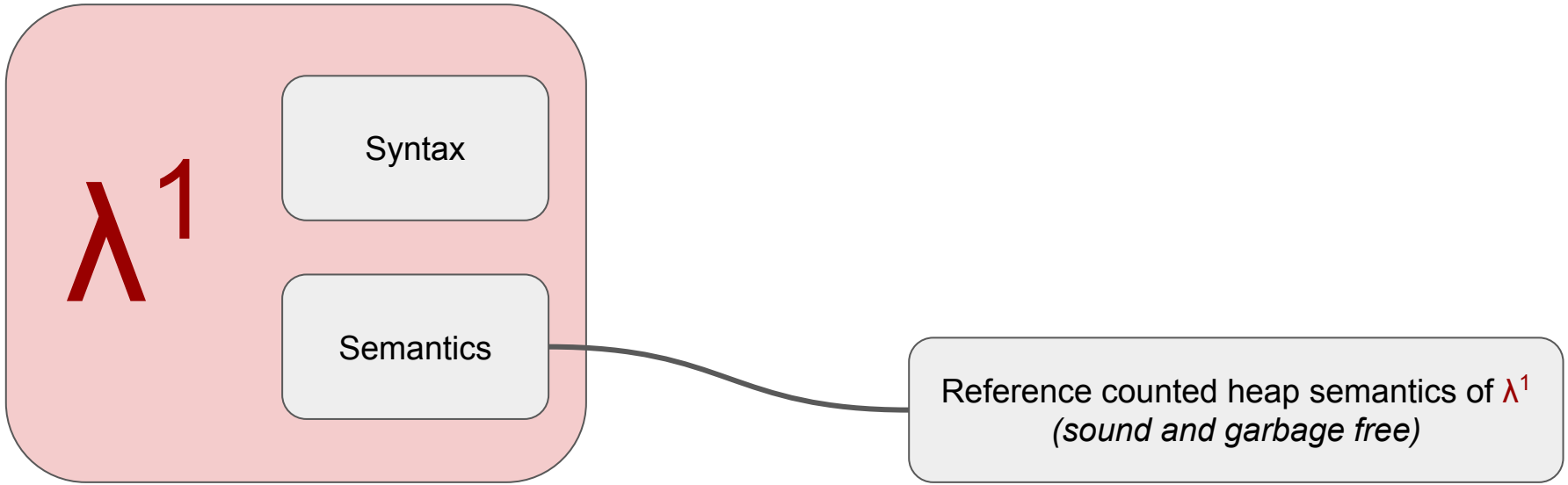
Declarative linear resources rules

$$\frac{\Delta \mid \boxed{\Gamma, x} \vdash e \rightsquigarrow e' \qquad \boxed{x \in \Delta, \Gamma}}{\Delta \mid \Gamma \vdash e \rightsquigarrow \text{dup } x;\ e'} \ [\textsc{dup}]$$

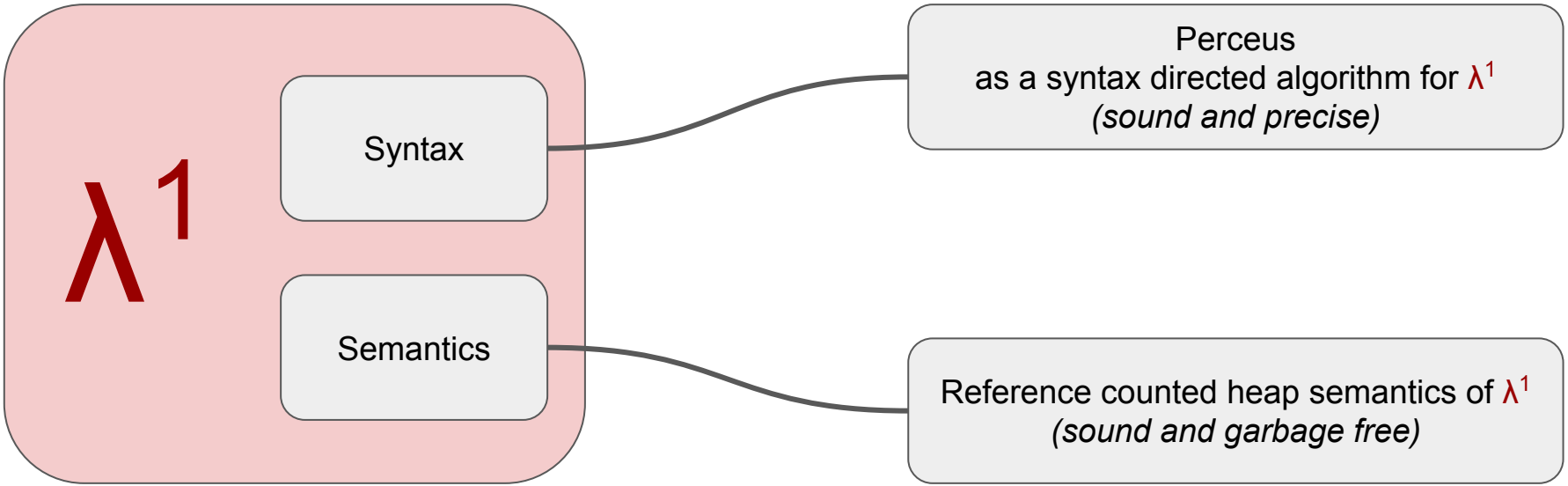
$$\frac{\Delta \mid \Gamma \vdash e \rightsquigarrow e'}{\Delta \mid \boxed{\Gamma, x} \vdash e \rightsquigarrow \text{drop } x;\ e'} \ [\textsc{drop}]$$

$$\frac{\boxed{\Delta, \Gamma_2} \mid \boxed{\Gamma_1} \vdash e_1 \rightsquigarrow e_1' \qquad \Delta \mid \boxed{\Gamma_2} \vdash e_2 \rightsquigarrow e_2'}{\Delta \mid \boxed{\Gamma_1}, \boxed{\Gamma_2} \vdash e_1\ e_2 \rightsquigarrow e_1'\ e_2'} \ [\textsc{app}]$$

Correctness of Perceus (1/3)



Correctness of Perceus (2/3)

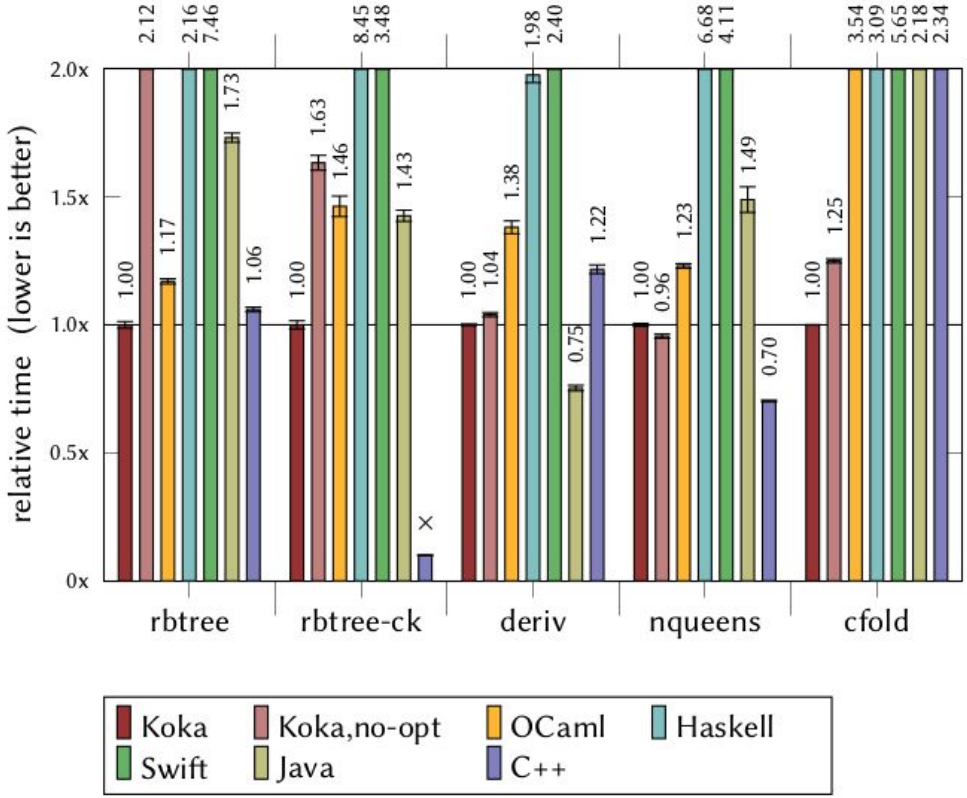


Correctness of Perceus (3/3)

Perceus
as a syntax directed algorithm for λ^1
(sound and precise)

Theorem: Perceus is *precise* and *garbage free*

Benchmarks


relative time (lower is better)
2.0x
1.5x
1.0x
0.5x
0x
rbtree
rbtree-ck
deriv
nqueens
cfold
1.00 2.12 1.17 2.16 7.46 1.73 1.06
1.00 1.63 1.46 8.45 3.48 1.43 ×
1.00 1.04 1.38 1.98 2.40 0.75 1.22
1.00 0.96 1.23 6.68 4.11 1.49 0.70
1.00 1.25 3.54 3.09 5.65 2.18 2.34
Koka
Koka,no-opt
OCaml
Haskell
Swift
Java
C++

Conclusion

Perceus: *garbage free* reference counting with *reuse*

Linear Resource Calculus λ^1

Syntax

Expressions

$$
\begin{array}{rcll}
e & ::= & v \mid e\ e & \text{(value, application)} \\
 & \mid & \text{val } x = e;\ e & \text{(bind)} \\
 & \mid & \text{match } x \{ \overline{p_i \to e_i} \} & \text{(match)} \\
 & \mid & \text{dup } x;\ e & \text{(duplicate)} \\
 & \mid & \text{drop } x;\ e & \text{(drop)} \\
 & \mid & \text{match } e \{ \overline{p_i \to e_i} \} & \text{(match expr)}
\end{array}
\qquad
\begin{array}{rcll}
v & ::= & x \mid \lambda x.\ e & \text{(variables, functions)} \\
 & \mid & C\ v_1 \ldots v_n & \text{(constructor of arity } n\text{)} \\
p & ::= & C\ b_1 \ldots b_n & \text{(pattern)} \\
b & ::= & x \mid \_ & \text{(binder or wildcard)}
\end{array}
$$

Contexts $\quad \Delta, \Gamma \ ::= \ \varnothing \mid \Delta \cup x$

Syntactic shorthands

$$
\begin{array}{lcll}
e_1;\ e_2 & \triangleq & \text{val } x = e_1;\ e_2 & \text{sequence, } x \notin \text{fv}(e_2) \\
\lambda\_.\ e & \triangleq & \lambda x.\ e & x \notin \text{fv}(e) \\
\lambda x.\ e & \triangleq & \lambda^{ys} x.\ e & ys = \text{fv}(e)
\end{array}
$$

Linear Resource Calculus λ^1

Declarative linear resources rules

$$x \notin \Delta, \Gamma_1, \Gamma_2 \quad \frac{\Delta, \Gamma_2 \mid \Gamma_1 \vdash e_1 \rightsquigarrow e_1' \qquad \Delta \mid \Gamma_2, x \vdash e_2 \rightsquigarrow e_2'}{\Delta \mid \Gamma_1, \Gamma_2 \vdash \mathsf{val}\ x = e_1;\ e_2 \rightsquigarrow \mathsf{val}\ x = e_1';\ e_2'} \ [\textsc{bind}]$$

$$\frac{[\textsc{match}] \qquad \Delta \mid \Gamma, \mathrm{bv}(p_i) \vdash e_i \rightsquigarrow e_i'}{\Delta \mid \Gamma, x \vdash \mathsf{match}\ x\ \{\ \overline{p_i \mapsto e_i}\ \} \rightsquigarrow \mathsf{match}\ x\ \{\ \overline{p_i \mapsto e_i'}\ \}}$$

$$\frac{\Delta, \Gamma_{i+1}, \ldots, \Gamma_n \mid \Gamma_i \vdash v_i \rightsquigarrow v_i' \qquad 1 \leqslant i \leqslant n}{\Delta \mid \Gamma_1, \ldots, \Gamma_n \vdash C\ v_1 \ldots v_n \rightsquigarrow C\ v_1' \ldots v_n'} \ [\textsc{con}]$$

Linear Resource Calculus λ^1

Reference-counted heap semantics

$$\begin{array}{lll} \mathsf{H} & : & x \to (\mathbb{N}^+, v) \\ \mathsf{E} & ::= & \Box \mid \mathsf{E}\ e \mid x\ \mathsf{E} \mid \mathsf{val}\ x\ =\ \mathsf{E};\ e \\ & \mid & C\ x_1 \ldots x_i\ \mathsf{E}\ v_j \ldots v_n \end{array}$$

$$\frac{\mathsf{H} \mid e \longrightarrow_r \mathsf{H}' \mid e'}{\mathsf{H} \mid \mathsf{E}[e] \longmapsto_r \mathsf{H}' \mid \mathsf{E}[e']}\ [\textsc{eval}]$$

$$\begin{array}{llll} (lam_r) & \mathsf{H} \mid (\lambda^{ys}\ x.\ e) & \longrightarrow_r\ \mathsf{H}, f \mapsto^1 \lambda^{ys} x.\ e & \mid f \qquad \text{fresh } f \\ (con_r) & \mathsf{H} \mid C\ x_1 \ldots x_n & \longrightarrow_r\ \mathsf{H}, z \mapsto^1 C\ x_1 \ldots x_n & \mid z \qquad \text{fresh } z \end{array}$$

$$\begin{array}{llll} (app_r) & \mathsf{H} \mid f\ z & \longrightarrow_r\ \mathsf{H} \mid \mathsf{dup}\ ys;\ \mathsf{drop}\ f;\ e[x{:=}z] & (f \mapsto^n \lambda^{ys} x.\ e) \in \mathsf{H} \\ (match_r) & \mathsf{H} \mid \mathsf{match}\ x\ \{\overline{p_i \to e_i}\} & \longrightarrow_r\ \mathsf{H} \mid \mathsf{dup}\ ys;\ \mathsf{drop}\ x;\ e_i[xs{:=}ys] & \text{with } p_i = C\ xs \text{ and } (x \mapsto^n C\ ys) \in \mathsf{H} \\ (bind_r) & \mathsf{H} \mid \mathsf{val}\ x\ =\ y\ ;\ e & \longrightarrow_r\ \mathsf{H} \mid e[x{:=}y] & \end{array}$$

$$\begin{array}{llll} (dup_r) & \mathsf{H}, x \mapsto^n v \mid \mathsf{dup}\ x;\ e & \longrightarrow_r\ \mathsf{H}, x \mapsto^{n+1} v \mid e & \\ (drop_r) & \mathsf{H}, x \mapsto^{n+1} v \mid \mathsf{drop}\ x;\ e & \longrightarrow_r\ \mathsf{H}, x \mapsto^n v \mid e & \text{if } n \geqslant 1 \\ (dlam_r) & \mathsf{H}, x \mapsto^1 \lambda^{ys} z.e \mid \mathsf{drop}\ x;\ e & \longrightarrow_r\ \mathsf{H} \mid \mathsf{drop}\ ys;\ e & \\ (dcon_r) & \mathsf{H}, x \mapsto^1 C\ ys \mid \mathsf{drop}\ x;\ e & \longrightarrow_r\ \mathsf{H} \mid \mathsf{drop}\ ys;\ e & \end{array}$$