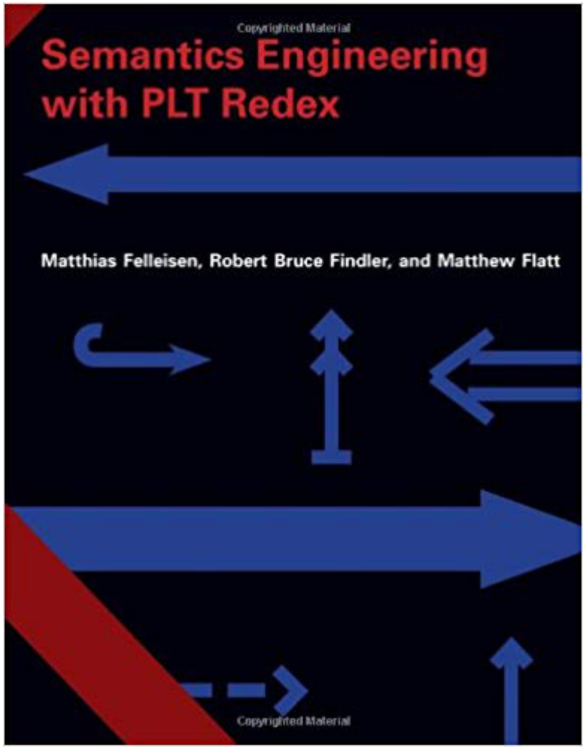
# A Wip, Executable Semantic Model for Linear AARA

# A Wip, Executable Semantic Model for Linear AARA

And a Showcase of Redex

# A Wip, Executable Semantic Model for Linear AARA

And a Showcase of Redex



# A Wip, Executable Semantic Model for Linear AARA

And a Showcase of Redex

## Run Your Research

On the Effectiveness of Lightweight Mechanization

Casey Klein[1] John Clements[2] Christos Dimoulas[3] Carl Eastlund[3] Matthias Felleisen[3]
Matthew Flatt[4] Jay A. McCarthy[5] Jon Rafkind[4] Sam Tobin-Hochstadt[3] Robert Bruce Findler[1]

PLT

[1]Northwestern University, Evanston, IL [2]California Polytechnic State University, San Luis Obispo, CA
[3]Northeastern University, Boston, MA [4]University of Utah, Salt Lake City, UT [5]Brigham Young University, Provo, UT

### Abstract

Formal models serve in many roles in the programming language community. In its primary role, a model communicates the idea of a language design; the architecture of a language tool; or the essence of a program analysis. No matter which role it plays, however, a faulty model doesn't serve its purpose.

One way to eliminate flaws from a model is to write it down in a mechanized formal language. It is then possible to state theorems about the model, to prove them, and to check the proofs. Over the past nine years, PLT has developed and explored a lightweight version of this approach, dubbed Redex. In a nutshell, Redex is a domain-specific language for semantic models that is embedded in the Racket programming language. The effort of creating a model in Redex is often no more burdensome than typesetting it used paper and pencil to develop these models. Paper-and-pencil models come with flaws, however. Since flawed models can lead to miscommunications, researchers state and prove theorems about models, which forces them to "debug" the model.

Some flaws nevertheless survive this paper-only validation step, and others are introduced during typesetting. These mistakes become obstacles to communication. For example, Martin Henz from National University of Singapore recently shared with one of this paper's authors his frustration with a historic paper (Plotkin 1975):

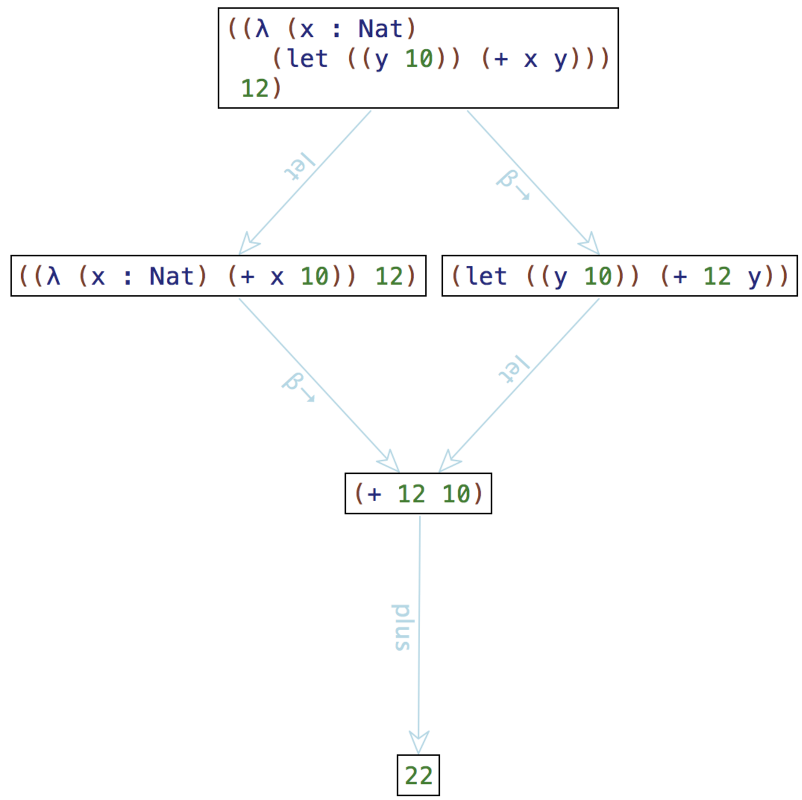
> *The readability is not helped by the fact that there are lots of typos, e.g. page 134, Rule II 1: M = N should be M = M. The rule II 3 on page 136 is missing the subscript 1 above the bar. [personal communication, 6/4/2011]*

# Redex: Design Programming Languages as Executable Semantics

Our language expressions, operations, type envs, and more

$$
\begin{array}{rl}
e ::= & x \mid \mathsf{c} \mid n \mid l \mid one \mid nat \\
& \mid (+\ e\ e) \\
& \mid (\mathsf{cons}\ e\ e) \\
& \mid () \\
& \mid (e\ e) \\
& \mid (\mathsf{tick}\ t) \\
& \mid (\mathsf{tick}\ t\ \mathsf{in}\ e) \\
& \mid (\mathsf{let}\ ([y\ e])\ e) \\
& \mid (\mathsf{case}\ e\ [\mathsf{nil} = e]\ [(\mathsf{cons}\ x\ x) = e]) \\
& \mid (o\ e) \\
& \mid (\mathsf{nil}\ \tau) \\
o ::= & \mathsf{hd} \mid \mathsf{tl} \\
pot ::= & (p \rightarrow q) \\
one ::= & \mathsf{triv} \mid \langle\rangle \\
t ::= & (\mathsf{side\text{-}condition}\ n_1\ (\mathsf{rational?}\ (\mathsf{term}\ n_1))) \\
p, q, r ::= & (\mathsf{side\text{-}condition} \\
& \quad n_1\ (\mathsf{and}\ (\mathsf{rational?}\ (\mathsf{term}\ n_1))\ ({>=}\ (\mathsf{term}\ n_1)\ 0))) \\
n ::= & number \\
nat ::= & natural \\
l ::= & (\lambda\ (x : \tau)\ e) \\
A, B ::= & [\tau\ p] \\
T, \tau ::= & \mathsf{Unit} \mid \mathsf{Nat} \mid (A \rightarrow A) \mid (\tau * \tau) \mid (\mathsf{List}\ A) \mid A \\
x, y, z ::= & variable\text{-}not\text{-}otherwise\text{-}mentioned \\
\Gamma ::= & \bullet \mid (\Gamma\ (x : \tau)\ \ldots\ (e : \tau)\ \ldots) \\
Venv ::= & \mathsf{V} \mid (\mathsf{V}\ x) \\
v ::= & n \mid l \mid (\mathsf{cons}\ v\ v) \mid \langle\rangle \mid [] \\
E ::= & [] \\
& \mid (E\ e\ \ldots) \mid (v\ \ldots\ E\ e\ \ldots) \mid (E\ e) \mid (v\ E) \\
& \mid (\lambda\ (x : \tau)\ E) \\
& \mid (\mathsf{cons}\ v\ \ldots\ E\ e\ \ldots) \\
& \mid (\mathsf{hd}\ E) \\
& \mid (\mathsf{tl}\ E) \\
& \mid (\mathsf{let}\ ([x\ E])\ e) \mid (\mathsf{let}\ ([x\ v])\ E) \\
& \mid (\mathsf{case}\ E\ [\mathsf{nil} = e]\ [(\mathsf{cons}\ x\ x) = e])
\end{array}
$$

# Redex: Design Programming Languages as Executable Semantics



Let’s build a derivation tree for type-checking our `id` fn

```
(define-term dumb-id
  (λ (l : [(List [Nat 2]) 0])
    (case l
      [nil = (nil [(List [Nat 2]) 0])]
      [(cons x xs) = (tick 2 in (let ([ys (id xs)])
                                  (cons x ys)))])))
```

# Let’s build a derivation tree for type-checking our `id` fn

```
((list
  (derivation
   '(type-infer
     •
     (0 → 0)
     ⊢
     (λ (l : ((List (Nat 2)) 0))
       (case l (nil = (nil ((List (Nat 2)) 0))) ((cons x xs) = xs)))
     :
     ((List (Nat 2)) 0))
   "L:Fun"
   (list
    (derivation
     '(type-infer
       (• (l«77» : ((List (Nat 2)) 0)))
       (0 → 0)
       ⊢
       (case l«77» (nil = (nil ((List (Nat 2)) 0))) ((cons x xs) = xs))
       :
       ((List (Nat 2)) 0))
     "L:MatL"
     (list
      (derivation
       '(type-infer • (0 → 0) ⊢ (nil ((List (Nat 2)) 0)) : ((List (Nat 2)) 0))
       "L:Nil"
       '())
      (derivation
       '(type-infer
         (• (xs : ((List (Nat 2)) 0)))
         (2 → 0)
         ⊢
         xs
         :
         ((List (Nat 2)) 0))
       "L:Var"
       '())))))))
```

# Let's build a derivation tree for type-checking our `id` fn

```
——————————————————————————— [L:Nil]    ———————————————————————————— [L:Var]
(type-infer                            (type-infer
 •                                      (• (xs : ((List (Nat 2)) 0)))
 (0 → 0)                                (2 → 0)
 ⊢                                      ⊢
 (nil ((List (Nat 2)) 0))               xs
 :                                      :
 ((List (Nat 2)) 0))                    ((List (Nat 2)) 0))
———————————————————————————————————————————————————————————————————— [L:MatL]
                    (type-infer
                     (•
                      (l«77»
                       :
                       ((List (Nat 2)) 0)))
                     (0 → 0)
                     ⊢
                     (case l«77»
                       (nil
                        =
                        (nil ((List (Nat 2)) 0)))
                       ((cons x xs) = xs))
                     :
                     ((List (Nat 2)) 0))
——————————————————————————————————————————————————————————————————————— [L:Fun]
                        (type-infer
                         •
                         (0 → 0)
                         ⊢
                         (λ (l : ((List (Nat 2)) 0))
                           (case l
                             (nil
                              =
                              (nil
                               ((List (Nat 2)) 0)))
                             ((cons x xs) = xs)))
                         :
                         ((List (Nat 2)) 0))
```

# Our Static Semantics for Linear AARA (Lists)

$$\frac{[\tau\ q] = A}{\begin{array}{l}\text{type-infer}\llbracket(\Gamma\ (e_1 : (A \to B))\ (e_2 : \tau)),\\ (q \to q_1),\\ \vdash, (e_1\ e_2), :, B\rrbracket\end{array}}\ [\text{L:App}]$$

$$\frac{\begin{array}{c}[\tau\ p] = A\\ \text{type-infer}\llbracket(\Gamma\ (x : \text{abs-env}\llbracket A\rrbracket)), (p \to q), \vdash, e, :, [\tau\ q]\rrbracket\end{array}}{\text{type-infer}\llbracket\Gamma, (r \to q), \vdash, (\lambda\ (x : A)\ e), :, [\tau\ q]\rrbracket}\ [\text{L:Fun}]$$

$$\frac{}{\text{type-infer}\llbracket\bullet, (p \to q), \vdash, (\text{nil}\ \tau), :, \tau\rrbracket}\ [\text{L:Nil}]$$

$$\frac{\begin{array}{c}(=\ q\ \ (+\ q_1\ \ t))\\ (=\ (-\ q\ \ t)\ \ q_1)\\ \text{type-infer}\llbracket(\bullet\ (e : \tau)), (q \to q_1), \vdash, e, :, \tau\rrbracket\end{array}}{\text{type-infer}\llbracket(\bullet\ (e : \tau)), (q \to q_1), \vdash, (\text{tick}\ t\ \text{in}\ e), :, \tau\rrbracket}\ [\text{L:Tick}]$$

$$\frac{\begin{array}{c}(>=\ q\ \ 0)\\ (=\ t\ \ q)\end{array}}{\text{type-infer}\llbracket\bullet, (q \to 0), \vdash, (\text{tick}\ t), :, \text{Unit}\rrbracket}\ [\text{L:Tick}{\geq}0]$$

$$\frac{\begin{array}{c}(<\ t\ \ 0)\\ (=\ (\text{abs}\ t)\ \ q)\end{array}}{\text{type-infer}\llbracket\bullet, (0 \to q), \vdash, (\text{tick}\ t), :, \text{Unit}\rrbracket}\ [\text{L:Tick}{<}0]$$

$$\frac{\begin{array}{c}[\tau\ p] = \text{lists-fn}\llbracket T\rrbracket\\ \text{type-infer}\llbracket\Gamma, (q \to p), \vdash, e_1, :, \tau\rrbracket\\ \text{type-infer}\llbracket(\Gamma\ (e_2 : T)\ (x : \tau)),\\ (p \to q_1), \vdash, e_2, :, T\rrbracket\end{array}}{\begin{array}{l}\text{type-infer}\llbracket(\Gamma\ (e_2 : T)), (q \to q_1),\\ \vdash, (\text{let}\ ([x\ e_1])\ e_2), :, T\rrbracket\end{array}}\ [\text{L:Let}]$$

$$\frac{\begin{array}{c}[\tau\ p] = \text{lists-fn}\llbracket T\rrbracket\\ r = (-\ q\ \ p)\\ \text{type-infer}\llbracket(\Gamma\ (e_1 : \tau)), (q \to r), \vdash, e_1, :, \tau\rrbracket\\ \text{type-infer}\llbracket(\Gamma\ (e_2 : T)), (r \to q_1),\\ \vdash, e_2, :, T\rrbracket\end{array}}{\begin{array}{l}\text{type-infer}\llbracket(\Gamma\ (e_2 : T)), (q \to q_1),\\ \vdash, (\text{cons}\ e_1\ e_2), :, T\rrbracket\end{array}}\ [\text{L:Cons}]$$

$$\frac{\begin{array}{c}[\tau\ p] = \text{lists-fn}\llbracket T\rrbracket\\ r = (+\ q\ \ p)\\ \text{type-infer}\llbracket\Gamma, (q \to q_1), \vdash, e_0, :, \tau_1\rrbracket\\ \text{type-infer}\llbracket\text{unique-env}\llbracket\text{env-set}\llbracket(\Gamma\ (x_1 : \tau)\ (x_2 : T))\rrbracket\rrbracket,\\ (r \to q_1), \vdash, e_1, :, \tau_1\rrbracket\end{array}}{\ }\ [\text{L:MatL}]$$

# Our Static Semantics for Linear AARA (Lists)

```
[(where [τ p] (lists-fn T))
 (where r ,(- (term q) (term p)))
 (type-infer (Γ (e₁ : τ)) (q → r) ⊢ e₁ : τ)
 (type-infer (Γ (e₂ : T)) (r → q₁)
             ⊢ e₂ : T)
 ------------------------------------------ "L:Cons"
 (type-infer (Γ (e₂ : T)) (q → q₁)
             ⊢ (cons e₁ e₂) : T)]
```

# Do (Typing) Judgments Hold?

```
(printf "tick -> ")
(judgment-holds (type-infer (• (x : Nat)) (8 → 4) ⊢ (tick 4 in x) : τ) τ)
(printf "tick -> ")
(judgment-holds (type-infer • (4 → 0) ⊢ (tick 4) : τ) τ)
(printf "tick -> ")
(judgment-holds (type-infer • (0 → 4) ⊢ (tick -4) : τ) τ)
(printf "unit -> ")
(judgment-holds (type-infer • (0 → 0) ⊢ triv : τ) τ)
(printf "cons -> ")
(judgment-holds (type-infer (• (y : (List [Nat 4]))) (8 → 0) ⊢ (cons x y) : τ) τ)
(printf "fun -> ")
(judgment-holds (type-infer • (0 → 0) ⊢ (λ (x : [(List [Nat 4]) 5]) x) : τ) τ)
(printf "let -> ")
(judgment-holds (type-infer (• (e : Unit)) (0 → 0) ⊢ (let ([y triv]) e) : τ) τ)
(printf "case -> ")
(judgment-holds (type-infer (• (x : (List [Nat 4]))) (0 → 0)
                            ⊢ (case x
                                [nil = (nil (List [Nat 4]))]
                                [(cons x₁ x₂) = x₂]) : τ) τ)
```

# Do (Typing) Judgments Hold?

```
(printf "tick -> ")
(judgment-holds (type-infer (• (x : Nat)) (8 → 4) ⊢ (tick 4 in x) : τ) '(Nat))
(printf "tick -> ")
(judgment-holds (type-infer • (4 → 0) ⊢ (tick 4) : τ) '(Unit))
(printf "tick -> ")
(judgment-holds (type-infer • (0 → 4) ⊢ (tick -4) : τ) '(Unit))
(printf "unit -> ")
(judgment-holds (type-infer • (0 → 0) ⊢ triv : τ) '(Unit))
(printf "cons -> ")
(judgment-holds (type-infer (• (y : (List [Nat 4]))) (8 → 0) ⊢ (cons x y) : τ) '((List (Nat 4))))
(printf "fun -> ")
(judgment-holds (type-infer • (0 → 0) ⊢ (λ (x : [(List [Nat 4]) 5]) x) : τ) '(((List (Nat 4)) 0)))
(printf "let -> ")
(judgment-holds (type-infer (• (e : Unit)) (0 → 0) ⊢ (let ([y triv]) e) : τ) '(Unit))
(printf "case -> ")
(judgment-holds (type-infer (• (x : (List [Nat 4]))) (0 → 0)
                            ⊢ (case x
                                [nil = (nil (List [Nat 4]))]
                                [(cons x₁ x₂) = x₂]) : τ) '((List (Nat 4))))
```

# Redex Has So Much More

- Testing is Crucial

# Redex Has So Much More

- Testing is Crucial

```
(redex-check
 LinearAARAL
 v
 (redex-match? LinearAARAL e (term v))
 #:attempts 1000)
```

```
(test-equal (redex-match? LinearAARAL e (term dumb-id)) #t)
```

# Redex Has So Much More

- Testing is Crucial

```
(redex-check
 LinearAARAL
 v
 (redex-match? LinearAARAL e (term v))
 #:attempts 1000)
```

```
(test-equal (redex-match? LinearAARAL e (term dumb-id)) #t)
```

- Can easily model natural, small-step semantics, and programming paradigms like call-by-push-value
- Next up for our model: Share!, Sum, Products, and Recursive Types