

# Efficient ML Type Inference Using Ranked Type Variables

George Kuan, David MacQueen

University of Chicago

ML Workshop, October 5, 2007

# Introduction

- Testing whether unification type variables (**univariables**) can be generalized by searching entire environment is expensive. . . Use ranked type variables for efficient test
- SML/NJ, OCaml, and others have employed ranking for a long time, but most ranking techniques were ad-hoc tricks. How and why does ranking work?

## Main Ideas

- Ranking schemes are based on two variants:
  - Damas:  $\lambda$  nesting depth (e.g., SML/NJ)
  - Rémy: let nesting depth (e.g., OCaml)
- **Abstract machines** (CEK style) formalism for type inference (in the spirit of Reynolds 1972 and Kuan, MacQueen, Findler ESOP '07)
- Complications introduced by value restriction

# Classic $\mathcal{W}$ Abstract Machine

$(c, \Gamma, k)$

“Evaluating” expressions to types

$$c ::= e \mid \tau \qquad \Gamma ::= \emptyset \mid \Gamma[x : \tau] \mid \Gamma[x : \sigma]$$

$$\tau ::= \mathbf{b} \mid \tau \rightarrow \tau \mid \xi \qquad \sigma ::= \forall \bar{\alpha}. \tau$$

$$k ::= \bullet \mid f :: k$$

$$f ::= \lambda \mid (\square e) \mid (\tau \square)$$

let $x$ in $e$	(in definien)
let	(in body)

# Classic $\mathcal{W}$ Abstract Machine

$(c, \Gamma, k)$

“Evaluating” expressions to types

$$\begin{array}{ll}
 c ::= e \mid \tau & \Gamma ::= \emptyset \mid \Gamma[x : \tau] \mid \Gamma[x : \sigma] \\
 \tau ::= \mathbf{b} \mid \tau \rightarrow \tau \mid \xi & \sigma ::= \forall \bar{\alpha}. \tau
 \end{array}$$

$$\begin{array}{ll}
 k ::= \bullet \mid f :: k & \\
 f ::= \lambda \mid (\square e) \mid (\tau \square) & \\
 \quad \mid \text{let } x \text{ in } e & \text{(in definien)} \\
 \quad \mid \text{let} & \text{(in body)}
 \end{array}$$

# Classic $\mathcal{W}$ Abstract Machine

$$(c, \Gamma, k)$$

“Evaluating” expressions to types

$$c ::= e \mid \tau \qquad \Gamma ::= \emptyset \mid \Gamma[x : \tau] \mid \Gamma[x : \sigma]$$

$$\tau ::= \mathbf{b} \mid \tau \rightarrow \tau \mid \xi \qquad \sigma ::= \forall \bar{\alpha}. \tau$$

$$k ::= \bullet \mid f :: k$$

$$f ::= \lambda \mid (\square e) \mid (\tau \square)$$

let $x$ in $e$	(in definien)
let	(in body)

# $\mathcal{W}$ Abstract Machine

$$(c, \Gamma, k) \mapsto (c', \Gamma', k')$$

$$(\lambda x.e, \Gamma, k) \mapsto (e, \Gamma[x : \xi], \lambda :: k) \quad [\lambda\text{-in}]$$

( $\xi$  fresh)

$$(\tau, \Gamma[x : \tau'], \lambda :: k) \mapsto (\tau' \rightarrow \tau, \Gamma, k) \quad [\lambda\text{-out}]$$

$$(e_1 e_2, \Gamma, k) \mapsto (e_1, \Gamma, (\square e_2) :: k) \quad [\text{app-l}]$$

$$(\tau, \Gamma, (\tau' \square) :: k) \mapsto (\theta\xi, \theta\Gamma, \theta k) \quad [\text{app-out}]$$

$\theta = \mathcal{U}(\tau', \tau \rightarrow \xi)$  ( $\xi$  fresh)

# $\mathcal{W}$ Abstract Machine

$$(c, \Gamma, k) \mapsto (c', \Gamma', k')$$

$$(\lambda x.e, \Gamma, k) \mapsto (e, \Gamma[x : \xi], \lambda :: k) \quad [\lambda\text{-in}]$$

( $\xi$  fresh)

$$(\tau, \Gamma[x : \tau'], \lambda :: k) \mapsto (\tau' \rightarrow \tau, \Gamma, k) \quad [\lambda\text{-out}]$$

$$(e_1 e_2, \Gamma, k) \mapsto (e_1, \Gamma, (\square e_2) :: k) \quad [\text{app-l}]$$

$$(\tau, \Gamma, (\tau' \square) :: k) \mapsto (\theta\xi, \theta\Gamma, \theta k) \quad [\text{app-out}]$$

$\theta = \mathcal{U}(\tau', \tau \rightarrow \xi)$  ( $\xi$  fresh)

$\mathcal{W}$  Abstract Machine

$$(c, \Gamma, k) \mapsto (c', \Gamma', k')$$

$$(\lambda x.e, \Gamma, k) \mapsto (e, \Gamma[x : \xi], \lambda :: k) \quad [\lambda\text{-in}]$$

( $\xi$  fresh)

$$(\tau, \Gamma[x : \tau'], \lambda :: k) \mapsto (\tau' \rightarrow \tau, \Gamma, k) \quad [\lambda\text{-out}]$$

$$(e_1 e_2, \Gamma, k) \mapsto (e_1, \Gamma, (\square e_2) :: k) \quad [\text{app-l}]$$

$$(\tau, \Gamma, (\tau' \square) :: k) \mapsto (\theta\xi, \theta\Gamma, \theta k) \quad [\text{app-out}]$$

$\theta = \mathcal{U}(\tau', \tau \rightarrow \xi)$  ( $\xi$  fresh)

# $\mathcal{W}$ Abstract Machine

$$(c, \Gamma, k) \mapsto (c', \Gamma', k')$$

$$(\lambda x.e, \Gamma, k) \mapsto (e, \Gamma[x : \xi], \lambda :: k) \quad [\lambda\text{-in}]$$

( $\xi$  fresh)

$$(\tau, \Gamma[x : \tau'], \lambda :: k) \mapsto (\tau' \rightarrow \tau, \Gamma, k) \quad [\lambda\text{-out}]$$

$$(e_1 e_2, \Gamma, k) \mapsto (e_1, \Gamma, (\square e_2) :: k) \quad [\text{app-l}]$$

$$(\tau, \Gamma, (\tau' \square) :: k) \mapsto (\theta\xi, \theta\Gamma, \theta k) \quad [\text{app-out}]$$

$\theta = \mathcal{U}(\tau', \tau \rightarrow \xi)$  ( $\xi$  fresh)

# $\mathcal{W}$ Abstract Machine (Cont.)

$(\text{let } x = e_1 \text{ in } e_2, \Gamma, k) \mapsto (e_1, \Gamma, \text{let } x \text{ in } e_2 :: k)$	[let-def]
$(\tau, \Gamma, \text{let } x \text{ in } e_2 :: k) \mapsto (e_2, \Gamma[x : \sigma], \text{let} :: k)$ $\sigma = \forall \bar{\alpha}. \{\bar{\alpha} / \mathcal{G}_\Gamma(\tau)\} \tau$ $\mathcal{G}_\Gamma(\tau) = \text{ftv}(\tau) - \text{ftv}(\Gamma)$	[let-body] ( $\bar{\alpha}$ fresh)
$(\tau, \Gamma[x : \sigma], \text{let} :: k) \mapsto (\tau, \Gamma, k)$	[let-out]
$(x, \Gamma, k) \mapsto (\mathcal{I}(\Gamma(x)), \Gamma, k)$ $\mathcal{I}(\forall \bar{\alpha}. \tau) = \{\bar{\xi} / \bar{\alpha}\} \tau$ $\mathcal{I}(\tau) = \tau$	[var] ( $\bar{\xi}$ fresh)

# $\mathcal{W}$ Abstract Machine (Cont.)

$(\text{let } x = e_1 \text{ in } e_2, \Gamma, k) \mapsto (e_1, \Gamma, \text{let } x \text{ in } e_2 :: k)$	[let-def]
$(\tau, \Gamma, \text{let } x \text{ in } e_2 :: k) \mapsto (e_2, \Gamma[x : \sigma], \text{let} :: k)$ $\sigma = \forall \bar{\alpha}. \{\bar{\alpha} / \mathcal{G}_\Gamma(\tau)\} \tau$ $\mathcal{G}_\Gamma(\tau) = \text{ftv}(\tau) - \text{ftv}(\Gamma)$	[let-body] ( $\bar{\alpha}$ fresh)
$(\tau, \Gamma[x : \sigma], \text{let} :: k) \mapsto (\tau, \Gamma, k)$	[let-out]
$(x, \Gamma, k) \mapsto (\mathcal{I}(\Gamma(x)), \Gamma, k)$ $\mathcal{I}(\forall \bar{\alpha}. \tau) = \{\bar{\xi} / \bar{\alpha}\} \tau$ $\mathcal{I}(\tau) = \tau$	[var] ( $\bar{\xi}$ fresh)

# $\mathcal{W}$ Abstract Machine (Cont.)

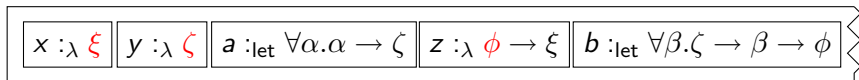
$(\text{let } x = e_1 \text{ in } e_2, \Gamma, k) \mapsto (e_1, \Gamma, \text{let } x \text{ in } e_2 :: k)$	[let-def]
$(\tau, \Gamma, \text{let } x \text{ in } e_2 :: k) \mapsto (e_2, \Gamma[x : \sigma], \text{let} :: k)$ $\sigma = \forall \bar{\alpha}. \{\bar{\alpha} / \mathcal{G}_\Gamma(\tau)\} \tau$ $\mathcal{G}_\Gamma(\tau) = \text{ftv}(\tau) - \text{ftv}(\Gamma)$	[let-body] ( $\bar{\alpha}$ fresh)
$(\tau, \Gamma[x : \sigma], \text{let} :: k) \mapsto (\tau, \Gamma, k)$	[let-out]
$(x, \Gamma, k) \mapsto (\mathcal{I}(\Gamma(x)), \Gamma, k)$ $\mathcal{I}(\forall \bar{\alpha}. \tau) = \{\bar{\xi} / \bar{\alpha}\} \tau$ $\mathcal{I}(\tau) = \tau$	[var] ( $\bar{\xi}$ fresh)

## $\mathcal{W}$ Abstract Machine (Cont.)

$(\text{let } x = e_1 \text{ in } e_2, \Gamma, k) \mapsto (e_1, \Gamma, \text{let } x \text{ in } e_2 :: k)$	[let-def]
$(\tau, \Gamma, \text{let } x \text{ in } e_2 :: k) \mapsto (e_2, \Gamma[x : \sigma], \text{let} :: k)$ $\sigma = \forall \bar{\alpha}. \{\bar{\alpha} / \mathcal{G}_\Gamma(\tau)\} \tau$ $\mathcal{G}_\Gamma(\tau) = \text{ftv}(\tau) - \text{ftv}(\Gamma)$	[let-body] ( $\bar{\alpha}$ fresh)
$(\tau, \Gamma[x : \sigma], \text{let} :: k) \mapsto (\tau, \Gamma, k)$	[let-out]
$(x, \Gamma, k) \mapsto (\mathcal{I}(\Gamma(x)), \Gamma, k)$ $\mathcal{I}(\forall \bar{\alpha}. \tau) = \{\bar{\xi} / \bar{\alpha}\} \tau$ $\mathcal{I}(\tau) = \tau$	[var] ( $\bar{\xi}$ fresh)

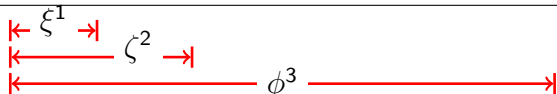
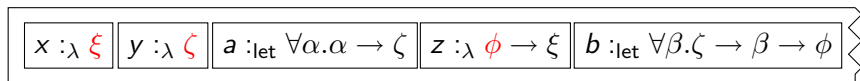
# $\lambda$ -ranking

## $\lambda$ -ranking: Some Observations



The first occurrence of a univariable must be in a  $\lambda$ -binding

## $\lambda$ -ranking: Some Observations



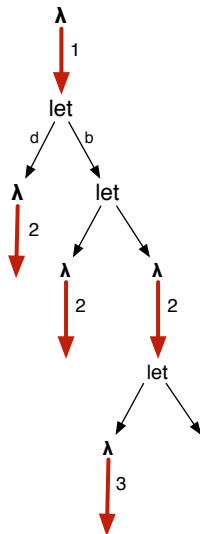
**intrinsic rank** of a univariable,  $R(\xi, \Gamma)$  is the count of  $\lambda$ -bindings ( $|\Gamma|_\lambda$ ) from top level to first binding containing that univariable, inclusive ( $\infty$  if it was never in  $\Gamma$ )

Intuition: Unvariables with finite intrinsic rank  $> |\Gamma|_\lambda$  must have been popped off.

## Visualizing Lambda Ranks

But intrinsic rank is still a property of the entire type environment.  
Is there a *syntactically* derivable equivalent?

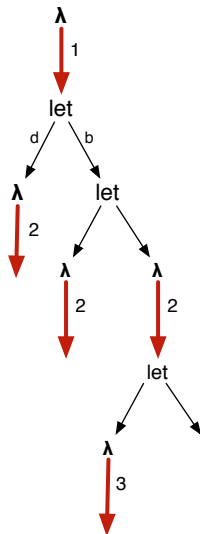
- The  $\lambda$ -nesting matches the intrinsic rank initially.
- Annotate univariates with  $\lambda$ -depth (e.g.,  $\xi^2$ ).
- The abstract machine will keep it that way.



## Visualizing Lambda Ranks

But intrinsic rank is still a property of the entire type environment.  
Is there a *syntactically* derivable equivalent?

- The  $\lambda$ -nesting matches the intrinsic rank initially.
- Annotate univariables with  $\lambda$ -depth (e.g.,  $\xi^2$ ).
- The abstract machine will keep it that way.



$\mathcal{W}_\lambda$  Abstract Machine

$$(c, \Gamma, d, k) \mapsto_\lambda (c', \Gamma', d', k')$$

Trivial property:  $d = |k|_\lambda$

$$(\lambda x.e, \Gamma, d, k) \mapsto_\lambda (e, \Gamma[x : \xi^{d+1}], d+1, \lambda :: k) \quad [\lambda\text{-in}]$$

( $\xi$  is fresh)

$$(\tau, \Gamma[x : \tau'], d, \lambda :: k) \mapsto_\lambda (\tau' \rightarrow \tau, \Gamma, d-1, k) \quad [\lambda\text{-out}]$$

$$(x, \Gamma, d, k) \mapsto_\lambda (\mathcal{I}(\Gamma(x)), \Gamma, d, k) \quad [\text{var}]$$

$$\mathcal{I}(\forall \bar{\alpha}. \tau) = \{\bar{\xi}^\infty / \bar{\alpha}\} \tau$$

$$\mathcal{I}(\tau) = \tau$$

$$(\bar{\xi} \text{ fresh})$$

## $\mathcal{W}_\lambda$ Abstract Machine (Cont.)

$$\begin{aligned}
 (\tau, \Gamma, d, \text{let } x \text{ in } e_2 :: k) &\mapsto_\lambda (e_2, \Gamma[x : \sigma], d, \text{let} :: k) \quad [\text{let-body}] \\
 \sigma &= \forall \bar{\alpha}. \{\bar{\alpha} / \mathcal{G}_d(\tau)\} \tau \\
 \mathcal{G}_d(\tau) &= \{\xi^m \in \text{ftv}(\tau) \mid m > d\} \\
 &(\bar{\alpha} \text{ is fresh})
 \end{aligned}$$

$$\begin{aligned}
 (\tau, \Gamma, d, (\tau' \square) :: k) &\mapsto_\lambda (\theta \xi^\infty, \theta \Gamma, d, \theta k) \quad [\text{app-out}] \\
 \theta &= \mathcal{U}(\tau', \tau \rightarrow \xi^\infty) \\
 &(\xi \text{ is fresh})
 \end{aligned}$$

## $\mathcal{W}_\lambda$ Abstract Machine (Cont.)

$$\begin{aligned}
 (\tau, \Gamma, d, \text{let } x \text{ in } e_2 :: k) &\mapsto_\lambda (e_2, \Gamma[x : \sigma], d, \text{let} :: k) \quad [\text{let-body}] \\
 \sigma &= \forall \bar{\alpha}. \{\bar{\alpha} / \mathcal{G}_d(\tau)\} \tau \\
 \mathcal{G}_d(\tau) &= \{\xi^m \in \text{ftv}(\tau) \mid m > d\} \\
 &(\bar{\alpha} \text{ is fresh})
 \end{aligned}$$

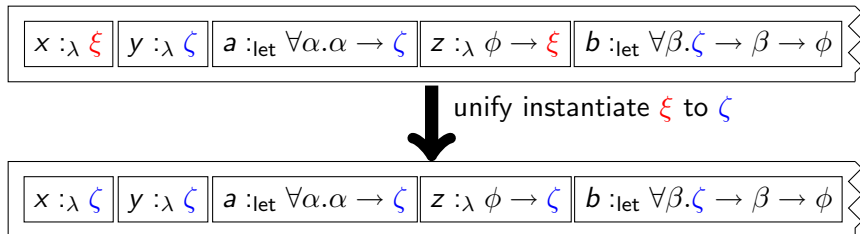
$$\begin{aligned}
 (\tau, \Gamma, d, (\tau' \square) :: k) &\mapsto_\lambda (\theta \xi^\infty, \theta \Gamma, d, \theta k) \quad [\text{app-out}] \\
 \theta &= \mathcal{U}(\tau', \tau \rightarrow \xi^\infty) \\
 &(\xi \text{ is fresh})
 \end{aligned}$$

# $\lambda$ -ranking: Unification Substitutions

Unification substitutions. . .

- eliminate some univariables (in the domain of the subst)
- produce new occurrences of univariables in the range

Special case: promote some univariable  $\zeta$  by introducing new occurrences of  $\zeta$  that precede the previous first occurrence (i.e., decreases the intrinsic rank of  $\zeta$ )

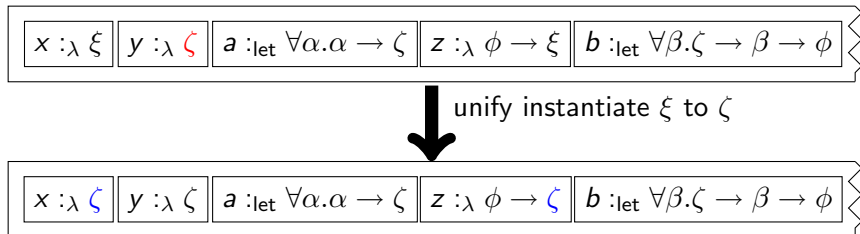


## $\lambda$ -ranking: Unification Substitutions

Unification substitutions. . .

- eliminate some univariables (in the domain of the subst)
- produce new occurrences of univariables in the range

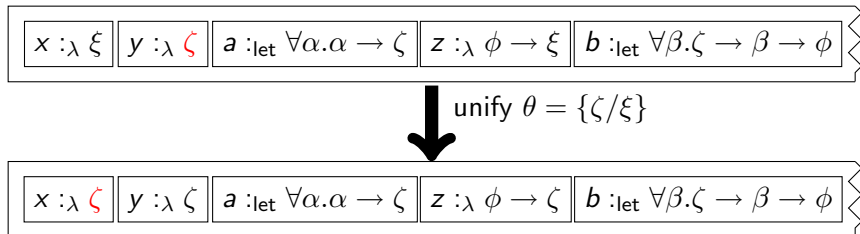
Special case: **promote** some univariable  $\zeta$  by introducing new occurrences of  $\zeta$  that precede the previous first occurrence (i.e., decreases the intrinsic rank of  $\zeta$ )



## $\lambda$ -ranking: Unification Substitutions

Unification substitutions. . .

- eliminate some univariables (in the domain of the subst)
  - produce new occurrences of univariables in the range
- Special case: promote** some univariable  $\zeta$  by introducing new occurrences of  $\zeta$  that precede the previous first occurrence (i.e., decreases the intrinsic rank of  $\zeta$ )



# Unification

$$\mathcal{U}(\xi^m, \tau) = \mathcal{L}_m(\tau) \circ \{\tau/\xi^m\} \quad [\text{instantiation}]$$

$$\mathcal{L}_m(\tau) = \{\zeta^m/\zeta^d \mid \zeta^d \in \text{ftv}(\tau) \wedge d > m\}$$

## $\mathcal{W}_\lambda$ Invariants

The following properties hold for any machine state  $(c, \Gamma, d, k)$  reachable from an initial machine state  $(e, \emptyset, 0, \bullet)$ :

1.  $\xi^m \in \text{ftv}(c)$  such that  $m \leq |\Gamma|_\lambda$ ,  $\xi^m$  must occur in  $\Gamma$
2.  $\xi^m \in \text{ftv}(\Gamma)$ ,  $R(\xi^m, \Gamma) = m$
3.  $|\Gamma|_\lambda = |k|_\lambda$
4. A technical fact constraining ranks of univariables on  $k$

## $\mathcal{W}_\lambda$ Invariants

The following properties hold for any machine state  $(c, \Gamma, d, k)$  reachable from an initial machine state  $(e, \emptyset, 0, \bullet)$ :

1.  $\xi^m \in \text{ftv}(c)$  such that  $m \leq |\Gamma|_\lambda$ ,  $\xi^m$  must occur in  $\Gamma$
2.  $\xi^m \in \text{ftv}(\Gamma)$ ,  $R(\xi^m, \Gamma) = m$
3.  $|\Gamma|_\lambda = |k|_\lambda$
4. A technical fact constraining ranks of univariables on  $k$

## $\mathcal{W}_\lambda$ Invariants

The following properties hold for any machine state  $(c, \Gamma, d, k)$  reachable from an initial machine state  $(e, \emptyset, 0, \bullet)$ :

1.  $\xi^m \in \text{ftv}(c)$  such that  $m \leq |\Gamma|_\lambda$ ,  $\xi^m$  must occur in  $\Gamma$
2.  $\xi^m \in \text{ftv}(\Gamma)$ ,  $R(\xi^m, \Gamma) = m$
3.  $|\Gamma|_\lambda = |k|_\lambda$
4. A technical fact constraining ranks of univariables on  $k$

## $\mathcal{W}_\lambda$ Invariants

The following properties hold for any machine state  $(c, \Gamma, d, k)$  reachable from an initial machine state  $(e, \emptyset, 0, \bullet)$ :

1.  $\xi^m \in \text{ftv}(c)$  such that  $m \leq |\Gamma|_\lambda$ ,  $\xi^m$  must occur in  $\Gamma$
2.  $\xi^m \in \text{ftv}(\Gamma)$ ,  $R(\xi^m, \Gamma) = m$
3.  $|\Gamma|_\lambda = |k|_\lambda$
4. A technical fact constraining ranks of univariables on  $k$

# $\mathcal{W}_\lambda$ And $\mathcal{W}$ Are Equivalent

## Theorem

$$(e, \emptyset, 0, \bullet) \mapsto_\lambda^* (\tau, \emptyset, 0, \bullet)$$

$$\mathbf{iff} (e, \emptyset, \bullet) \mapsto^* (\tau, \emptyset, \bullet)$$

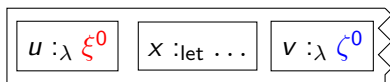
# Let-ranking

# Let-ranking

When to bump up the rank? What does the rank calculate?

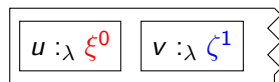
$$\lambda u.\text{let } x = u$$

$$\text{in } \lambda v.\text{let } y = (u,v)$$

$$\text{in } \dots$$


$$\lambda u.\text{let } x = \lambda v.\text{let } y = (u,v)$$

$$\text{in } \dots$$

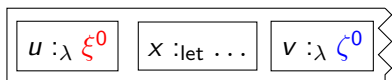
$$\text{in } \dots$$


# Let-ranking

When to bump up the rank? What does the rank calculate?

$$\lambda u. \text{let } x = u$$

$$\text{in } \lambda v. \text{let } y = (u, v)$$

$$\text{in } \dots$$


$$\lambda u. \text{let } x = \lambda v. \text{let } y = (u, v)$$

$$\text{in } \dots$$

$$\text{in } \dots$$


1. **Punctuate** environment with **letd (definien)** mark when entering let-definien
2. Intrinsic rank counts let-definiens preceding first occurrence of a univariable

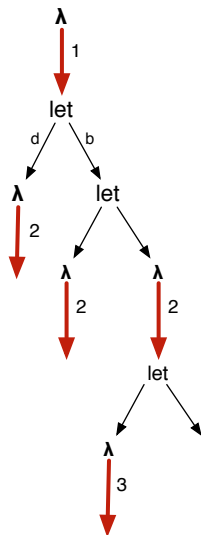
# Let-ranking

$(\text{let } x = e_1 \text{ in } e_2, \Gamma, d, k) \mapsto_L (e_1, \Gamma[\text{letd}], d + 1, \text{let } x \text{ in } e_2 :: k)$

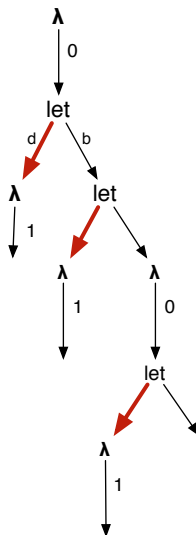
Derive rank annotation from let definien nesting depth

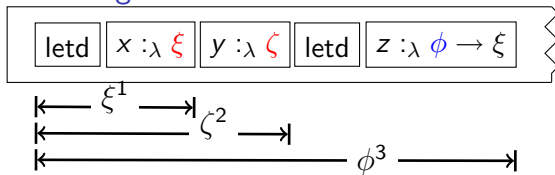
# $\lambda$ Versus Let-ranks

Lambda Rank

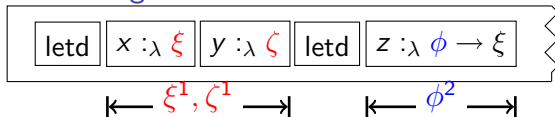


Let Rank



$\lambda$  Versus Let-ranks (2) $\lambda$ -ranking

## let-ranking



# Matrix of Abstract Machines

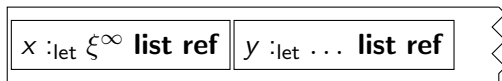
	Pure	Impure
Classic $\mathcal{W}$	$\mathcal{W}$	$\mathcal{W}_V$
$\lambda$ -ranking	$\mathcal{W}_\lambda$	$\mathcal{W}_{\lambda V}$
let-ranking	$\mathcal{W}_L$	$\mathcal{W}_{LV}$

# The Value Restriction

# Value Restriction

## Complication #1

let  $x = \text{ref nil}$   
 in let  $y = x$  in ...

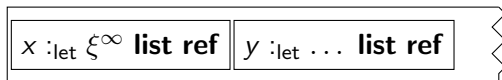


Violates an observation about the environment  
 Univariable can first occur in non-value let binding (in environment)

# Value Restriction

## Complication #1

let  $x = \text{ref nil}$   
in let  $y = x$  in ...



Violates an observation about the environment  
Univariable can first occur in non-value let binding (in environment)

## Solution

Treat non-value let bindings as we do  $\lambda s$

$f ::= \dots \mid \text{let}_n x \text{ in } e \mid \text{let}_n$

$$(\tau, \Gamma, d, \text{let}_n x \text{ in } e' :: k) \mapsto_{\lambda V} (e', \Gamma[x : \tau], d + 1, \text{let}_n :: k)$$

# Value Restriction

## Complication #2

`let`  $x = \text{ref nil}$        $x : \xi^\infty$  **list ref**  
in `let`  $y = x$  in ...

High ranked univariables may leak out from a non-value let and get generalized by a value let at a later point

# Value Restriction

## Complication #2

$\text{let } x = \text{ref nil} \quad x : \xi^\infty \text{ list ref}$   
 in  $\text{let } y = x \text{ in } \dots$

High ranked univariables may leak out from a non-value let and get generalized by a value let at a later point

## Solution

Promote high ranked univariables to non-value let's  $\lambda$ /let-depth to prevent unsound generalization

$$(\tau, \Gamma, d, \text{let}_n x \text{ in } e :: k) \mapsto_{\lambda V} (e, \theta\Gamma[x : \theta\tau], d + 1, \text{let}_n :: \theta k)$$

$$\theta = \mathcal{L}_d(\tau)$$

# Issues

- Support for alternative typechecking traversal orders
- Scalability of formalization (E.g. try adding SML-style overloading of operators and literals, or equality types, or ...).
- Relate to refunctionalization of abstract machines (Danvy and Millikin 07)

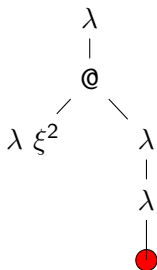
## Concluding Remarks

- Same lightweight formal framework works for both  $\lambda$  and let-ranking techniques
- Extension to value restriction is straightforward
- Desired equivalence theorem holds for all variants and proofs are very similar

# Thanks!

## Technical Fact About $k$

How univariates on the stack are constrained



Univariable is low ranked only if rank  $\leq |k'|_\lambda$  where  $k'$  is the segment of  $k$  below the occurrence of that univariable in  $k$

$\mathbf{k} \lambda :: \lambda :: (\xi^2 \square) :: \lambda$

## $\lambda$ Versus Let-ranks

- $\lambda$ -ranks easier intuition, straightforward adaptation for value restriction
- let-ranks more parsimonious, only differentiating between different scopes of generalization

# Generic Univariables

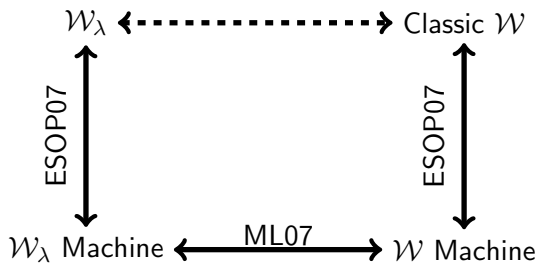
## Why give a rank $\infty$ ?

- Rank  $\infty$  corresponds to the fact that univariable is not in the type environment.
- Get that rank  $>$  all univariable ranks for univariables in type environment for free. (Otherwise, we need a lemma that says rank  $d + 1 >$  all ranks for univariables in type environment at all times)

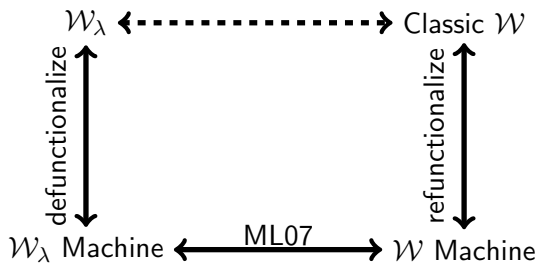
# Difference Between Classic $\mathcal{W}$ Machine and Algorithm $\mathcal{W}$

- Refunctionalization reveals that the Classic Algorithm  $\mathcal{W}$  machine is essentially equivalent to Algorithm  $\mathcal{W}$
- Sole exception: A space efficiency optimization — the machine pops the type environment after typechecking with an extended environment instead of *saving* a copy of the environment before extending it.

## Where This Work Fits



## Where This Work Fits



## Question

Q: Is the  $\text{let}_n$  necessary?

A: No, the  $\lambda$  frame would be sufficient, but the  $\text{let}_n x$  in  $e$  frame is necessary so we can do the right thing after typechecking the definien.