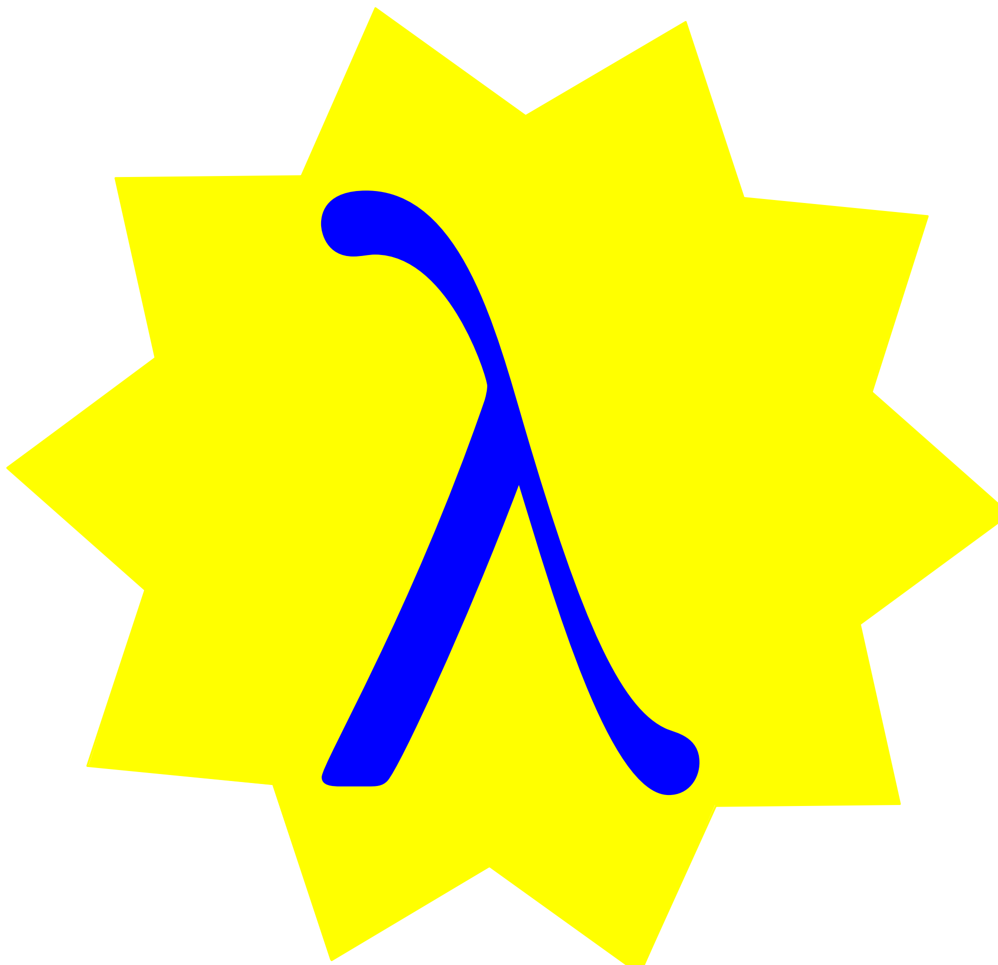
The Call-by-need Lambda Calculus, Revisited

Stephen Chang and Matthias Felleisen

Northeastern University

26/3/2012



Church

Theoretical Computer Science 1 (1975) 125–159.

CALL-BY-NAME, CALL-BY-VALUE AND THE λ-CALCULUS

G. D. PLOTKIN

Department of Machine Intelligence, School of Artificial Intelligence, University of Edinburgh, Edinburgh, United Kingdom

Communicated by R. Milner
Received 1 August 1974

Abstract. This paper examines the old question of the relationship between ISWIM and the λ-calculus, using the distinction between call-by-value and call-by-name. It is held that the relationship should be mediated by a standardisation theorem. Since this leads to difficulties, a new λ-calculus is introduced whose standardisation theorem gives a good correspondence with ISWIM as given by the SECD machine, but without the *letrec* feature. Next a call-by-name variant of ISWIM is introduced which is in an analogous correspondence with the usual λ-calculus. The relation between call-by-value and call-by-name is then studied by giving simulations of each language by the other and interpretations of each calculus in the other. These are obtained as another application of the continuation technique. Some emphasis is placed throughout on the notion of operational equality (or contextual equality). If terms can be proved equal in a calculus they are operationally equal in the corresponding language. Unfortunately, operational equality is not preserved by either of the simulations.

1. Introduction

Our intention is to study call-by-value and call-by-name in the setting of the lambda-calculus which was first used to explicate programming language features by Landin [5, 6, 7]. To this end, for each calling mechanism we set up a programming language and a formal calculus and then show how each determines the other. After that we give simulations of the call-by-value programming language by the call-by-name one and vice versa — this also provides interpretations of each calculus in the other one.

If the terms of the λ-calculus (we have in mind the λK-β calculus for the moment) are regarded as rules, with a reduction relation showing how they may be carried out and indeed with a normal order reduction sequence capturing, in deterministic fashion, all possible normal forms, then we have already pretty well determined a programming language.

On the other hand, the language can be regarded as giving true equations between programs (= terms of the calculus). Informally, one program equals another, operationally, if it can be substituted for the other in all contexts without "changing

Theoretical Computer Science 1 (1975) 125–159.

If the terms of the λ-calculus are regarded as rules, with a reduction relation showing how they may be carried out, then we have already pretty well determined a programming language.

Department of Machine Intelligence, School of Artificial Intelligence, University of Edinburgh, Edinburgh, United Kingdom

Communicated by R. Milner
Received 1 August 1974

Abstract. This paper examines the old question of the relationship between ISWIM and the λ-calculus, using the distinction between call-by-value and call-by-name. It is held that the relationship should be mediated by a standardisation theorem. Since this leads to difficulties, a new λ-calculus is introduced whose standardisation theorem gives a good correspondence with ISWIM as given by the SECD machine, but without the *letrec* feature. Next a call-by-name variant of ISWIM is introduced which is in an analogous correspondence with the usual λ-calculus. The relation between call-by-value and call-by-name is then studied by giving simulations of each language by the other and interpretations of each calculus in the other. These are obtained as another application of the continuation technique. Some emphasis is placed throughout on the notion of operational equality (or contextual equality). If terms can be proved equal in a calculus they are operationally equal in the corresponding language. Unfortunately, operational equality is not preserved by either of the simulations.

1. Introduction

Our intention is to study call-by-value and call-by-name in the setting of the lambda-calculus which was first used to explicate programming language features by Landin [5, 6, 7]. To this end, for each calling mechanism we set up a programming language and a formal calculus and then show how each determines the other. After that we give simulations of the call-by-value programming language by the call-by-name one and vice versa — this also provides interpretations of each calculus in the other one.

If the terms of the λ-calculus (we have in mind the λK-β calculus for the moment) are regarded as rules, with a reduction relation showing how they may be carried out and indeed with a normal order reduction sequence capturing, in deterministic fashion, all possible normal forms, then we have already pretty well determined a programming language.

On the other hand, the language can be regarded as giving true equations between programs (= terms of the calculus). Informally, one program equals another, operationally, if it can be substituted for the other in all contexts without "changing

Theoretical Computer Science 1 (1975) 125–159.

If the terms of the λ-calculus are regarded as rules, with a reduction relation showing how they may be carried out, then we have already pretty well determined a programming language.

Department of Machine Intelligence, School of Artificial Intelligence, University of Edinburgh, Edinburgh, United Kingdom

Communicated by R. Milner
Received 1 August 1974

a new λ-calculus is introduced whose standardisation theorem gives a good correspondence with ISWIM as given by the SECD machine . . . Next a call-by-name variant of ISWIM is introduced which is in an analogous correspondence with the usual λ-calculus.

variant of ISWIM is introduced which is in an analogous correspondence with the usual λ-calculus. The relation between call-by-value and call-by-name is then studied by giving simulations of each language by the other and interpretations of each calculus in the other. These are obtained as another application of the continuation technique. Some emphasis is placed throughout on the notion of operational equality (or contextual equality). If terms can be proved equal in a calculus they are operationally equal in the corresponding language. Unfortunately, operational equality is not preserved by either of the simulations.

1. Introduction

Our intention is to study call-by-value and call-by-name in the setting of the lambda-calculus which was first used to explicate programming language features by Landin [5, 6, 7]. To this end, for each calling mechanism we set up a programming language and a formal calculus and then show how each determines the other. After that we give simulations of the call-by-value programming language by the call-by-name one and vice versa — this also provides interpretations of each calculus in the other one.

If the terms of the λ-calculus (we have in mind the λK-β calculus for the moment) are regarded as rules, with a reduction relation showing how they may be carried out and indeed with a normal order reduction sequence capturing, in deterministic fashion, all possible normal forms, then we have already pretty well determined a programming language.

On the other hand, the language can be regarded as giving true equations between programs (= terms of the calculus). Informally, one program equals another, operationally, if it can be substituted for the other in all contexts without "changing

Theoretical Computer Science 1 (1975) 125–159.

If the terms of the λ-calculus are regarded as rules, with a reduction relation showing how they may be carried out, then we have already pretty well determined a programming language.

Department of Machine Intelligence, School of Artificial Intelligence, University of Edinburgh, Edinburgh, United Kingdom

Communicated by R. Milner
Received 1 August 1974

a new λ-calculus is introduced whose standardisation theorem gives a good correspondence with ISWIM as given by the SECD machine . . . Next a call-by-name variant of ISWIM is introduced which is in an analogous correspondence with the usual λ-calculus.

variant of ISWIM is introduced which is in an analogous correspondence with the usual λ-calculus. The relation between call-by-value and call-by-name is then studied by giving simulations of each language by the other and interpretations of each calculus in the other. These are obtained as another application of the continuation technique. Some emphasis is placed throughout on the notion of operational equality (or contextual equality). If terms can be proved equal in a calculus they are operationally equal in the corresponding language. Unfortunately, operational equality is not preserved by either of the simulations.

In both cases the calculi are seen to be correct from the point of view of the programming languages.

Our intention is to study call-by-value and call-by-name in the setting of the lambda-calculus which was first used to explicate programming language features by Landin [5, 6, 7]. To this end, for each calling mechanism we set up a programming language and a formal calculus and then show how each determines the other. After that we give simulations of the call-by-value programming language by the call-by-name one and vice versa — this also provides interpretations of each calculus in the other one.

If the terms of the λ-calculus (we have in mind the λK-β calculus for the moment) are regarded as rules, with a reduction relation showing how they may be carried out and indeed with a normal order reduction sequence capturing, in deterministic fashion, all possible normal forms, then we have already pretty well determined a programming language.

On the other hand, the language can be regarded as giving true equations between programs (= terms of the calculus). Informally, one program equals another, operationally, if it can be substituted for the other in all contexts without "changing

Theoretical Computer Science 1 (1975) 125–159.

If the terms of the λ-calculus are regarded as rules, with a reduction relation showing how they may be carried out, then we have already pretty well determined a programming language.

Department of Machine Intelligence, School of Artificial Intelligence, University of Edinburgh, Edinburgh, United Kingdom

Communicated by R. Milner
Received 1 August 1974

a new λ-calculus is introduced whose standardisation theorem gives a good correspondence with ISWIM as given by the SECD machine . . . Next a call-by-name variant of ISWIM is introduced which is in an analogous correspondence with the usual λ-calculus.

variant of ISWIM is introduced which is in an analogous correspondence with the usual λ-calculus. The relation between call-by-value and call-by-name is then studied by giving simulations of each language by the other and interpretations of each calculus in the other. These are obtained as another application of the continuation technique. Some emphasis is placed throughout on the notion of operational equality (or contextual equality). If terms can be proved equal in a calculus they are operationally equal in the corresponding language. Unfortunately, operational equality is not preserved by either of the simulations.

In both cases the calculi are seen to be correct from the point of view of the programming languages.

Our intention is to study call-by-value and call-by-name in the setting of the lambda-calculus which was first used to explicate programming language features by Landin [5, 6, 7]. To this end, for each calling mechanism we set up a programming language and a formal calculus and then show how each determines the other. After that we give simulations of the call-by-value programming language by the call-by-

So one has to look for programming language/calculus pairs.

If the terms of the λ-calculus (we have in mind the λK-β calculus for the moment) are regarded as rules, with a reduction relation showing how they may be carried out and indeed with a normal order reduction sequence capturing, in deterministic fashion, all possible normal forms, then we have already pretty well determined a programming language.

On the other hand, the language can be regarded as giving true equations between programs (= terms of the calculus). Informally, one program equals another, operationally, if it can be substituted for the other in all contexts without "changing

λ

call-by-name

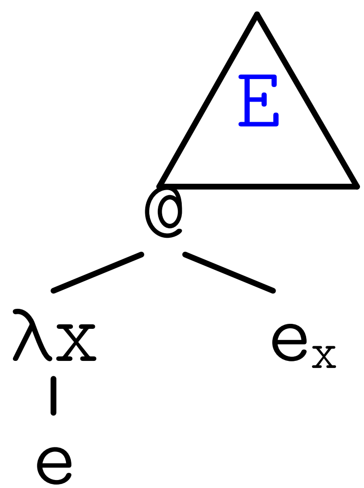
$$(\lambda x.e)\ e_x \quad \rightarrow \quad e\{x:=e_x\} \qquad (\beta)$$

λ

call-by-name

$$E[(\lambda x.e)\ e_x] \rightarrow E[e\{x:=e_x\}] \qquad (\beta)$$

"leftmost-outermost"

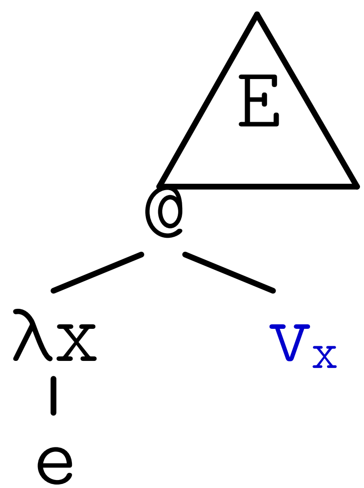


λ_v

call-by-value

$$E[(\lambda x.e)\ v_x] \rightarrow E[e\{x:=v_x\}] \qquad (\beta_v)$$

"leftmost-outermost"



call-by-need

call-by-need

1) Evaluate argument only when needed.

call-by-need

1) Evaluate argument only when needed.
2) Evaluate argument at most once.

? $\lambda_{\mathtt{need}}$?

call-by-need

1) Evaluate argument only when needed.
2) Evaluate argument at most once.

$\lambda_{\texttt{need-af}}$ Ariola/Felleisen '94,'95,'97 $\stackrel{?}{=}$ $\lambda_{\texttt{need}}$ $\stackrel{?}{=}$ $\lambda_{\texttt{need-mow}}$ Maraist/Odersky/Wadler '94,'95,'98

call-by-need

1) Evaluate argument only when needed.
2) Evaluate argument at most once.

$\lambda_{\texttt{need-af}}$
Ariola/Felleisen
'94,'95,'97

$\neq$

$\lambda_{\texttt{need}}$

$\neq$

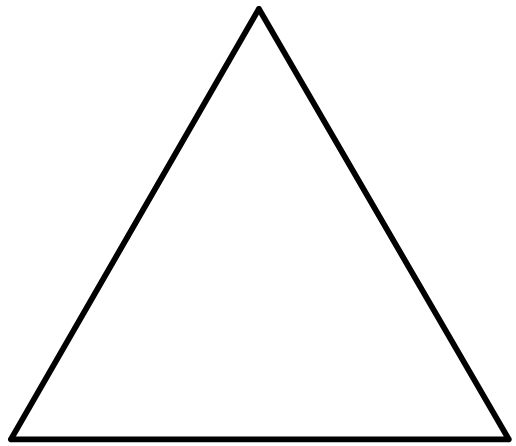
$\lambda_{\texttt{need-mow}}$
Maraist/Odersky/Wadler
'94,'95,'98

call-by-need

1) Evaluate argument only when needed.
2) Evaluate argument at most once.

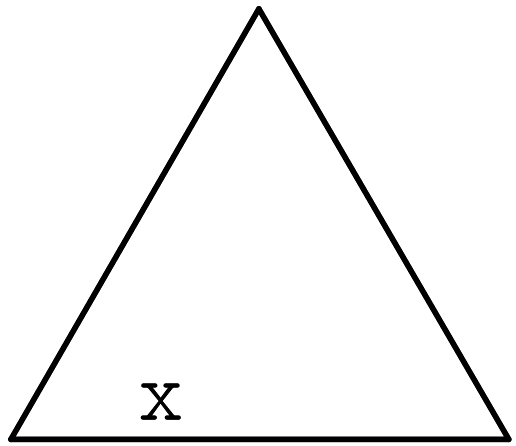
$\lambda_{\texttt{need-af}}$ Ariola/Felleisen '94,'95,'97 $\neq$ $\lambda_{\texttt{need}}$ $\neq$ $\lambda_{\texttt{need-mow}}$ Maraist/Odersky/Wadler '94,'95,'98

call-by-need



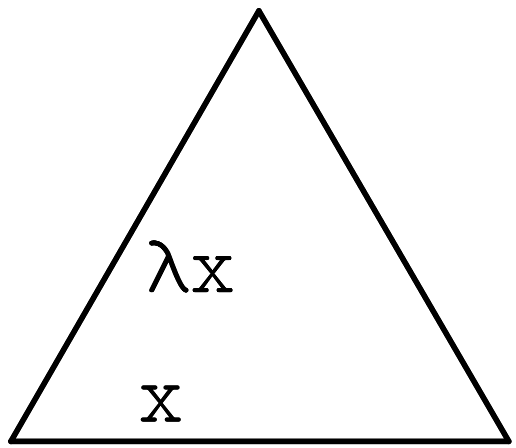
$\lambda_{\texttt{need-af}}$ Ariola/Felleisen '94,'95,'97 $\neq$ $\lambda_{\texttt{need}}$ $\neq$ $\lambda_{\texttt{need-mow}}$ Maraist/Odersky/Wadler '94,'95,'98

call-by-need



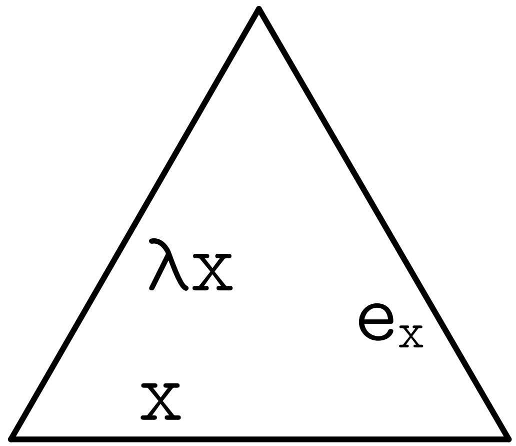
$\lambda_{\texttt{need-af}}$ Ariola/Felleisen '94,'95,'97 $\neq$ $\lambda_{\texttt{need}}$ $\neq$ $\lambda_{\texttt{need-mow}}$ Maraist/Odersky/Wadler '94,'95,'98

call-by-need



$\lambda_{\texttt{need-af}}$ Ariola/Felleisen '94,'95,'97 $\neq$ $\lambda_{\texttt{need}}$ $\neq$ $\lambda_{\texttt{need-mow}}$ Maraist/Odersky/Wadler '94,'95,'98

call-by-need

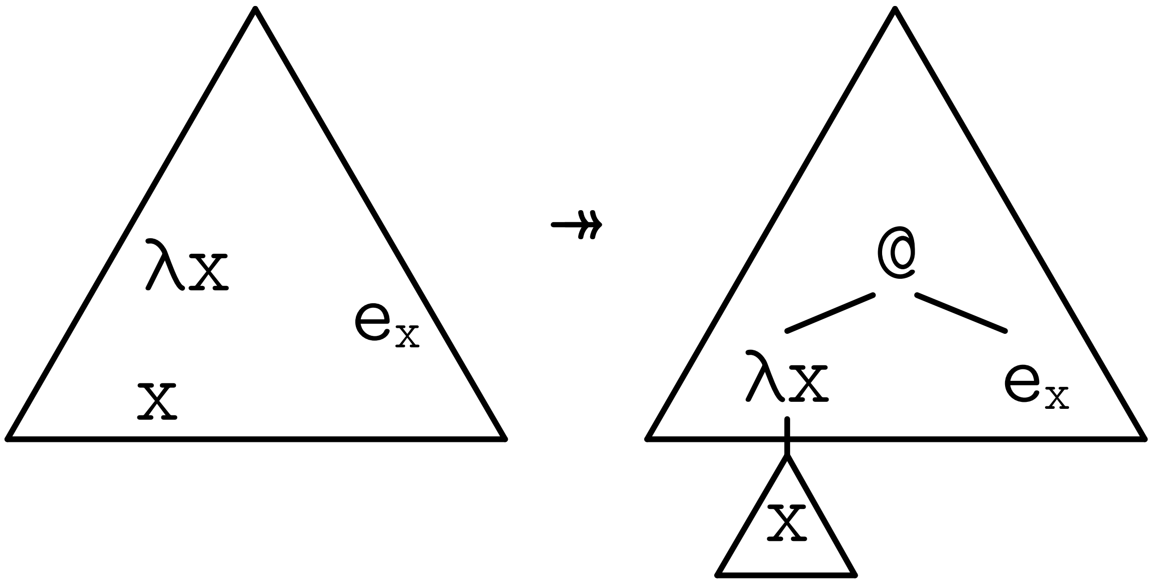


$\lambda_{\texttt{need-af}}$ $\neq$ $\lambda_{\texttt{need}}$ $\neq$ $\lambda_{\texttt{need-mow}}$

Ariola/Felleisen '94,'95,'97

Maraist/Odersky/Wadler '94,'95,'98

call-by-need



(reshuffle)

$\lambda_{\texttt{need-af}}$
Ariola/Felleisen
'94,'95,'97

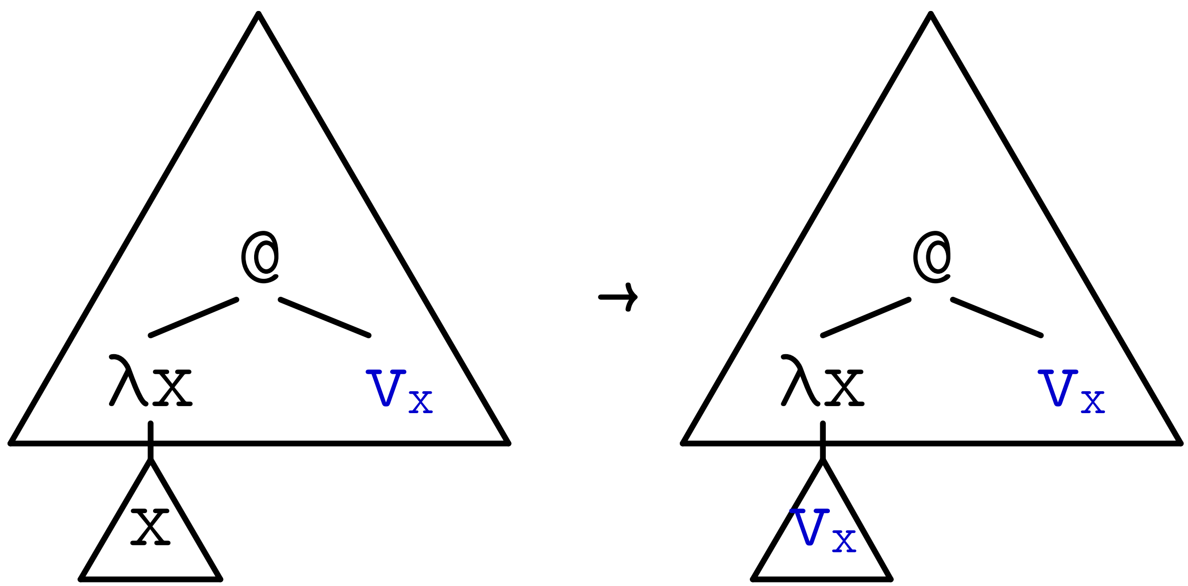
$\neq$

$\lambda_{\texttt{need}}$

$\neq$

$\lambda_{\texttt{need-mow}}$
Maraist/Odersky/Wadler
'94,'95,'98

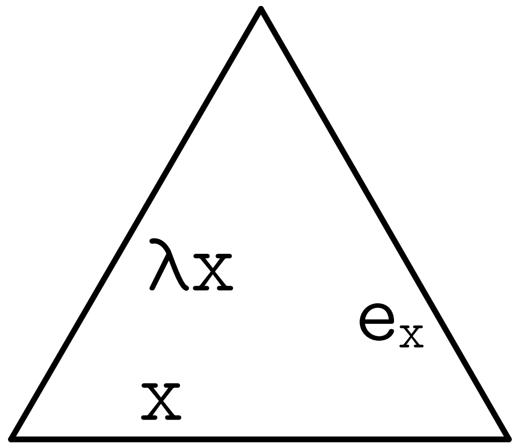
call-by-need



(like β)

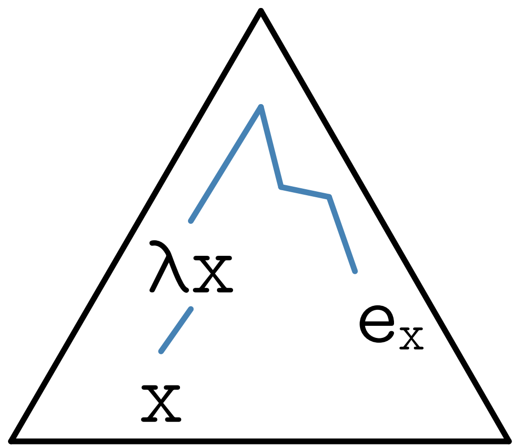
our $\lambda_{\texttt{need}}$

call-by-need



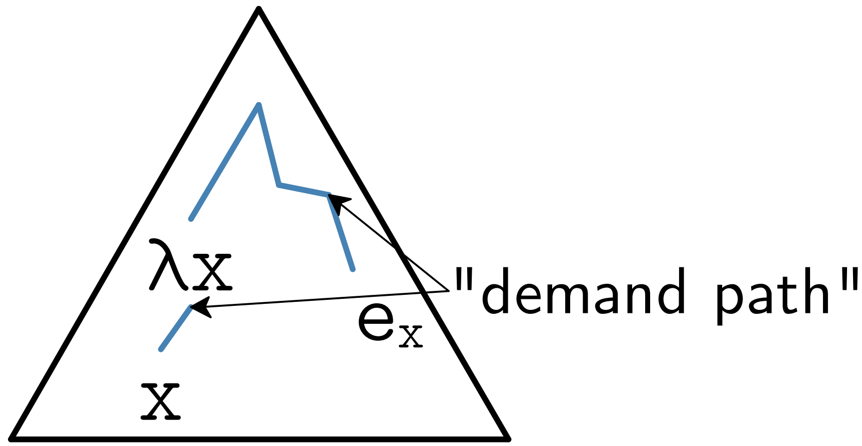
our $\lambda_{\texttt{need}}$

call-by-need



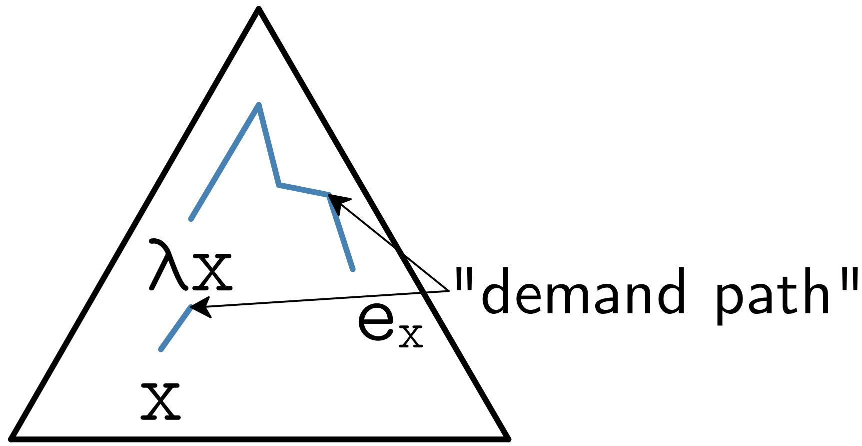
our $\lambda_{\texttt{need}}$

call-by-need



our $\lambda_{\mathtt{need}}$

call-by-need



($\beta_{\mathtt{need}}$)

Old $\lambda_{\texttt{need}}$: Operational Overview

1) Find the next demanded variable.

2) Find its corresponding argument and evaluate it.

3) Substitute evaluated argument for demanded variable.

OLD $\lambda_{\texttt{need}}$: OPERATIONAL OVERVIEW

1) Find the next demanded variable.

2) Find its corresponding argument and evaluate it.

3) Substitute evaluated argument for demanded variable.

OLD $\lambda_{\texttt{need}}$: DEMAND CONTEXTS

```
D = [ ] | D e
```

Old λ_{need}: Demand Contexts

$$D = [\ ] \mid D\ e$$

Old $\lambda_{\texttt{need}}$: Demand Contexts

```
D = [ ] | D e | (λx.D) e
```

OLD $\lambda_{\texttt{need}}$: DEMAND CONTEXTS

$$D = [\ ] \mid D\ e \mid (\lambda x.D)\ e$$

Old $\lambda_{\texttt{need}}$: Demand Contexts

$$\texttt{D = [ ] | D e | (}\lambda\texttt{x.D) e}$$

OLD $\lambda_{\texttt{need}}$: DEMAND CONTEXTS

```
D = [ ] | D e | (λx.D) e
```

↑

binding structure

Old $\lambda_{\texttt{need}}$: Demand Contexts

$$\texttt{D = [ ] | D e | B[D]}$$

⬆

binding structure

⬇

$$\texttt{B = [ ] | (}\lambda\texttt{x.B) e}$$

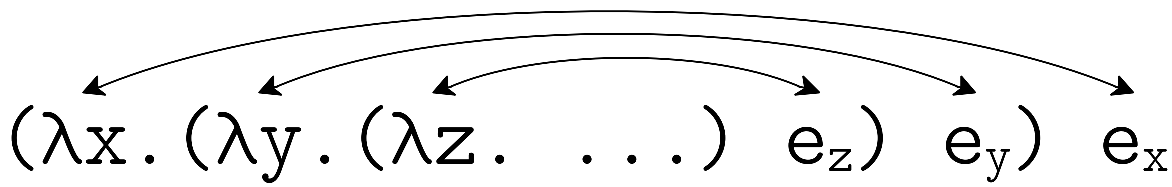
OLD $\lambda_{\texttt{need}}$: BINDING STRUCTURE

$$\texttt{B = [ ] | (}\lambda\texttt{x.B) e}$$

$$\texttt{(}\lambda\texttt{x.(}\lambda\texttt{y.(}\lambda\texttt{z. ...) e}_\texttt{z}\texttt{) e}_\texttt{y}\texttt{) e}_\texttt{x}$$

Old $\lambda_{\texttt{need}}$: Binding Structure

`B = [ ] | (λx.B) e`



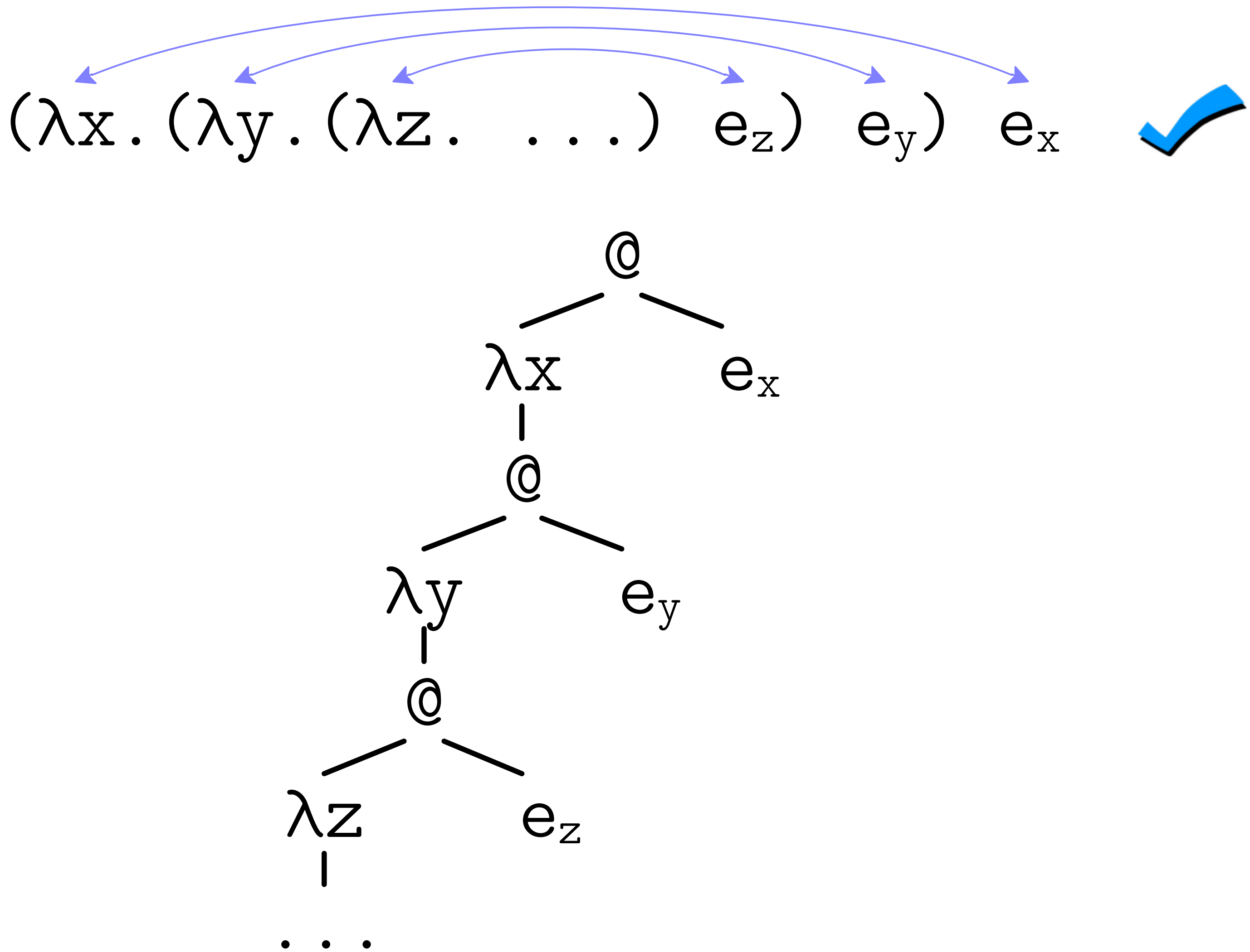
OLD $\lambda_{\texttt{need}}$: BINDING STRUCTURE

`B = [ ] | (λx.B) e`

(λx.(λy.(λz. ...) e_z) e_y) e_x ✓

OLD λ_{need}: BINDING STRUCTURE

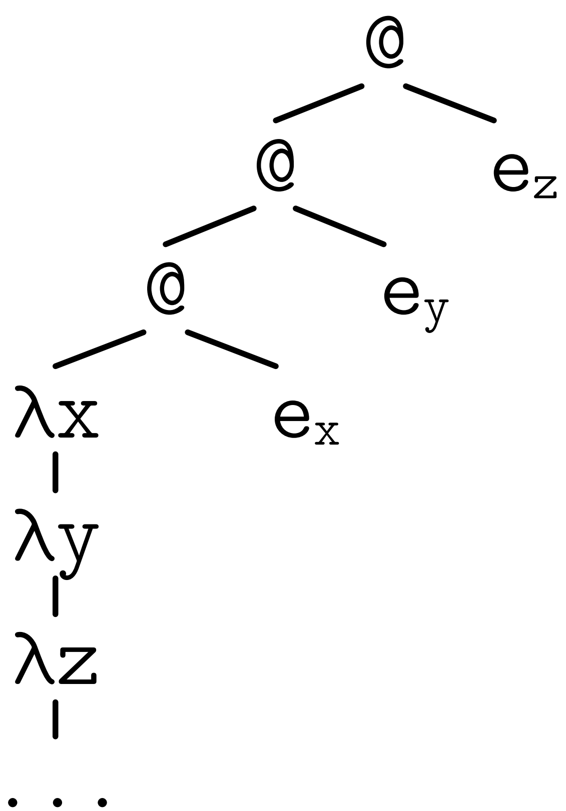
$$B = [\ ]\ |\ (\lambda x.B)\ e$$



Old λ_{need}: Reshuffling of Bindings

B = [ ] | (λx.B) e

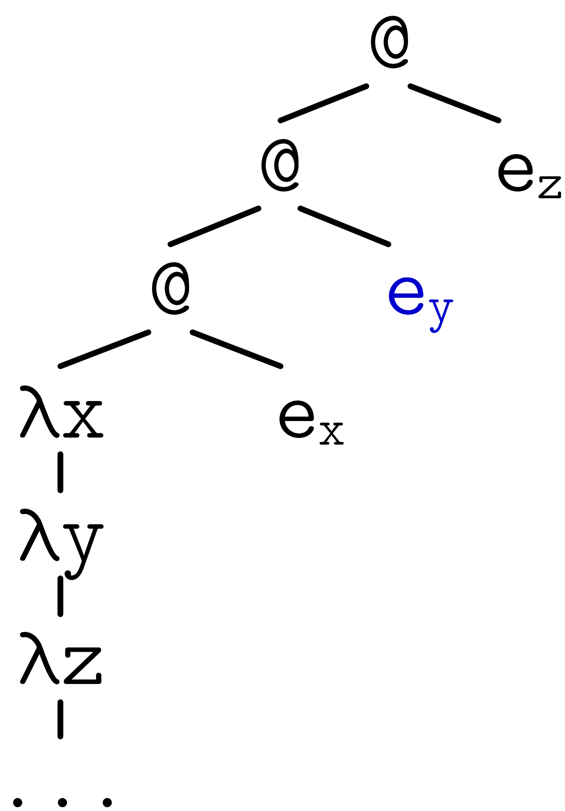
$(((\lambda x.\lambda y.\lambda z.\ \ldots)\ e_x)\ e_y)\ e_z$



Old $\lambda_{\mathtt{need}}$: Reshuffling of Bindings

`B = [ ] | (λx.B) e`

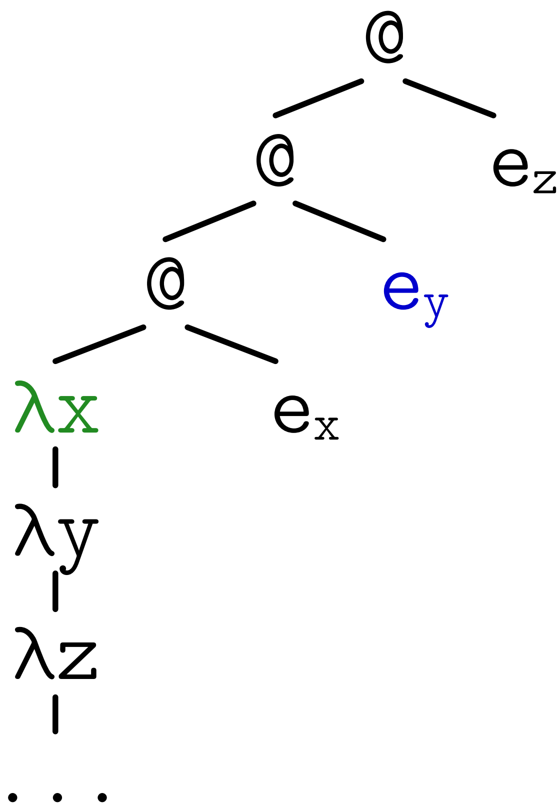
$(((\lambda x.\lambda y.\lambda z.\ \ldots)\ e_x)\ e_y)\ e_z$ ✗



OLD λ_{need}: RESHUFFLING OF BINDINGS

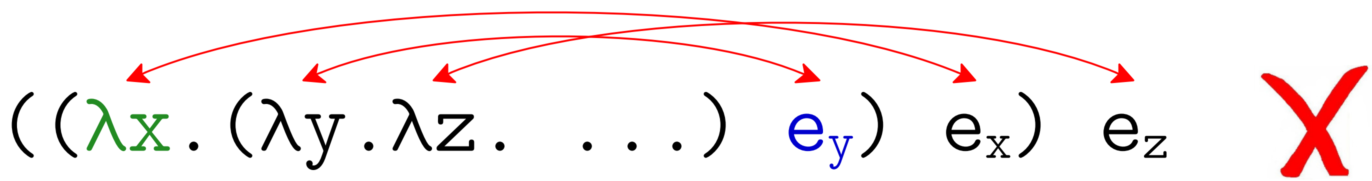
`B = [ ] | (λx.B) e`

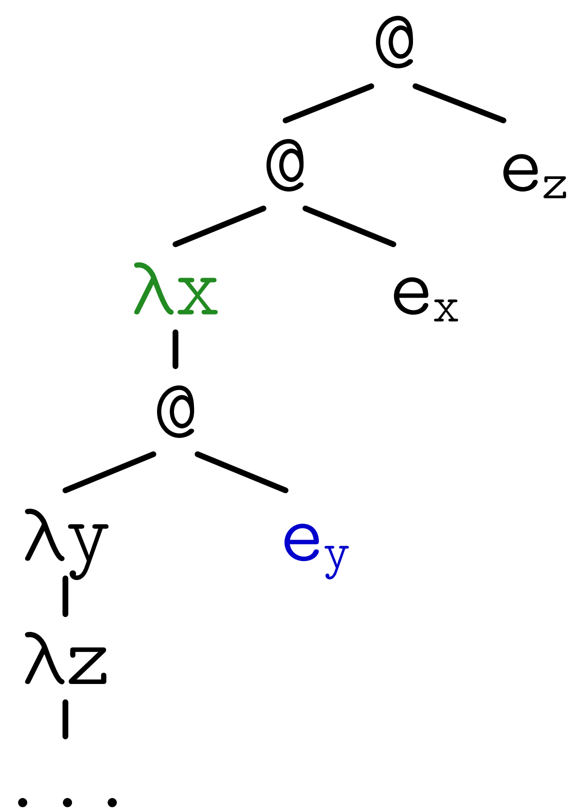
$(((\lambda x.\lambda y.\lambda z.\ \ldots)\ e_x)\ e_y)\ e_z$ ✗



Old $\lambda_{\texttt{need}}$: Reshuffling of Bindings

`B = [ ] | (λx.B) e`

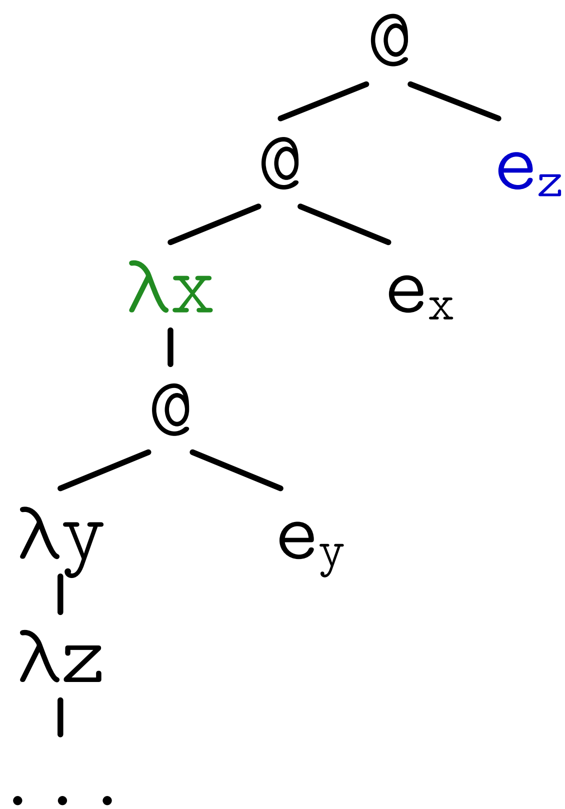




Old $\lambda_{\texttt{need}}$: Reshuffling of Bindings

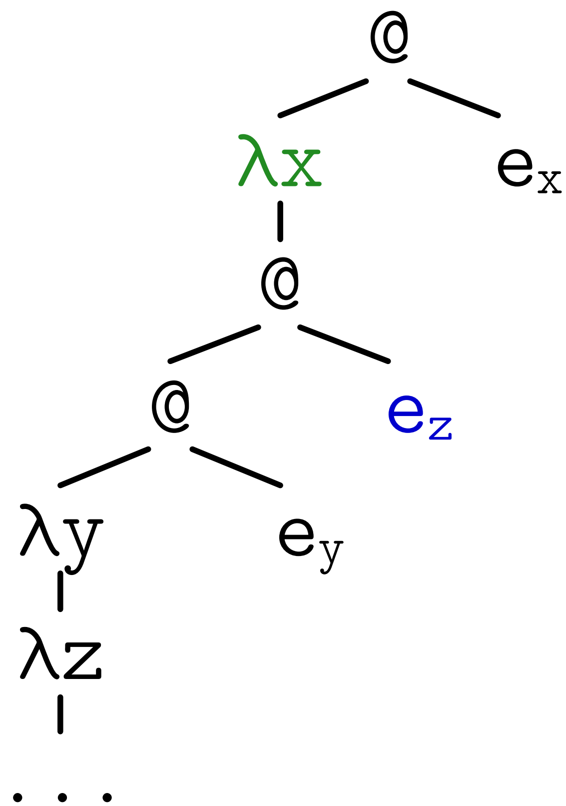
$\texttt{B = [ ] | (}\lambda\texttt{x.B) e}$

$((\lambda x.(\lambda y.\lambda z.\ \ldots)\ e_y)\ e_x)\ e_z$ ✗



OLD λ_{need}: RESHUFFLING OF BINDINGS

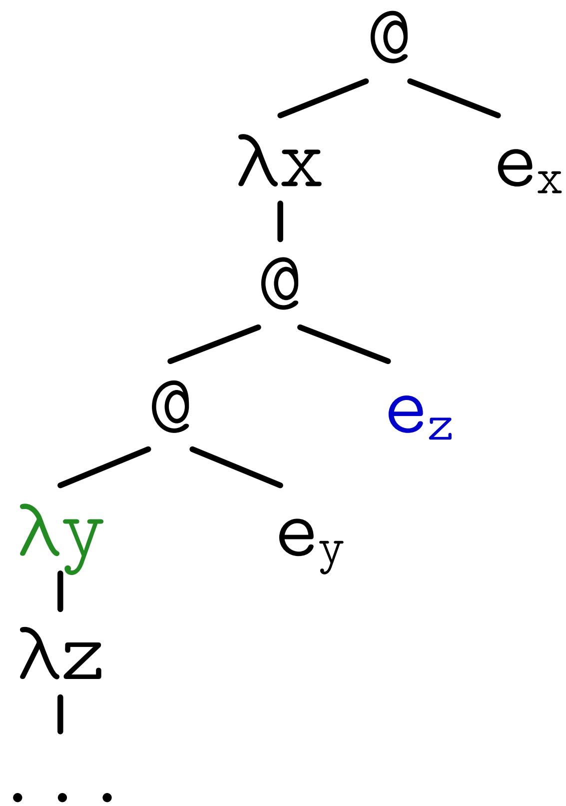
B = [ ] | (λx.B) e



Old λ_{need}: Reshuffling of Bindings

`B = [ ] | (λx.B) e`

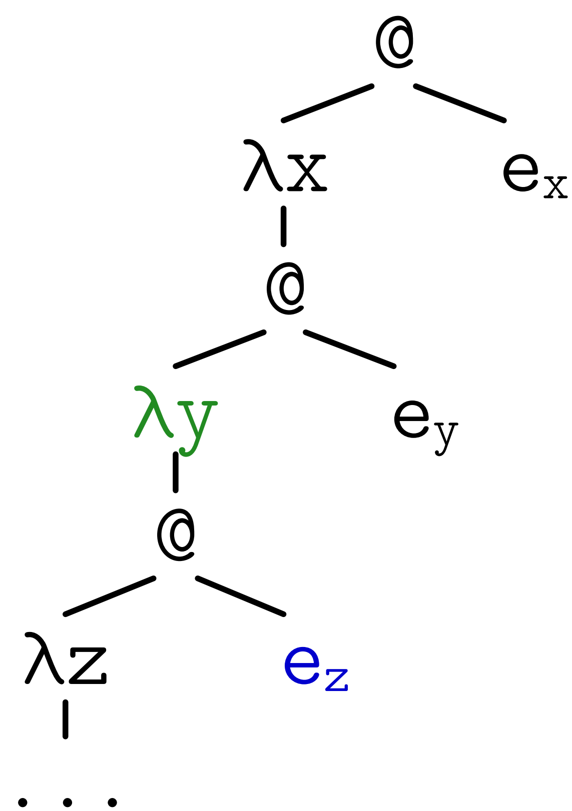
$(\lambda x.((\lambda y.\lambda z.\ \ldots)\ e_y)\ e_z)\ e_x$ ✗



OLD λ_{need}: RESHUFFLING OF BINDINGS

`B = [ ] | (λx.B) e`

$(\lambda x.(\lambda y.(\lambda z. \ \ldots) \ e_z) \ e_y) \ e_x$ ✔



Problems with Old Call-by-need Calculus

1) Reshuffling rules.

OLD $\lambda_{\texttt{need}}$: OPERATIONAL OVERVIEW

1) Find the next demanded variable.

2) Find its corresponding argument and evaluate it.

3) Substitute evaluated argument for demanded variable.

Old $\lambda_{\mathtt{need}}$: Operational Overview

1) Find the next demanded variable.

2) Find its corresponding argument and evaluate it.

3) Substitute evaluated argument for demanded variable.

OLD $\lambda_{\texttt{need}}$: OPERATIONAL OVERVIEW

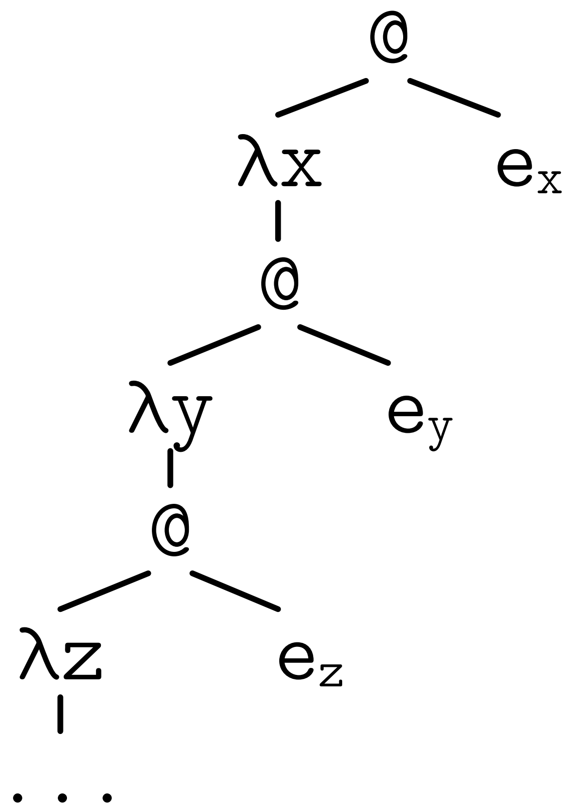
1) Find the next demanded variable.

2) Find its corresponding argument and evaluate it.

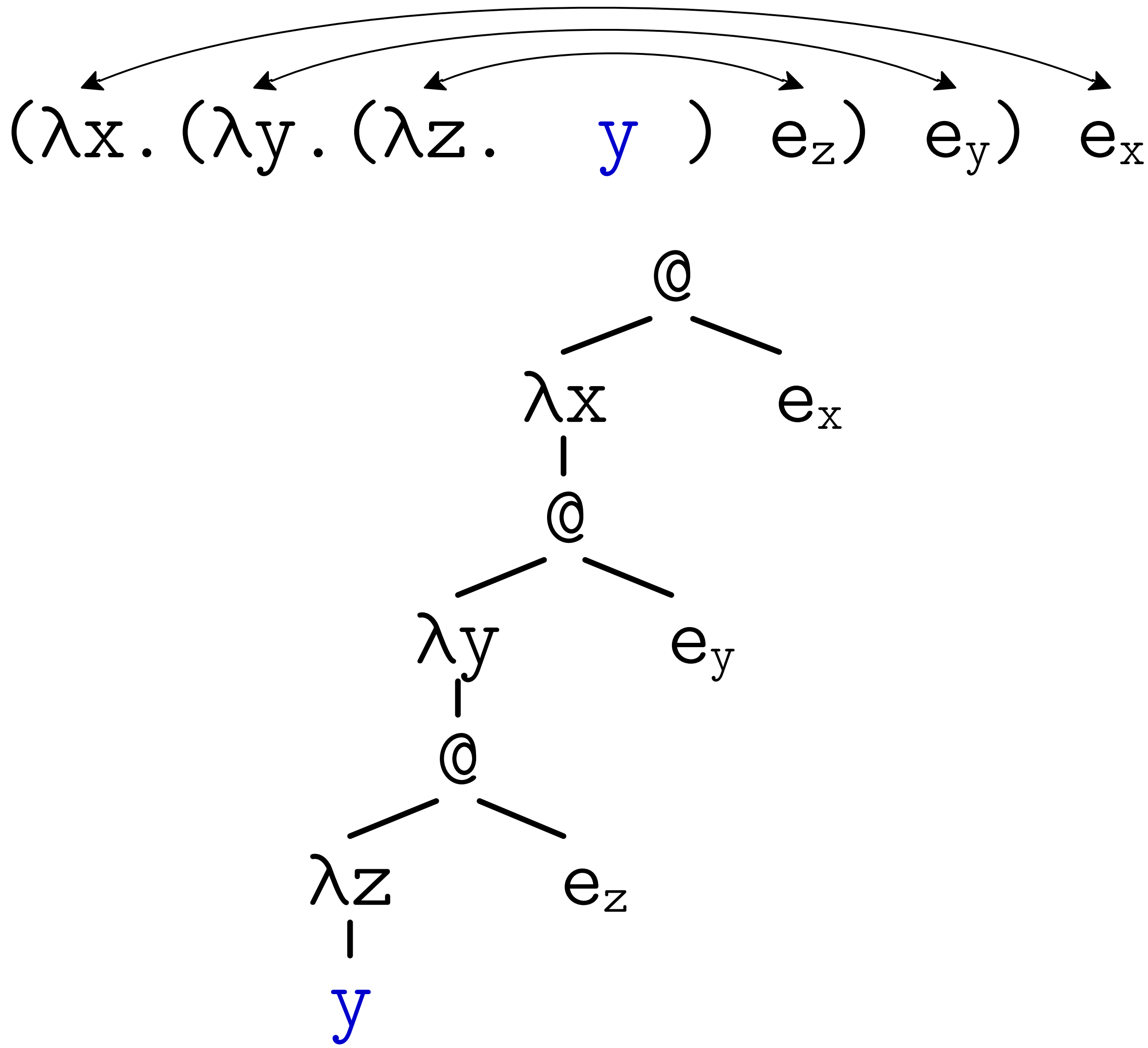
3) Substitute evaluated argument for demanded variable.

OLD λ_{need}: DEREFERENCING

$$(\lambda x.(\lambda y.(\lambda z.\ \ldots)\ e_z)\ e_y)\ e_x$$

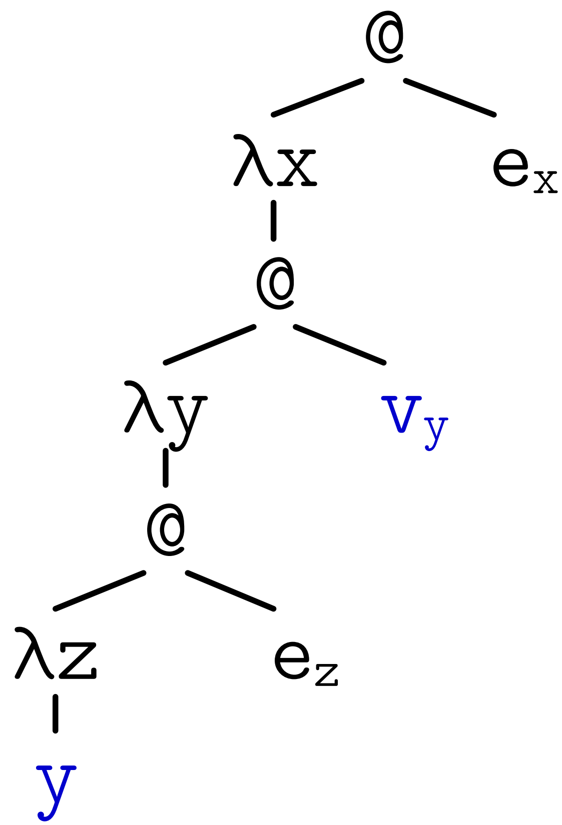


OLD $\lambda_{\texttt{need}}$: DEREFERENCING

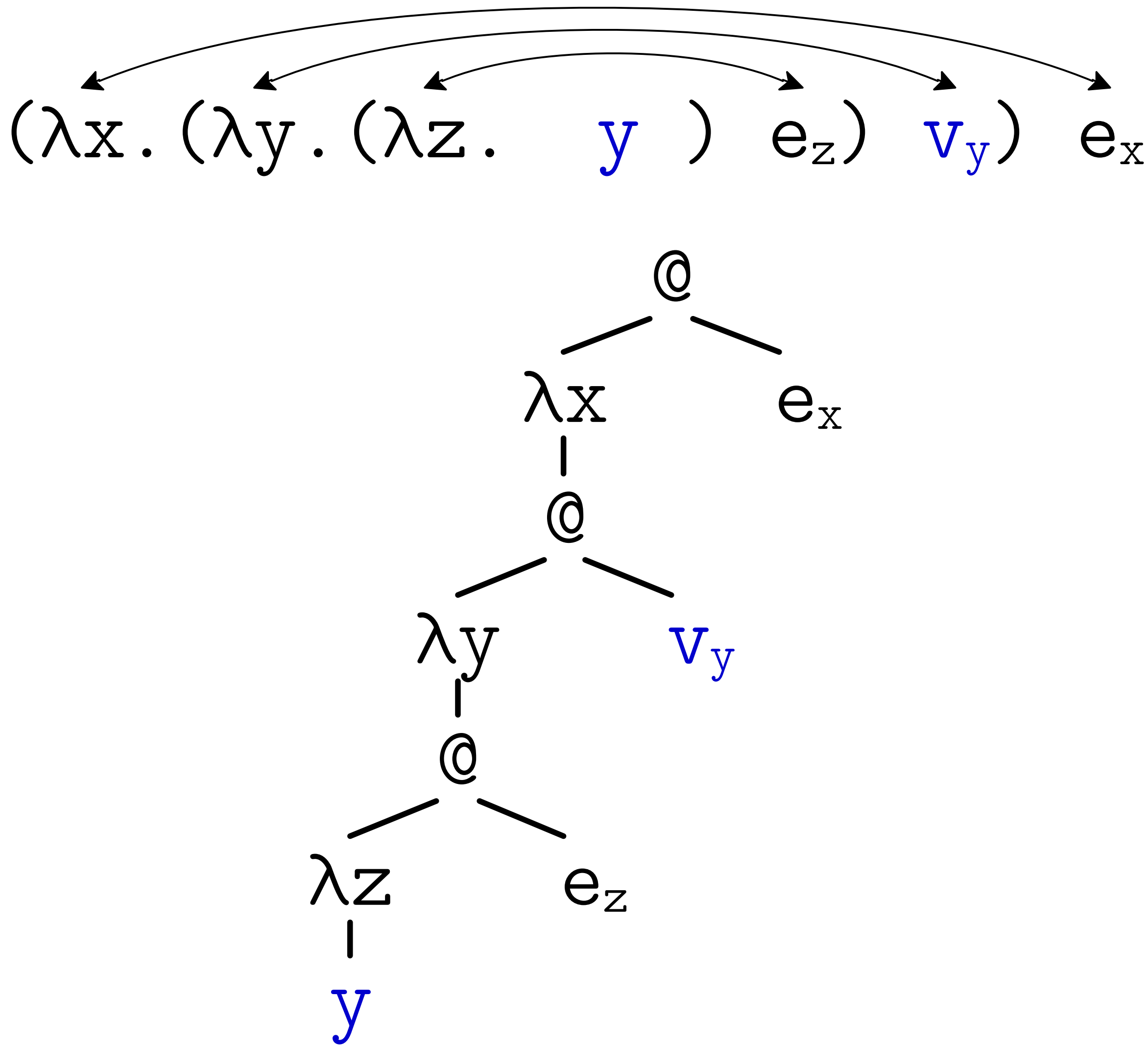


OLD $\lambda_{\texttt{need}}$: DEREFERENCING

$$(\lambda x.(\lambda y.(\lambda z.\ {\color{blue}y}\ )\ e_z)\ {\color{blue}v_y})\ e_x$$

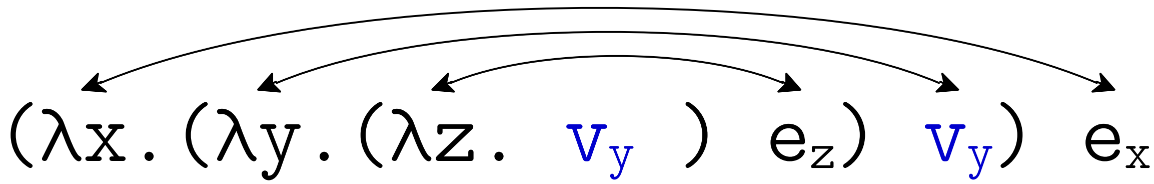


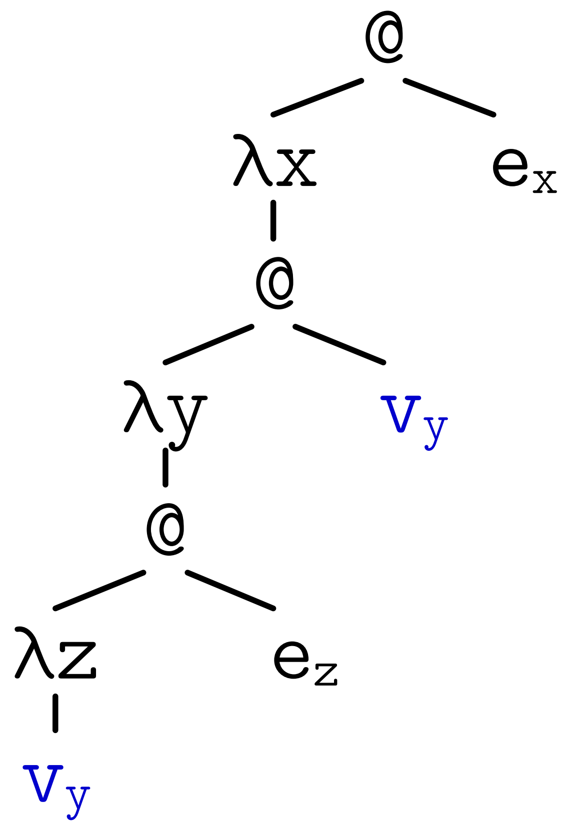
OLD $\lambda_{\texttt{need}}$: DEREFERENCING



$$\boxed{(\lambda y.D[y])\ v \rightarrow (\lambda y.D[v])\ v} \qquad (\texttt{deref})$$

OLD $\lambda_{\texttt{need}}$: DEREFERENCING



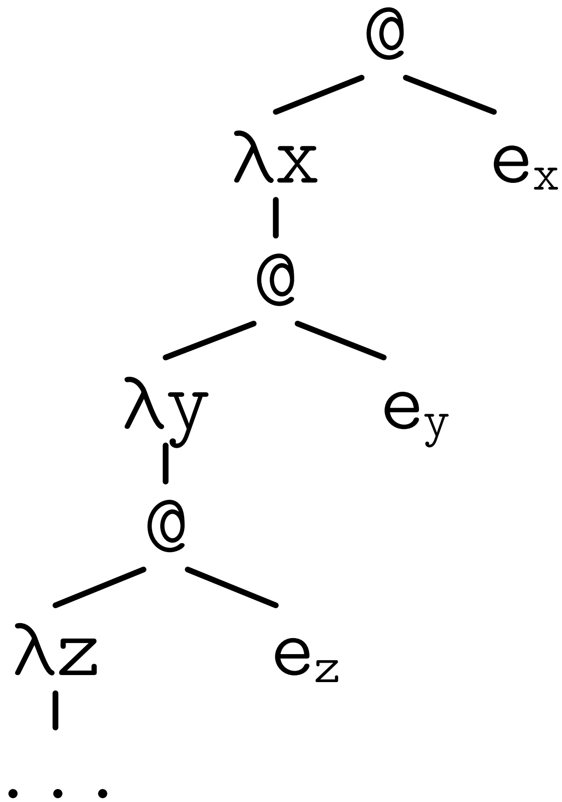


$$(\lambda y.D[y])\ v \rightarrow (\lambda y.D[v])\ v \qquad \texttt{(deref)}$$

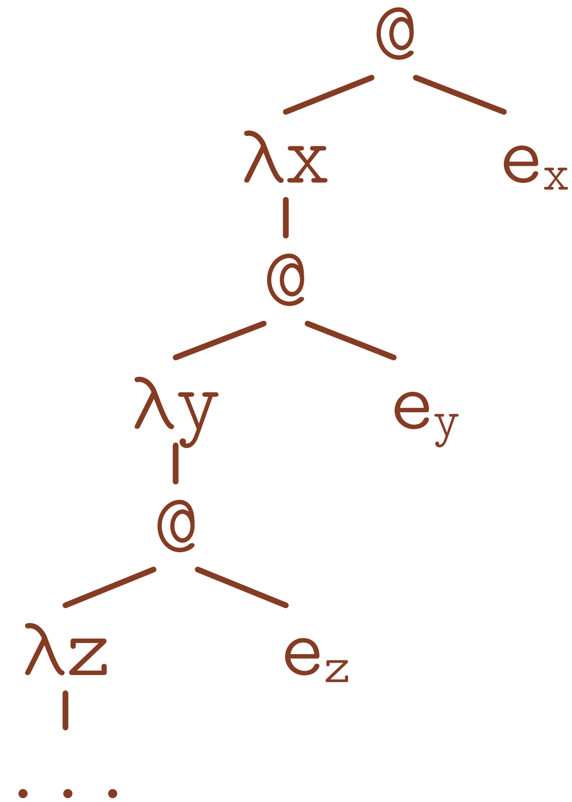
Problems with Old Call-by-need Calculus

1) Reshuffling rules.
2) Arguments and applications never go away.

NEW $\lambda_{\texttt{need}}$: HANDLING ARBITRARY BINDING STRUCTURE

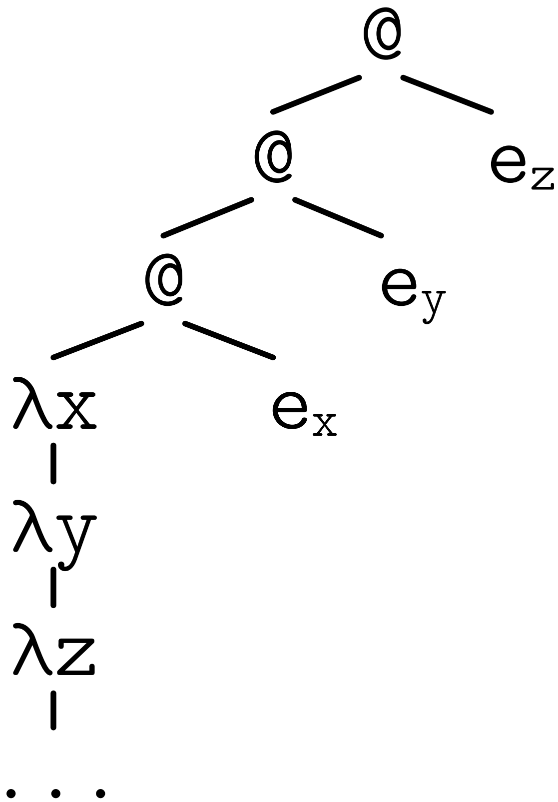


New λ_{need}: Handling Arbitrary Binding Structure



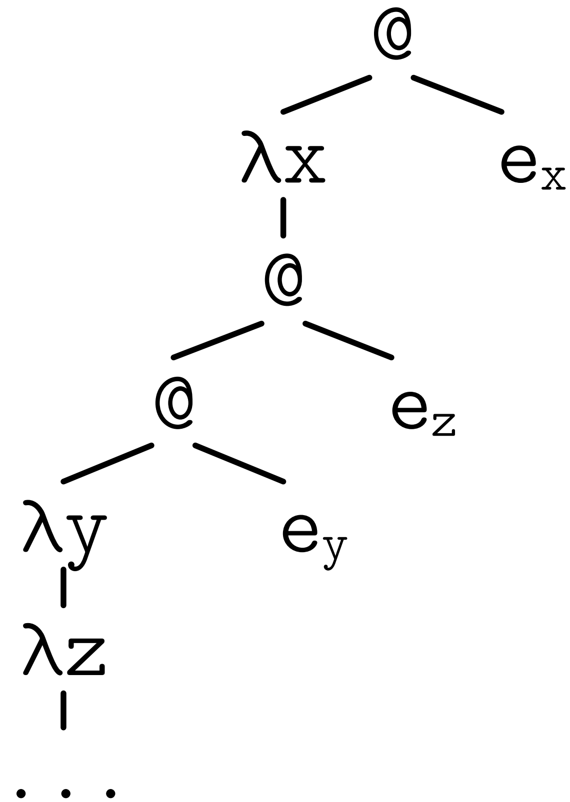
B = [ ] | (λx.B) e

New $\lambda_{\texttt{need}}$: Handling Arbitrary Binding Structure



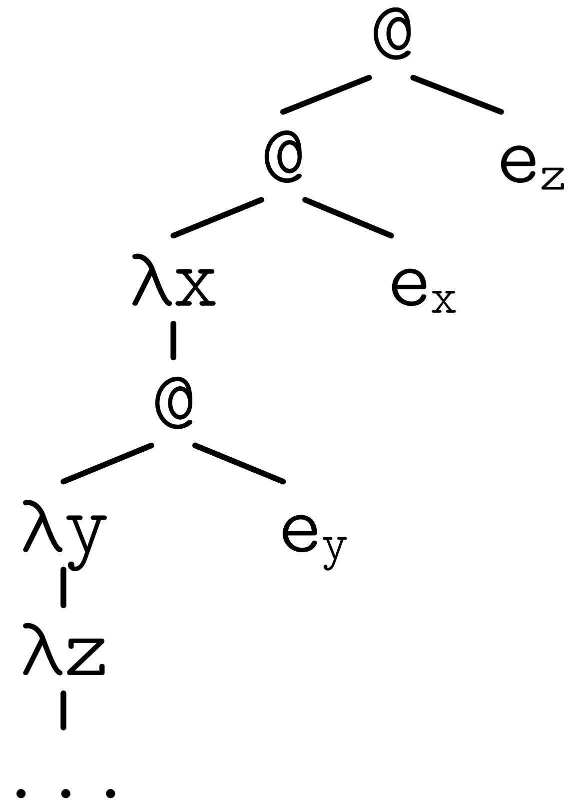
A = [ ] | ???

New λ_{need}: Handling Arbitrary Binding Structure



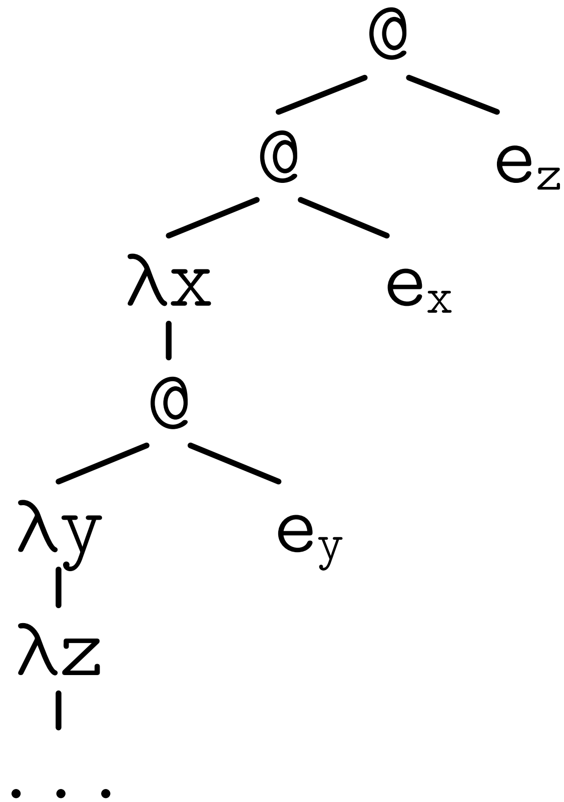
A = [ ] | ???

New λ_{need}: Handling Arbitrary Binding Structure



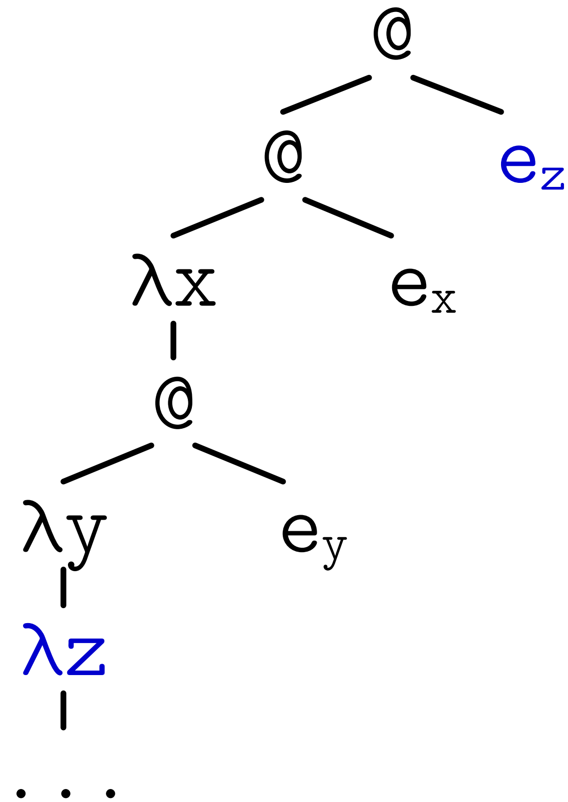
A = [ ] | ???

New λ_{need}: Handling Arbitrary Binding Structure



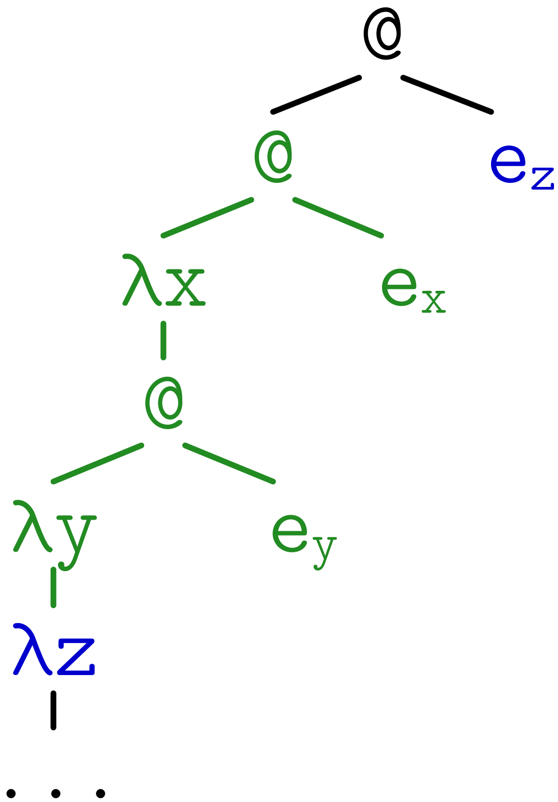
$$A = [\ ] \mid (\lambda x.A)\ e$$

New λ_{need}: Handling Arbitrary Binding Structure



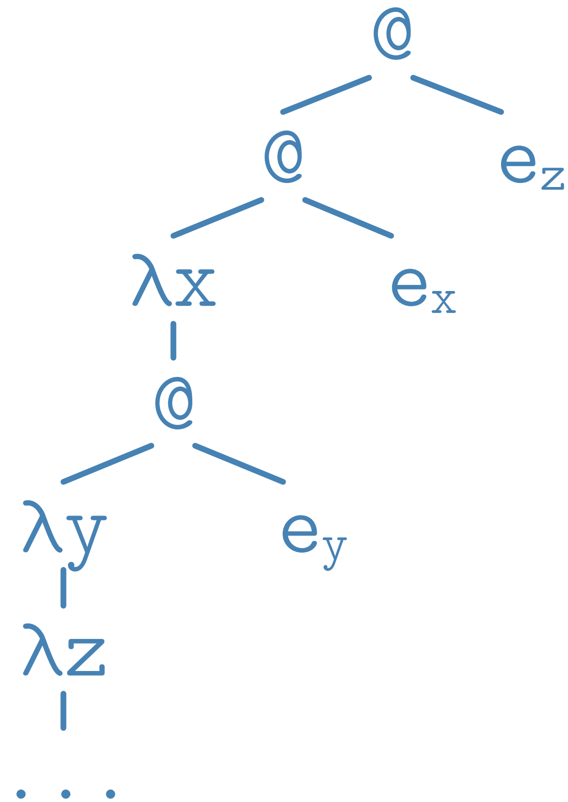
$$A = [\ ] \mid (\lambda x.A)\ e$$

New λ_{need}: Handling Arbitrary Binding Structure



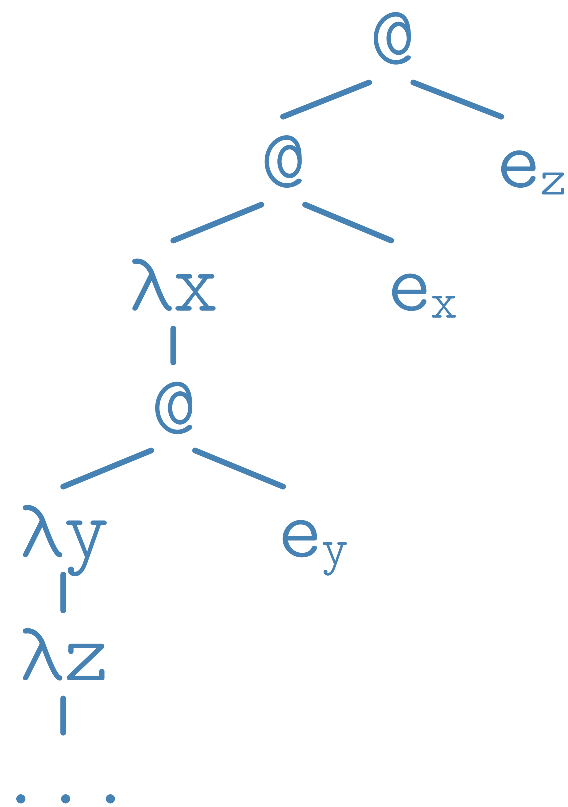
$$A = [\;] \mid (\lambda x.A)\ e$$

NEW $\lambda_{\mathtt{need}}$: HANDLING ARBITRARY BINDING STRUCTURE



$$A = [\ ]\ |\ A[(\lambda x.A)]\ e$$

New $\lambda_{\mathtt{need}}$: Handling Arbitrary Binding Structure

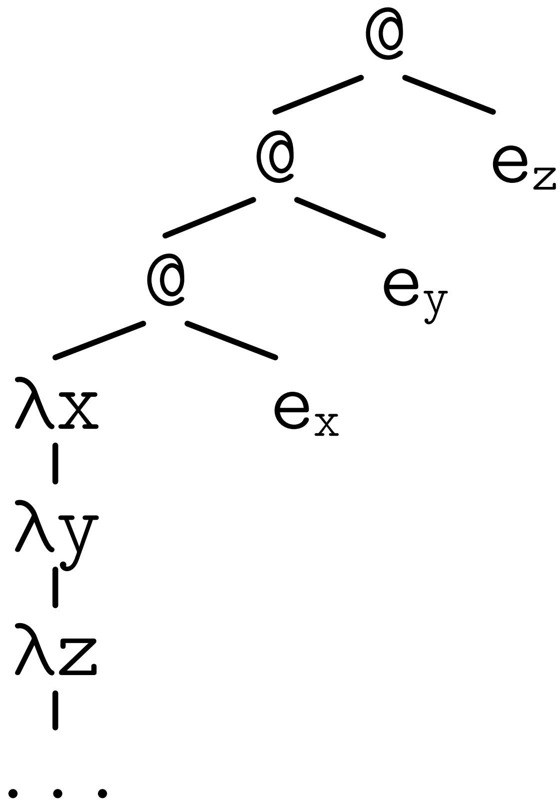


$$A = [\ ] \mid A[(\lambda x.A)]\ e$$
$$D = [\ ] \mid D\ e \mid A[D]$$

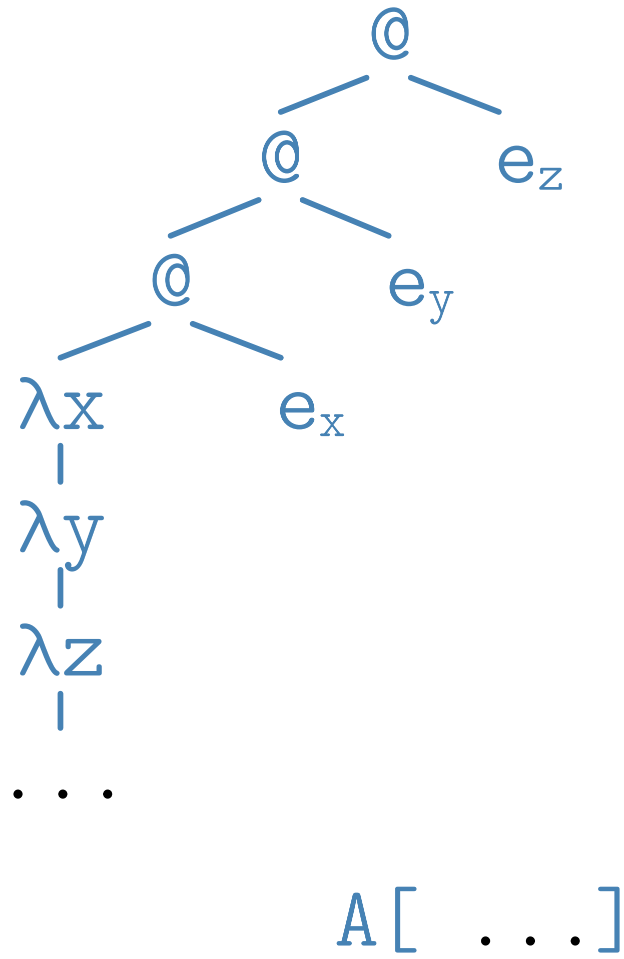
Problems with Old Call-by-need Calculus

~~1) Reshuffling rules.~~
2) Arguments and applications never go away.

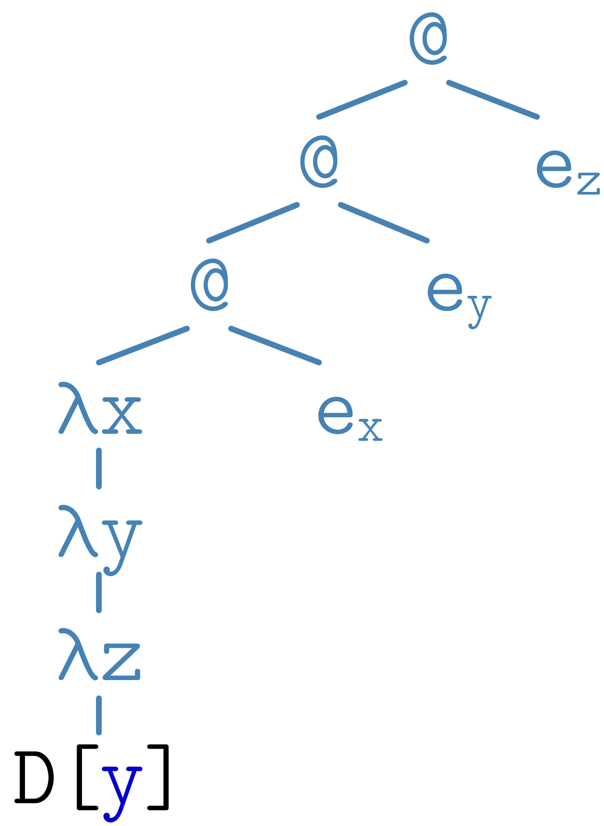
NEW $\lambda_{\texttt{need}}$: SPLITTING CONTEXTS



New λ_{need}: Splitting Contexts

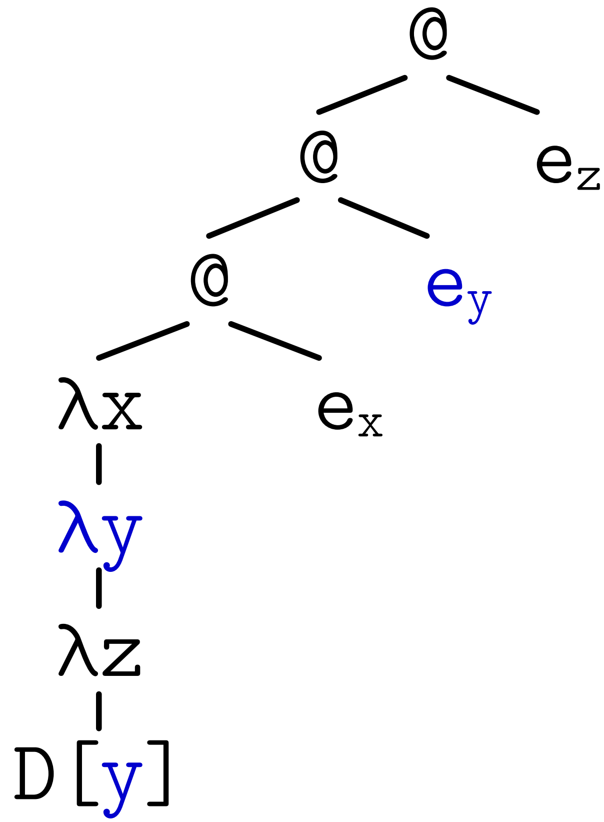


NEW λ_{need}: SPLITTING CONTEXTS



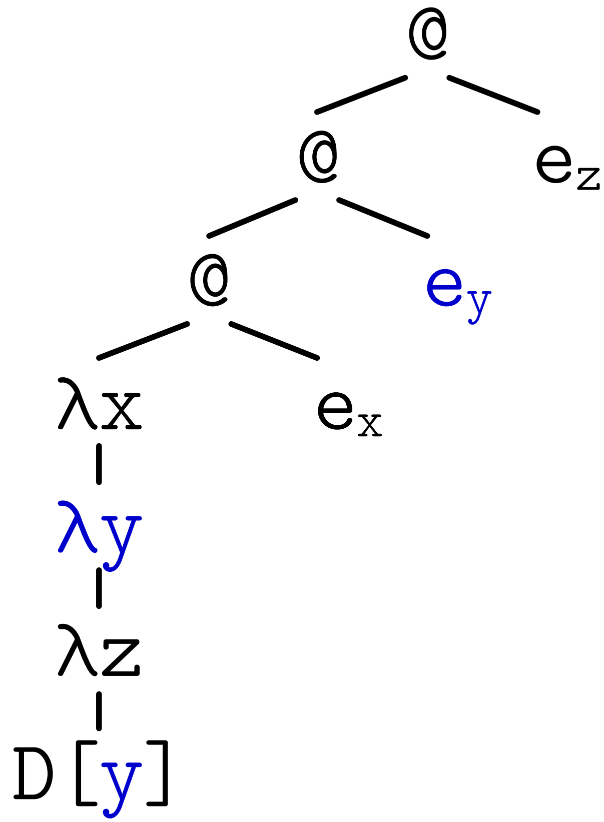
A[D[y]]

New λ_{need}: Splitting Contexts



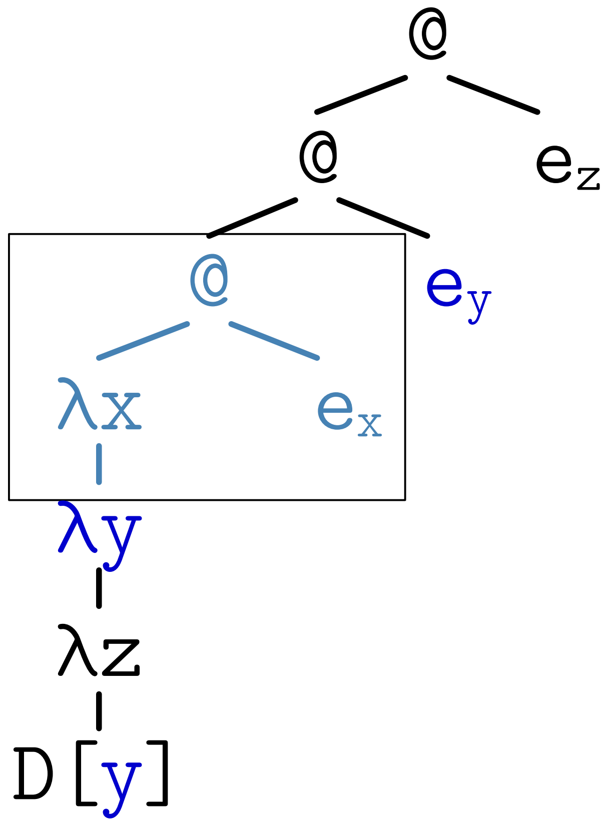
$$A[D[y]]$$

NEW λ_{need}: SPLITTING CONTEXTS



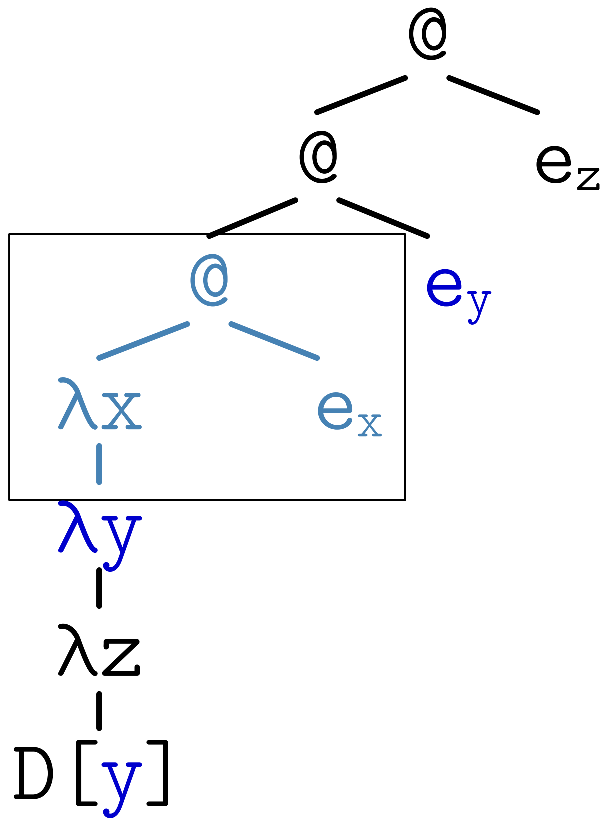
$$\ldots\ (\lambda y \ldots D[y]) \ldots e_y \ldots$$

NEW λ_{need}: SPLITTING CONTEXTS



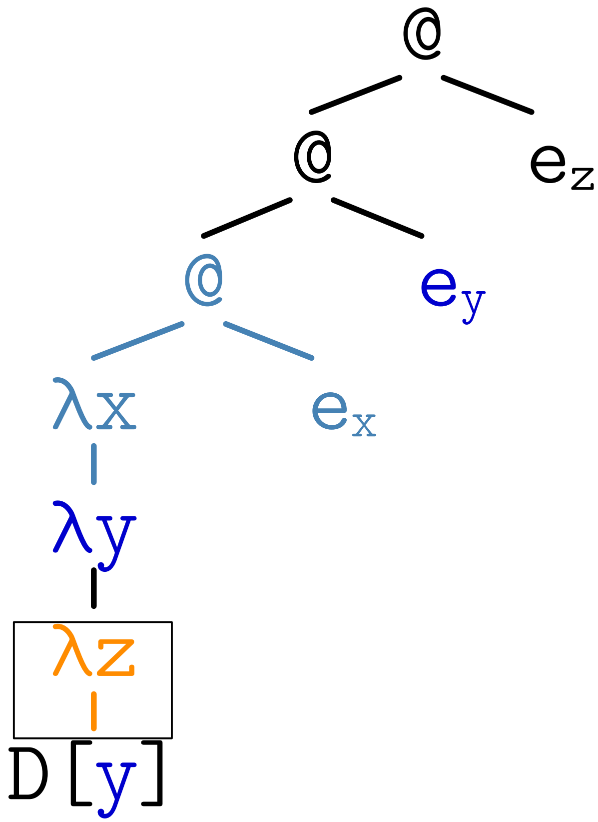
$$\ldots\ (\lambda y \ldots D[y]) \ldots e_y \ldots$$

NEW λ_{need}: SPLITTING CONTEXTS



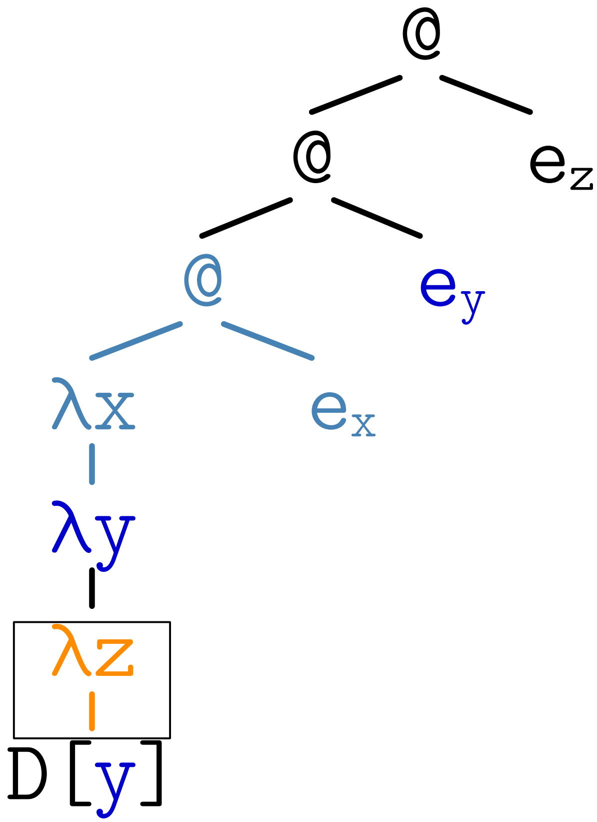
$\ldots A[\lambda y \ldots D[y]\ ]\ e_y\ \ldots$

New λ_{need}: Splitting Contexts



$$\ldots A[\lambda y \ldots D[y]\ ]\ e_y\ \ldots$$

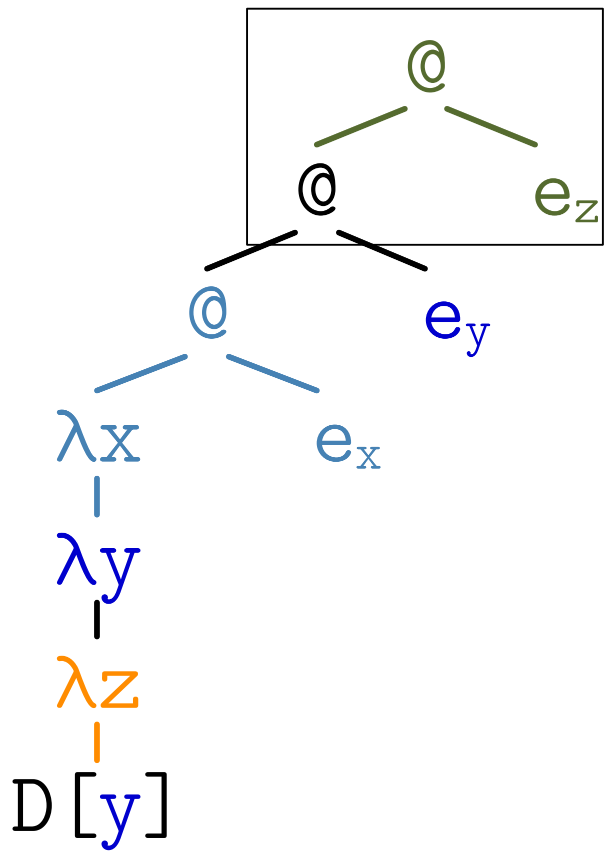
New λ_{need}: Splitting Contexts



$$\ldots A[\lambda y.\check{A}[D[y]]]\ e_y\ \ldots$$

$$\check{A} = [\ ]\ |\ A[\lambda x.\check{A}]$$

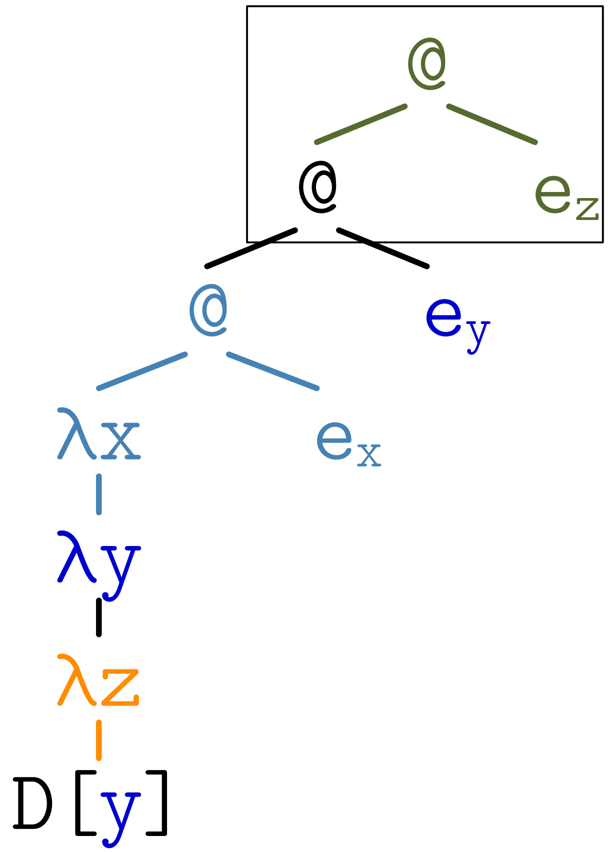
New λ_{need}: Splitting Contexts



$$\ldots A[\lambda y.\check{A}[D[y]]]\ e_y\ \ldots$$

$$\check{A} = [\ ]\ \mid\ A[\lambda x.\check{A}]$$

New λ_{need}: Splitting Contexts

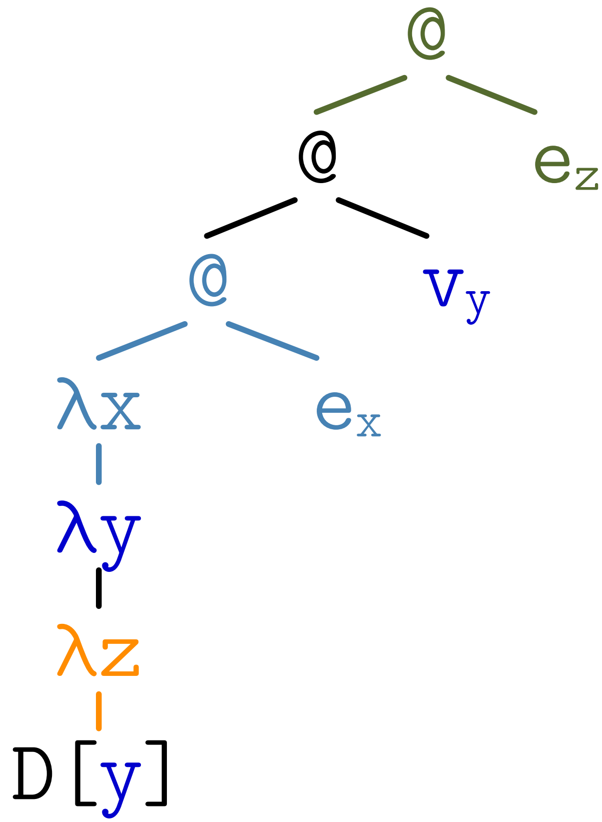


$$\hat{A}[A[\lambda y.\check{A}[D[y]]]\ e_y]$$

$$\check{A} = [\ ] \mid A[\lambda x.\check{A}]$$

$$\hat{A} = [\ ] \mid A[\hat{A}]\ e$$

New λ_{need}: Splitting Contexts

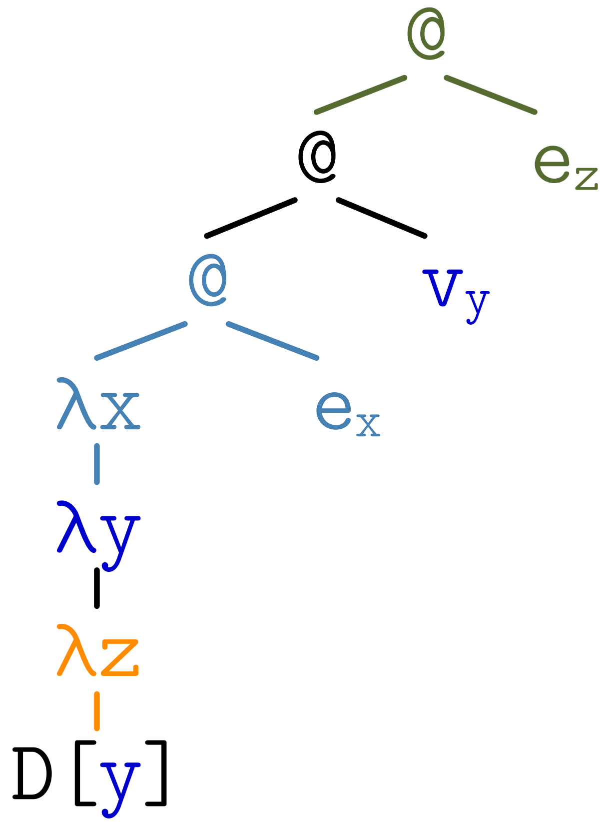


$$\hat{A}[A[\lambda y.\check{A}[D[y]]]\ v_y]$$

$$\check{A} = [\ ] \mid A[\lambda x.\check{A}]$$

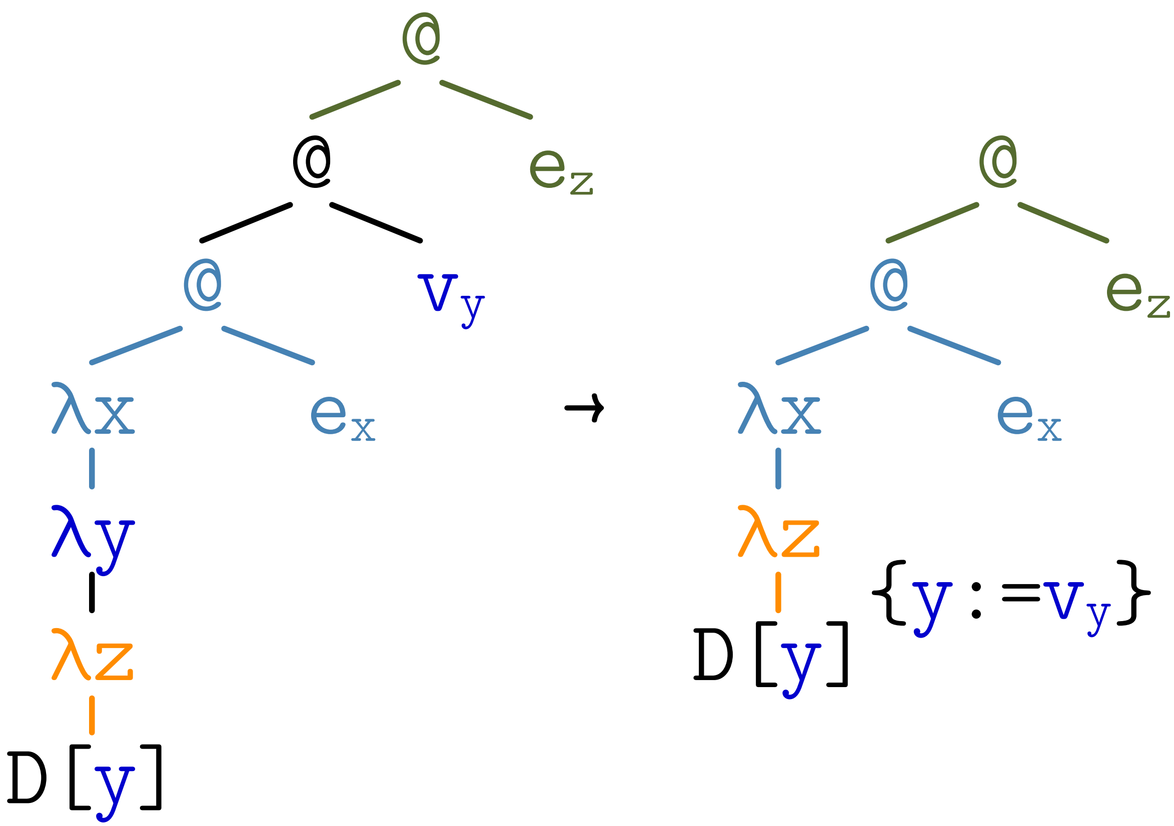
$$\hat{A} = [\ ] \mid A[\hat{A}]\ e$$

NEW λ_{need}: β_{need}



$$\hat{A}[A[\lambda y.\check{A}[D[y]]]\ \ v_y] \;\rightarrow\; \hat{A}[A[\check{A}[D[y]]\{y:=v_y\}]] \qquad (\beta_{need})$$

New $\lambda_{\mathtt{need}}$: $\beta_{\mathtt{need}}$



$$\hat{A}[A[\lambda y.\check{A}[D[y]]]\ \ v_y]$$
$$\rightarrow$$
$$\hat{A}[A[\check{A}[D[y]]\{y{:=}v_y\}]] \qquad (\beta_{\mathtt{need}})$$

Problems with Previous Call-by-need Calculus

~~1) Reassociation rules.~~
~~2) Function calls not resolved.~~

New $\lambda_{\texttt{need}}$: Evaluating Arguments

$$\texttt{D} = [\ ]\ |\ \texttt{D e}\ |\ A[\texttt{D}]$$

New $\lambda_{\mathtt{need}}$: Evaluating Arguments

$$\mathtt{D} = [\ ]\ |\ \mathtt{D}\ \mathtt{e}\ |\ \mathtt{A}[\mathtt{D}]\ |\ \hat{\mathtt{A}}[\mathtt{A}[\lambda \mathtt{y}.\check{\mathtt{A}}[\mathtt{D}[\mathtt{y}]]]\ \mathtt{D}]$$

Other interesting things in the paper . . .

- Correspondence to Launchbury's (1993) machine semantics.

OTHER INTERESTING THINGS IN THE PAPER . . .

- Correspondence to Launchbury's (1993) machine semantics.
- Confluence, Standardization properties.

Other interesting things in the paper . . .

- Correspondence to Launchbury's (1993) machine semantics.
- Confluence, Standardization properties.
- Soundness with respect to observational equivalence.

Thanks!