Divide-and-Conquer Checkpointing for Arbitrary Programs with No User Annotation

Jeffrey Mark Siskind, qobi@purdue.edu

Purdue University

NIPS 2017 Workshop on
The Future of Gradient-Based Machine Learning Software & Techniques
Saturday 9 December 2017

Joint work with Barak Avrum Pearlmutter

Barak and my Work

Barak and my Work

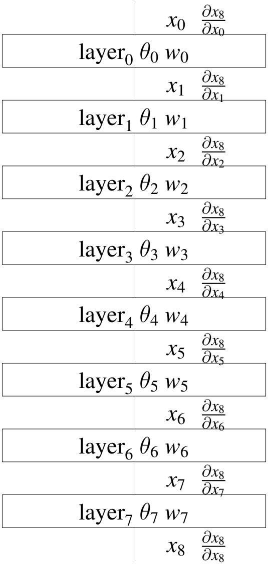
AD in functional programs.

Barak and my Work

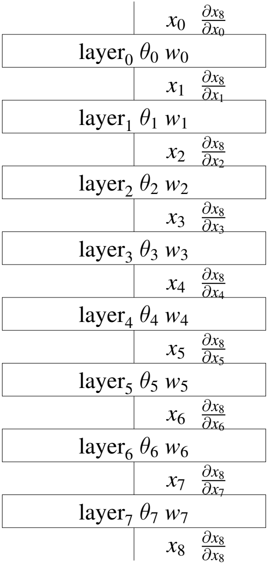
AD in functional programs.

AD is easier in functional programs.

A Neural Network



A Neural Network is a (Functional) Program



$$
\begin{array}{l}
\text{net}\ [\theta_0,\ldots,\theta_7]\ [w_0,\ldots,w_7]\ x_0 \triangleq \\
\quad \textbf{let}\ \begin{array}{rcl}
x_1 & = & \text{layer}_0\ \theta_0\ w_0\ x_0 \\
x_2 & = & \text{layer}_1\ \theta_1\ w_1\ x_1 \\
x_3 & = & \text{layer}_2\ \theta_2\ w_2\ x_2 \\
x_4 & = & \text{layer}_3\ \theta_3\ w_3\ x_3 \\
x_5 & = & \text{layer}_4\ \theta_4\ w_4\ x_4 \\
x_6 & = & \text{layer}_5\ \theta_5\ w_5\ x_5 \\
x_7 & = & \text{layer}_6\ \theta_6\ w_6\ x_6 \\
x_8 & = & \text{layer}_7\ \theta_7\ w_7\ x_7
\end{array} \\
\quad \textbf{in}\ x_8
\end{array}
$$

A (Functional) Program

$$
\begin{aligned}
& f\ [w_0, w_1]\ [x_0, x_1] \triangleq \\
& \quad \textbf{let}\ \ t_0 = w_0 \times x_0 \\
& \qquad\quad\ t_1 = w_1 \times x_1 \\
& \qquad\quad\ \ y = t_0 + t_1 \\
& \quad \textbf{in}\ y
\end{aligned}
$$

A (Functional) Program is a (Neural) Network

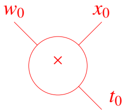
$$
\begin{array}{l}
f\ [w_0, w_1]\ [x_0, x_1] \triangleq \\
\quad \textbf{let}\ \begin{array}{rcl} t_0 & = & w_0 \times x_0 \\ t_1 & = & w_1 \times x_1 \\ y & = & t_0 + t_1 \end{array} \\
\quad \textbf{in}\ y
\end{array}
$$

A (Functional) Program is a (Neural) Network

$$f\ [w_0, w_1]\ [x_0, x_1] \triangleq$$

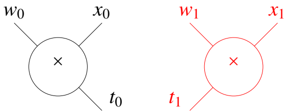
$$\begin{aligned}
\textbf{let}\quad t_0 &= w_0 \times x_0\\
t_1 &= w_1 \times x_1\\
y &= t_0 + t_1
\end{aligned}$$

$$\textbf{in}\ y$$



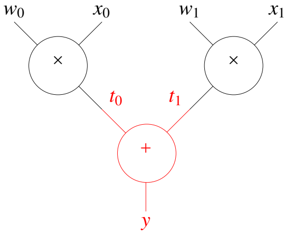
A (Functional) Program is a (Neural) Network

$$
\begin{aligned}
&f\ [w_0, w_1]\ [x_0, x_1] \triangleq\\
&\quad \textbf{let}\ t_0 = w_0 \times x_0\\
&\quad\quad\ \ t_1 = w_1 \times x_1\\
&\quad\quad\ \ y = t_0 + t_1\\
&\quad \textbf{in}\ y
\end{aligned}
$$



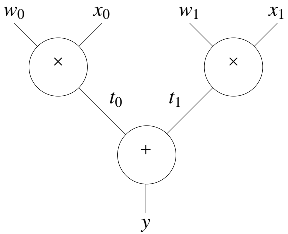
A (Functional) Program is a (Neural) Network

$$
\begin{aligned}
&f\ [w_0, w_1]\ [x_0, x_1] \triangleq \\
&\quad \textbf{let}\ \ t_0 = w_0 \times x_0 \\
&\qquad\quad\ t_1 = w_1 \times x_1 \\
&\qquad\quad\ y = t_0 + t_1 \\
&\quad \textbf{in}\ y
\end{aligned}
$$



A (Functional) Program is a (Neural) Network

$$f\ [w_0, w_1]\ [x_0, x_1] \triangleq$$
$$\textbf{let}\ \ t_0 = w_0 \times x_0$$
$$t_1 = w_1 \times x_1$$
$$y = t_0 + t_1$$
$$\textbf{in}\ y$$



Some Observations

Some Observations

- Deep learning network ‘frameworks’ are domain specific (functional) programming languages.

Some Observations

- Deep learning network 'frameworks' are domain specific (functional) programming languages.
- A deep neural network is a long running (functional) program.

Some Observations

- Deep learning network ‘frameworks’ are domain specific (functional) programming languages.
- A deep neural network is a long running (functional) program.
- Can perform backpropagation on (functional) programs

Some Observations

- Deep learning network ‘frameworks’ are domain specific (functional) programming languages.
- A deep neural network is a long running (functional) program.
- Can perform backpropagation on (functional) programs by having an execution of the program generate a network.

Some Observations

- Deep learning network 'frameworks' are domain specific (functional) programming languages.
- A deep neural network is a long running (functional) program.
- Can perform backpropagation on (functional) programs by having an execution of the program generate a network. This is called reverse-mode automatic differentiation (AD).

A (Brief) History of Backpropagation aka Reverse-Mode AD

A (Brief) History of Backpropagation aka Reverse-Mode AD

D.E. Rumelhart, G.E. Hinton, and R.J. Williams, *Learning representations by back-propagating errors*, Nature, 323:533–536, 1986.

A (Brief) History of Backpropagation aka Reverse-Mode AD

D.E. Rumelhart, G.E. Hinton, and R.J. Williams, *Learning representations by back-propagating errors*, Nature, 323:533–536, 1986.

B. Speelpenning, *Compiling Fast Partial Derivatives of Functions Given by Algorithms*, Department of Computer Science, University of Illinois at Urbana-Champaign, 1980.

A (Brief) History of Backpropagation aka Reverse-Mode AD

D.E. Rumelhart, G.E. Hinton, and R.J. Williams, *Learning representations by back-propagating errors*, Nature, 323:533–536, 1986.

B. Speelpenning, *Compiling Fast Partial Derivatives of Functions Given by Algorithms*, Department of Computer Science, University of Illinois at Urbana-Champaign, 1980.

P.J. Werbos, *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*, Harvard University, 1974.

A (Brief) History of Backpropagation aka Reverse-Mode AD

D.E. Rumelhart, G.E. Hinton, and R.J. Williams, *Learning representations by back-propagating errors*, Nature, 323:533–536, 1986.

B. Speelpenning, *Compiling Fast Partial Derivatives of Functions Given by Algorithms*, Department of Computer Science, University of Illinois at Urbana-Champaign, 1980.

P.J. Werbos, *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*, Harvard University, 1974.

G.M. Ostrovskii, Y.M. Volin, and W.W. Borisov, *Über die Berechnung von Ableitungen, Wissenschaftliche Zeitschrift der Technischen Hochschule für Chemie, Leuna-Merseburg*, 13(4):382–384, 1971.

A (Brief) History of Backpropagation aka Reverse-Mode AD

D.E. Rumelhart, G.E. Hinton, and R.J. Williams, *Learning representations by back-propagating errors*, Nature, 323:533–536, 1986.

B. Speelpenning, *Compiling Fast Partial Derivatives of Functions Given by Algorithms*, Department of Computer Science, University of Illinois at Urbana-Champaign, 1980.

P.J. Werbos, *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*, Harvard University, 1974.

G.M. Ostrovskii, Y.M. Volin, and W.W. Borisov, *Über die Berechnung von Ableitungen, Wissenschaftliche Zeitschrift der Technischen Hochschule für Chemie, Leuna-Merseburg*, 13(4):382–384, 1971.

S. Linnainmaa, *The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors* (in Finnish), Department of Computer Science, University of Helsinki, 1970.

A (Brief) History of Backpropagation aka Reverse-Mode AD

D.E. Rumelhart, G.E. Hinton, and R.J. Williams, *Learning representations by back-propagating errors*, Nature, 323:533–536, 1986.

B. Speelpenning, *Compiling Fast Partial Derivatives of Functions Given by Algorithms*, Department of Computer Science, University of Illinois at Urbana-Champaign, 1980.

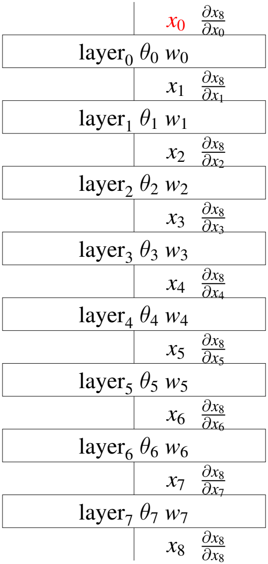
P.J. Werbos, *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*, Harvard University, 1974.

G.M. Ostrovskii, Y.M. Volin, and W.W. Borisov, *Über die Berechnung von Ableitungen, Wissenschaftliche Zeitschrift der Technischen Hochschule für Chemie, Leuna-Merseburg*, 13(4):382–384, 1971.

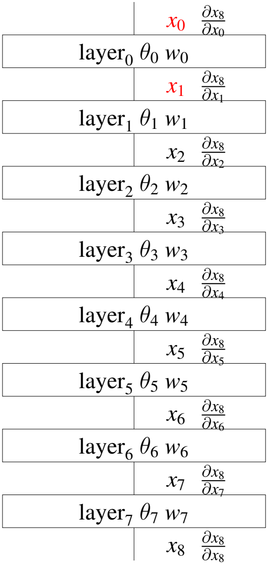
S. Linnainmaa, *The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors* (in Finnish), Department of Computer Science, University of Helsinki, 1970.

A.E. Bryson, Jr. and Y.-C. Ho, *Applied optimal control*, Blaisdell, 1969.

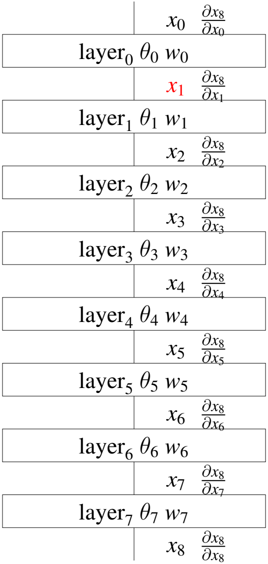
Evaluating a Neural Network



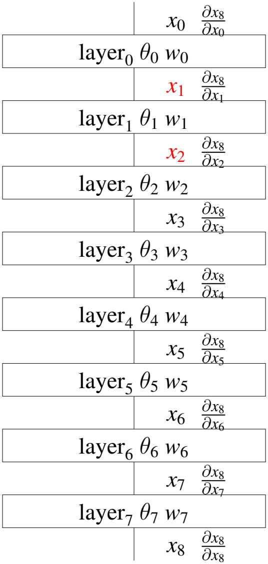
Evaluating a Neural Network



Evaluating a Neural Network



Evaluating a Neural Network



Evaluating a Neural Network

x_0 $\frac{\partial x_8}{\partial x_0}$

layer$_0$ θ_0 w_0

x_1 $\frac{\partial x_8}{\partial x_1}$

layer$_1$ θ_1 w_1

$\color{red}{x_2}$ $\frac{\partial x_8}{\partial x_2}$

layer$_2$ θ_2 w_2

x_3 $\frac{\partial x_8}{\partial x_3}$

layer$_3$ θ_3 w_3

x_4 $\frac{\partial x_8}{\partial x_4}$

layer$_4$ θ_4 w_4

x_5 $\frac{\partial x_8}{\partial x_5}$

layer$_5$ θ_5 w_5

x_6 $\frac{\partial x_8}{\partial x_6}$

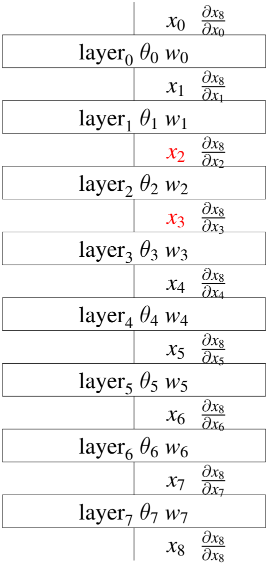
layer$_6$ θ_6 w_6

x_7 $\frac{\partial x_8}{\partial x_7}$

layer$_7$ θ_7 w_7

x_8 $\frac{\partial x_8}{\partial x_8}$

Evaluating a Neural Network



Evaluating a Neural Network

x_0 $\frac{\partial x_8}{\partial x_0}$

layer$_0$ θ_0 w_0

x_1 $\frac{\partial x_8}{\partial x_1}$

layer$_1$ θ_1 w_1

x_2 $\frac{\partial x_8}{\partial x_2}$

layer$_2$ θ_2 w_2

x_3 $\frac{\partial x_8}{\partial x_3}$

layer$_3$ θ_3 w_3

x_4 $\frac{\partial x_8}{\partial x_4}$

layer$_4$ θ_4 w_4

x_5 $\frac{\partial x_8}{\partial x_5}$

layer$_5$ θ_5 w_5

x_6 $\frac{\partial x_8}{\partial x_6}$

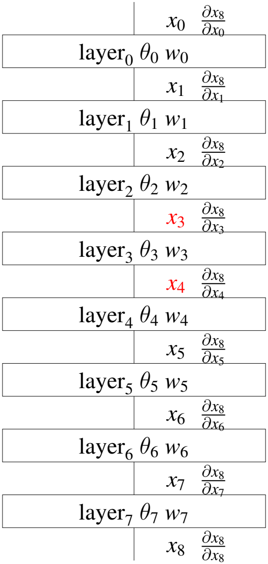
layer$_6$ θ_6 w_6

x_7 $\frac{\partial x_8}{\partial x_7}$

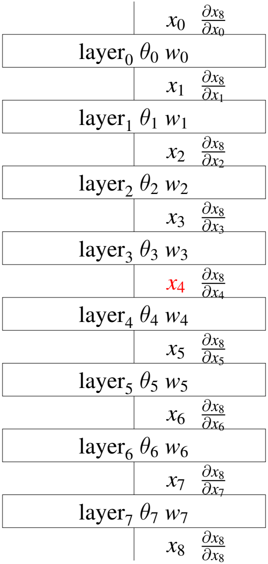
layer$_7$ θ_7 w_7

x_8 $\frac{\partial x_8}{\partial x_8}$

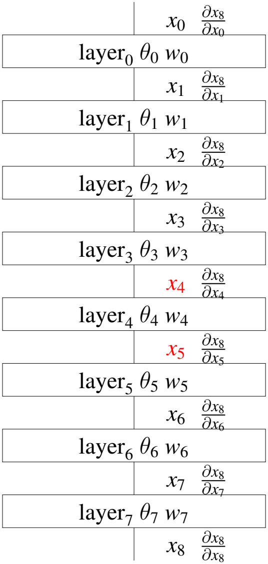
Evaluating a Neural Network



Evaluating a Neural Network



Evaluating a Neural Network



Evaluating a Neural Network

x_0 $\frac{\partial x_8}{\partial x_0}$

layer$_0$ θ_0 w_0

x_1 $\frac{\partial x_8}{\partial x_1}$

layer$_1$ θ_1 w_1

x_2 $\frac{\partial x_8}{\partial x_2}$

layer$_2$ θ_2 w_2

x_3 $\frac{\partial x_8}{\partial x_3}$

layer$_3$ θ_3 w_3

x_4 $\frac{\partial x_8}{\partial x_4}$

layer$_4$ θ_4 w_4

x_5 $\frac{\partial x_8}{\partial x_5}$

layer$_5$ θ_5 w_5

x_6 $\frac{\partial x_8}{\partial x_6}$

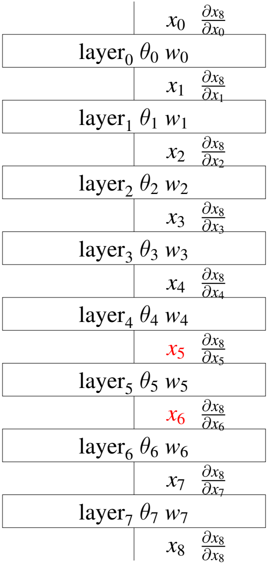
layer$_6$ θ_6 w_6

x_7 $\frac{\partial x_8}{\partial x_7}$

layer$_7$ θ_7 w_7

x_8 $\frac{\partial x_8}{\partial x_8}$

Evaluating a Neural Network



Evaluating a Neural Network

$x_0 \quad \frac{\partial x_8}{\partial x_0}$

layer$_0$ θ_0 w_0

$x_1 \quad \frac{\partial x_8}{\partial x_1}$

layer$_1$ θ_1 w_1

$x_2 \quad \frac{\partial x_8}{\partial x_2}$

layer$_2$ θ_2 w_2

$x_3 \quad \frac{\partial x_8}{\partial x_3}$

layer$_3$ θ_3 w_3

$x_4 \quad \frac{\partial x_8}{\partial x_4}$

layer$_4$ θ_4 w_4

$x_5 \quad \frac{\partial x_8}{\partial x_5}$

layer$_5$ θ_5 w_5

$x_6 \quad \frac{\partial x_8}{\partial x_6}$

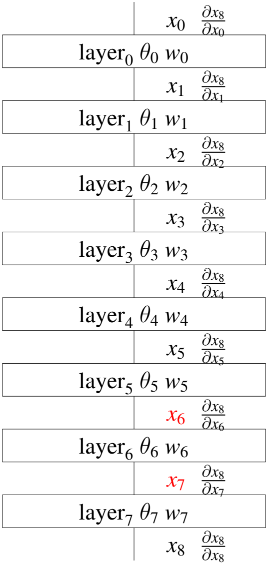
layer$_6$ θ_6 w_6

$x_7 \quad \frac{\partial x_8}{\partial x_7}$

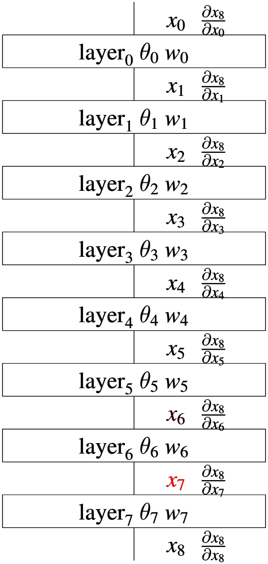
layer$_7$ θ_7 w_7

$x_8 \quad \frac{\partial x_8}{\partial x_8}$

Evaluating a Neural Network



Evaluating a Neural Network



Evaluating a Neural Network

x_0 $\frac{\partial x_8}{\partial x_0}$

layer$_0$ θ_0 w_0

x_1 $\frac{\partial x_8}{\partial x_1}$

layer$_1$ θ_1 w_1

x_2 $\frac{\partial x_8}{\partial x_2}$

layer$_2$ θ_2 w_2

x_3 $\frac{\partial x_8}{\partial x_3}$

layer$_3$ θ_3 w_3

x_4 $\frac{\partial x_8}{\partial x_4}$

layer$_4$ θ_4 w_4

x_5 $\frac{\partial x_8}{\partial x_5}$

layer$_5$ θ_5 w_5

x_6 $\frac{\partial x_8}{\partial x_6}$

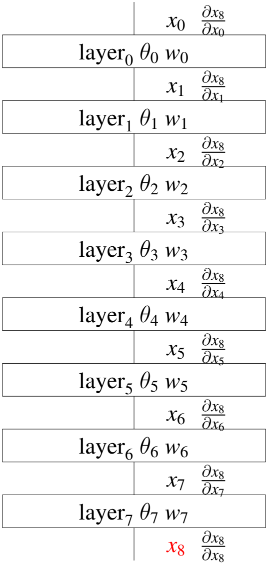
layer$_6$ θ_6 w_6

x_7 $\frac{\partial x_8}{\partial x_7}$

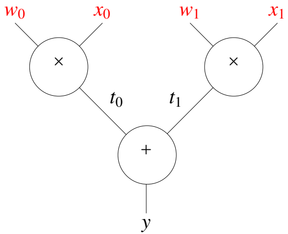
layer$_7$ θ_7 w_7

x_8 $\frac{\partial x_8}{\partial x_8}$

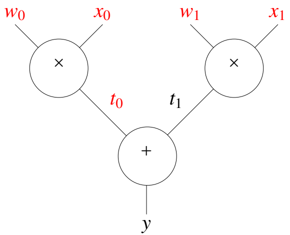
Evaluating a Neural Network



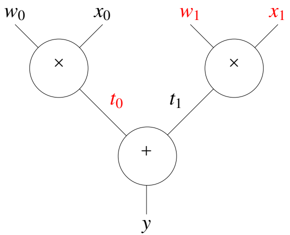
Evaluating a Network



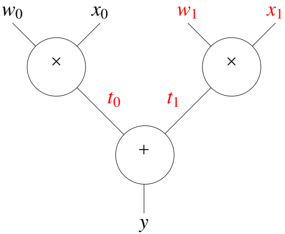
Evaluating a Network



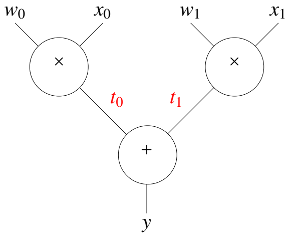
Evaluating a Network



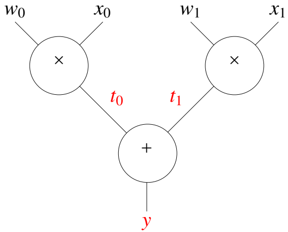
Evaluating a Network



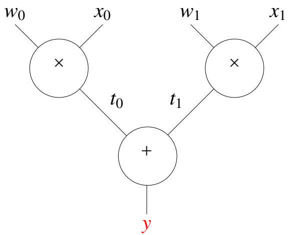
Evaluating a Network



Evaluating a Network



Evaluating a Network



Some Observations

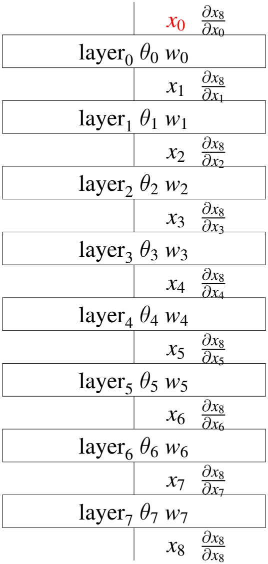
Some Observations

- Only need to store live variables.

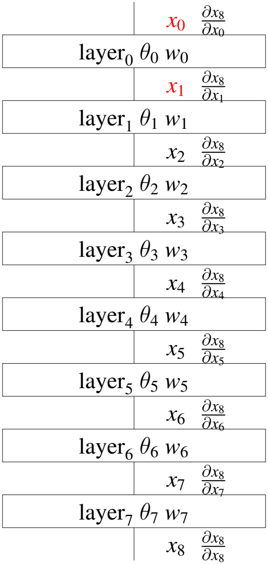
Some Observations

- Only need to store live variables.
- Most deep learning frameworks store all intermediate variables to allow subsequent backpropagation.

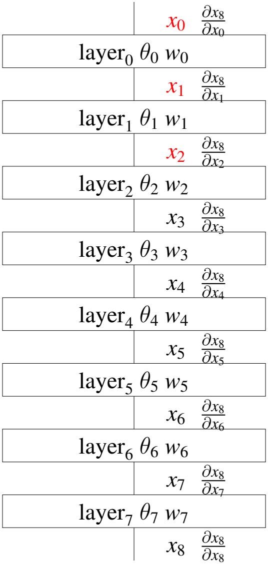
Evaluating a Neural Network to Allow Subsequent Backpropagation



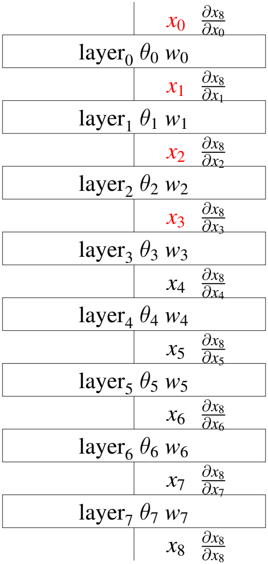
Evaluating a Neural Network to Allow Subsequent Backpropagation



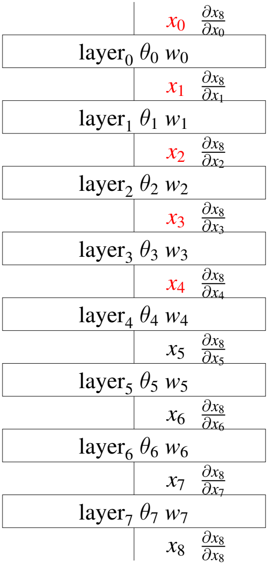
Evaluating a Neural Network to Allow Subsequent Backpropagation



Evaluating a Neural Network to Allow Subsequent Backpropagation



Evaluating a Neural Network to Allow Subsequent Backpropagation



Evaluating a Neural Network to Allow Subsequent Backpropagation

x_0 $\frac{\partial x_8}{\partial x_0}$

layer$_0$ θ_0 w_0

x_1 $\frac{\partial x_8}{\partial x_1}$

layer$_1$ θ_1 w_1

x_2 $\frac{\partial x_8}{\partial x_2}$

layer$_2$ θ_2 w_2

x_3 $\frac{\partial x_8}{\partial x_3}$

layer$_3$ θ_3 w_3

x_4 $\frac{\partial x_8}{\partial x_4}$

layer$_4$ θ_4 w_4

x_5 $\frac{\partial x_8}{\partial x_5}$

layer$_5$ θ_5 w_5

x_6 $\frac{\partial x_8}{\partial x_6}$

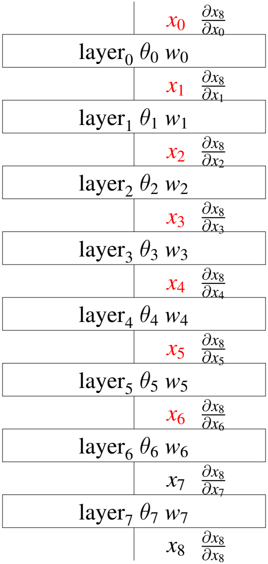
layer$_6$ θ_6 w_6

x_7 $\frac{\partial x_8}{\partial x_7}$

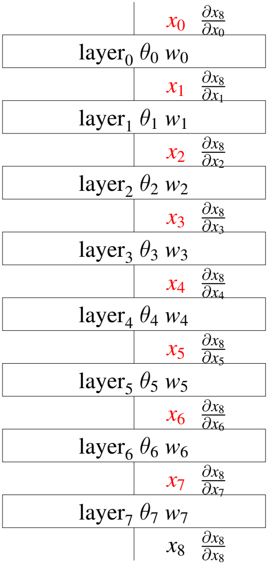
layer$_7$ θ_7 w_7

x_8 $\frac{\partial x_8}{\partial x_8}$

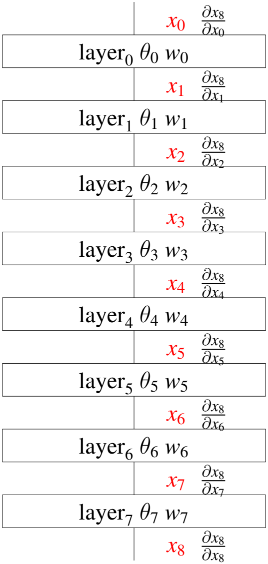
Evaluating a Neural Network to Allow Subsequent Backpropagation



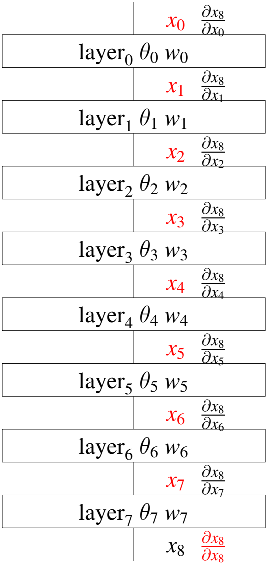
Evaluating a Neural Network to Allow Subsequent Backpropagation



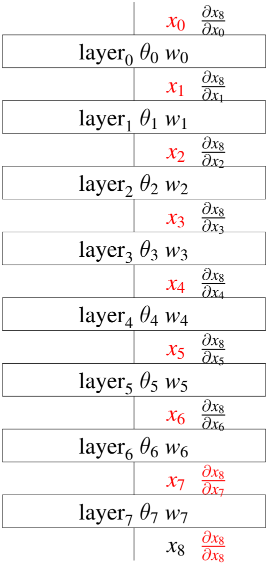
Evaluating a Neural Network to Allow Subsequent Backpropagation



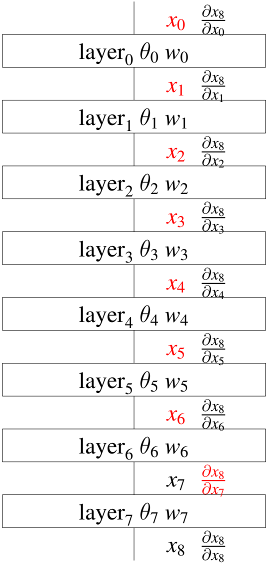
Evaluating a Neural Network to Allow Subsequent Backpropagation



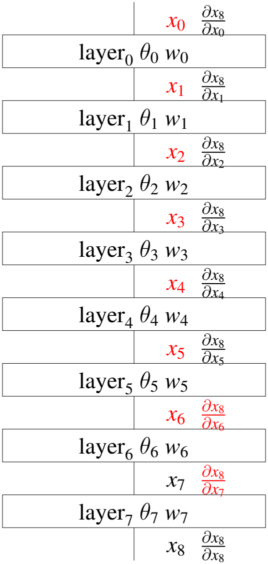
Evaluating a Neural Network to Allow Subsequent Backpropagation



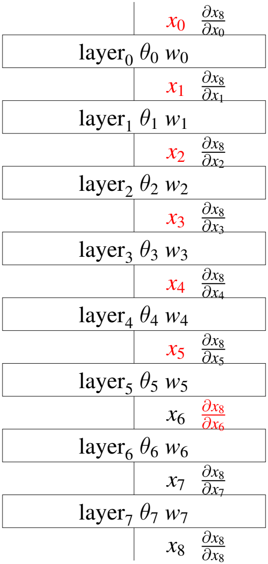
Evaluating a Neural Network to Allow Subsequent Backpropagation



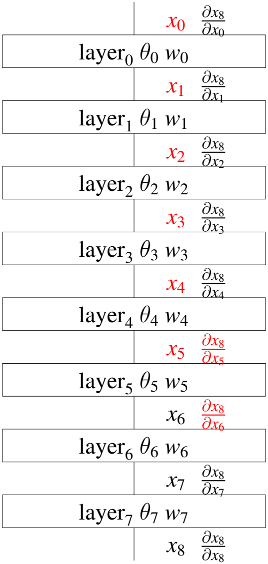
Evaluating a Neural Network to Allow Subsequent Backpropagation



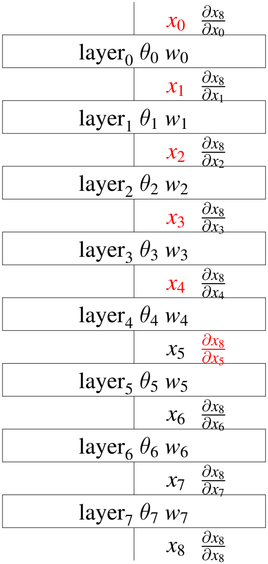
Evaluating a Neural Network to Allow Subsequent Backpropagation



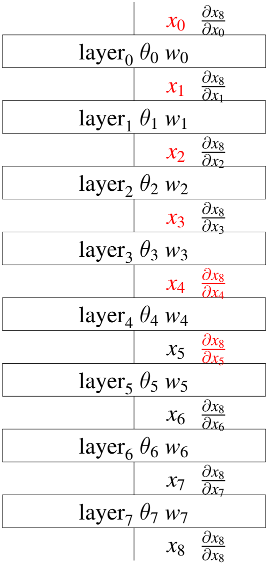
Evaluating a Neural Network to Allow Subsequent Backpropagation



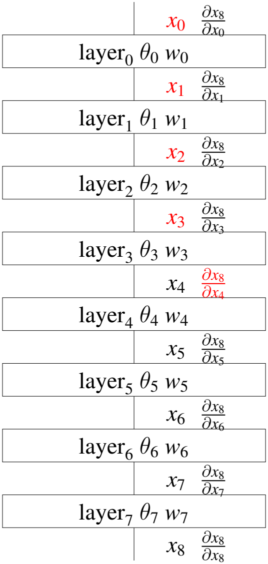
Evaluating a Neural Network to Allow Subsequent Backpropagation



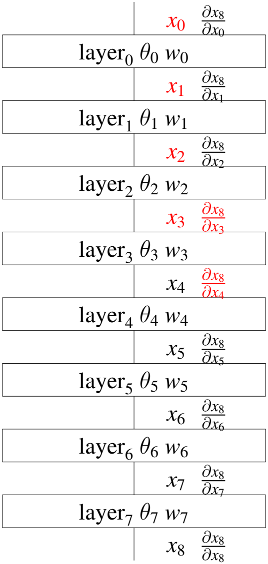
Evaluating a Neural Network to Allow Subsequent Backpropagation



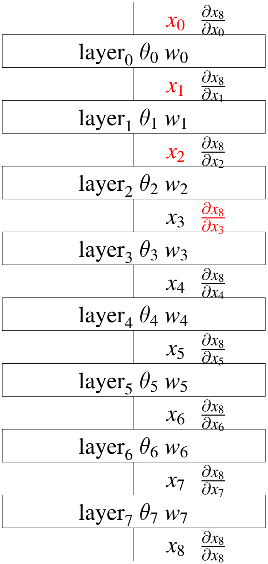
Evaluating a Neural Network to Allow Subsequent Backpropagation



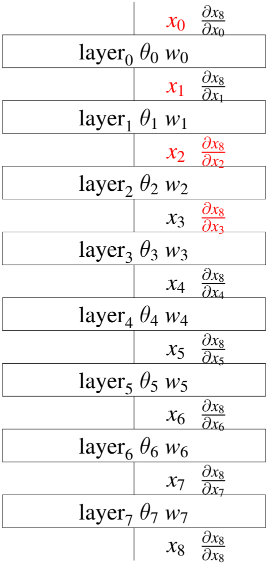
Evaluating a Neural Network to Allow Subsequent Backpropagation



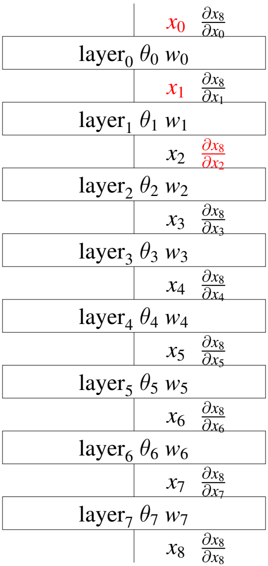
Evaluating a Neural Network to Allow Subsequent Backpropagation



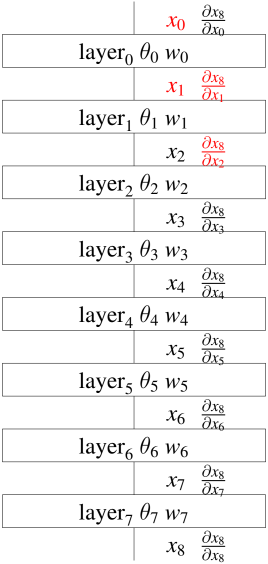
Evaluating a Neural Network to Allow Subsequent Backpropagation



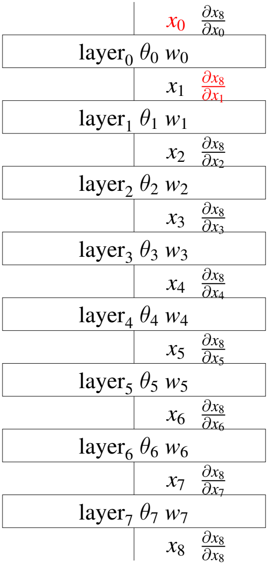
Evaluating a Neural Network to Allow Subsequent Backpropagation



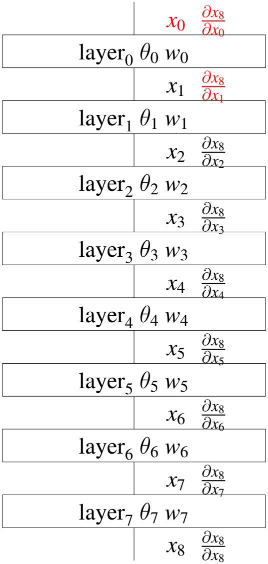
Evaluating a Neural Network to Allow Subsequent Backpropagation



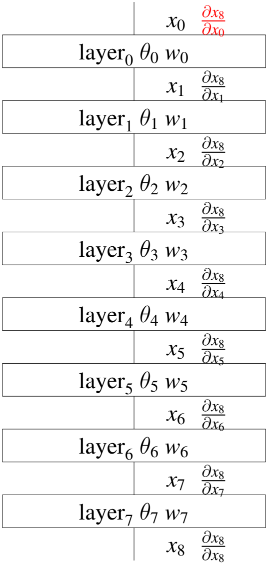
Evaluating a Neural Network to Allow Subsequent Backpropagation



Evaluating a Neural Network to Allow Subsequent Backpropagation



Evaluating a Neural Network to Allow Subsequent Backpropagation



Some Observations

Some Observations

- Only need to store live variables from forward pass until they are used in reverse pass.

Some Observations

- Only need to store live variables from forward pass until they are used in reverse pass.
- Only need to store live variables during reverse pass.

Some Observations

- Only need to store live variables from forward pass until they are used in reverse pass.
- Only need to store live variables during reverse pass.
- Most deep learning frameworks store all intermediate forward and reverse pass variables for simplicity of implementation.

Some Observations

- Only need to store live variables from forward pass until they are used in reverse pass.
- Only need to store live variables during reverse pass.
- Most deep learning frameworks store all intermediate forward and reverse pass variables for simplicity of implementation.
- It doesn't matter because storage use is dominated by maximal use.

Some Observations

- Only need to store live variables from forward pass until they are used in reverse pass.
- Only need to store live variables during reverse pass.
- Most deep learning frameworks store all intermediate forward and reverse pass variables for simplicity of implementation.
- It doesn't matter because storage use is dominated by maximal use.
- Maximal use is proportional to the depth of the network *i.e.*, the running time of the program.

Complexity of Reverse-Mode AD

Complexity of Reverse-Mode AD

- If running time of primal is $O(t)$

Complexity of Reverse-Mode AD

- If running time of primal is $O(t)$
  and primal has maximal live storage $O(w)$

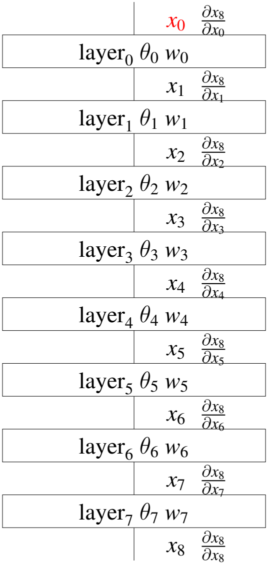
Complexity of Reverse-Mode AD

- If running time of primal is $O(t)$
  and primal has maximal live storage $O(w)$
- then reverse mode takes $O(wt)$ space

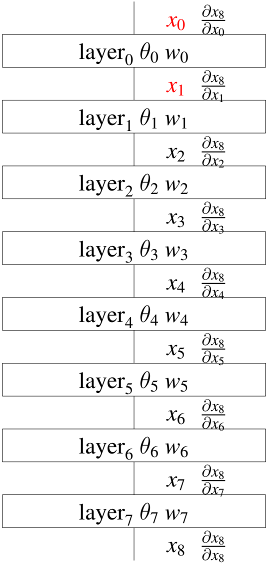
Complexity of Reverse-Mode AD

- If running time of primal is $O(t)$
  and primal has maximal live storage $O(w)$
- then reverse mode takes $O(wt)$ space
  and $O(t)$ time.

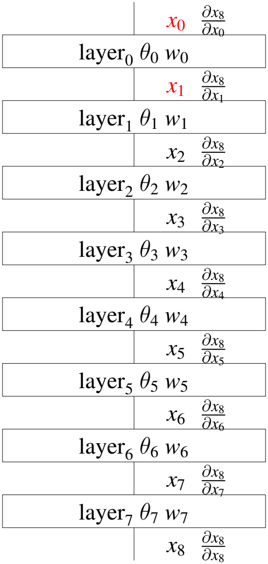
Backpropagation in a Neural Network with Checkpointing



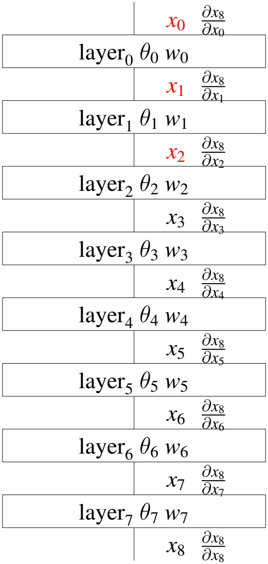
Backpropagation in a Neural Network with Checkpointing



Backpropagation in a Neural Network with Checkpointing



Backpropagation in a Neural Network with Checkpointing



Backpropagation in a Neural Network with Checkpointing

x_0 $\frac{\partial x_8}{\partial x_0}$

layer$_0$ θ_0 w_0

x_1 $\frac{\partial x_8}{\partial x_1}$

layer$_1$ θ_1 w_1

x_2 $\frac{\partial x_8}{\partial x_2}$

layer$_2$ θ_2 w_2

x_3 $\frac{\partial x_8}{\partial x_3}$

layer$_3$ θ_3 w_3

x_4 $\frac{\partial x_8}{\partial x_4}$

layer$_4$ θ_4 w_4

x_5 $\frac{\partial x_8}{\partial x_5}$

layer$_5$ θ_5 w_5

x_6 $\frac{\partial x_8}{\partial x_6}$

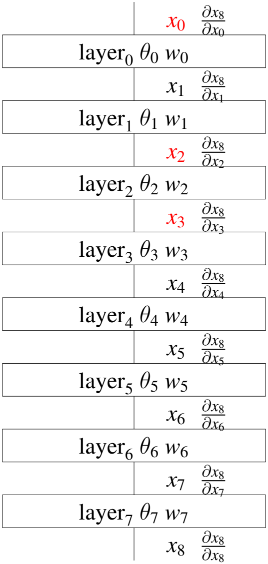
layer$_6$ θ_6 w_6

x_7 $\frac{\partial x_8}{\partial x_7}$

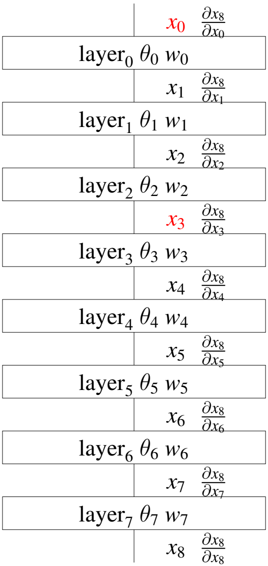
layer$_7$ θ_7 w_7

x_8 $\frac{\partial x_8}{\partial x_8}$

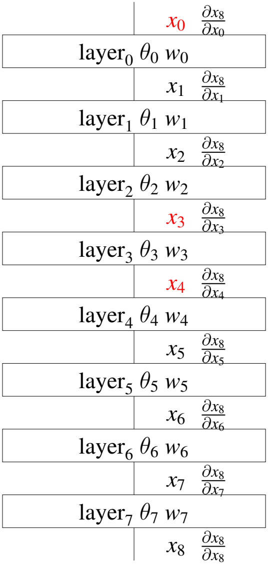
Backpropagation in a Neural Network with Checkpointing



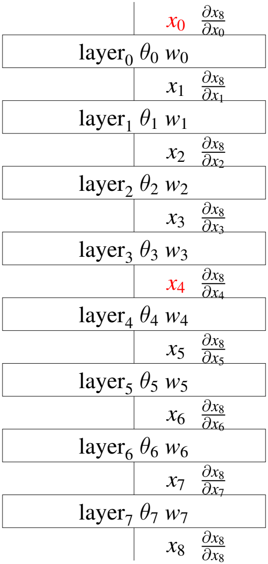
Backpropagation in a Neural Network with Checkpointing



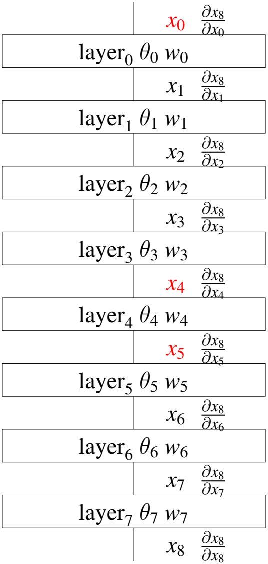
Backpropagation in a Neural Network with Checkpointing



Backpropagation in a Neural Network with Checkpointing



Backpropagation in a Neural Network with Checkpointing



Backpropagation in a Neural Network with Checkpointing

x_0 $\frac{\partial x_8}{\partial x_0}$

layer$_0$ θ_0 w_0

x_1 $\frac{\partial x_8}{\partial x_1}$

layer$_1$ θ_1 w_1

x_2 $\frac{\partial x_8}{\partial x_2}$

layer$_2$ θ_2 w_2

x_3 $\frac{\partial x_8}{\partial x_3}$

layer$_3$ θ_3 w_3

x_4 $\frac{\partial x_8}{\partial x_4}$

layer$_4$ θ_4 w_4

x_5 $\frac{\partial x_8}{\partial x_5}$

layer$_5$ θ_5 w_5

x_6 $\frac{\partial x_8}{\partial x_6}$

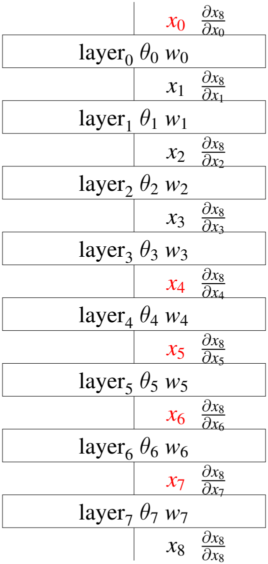
layer$_6$ θ_6 w_6

x_7 $\frac{\partial x_8}{\partial x_7}$

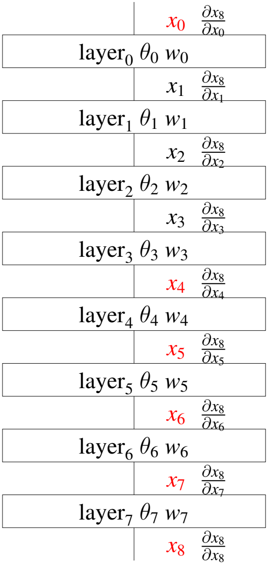
layer$_7$ θ_7 w_7

x_8 $\frac{\partial x_8}{\partial x_8}$

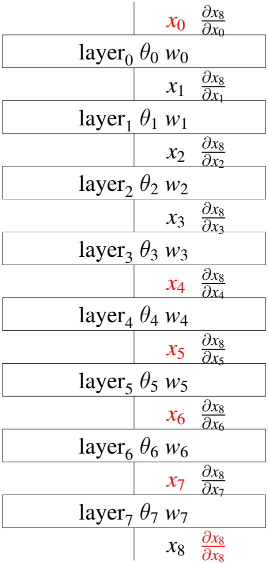
Backpropagation in a Neural Network with Checkpointing



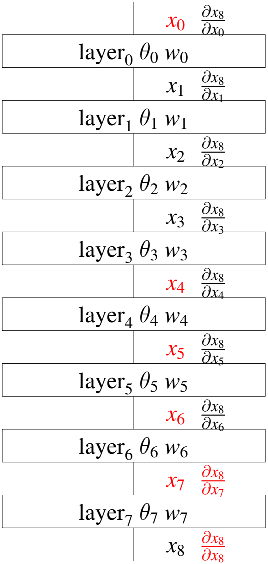
Backpropagation in a Neural Network with Checkpointing



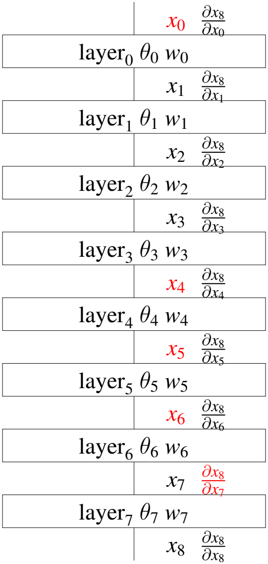
Backpropagation in a Neural Network with Checkpointing



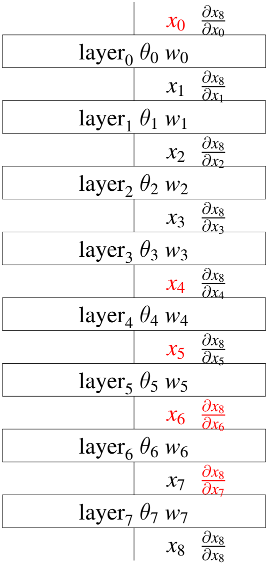
Backpropagation in a Neural Network with Checkpointing



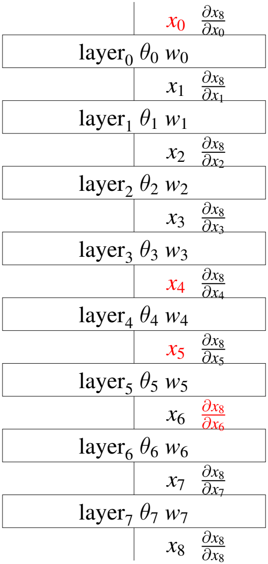
Backpropagation in a Neural Network with Checkpointing



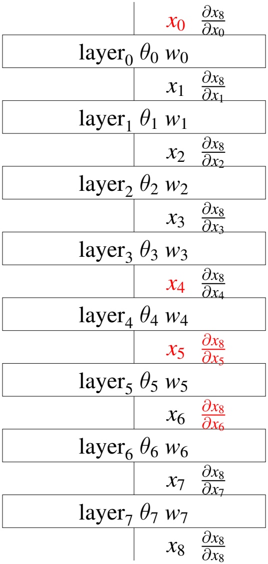
Backpropagation in a Neural Network with Checkpointing



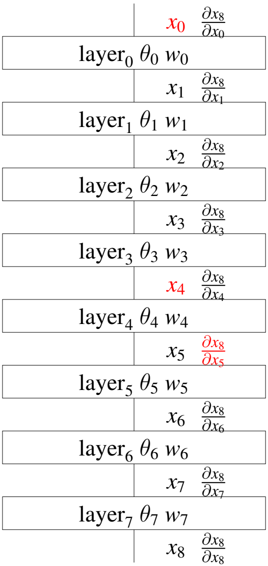
Backpropagation in a Neural Network with Checkpointing



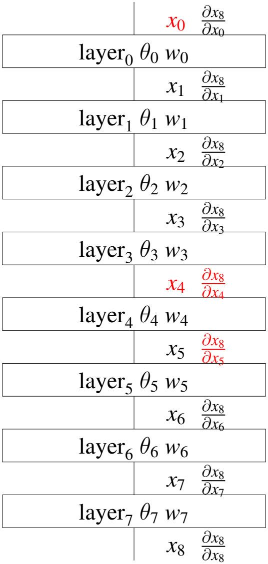
Backpropagation in a Neural Network with Checkpointing



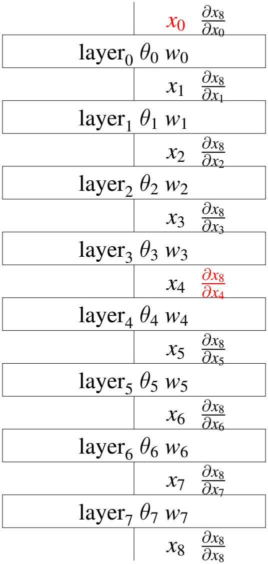
Backpropagation in a Neural Network with Checkpointing



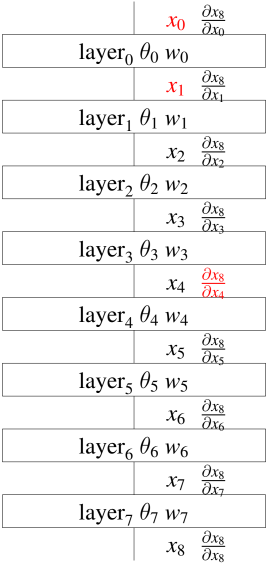
Backpropagation in a Neural Network with Checkpointing



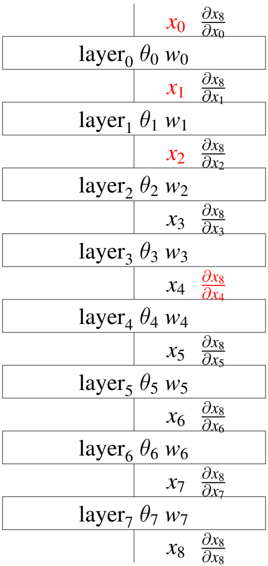
Backpropagation in a Neural Network with Checkpointing



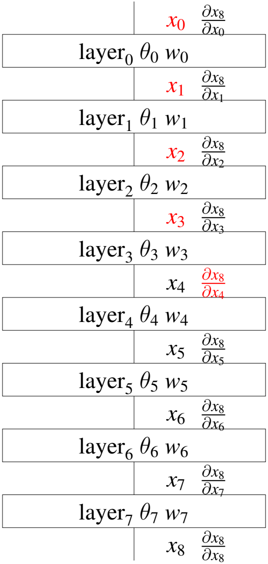
Backpropagation in a Neural Network with Checkpointing



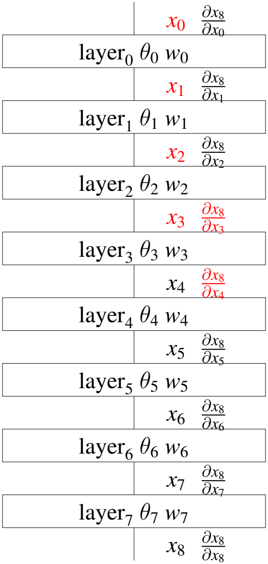
Backpropagation in a Neural Network with Checkpointing



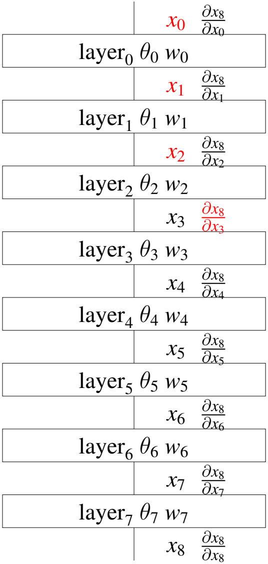
Backpropagation in a Neural Network with Checkpointing



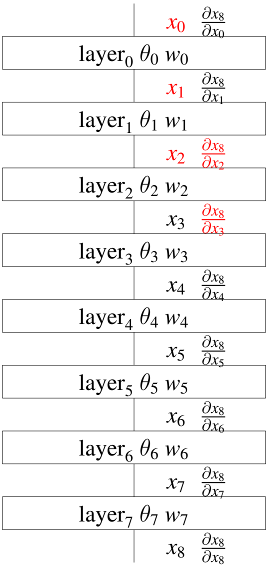
Backpropagation in a Neural Network with Checkpointing



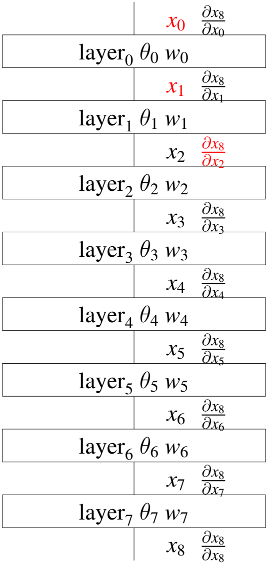
Backpropagation in a Neural Network with Checkpointing



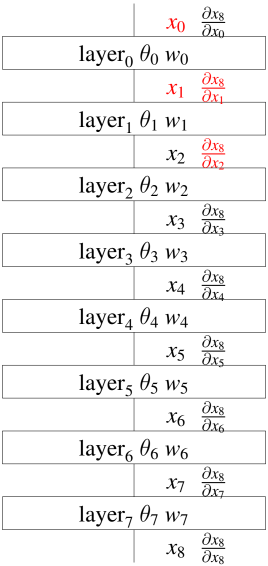
Backpropagation in a Neural Network with Checkpointing



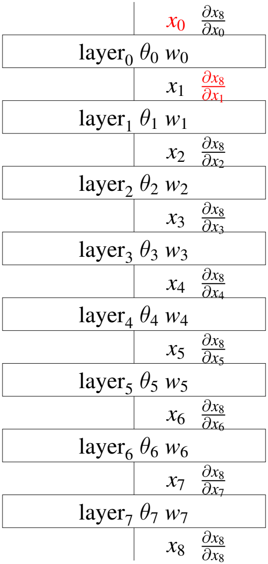
Backpropagation in a Neural Network with Checkpointing



Backpropagation in a Neural Network with Checkpointing



Backpropagation in a Neural Network with Checkpointing



Backpropagation in a Neural Network with Checkpointing

$x_0 \quad \frac{\partial x_8}{\partial x_0}$

$\text{layer}_0\ \theta_0\ w_0$

$x_1 \quad \frac{\partial x_8}{\partial x_1}$

$\text{layer}_1\ \theta_1\ w_1$

$x_2 \quad \frac{\partial x_8}{\partial x_2}$

$\text{layer}_2\ \theta_2\ w_2$

$x_3 \quad \frac{\partial x_8}{\partial x_3}$

$\text{layer}_3\ \theta_3\ w_3$

$x_4 \quad \frac{\partial x_8}{\partial x_4}$

$\text{layer}_4\ \theta_4\ w_4$

$x_5 \quad \frac{\partial x_8}{\partial x_5}$

$\text{layer}_5\ \theta_5\ w_5$

$x_6 \quad \frac{\partial x_8}{\partial x_6}$

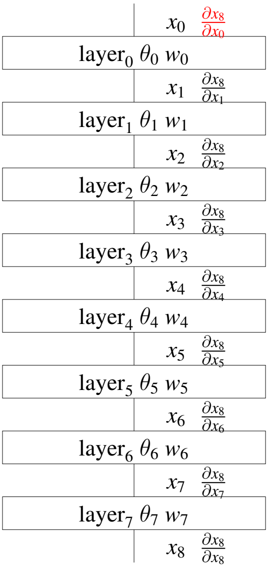
$\text{layer}_6\ \theta_6\ w_6$

$x_7 \quad \frac{\partial x_8}{\partial x_7}$

$\text{layer}_7\ \theta_7\ w_7$

$x_8 \quad \frac{\partial x_8}{\partial x_8}$

Backpropagation in a Neural Network with Checkpointing



Checkpointing

Checkpointing

- Trades off extra running time for reduction in space.

Checkpointing

- Trades off extra running time for reduction in space.
- Forward pass of first half performed twice.

Checkpointing

- Trades off extra running time for reduction in space.
- Forward pass of first half performed twice.
  Once without saving intermediate variables.

Checkpointing

- Trades off extra running time for reduction in space.
- Forward pass of first half performed twice.
  Once without saving intermediate variables.
  Once with saving intermediate variables.

Checkpointing

- Trades off extra running time for reduction in space.
- Forward pass of first half performed twice.
  Once without saving intermediate variables.
  Once with saving intermediate variables.
- Backpropagation done in stages.

Checkpointing

- Trades off extra running time for reduction in space.
- Forward pass of first half performed twice.
  Once without saving intermediate variables.
  Once with saving intermediate variables.
- Backpropagation done in stages.
  Interleaved with (re)running forward pass.

Checkpointing

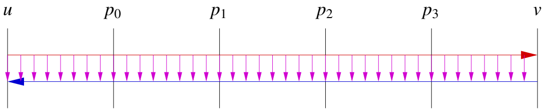
- Trades off extra running time for reduction in space.
- Forward pass of first half performed twice.
  Once without saving intermediate variables.
  Once with saving intermediate variables.
- Backpropagation done in stages.
  Interleaved with (re)running forward pass.
  Only need saved intermediate variables from forward pass for current stage.

Checkpointing

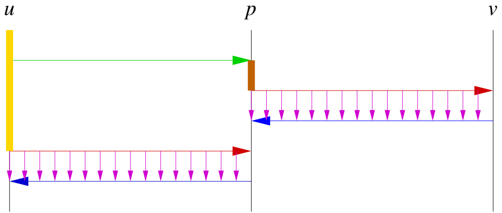
- Trades off extra running time for reduction in space.
- Forward pass of first half performed twice.
  Once without saving intermediate variables.
  Once with saving intermediate variables.
- Backpropagation done in stages.
  Interleaved with (re)running forward pass.
  Only need saved intermediate variables from forward pass for current stage.
- Can perform divide-and-conquer.

Divide-and-Conquer Checkpointing

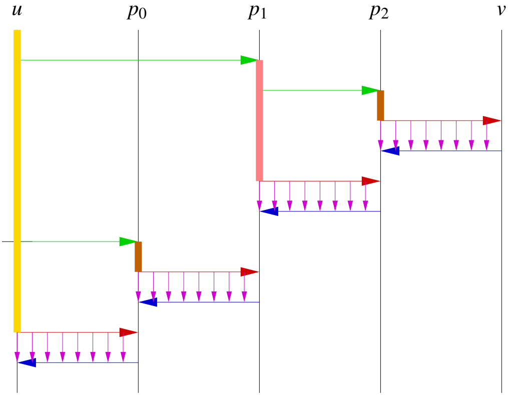
Divide-and-Conquer Checkpointing



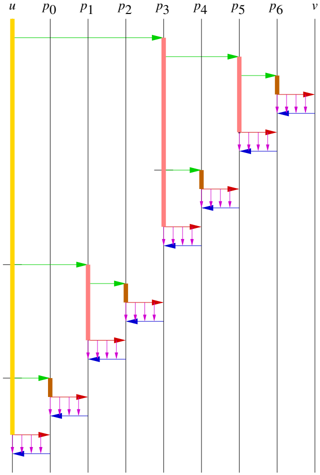
Divide-and-Conquer Checkpointing



Divide-and-Conquer Checkpointing



Divide-and-Conquer Checkpointing



Complexity of Divide-and-Conquer Checkpointing

Complexity of Divide-and-Conquer Checkpointing

- If running time of primal is $O(t)$

Complexity of Divide-and-Conquer Checkpointing

- If running time of primal is $O(t)$
  and primal has maximal live storage $O(w)$

Complexity of Divide-and-Conquer Checkpointing

- If running time of primal is $O(t)$
  and primal has maximal live storage $O(w)$
- then reverse mode takes $O(w \log t)$ space

Complexity of Divide-and-Conquer Checkpointing

- If running time of primal is $O(t)$
  and primal has maximal live storage $O(w)$
- then reverse mode takes $O(w \log t)$ space
  and $O(t \log t)$ time.

A (Brief) History of Divide-and-Conquer Checkpointing

A (Brief) History of Divide-and-Conquer Checkpointing

T. Chen, B. Xu, Z. Zhang, and C. Guestrin, *Training deep nets with sublinear memory cost*, arXiv 1604.06174, 2016.

A. Gruslys, R. Munos, I. Danihelka, M. Lanctot, and A. Graves, *Memory-Efficient Backpropagation Through Time*, NIPS, 2016.

A (Brief) History of Divide-and-Conquer Checkpointing

T. Chen, B. Xu, Z. Zhang, and C. Guestrin, *Training deep nets with sublinear memory cost*, arXiv 1604.06174, 2016.

A. Gruslys, R. Munos, I. Danihelka, M. Lanctot, and A. Graves, *Memory-Efficient Backpropagation Through Time*, NIPS, 2016.

A. Griewank, *Achieving Logarithmic Growth of Temporal and Spatial Complexity in Reverse Automatic Differentiation*, Optimization Methods and Software, 1:35-54, 1992.

Implemented for DO Loops

L. Hascoët and V. Pascual, *TAPENADE 2.1 User's Guide*, Rapport technique 300, INRIA, 2004.

Implemented for DO Loops

L. Hascoët and V. Pascual, *TAPENADE 2.1 User's Guide*, Rapport technique 300, INRIA, 2004.

```
      do 10 i=1, n
         ...
10    continue
```

Implemented for DO Loops

L. Hascoët and V. Pascual, *TAPENADE 2.1 User's Guide*, Rapport technique 300, INRIA, 2004.

```
      do 10 i=1, n            c$ad binomial-ckp n+1 30 1
         ...             ⇝          do 10 i=1, n
10    continue                          ...
                              10    continue
```

Implemented for DO Loops

L. Hascoët and V. Pascual, *TAPENADE 2.1 User's Guide*, Rapport technique 300, INRIA, 2004.

```
      do 10 i=1, n        }      c$ad binomial-ckp n+1 30 1
         ...              } ⇝         do 10 i=1, n
 10   continue            }              ...
                                 10   continue
```

https://www-sop.inria.fr/tropics/tapenade/faq.html

> *Assuming that the final number of iterations N is known, and assuming that each iteration has the same runtime cost,*

Desiderata

Desiderata

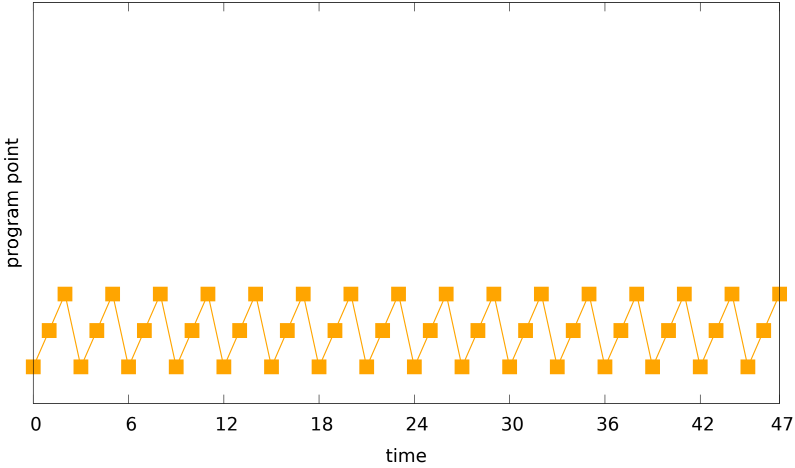
A (deep) neural network has no loops (except inside primitives).

Desiderata

A (deep) neural network has no loops (except inside primitives).

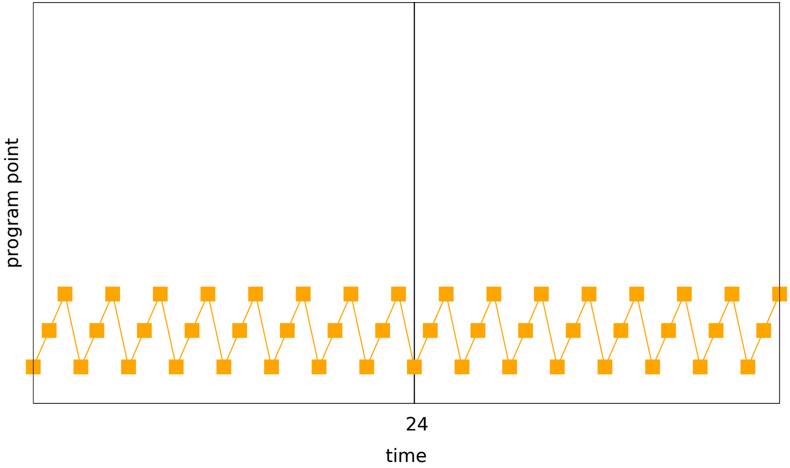
Want to implement for arbitrary code (not just a single DO loop).

Execution Trace of Loop



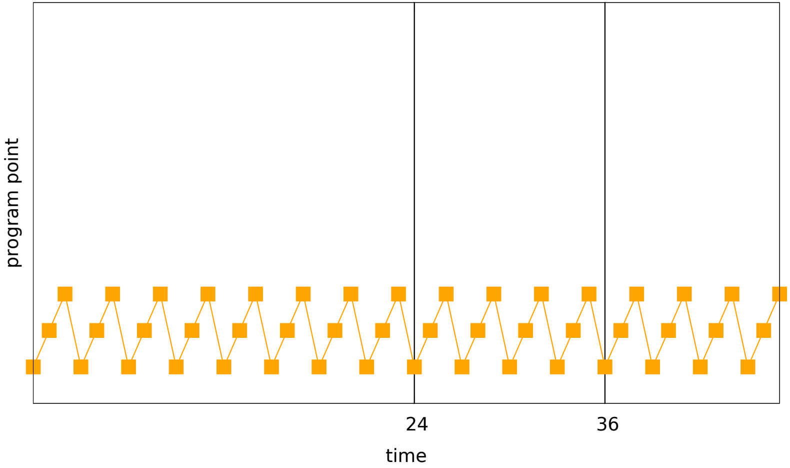
Execution Trace of Loop

Easy to make regular and uniform checkpoints



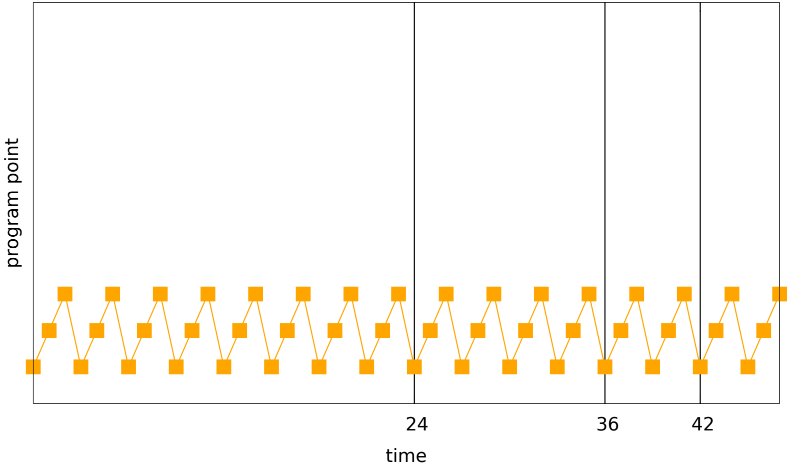
Execution Trace of Loop

Easy to make regular and uniform checkpoints



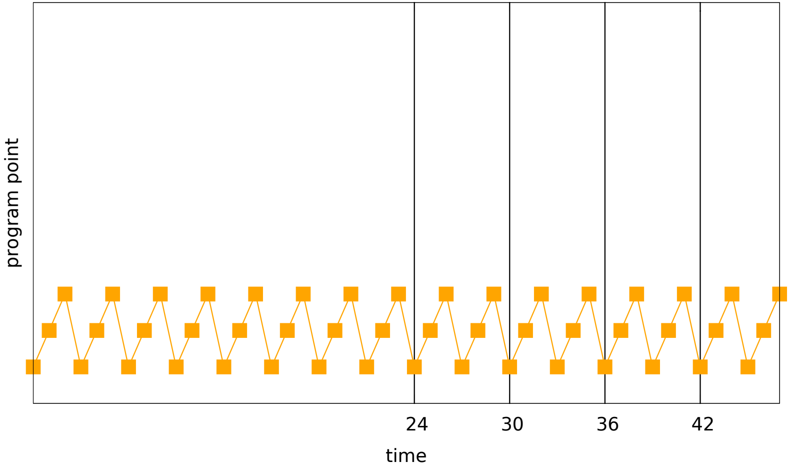
Execution Trace of Loop

Easy to make regular and uniform checkpoints



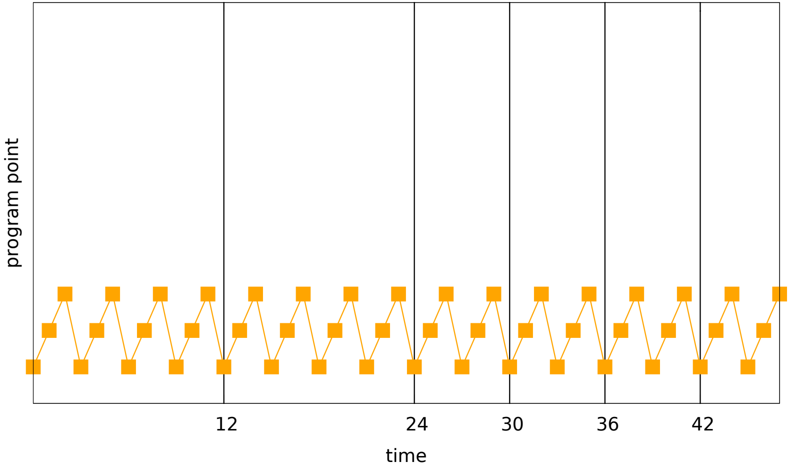
Execution Trace of Loop

Easy to make regular and uniform checkpoints



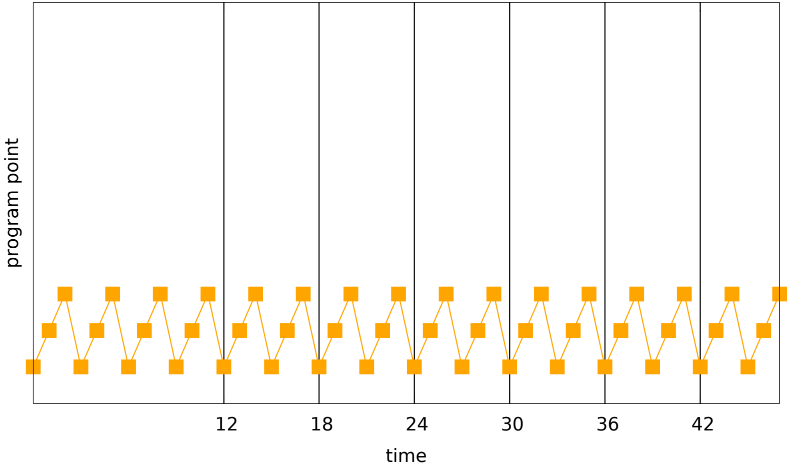
Execution Trace of Loop

Easy to make regular and uniform checkpoints



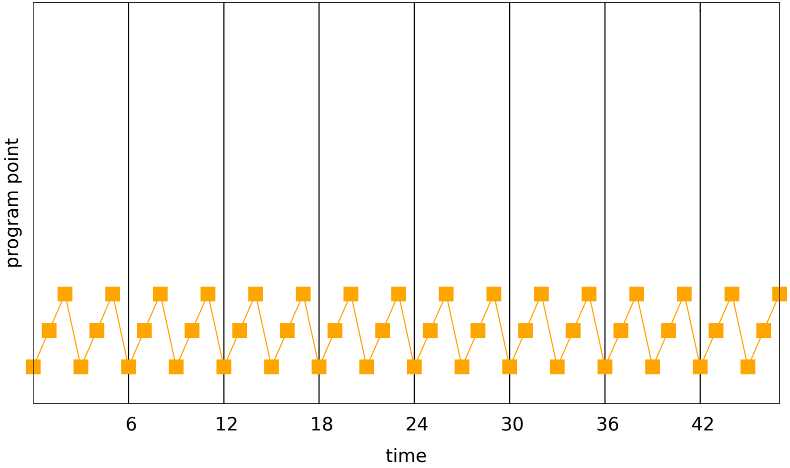
Execution Trace of Loop

Easy to make regular and uniform checkpoints

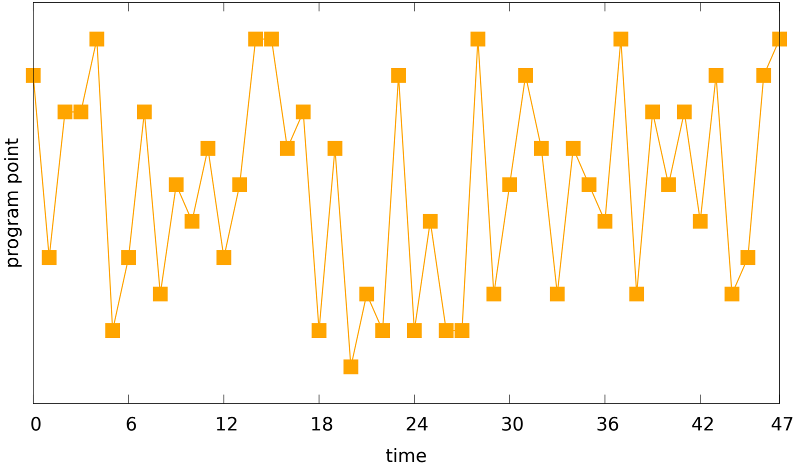


Execution Trace of Loop

Easy to make regular and uniform checkpoints

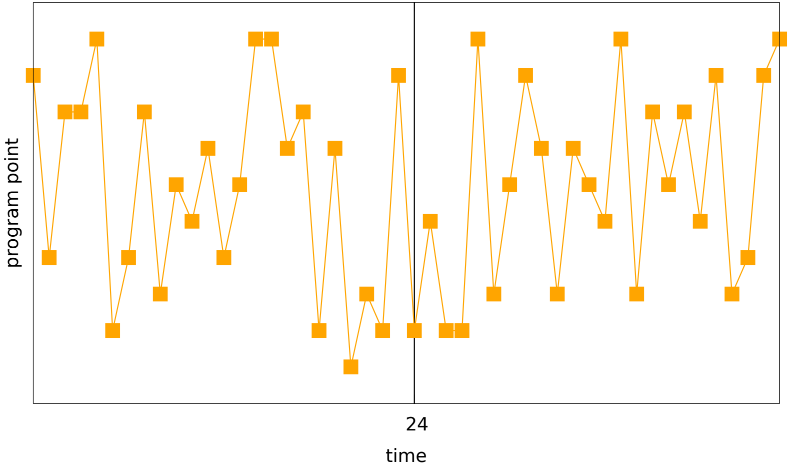


Execution Trace of Arbitrary Code



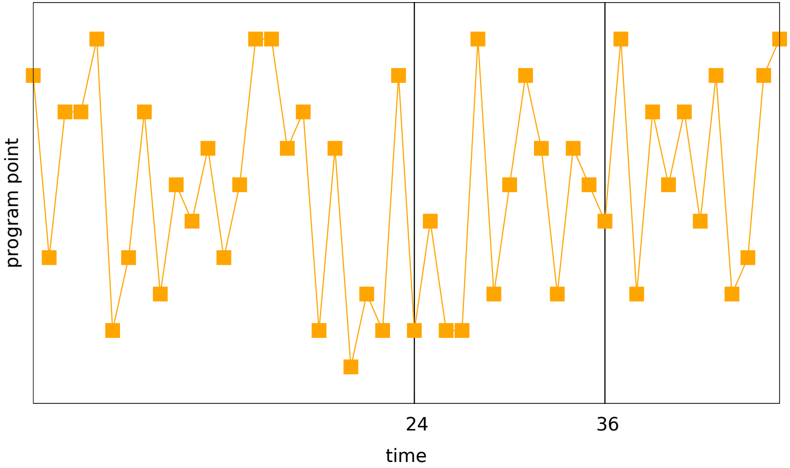
Execution Trace of Arbitrary Code

Difficult to make regular and uniform checkpoints



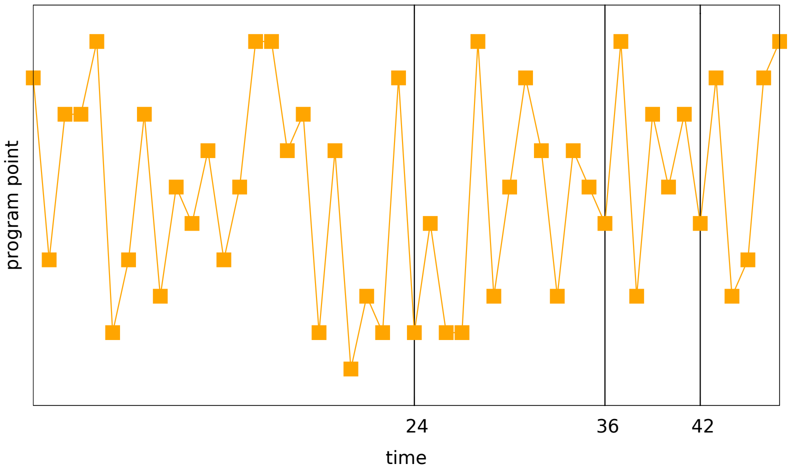
Execution Trace of Arbitrary Code

Difficult to make regular and uniform checkpoints



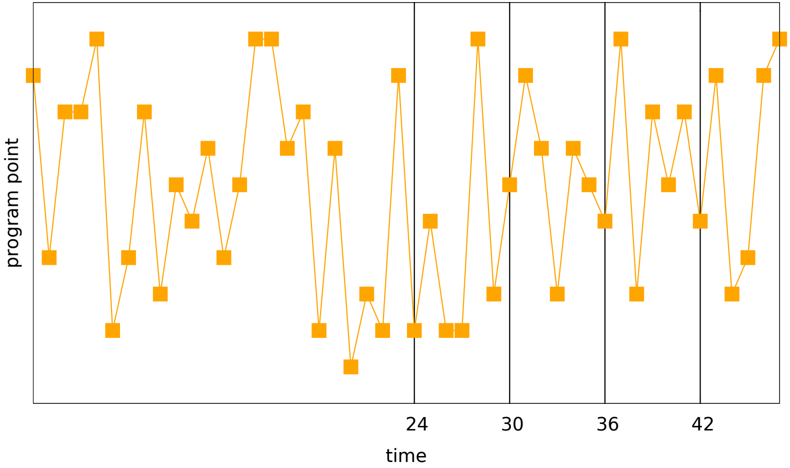
Execution Trace of Arbitrary Code

Difficult to make regular and uniform checkpoints



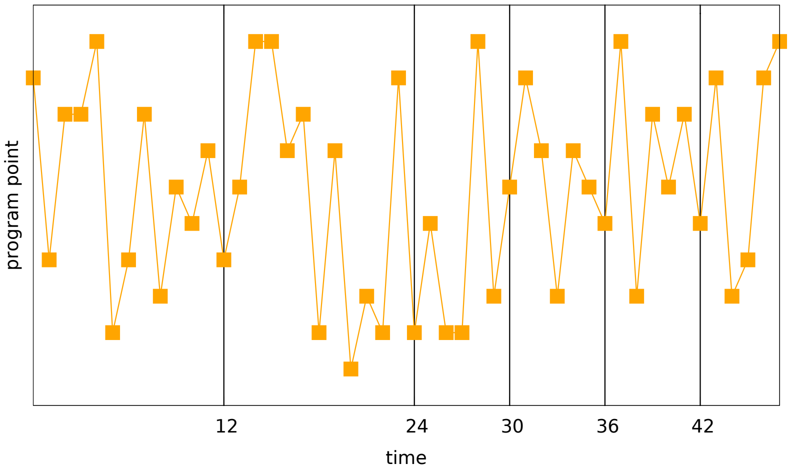
Execution Trace of Arbitrary Code

Difficult to make regular and uniform checkpoints



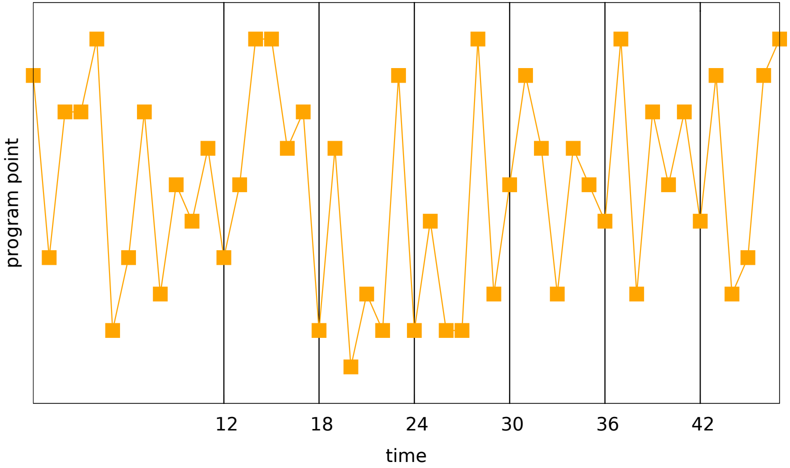
Execution Trace of Arbitrary Code

Difficult to make regular and uniform checkpoints



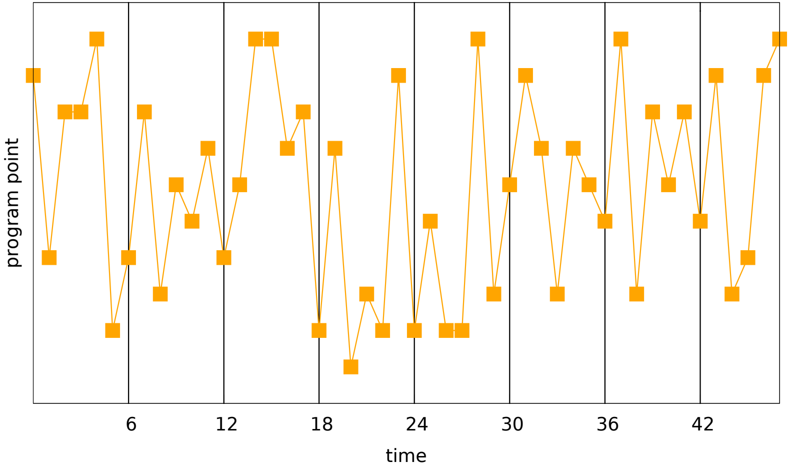
Execution Trace of Arbitrary Code

Difficult to make regular and uniform checkpoints



Execution Trace of Arbitrary Code

Difficult to make regular and uniform checkpoints



Key Challenges

Need to interleave generation of the network with forward and backward passes through the network.

Key Challenges

Need to interleave generation of the network with forward and backward passes through the network.

Portions of the network need to be (re)generated, and (re)evaluated with forward and backward passes, multiple times and out of order.

Key Idea

```
function main(w)
   local x = f(w)
   local y = h(g(x))
   local z = p(y)
   return z
end
```

Key Idea

```
function main(w)
    local x = f(w)
    local y = h(g(x))
    local z = p(y)
    return z
end
```

$\rightsquigarrow$

```
function main(w)
    for i = 1, 5
    if i==1 then
        local x = f(w)
    elseif i==2 then
        local t = g(x)
    elseif i==3 then
        local y = h(t)
    elseif i==4 then
        local z = p(y)
    elseif i==5 then
        return z
    end
end
```

Core Language

$$e ::= c \mid x \mid \lambda x.e \mid e_1\ e_2 \mid \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \mid \diamond e \mid e_1 \bullet e_2$$

Adding AD Operators to the Core Language

$$\overleftarrow{\mathcal{J}} : f\ x\ \grave{y} \mapsto (y, \grave{x})$$

$$\checkmark\!\!\!\mathcal{J} : f\ x\ \grave{y} \mapsto (y, \grave{x})$$

Algorithm for Divide-and-Conquer Checkpointing

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

base case ($f\ x$ fast): $(y, \grave{x}) = \overleftarrow{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

inductive case: $h \circ g = f$ (step 1)

$z = g\ x$ (step 2)

$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ h\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

Algorithm for Divide-and-Conquer Checkpointing

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\; f\; x\; \grave{y}$:

base case ($f\ x$ fast): $(y, \grave{x}) = \overleftarrow{\mathcal{J}}\; f\; x\; \grave{y}$ (step 0)

inductive case: $h \circ g = f$ (step 1)

$z = g\; x$ (step 2)

$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\; h\; z\; \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\; g\; x\; \grave{z}$ (step 4)

Algorithm for Divide-and-Conquer Checkpointing

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

base case (f x fast): $(y, \grave{x}) = \overleftarrow{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

inductive case:

$h \circ g = f$ (step 1)

$z = g\ x$ (step 2)

$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ h\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

Algorithm for Divide-and-Conquer Checkpointing

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

base case (f x fast): $(y, \grave{x}) = \overleftarrow{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

inductive case:

$h \circ g = f$ (step 1)

$z = g\ x$ (step 2)

$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ h\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

Algorithm for Divide-and-Conquer Checkpointing

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ {\color{red}x}\ {\color{red}\grave{y}}$:

base case ($f\ x$ fast): $(y, \grave{x}) = \overleftarrow{\mathcal{J}}\ f\ {\color{red}x}\ {\color{red}\grave{y}}$ (step 0)

inductive case: $h \circ g = f$ (step 1)

$z = g\ x$ (step 2)

$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ h\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

Algorithm for Divide-and-Conquer Checkpointing

To compute $({\color{red}y}, {\color{red}\grave{x}}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

base case ($f\ x$ fast):	$({\color{red}y}, {\color{red}\grave{x}}) = \overleftarrow{\mathcal{J}}\ f\ x\ \grave{y}$	(step 0)
inductive case:	$h \circ g = f$	(step 1)
	$z = g\ x$	(step 2)
	$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ h\ z\ \grave{y}$	(step 3)
	$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z}$	(step 4)

Algorithm for Divide-and-Conquer Checkpointing

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

base case ($f\ x$ fast):	$(y, \grave{x}) = \overleftarrow{\mathcal{J}}\ f\ x\ \grave{y}$	(step 0)
inductive case:	$h \circ g = f$	(step 1)
	$z = g\ x$	(step 2)
	$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ h\ z\ \grave{y}$	(step 3)
	$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z}$	(step 4)

Algorithm for Divide-and-Conquer Checkpointing

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

base case (f x fast): $(y, \grave{x}) = \overleftarrow{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

inductive case:

$$h \circ g = f \qquad \text{(step 1)}$$

$$z = g\ x \qquad \text{(step 2)}$$

$$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ h\ z\ \grave{y} \qquad \text{(step 3)}$$

$$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z} \qquad \text{(step 4)}$$

Algorithm for Divide-and-Conquer Checkpointing

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

base case ($f\ x$ fast): $(y, \grave{x}) = \overleftarrow{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

inductive case: $h \circ g = f$ (step 1)

$z = g\ x$ (step 2)

$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ h\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

Algorithm for Divide-and-Conquer Checkpointing

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

base case ($f\ x$ fast):

$$(y, \grave{x}) = \overleftarrow{\mathcal{J}}\ f\ x\ \grave{y} \qquad \text{(step 0)}$$

inductive case:

$$h \circ g = f \qquad \text{(step 1)}$$

$$z = g\ x \qquad \text{(step 2)}$$

$$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ h\ z\ \grave{y} \qquad \text{(step 3)}$$

$$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z} \qquad \text{(step 4)}$$

Algorithm for Divide-and-Conquer Checkpointing

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

base case ($f\ x$ fast): $(y, \grave{x}) = \overleftarrow{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

inductive case: $h \circ g = f$ (step 1)

$z = g\ x$ (step 2)

$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ h\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

Algorithm for Divide-and-Conquer Checkpointing

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

base case ($f\ x$ fast): $(y, \grave{x}) = \overleftarrow{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

inductive case: $h \circ g = f$ (step 1)

$z = g\ x$ (step 2)

$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ h\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

Algorithm for Divide-and-Conquer Checkpointing

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

base case ($f\ x$ fast): $(y, \grave{x}) = \overleftarrow{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

inductive case:

$h \circ g = f$ (step 1)

$z = g\ x$ (step 2)

$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ h\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

Algorithm for Divide-and-Conquer Checkpointing

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

base case ($f\ x$ fast): $(y, \grave{x}) = \overleftarrow{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

inductive case: $h \circ g = f$ (step 1)

$z = g\ x$ (step 2)

$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ h\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

Algorithm for Divide-and-Conquer Checkpointing

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

base case ($f\ x$ fast): $(y, \grave{x}) = \overleftarrow{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

inductive case: $h \circ g = f$ (step 1)

$z = g\ x$ (step 2)

$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ h\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

Algorithm for Divide-and-Conquer Checkpointing

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\; f\; x\; \grave{y}$:

base case ($f\ x$ fast):	$(y, \grave{x}) = \overleftarrow{\mathcal{J}}\; f\; x\; \grave{y}$	(step 0)
inductive case:	$h \circ g = f$	(step 1)
	$z = g\; x$	(step 2)
	$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\; h\; z\; \grave{y}$	(step 3)
	$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\; g\; x\; \grave{z}$	(step 4)

Algorithm for Divide-and-Conquer Checkpointing

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

base case ($f\ x$ fast): $(y, \grave{x}) = \overleftarrow{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

inductive case: $h \circ g = f$ (step 1)

$z = g\ x$ (step 2)

$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ h\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

Algorithm for Divide-and-Conquer Checkpointing

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\; f\; x\; \grave{y}$:

base case (f x fast): $(y, \grave{x}) = \overleftarrow{\mathcal{J}}\; f\; x\; \grave{y}$ (step 0)

inductive case:

$h \circ g = f$ (step 1)

$z = g\; x$ (step 2)

$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\; h\; z\; \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\; g\; x\; \grave{z}$ (step 4)

Algorithm for Divide-and-Conquer Checkpointing

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

base case ($f\ x$ fast): $(y, \grave{x}) = \overleftarrow{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

inductive case: $h \circ g = f$ (step 1)

$z = g\ x$ (step 2)

$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ h\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

Algorithm for Divide-and-Conquer Checkpointing

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

base case (f x fast): $(y, \grave{x}) = \overleftarrow{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

inductive case: $h \circ g = f$ (step 1)

$z = g\ x$ (step 2)

$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ h\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

What is Needed to Implement the Algorithm?

What is Needed to Implement the Algorithm?

1. measure the length of the primal computation

What is Needed to Implement the Algorithm?

1. measure the length of the primal computation
2. interrupt the primal computation at a portion of the measured length

What is Needed to Implement the Algorithm?

1. measure the length of the primal computation
2. interrupt the primal computation at a portion of the measured length
3. save the state of the interrupted computation as a capsule

What is Needed to Implement the Algorithm?

1. measure the length of the primal computation
2. interrupt the primal computation at a portion of the measured length
3. save the state of the interrupted computation as a capsule
4. resume an interrupted computation from a capsule

General-Purpose Interruption and Resumption Interface

PRIMOPS $f\ x \mapsto l$	Return the number l of evaluation steps needed to compute $y = f(x)$.
INTERRUPT $f\ x\ l \mapsto z$	Run the first l steps of the computation of $f(x)$ and return a capsule z.
RESUME $z \mapsto y$	If $z = (\text{INTERRUPT}\ f\ x\ l)$, return $y = f(x)$.

General-Purpose Interruption and Resumption Interface

PRIMOPS $f\ x \mapsto l$	Return the number l of evaluation steps needed to compute $y = f(x)$.
INTERRUPT $f\ x\ l \mapsto z$	Run the first l steps of the computation of $f(x)$ and return a capsule z.
RESUME $z \mapsto y$	If $z = (\text{INTERRUPT}\ f\ x\ l)$, return $y = f(x)$.

General-Purpose Interruption and Resumption Interface

PRIMOPS $f\ x \mapsto l$	Return the number l of evaluation steps needed to compute $y = f(x)$.
INTERRUPT $f\ x\ l \mapsto z$	Run the first l steps of the computation of $f(x)$ and return a capsule z.
RESUME $z \mapsto y$	If $z = (\text{INTERRUPT}\ f\ x\ l)$, return $y = f(x)$.

General-Purpose Interruption and Resumption Interface

PRIMOPS $f\ x \mapsto l$	Return the number l of evaluation steps needed to compute $y = f(x)$.
INTERRUPT $f\ x\ l \mapsto z$	Run the first l steps of the computation of $f(x)$ and return a capsule z.
RESUME $z \mapsto y$	If $z = (\text{INTERRUPT}\ f\ x\ l)$, return $y = f(x)$.

General-Purpose Interruption and Resumption Interface

PRIMOPS $f\ x \mapsto l$	Return the number l of evaluation steps needed to compute $y = f(x)$.
INTERRUPT $f\ x\ l \mapsto z$	Run the first l steps of the computation of $f(x)$ and return a capsule z.
RESUME $z \mapsto y$	If $z = (\text{INTERRUPT}\ f\ x\ l)$, return $y = f(x)$.

General-Purpose Interruption and Resumption Interface

$\text{PRIMOPS}\, f\ x \mapsto l$	Return the number l of evaluation steps needed to compute $y = f(x)$.
$\text{INTERRUPT}\, f\ x\ l \mapsto z$	Run the first l steps of the computation of $f(x)$ and return a capsule z.
$\text{RESUME}\ z \mapsto y$	If $z = (\text{INTERRUPT}\, f\ x\ l)$, return $y = f(x)$.

General-Purpose Interruption and Resumption Interface

PRIMOPS $f\ x \mapsto l$	Return the number l of evaluation steps needed to compute $y = f(x)$.
INTERRUPT $f\ x\ l \mapsto z$	Run the first l steps of the computation of $f(x)$ and return a capsule z.
RESUME $z \mapsto y$	If $z = (\text{INTERRUPT}\ f\ x\ l)$, return $y = f(x)$.

General-Purpose Interruption and Resumption Interface

PRIMOPS $f\ x \mapsto l$	Return the number l of evaluation steps needed to compute $y = f(x)$.
INTERRUPT $f\ x\ l \mapsto z$	Run the first l steps of the computation of $f(x)$ and return a capsule z.
RESUME $z \mapsto y$	If $z = (\text{INTERRUPT}\ f\ x\ l)$, return $y = f(x)$.

General-Purpose Interruption and Resumption Interface

PRIMOPS $f\ x \mapsto l$	Return the number l of evaluation steps needed to compute $y = f(x)$.
INTERRUPT $f\ x\ l \mapsto z$	Run the first l steps of the computation of $f(x)$ and return a capsule z.
RESUME $z \mapsto y$	If $z = (\text{INTERRUPT}\ f\ x\ l)$, return $y = f(x)$.

General-Purpose Interruption and Resumption Interface

PRIMOPS $f\ x \mapsto l$	Return the number l of evaluation steps needed to compute $y = f(x)$.
INTERRUPT $f\ x\ l \mapsto z$	Run the first l steps of the computation of $f(x)$ and return a capsule z.
RESUME $z \mapsto y$	If $z = (\text{INTERRUPT}\ f\ x\ l)$, return $y = f(x)$.

General-Purpose Interruption and Resumption Interface

$\text{PRIMOPS}\, f\ x \mapsto l$	Return the number l of evaluation steps needed to compute $y = f(x)$.
$\text{INTERRUPT}\, f\ x\ l \mapsto z$	Run the first l steps of the computation of $f(x)$ and return a capsule z.
$\text{RESUME}\ z \mapsto y$	If $z = (\text{INTERRUPT}\, f\ x\ l)$, return $y = f(x)$.

General-Purpose Interruption and Resumption Interface

PRIMOPS $f\ x \mapsto l$	Return the number l of evaluation steps needed to compute $y = f(x)$.
INTERRUPT $f\ x\ l \mapsto z$	Run the first l steps of the computation of $f(x)$ and return a capsule z.
RESUME $z \mapsto y$	If $z = (\text{INTERRUPT}\ f\ x\ l)$, return $y = f(x)$.

General-Purpose Interruption and Resumption Interface

PRIMOPS $f\ x \mapsto l$	Return the number l of evaluation steps needed to compute $y = f(x)$.
INTERRUPT $f\ x\ l \mapsto z$	Run the first l steps of the computation of $f(x)$ and return a capsule z.
RESUME $z \mapsto y$	If $z = (\text{INTERRUPT}\ f\ x\ l)$, return $y = f(x)$.

General-Purpose Interruption and Resumption Interface

PRIMOPS $f\ x \mapsto l$	Return the number l of evaluation steps needed to compute $y = f(x)$.
INTERRUPT $f\ x\ l \mapsto z$	Run the first l steps of the computation of $f(x)$ and return a capsule z.
RESUME $z \mapsto y$	If $z = (\text{INTERRUPT}\ f\ x\ l)$, return $y = f(x)$.

Algorithm for Divide-and-Conquer Checkpointing

via General-Purpose Interruption and Resumption Interface

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

base case ($f\ x$ fast): $(y, \grave{x}) = \overleftarrow{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

inductive case: $h \circ g = f$ (step 1)

$z = g\ x$ (step 2)

$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ h\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

Algorithm for Divide-and-Conquer Checkpointing

via General-Purpose Interruption and Resumption Interface

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

base case ($f\ x$ fast): $(y, \grave{x}) = \overset{\leftarrow}{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

inductive case: $h \circ g = f$ (step 1)

$z = g\ x$ (step 2)

$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ h\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

Algorithm for Divide-and-Conquer Checkpointing

via General-Purpose Interruption and Resumption Interface

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

base case (f x fast):	$(y, \grave{x}) = \overleftarrow{\mathcal{J}}\ f\ x\ \grave{y}$	(step 0)
inductive case:	$h \circ g = f$	(step 1)
	$z = g\ x$	(step 2)
	$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ h\ z\ \grave{y}$	(step 3)
	$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z}$	(step 4)

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

base case ($f\ x$ fast): $(y, \grave{x}) = \overleftarrow{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

inductive case: $l = \text{PRIMOPS}\ f\ x$ (step 1)

$z = g\ x$ (step 2)

$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ h\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

base case ($f\ x$ fast): $(y, \grave{x}) = \overleftarrow{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

inductive case: $l = \text{PRIMOPS}\ f\ x$ (step 1)

$z = g\ x$ (step 2)

$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ h\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

Algorithm for Divide-and-Conquer Checkpointing

via General-Purpose Interruption and Resumption Interface

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

base case ($f\ x$ fast):	$(y, \grave{x}) = \overleftarrow{\mathcal{J}}\ f\ x\ \grave{y}$	(step 0)
inductive case:	$l = \text{PRIMOPS}\ f\ x$	(step 1)
	$z = \color{red}{g}\ x$	(step 2)
	$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ h\ z\ \grave{y}$	(step 3)
	$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z}$	(step 4)

Algorithm for Divide-and-Conquer Checkpointing

via General-Purpose Interruption and Resumption Interface

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

base case ($f\ x$ fast): $(y, \grave{x}) = \overleftarrow{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

inductive case: $l = \text{PRIMOPS}\ f\ x$ (step 1)

$z = \text{INTERRUPT}\ f\ x\ \lfloor \frac{l}{2} \rfloor$ (step 2)

$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ h\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

base case ($f\ x$ fast): $(y, \grave{x}) = \overleftarrow{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

inductive case: $l = \text{PRIMOPS}\ f\ x$ (step 1)

$z = \text{INTERRUPT}\ f\ x\ \lfloor \frac{l}{2} \rfloor$ (step 2)

$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ h\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

Algorithm for Divide-and-Conquer Checkpointing

via General-Purpose Interruption and Resumption Interface

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\, f\; x\; \grave{y}$:

base case ($f\ x$ fast):	$(y, \grave{x}) = \overleftarrow{\mathcal{J}}\, f\; x\; \grave{y}$	(step 0)
inductive case:	$l = \text{PRIMOPS}\; f\; x$	(step 1)
	$z = \text{INTERRUPT}\; f\; x\; \lfloor \frac{l}{2} \rfloor$	(step 2)
	$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\; h\; z\; \grave{y}$	(step 3)
	$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\; g\; x\; \grave{z}$	(step 4)

Algorithm for Divide-and-Conquer Checkpointing

via General-Purpose Interruption and Resumption Interface

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

base case ($f\ x$ fast): $(y, \grave{x}) = \overleftarrow{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

inductive case: $l = \text{PRIMOPS}\ f\ x$ (step 1)

$z = \text{INTERRUPT}\ f\ x\ \lfloor \frac{l}{2} \rfloor$ (step 2)

$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ (\lambda z.\text{RESUME}\ z)\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

Algorithm for Divide-and-Conquer Checkpointing

via General-Purpose Interruption and Resumption Interface

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

base case ($f\ x$ fast): $(y, \grave{x}) = \overleftarrow{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

inductive case:

$l = \text{PRIMOPS}\ f\ x$ (step 1)

$z = \text{INTERRUPT}\ f\ x\ \left\lfloor \frac{l}{2} \right\rfloor$ (step 2)

$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ (\lambda z.\text{RESUME}\ z)\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z}$ (step 4)

Algorithm for Divide-and-Conquer Checkpointing

via General-Purpose Interruption and Resumption Interface

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

base case ($f\ x$ fast): $(y, \grave{x}) = \overset{\leftarrow}{\mathcal{J}}\ f\ x\ \grave{y}$ (step 0)

inductive case:

$l = \text{PRIMOPS}\ f\ x$ (step 1)

$z = \text{INTERRUPT}\ f\ x\ \lfloor \frac{l}{2} \rfloor$ (step 2)

$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ (\lambda z.\text{RESUME}\ z)\ z\ \grave{y}$ (step 3)

$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ {\color{red}g}\ x\ \grave{z}$ (step 4)

Algorithm for Divide-and-Conquer Checkpointing

via General-Purpose Interruption and Resumption Interface

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

base case ($f\ x$ fast):	$(y, \grave{x}) = \overleftarrow{\mathcal{J}}\ f\ x\ \grave{y}$	(step 0)
inductive case:	$l = \text{PRIMOPS}\ f\ x$	(step 1)
	$z = \text{INTERRUPT}\ f\ x\ \lfloor \frac{l}{2} \rfloor$	(step 2)
	$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ (\lambda z.\text{RESUME}\ z)\ z\ \grave{y}$	(step 3)
	$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ (\lambda x.\text{INTERRUPT}\ f\ x\ \lfloor \frac{l}{2} \rfloor)\ x\ \grave{z}$	(step 4)

Example of CPS Conversion

```
function f(x)
   return q(p(g(x), h(x)))
end
```

$\rightsquigarrow$

```
function f(c, x)
   return g(function(t1)
               return h(function(t2)
                           return p(function(t3)
                                       return q(c, t3)
                                    end, t1, t2)
                        end, x)
            end, x)
end
```

Implementation

Implementation

1. Convert source program to CPS.

Implementation

1. Convert source program to CPS.
2. Thread step count and limit.

Implementation

1. Convert source program to CPS.
2. Thread step count and limit.
3. Translate CPS to C.

Implementation

1. Convert source program to CPS.
2. Thread step count and limit.
3. Translate CPS to C.
4. Combine with general-purpose interruption and resumption interface and $\sqrt{\mathcal{J}}$ written in C.

Implementation

1. Convert source program to CPS.
2. Thread step count and limit.
3. Translate CPS to C.
4. Combine with general-purpose interruption and resumption interface and $\sqrt{\mathcal{J}}$ written in C.
5. Compile to machine code.

Implementation

1. Convert source program to CPS.
2. Thread step count and limit.
3. Translate CPS to C.
4. Combine with general-purpose interruption and resumption interface and $\sqrt{\mathcal{J}}$ written in C.
5. Compile to machine code.

CPS Conversion as a Program Transformation

$$\ulcorner x | k \urcorner \rightsquigarrow k\ x$$

$$\ulcorner (\lambda x.e) | k \urcorner \rightsquigarrow k\ (\lambda k'\ x. \ulcorner e | k' \urcorner)$$

$$\ulcorner (e_1\ e_2) | k \urcorner \rightsquigarrow \ulcorner e_1 | (\lambda x_1. \ulcorner e_2 | (\lambda x_2.(x_1\ k\ x_2)) \urcorner) \urcorner$$

$$e_0 \rightsquigarrow \ulcorner e_0 | (\lambda x.x) \urcorner$$

CPS Conversion as a Program Transformation

$$\ulcorner x|k \urcorner \rightsquigarrow k\ x$$

$$\ulcorner (\lambda x.e)|k \urcorner \rightsquigarrow k\ (\lambda k'\ x.\ulcorner e|k' \urcorner)$$

$$\ulcorner (e_1\ e_2)|k \urcorner \rightsquigarrow \ulcorner e_1|(\lambda x_1.\ulcorner e_2|(\lambda x_2.(x_1\ k\ x_2)) \urcorner) \urcorner$$

$$e_0 \rightsquigarrow \ulcorner e_0|(\lambda x.x) \urcorner$$

CPS Conversion as a Program Transformation

$$\ulcorner x|{\color{red}k}\urcorner \rightsquigarrow k\ x$$
$$\ulcorner(\lambda x.e)|k\urcorner \rightsquigarrow k\ (\lambda k'\ x.\ulcorner e|k'\urcorner)$$
$$\ulcorner(e_1\ e_2)|k\urcorner \rightsquigarrow \ulcorner e_1|(\lambda x_1.\ulcorner e_2|(\lambda x_2.(x_1\ k\ x_2))\urcorner)\urcorner$$
$$e_0 \rightsquigarrow \ulcorner e_0|(\lambda x.x)\urcorner$$

CPS Conversion as a Program Transformation

$$\ulcorner x | k \urcorner \rightsquigarrow k\ x$$

$$\ulcorner (\lambda x.e) | k \urcorner \rightsquigarrow k\ (\lambda k'\ x. \ulcorner e | k' \urcorner)$$

$$\ulcorner (e_1\ e_2) | k \urcorner \rightsquigarrow \ulcorner e_1 | (\lambda x_1. \ulcorner e_2 | (\lambda x_2.(x_1\ k\ x_2)) \urcorner) \urcorner$$

$$e_0 \rightsquigarrow \ulcorner e_0 | (\lambda x.x) \urcorner$$

CPS Conversion as a Program Transformation

$$\ulcorner x | k \urcorner \rightsquigarrow k\ x$$

$$\ulcorner (\lambda x.e) | k \urcorner \rightsquigarrow k\ (\lambda k'\ x. \ulcorner e | k' \urcorner)$$

$$\ulcorner (e_1\ e_2) | k \urcorner \rightsquigarrow \ulcorner e_1 | (\lambda x_1. \ulcorner e_2 | (\lambda x_2.(x_1\ k\ x_2)) \urcorner) \urcorner$$

$$e_0 \rightsquigarrow \ulcorner e_0 | (\lambda x.x) \urcorner$$

CPS Conversion as a Program Transformation

$$\ulcorner x|k \lrcorner \rightsquigarrow k\ x$$

$$\ulcorner (\lambda x.e)|k \lrcorner \rightsquigarrow k\ (\lambda k'\ x.\ulcorner e|k' \lrcorner)$$

$$\ulcorner (e_1\ e_2)|k \lrcorner \rightsquigarrow \ulcorner e_1|(\lambda x_1.\ulcorner e_2|(\lambda x_2.(x_1\ k\ x_2)) \lrcorner) \lrcorner$$

$$e_0 \rightsquigarrow \ulcorner e_0|(\lambda x.x) \lrcorner$$

CPS Conversion as a Program Transformation

$$\ulcorner x | k \urcorner \rightsquigarrow k\ x$$

$$\ulcorner (\lambda x.e) | k \urcorner \rightsquigarrow k\ (\lambda k'\ x. \ulcorner e | k' \urcorner)$$

$$\ulcorner (e_1\ e_2) | k \urcorner \rightsquigarrow \ulcorner e_1 | (\lambda x_1. \ulcorner e_2 | (\lambda x_2.(x_1\ k\ x_2)) \urcorner) \urcorner$$

$$e_0 \rightsquigarrow \ulcorner e_0 | (\lambda x.x) \urcorner$$

CPS Conversion as a Program Transformation

$$\begin{aligned}
\ulcorner x|k \urcorner &\rightsquigarrow k\ x \\
\ulcorner (\lambda x.e)|k \urcorner &\rightsquigarrow k\ (\lambda \textcolor{red}{k'}\ x.\ulcorner e|k' \urcorner) \\
\ulcorner (e_1\ e_2)|k \urcorner &\rightsquigarrow \ulcorner e_1|(\lambda x_1.\ulcorner e_2|(\lambda x_2.(x_1\ k\ x_2))\urcorner)\urcorner \\
e_0 &\rightsquigarrow \ulcorner e_0|(\lambda x.x)\urcorner
\end{aligned}$$

CPS Conversion as a Program Transformation

$$\ulcorner x|k \lrcorner \rightsquigarrow k\ x$$

$$\ulcorner (\lambda x.e)|k \lrcorner \rightsquigarrow k\ (\lambda k'\ x.\ulcorner e|k' \lrcorner)$$

$$\ulcorner (e_1\ e_2)|k \lrcorner \rightsquigarrow \ulcorner e_1|(\lambda x_1.\ulcorner e_2|(\lambda x_2.(x_1\ k\ x_2)) \lrcorner) \lrcorner$$

$$e_0 \rightsquigarrow \ulcorner e_0|(\lambda x.x) \lrcorner$$

CPS Conversion as a Program Transformation

$$
\begin{aligned}
\ulcorner x | k \urcorner &\rightsquigarrow k\ x \\
\ulcorner (\lambda x.e) | k \urcorner &\rightsquigarrow k\ (\lambda k'\ x. \ulcorner e | k' \urcorner) \\
\ulcorner (e_1\ e_2) | k \urcorner &\rightsquigarrow \ulcorner e_1 | (\lambda x_1. \ulcorner e_2 | (\lambda x_2.(x_1\ k\ x_2)) \urcorner) \urcorner \\
e_0 &\rightsquigarrow \ulcorner e_0 | (\lambda x.x) \urcorner
\end{aligned}
$$

CPS Conversion as a Program Transformation

$$\ulcorner x|k \lrcorner \rightsquigarrow k\ x$$

$$\ulcorner(\lambda x.e)|k\lrcorner \rightsquigarrow k\ (\lambda k'\ x.\ulcorner e|k'\lrcorner)$$

$$\ulcorner(e_1\ e_2)|k\lrcorner \rightsquigarrow \ulcorner e_1|(\lambda x_1.\ulcorner e_2|(\lambda x_2.(x_1\ k\ x_2))\lrcorner)\lrcorner$$

$$e_0 \rightsquigarrow \ulcorner e_0|(\lambda x.x)\lrcorner$$

CPS Conversion as a Program Transformation

$$\ulcorner x|k \urcorner \rightsquigarrow k\ x$$

$$\ulcorner (\lambda x.e)|k \urcorner \rightsquigarrow k\ (\lambda k'\ x.\ulcorner e|k' \urcorner)$$

$$\ulcorner (e_1\ e_2)|k \urcorner \rightsquigarrow \ulcorner e_1|(\lambda x_1.\ulcorner e_2|(\lambda x_2.(x_1\ k\ x_2))\urcorner)\urcorner$$

$$e_0 \rightsquigarrow \ulcorner e_0|(\lambda x.x)\urcorner$$

CPS Conversion as a Program Transformation

$$
\begin{aligned}
\ulcorner x|k \urcorner &\rightsquigarrow k\ x\\
\ulcorner (\lambda x.e)|k \urcorner &\rightsquigarrow k\ (\lambda k'\ x.\ulcorner e|k' \urcorner)\\
\ulcorner (e_1\ e_2)|k \urcorner &\rightsquigarrow \ulcorner e_1|(\lambda x_1.\ulcorner e_2|(\lambda x_2.(x_1\ k\ x_2))\urcorner)\urcorner\\
e_0 &\rightsquigarrow \ulcorner e_0|(\lambda x.x)\urcorner
\end{aligned}
$$

CPS Conversion as a Program Transformation

$$\begin{aligned}
\ulcorner x|k \urcorner &\leadsto k\ x \\
\ulcorner (\lambda x.e)|k \urcorner &\leadsto k\ (\lambda k'\ x.\ulcorner e|k' \urcorner) \\
\ulcorner (e_1\ e_2)|k \urcorner &\leadsto \ulcorner e_1|(\lambda x_1.\ulcorner e_2|(\lambda x_2.(x_1\ k\ x_2)) \urcorner) \urcorner \\
e_0 &\leadsto \ulcorner e_0|(\lambda x.x) \urcorner
\end{aligned}$$

CPS Conversion as a Program Transformation

$$\ulcorner x|k \urcorner \rightsquigarrow k\ x$$

$$\ulcorner (\lambda x.e)|k \urcorner \rightsquigarrow k\ (\lambda k'\ x.\ulcorner e|k' \urcorner)$$

$$\ulcorner ({\color{red}e_1}\ e_2)|k \urcorner \rightsquigarrow \ulcorner {\color{red}e_1}|(\lambda x_1.\ulcorner e_2|(\lambda x_2.(x_1\ k\ x_2))\urcorner)\urcorner$$

$$e_0 \rightsquigarrow \ulcorner e_0|(\lambda x.x)\urcorner$$

CPS Conversion as a Program Transformation

$$\ulcorner x|k \urcorner \rightsquigarrow k\ x$$

$$\ulcorner(\lambda x.e)|k\urcorner \rightsquigarrow k\ (\lambda k'\ x.\ulcorner e|k'\urcorner)$$

$$\ulcorner(e_1\ e_2)|k\urcorner \rightsquigarrow \ulcorner e_1|(\lambda x_1.\ulcorner e_2|(\lambda x_2.(x_1\ k\ x_2))\urcorner)\urcorner$$

$$e_0 \rightsquigarrow \ulcorner e_0|(\lambda x.x)\urcorner$$

CPS Conversion as a Program Transformation

$$\ulcorner x|k \urcorner \leadsto k\ x$$

$$\ulcorner (\lambda x.e)|k \urcorner \leadsto k\ (\lambda k'\ x. \ulcorner e|k' \urcorner)$$

$$\ulcorner (e_1\ e_2)|k \urcorner \leadsto \ulcorner e_1|(\lambda x_1.\ulcorner e_2|(\lambda x_2.(x_1\ k\ x_2))\urcorner)\urcorner$$

$$e_0 \leadsto \ulcorner e_0|(\lambda x.x)\urcorner$$

CPS Conversion as a Program Transformation

$$\ulcorner x|k \lrcorner \leadsto k\ x$$

$$\ulcorner (\lambda x.e)|k \lrcorner \leadsto k\ (\lambda k'\ x.\ulcorner e|k' \lrcorner)$$

$$\ulcorner (e_1\ e_2)|k \lrcorner \leadsto \ulcorner e_1|(\lambda x_1.\ulcorner e_2|(\lambda x_2.(x_1\ k\ x_2))\lrcorner)\lrcorner$$

$$e_0 \leadsto \ulcorner e_0|(\lambda x.x)\lrcorner$$

CPS Conversion as a Program Transformation

$$\ulcorner x | k \urcorner \rightsquigarrow k\ x$$

$$\ulcorner (\lambda x.e) | k \urcorner \rightsquigarrow k\ (\lambda k'\ x.\ulcorner e | k' \urcorner)$$

$$\ulcorner (e_1\ e_2) | k \urcorner \rightsquigarrow \ulcorner e_1 | (\lambda x_1.\ulcorner e_2 | (\lambda x_2.(x_1\ k\ x_2)) \urcorner) \urcorner$$

$$e_0 \rightsquigarrow \ulcorner e_0 | (\lambda x.x) \urcorner$$

CPS Conversion as a Program Transformation

$$\ulcorner x|k \lrcorner \rightsquigarrow k\ x$$

$$\ulcorner (\lambda x.e)|k \lrcorner \rightsquigarrow k\ (\lambda k'\ x.\ulcorner e|k' \lrcorner)$$

$$\ulcorner (e_1\ e_2)|k \lrcorner \rightsquigarrow \ulcorner e_1|(\lambda x_1.\ulcorner e_2|(\lambda x_2.(x_1\ k\ x_2)) \lrcorner) \lrcorner$$

$$e_0 \rightsquigarrow \ulcorner e_0|(\lambda x.x) \lrcorner$$

CPS Conversion as a Program Transformation

$$
\begin{aligned}
\ulcorner x|k \urcorner &\leadsto k\ x\\
\ulcorner (\lambda x.e)|k \urcorner &\leadsto k\ (\lambda k'\ x.\ulcorner e|k' \urcorner)\\
\ulcorner ({\color{red}e_1}\ e_2)|k \urcorner &\leadsto \ulcorner e_1|(\lambda {\color{red}x_1}.\ulcorner e_2|(\lambda x_2.({\color{red}x_1}\ k\ x_2))\urcorner)\urcorner\\
e_0 &\leadsto \ulcorner e_0|(\lambda x.x)\urcorner
\end{aligned}
$$

CPS Conversion as a Program Transformation

$$\ulcorner x|k \lrcorner \leadsto k\ x$$

$$\ulcorner(\lambda x.e)|k\lrcorner \leadsto k\ (\lambda k'\ x.\ulcorner e|k'\lrcorner)$$

$$\ulcorner(e_1\ e_2)|k\lrcorner \leadsto \ulcorner e_1|(\lambda x_1.\ulcorner e_2|(\lambda x_2.(x_1\ k\ x_2))\lrcorner)\lrcorner$$

$$e_0 \leadsto \ulcorner e_0|(\lambda x.x)\lrcorner$$

CPS Conversion as a Program Transformation

$$\ulcorner x|k \lrcorner \leadsto k\ x$$

$$\ulcorner (\lambda x.e)|k \lrcorner \leadsto k\ (\lambda k'\ x.\ulcorner e|k' \lrcorner)$$

$$\ulcorner (e_1\ e_2)|k \lrcorner \leadsto \ulcorner e_1|(\lambda x_1.\ulcorner e_2|(\lambda x_2.(x_1\ k\ x_2))\lrcorner)\lrcorner$$

$$e_0 \leadsto \ulcorner e_0|(\lambda x.x)\lrcorner$$

CPS Conversion as a Program Transformation

$$\ulcorner x|k \urcorner \rightsquigarrow k\ x$$

$$\ulcorner (\lambda x.e)|k \urcorner \rightsquigarrow k\ (\lambda k'\ x.\ulcorner e|k' \urcorner)$$

$$\ulcorner (e_1\ e_2)|k \urcorner \rightsquigarrow \ulcorner e_1|(\lambda x_1.\ulcorner e_2|(\lambda x_2.(x_1\ k\ x_2))\urcorner)\urcorner$$

$$e_0 \rightsquigarrow \ulcorner e_0|(\lambda x.x)\urcorner$$

CPS Conversion as a Program Transformation

$$\ulcorner x|k \urcorner \rightsquigarrow k\ x$$

$$\ulcorner (\lambda x.e)|k \urcorner \rightsquigarrow k\ (\lambda k'\ x.\ulcorner e|k' \urcorner)$$

$$\ulcorner (e_1\ e_2)|k \urcorner \rightsquigarrow \ulcorner e_1|(\lambda x_1.\ulcorner e_2|(\lambda x_2.(x_1\ k\ x_2))\urcorner)\urcorner$$

$$e_0 \rightsquigarrow \ulcorner {\color{red}e_0}|(\lambda x.x)\urcorner$$

CPS Conversion as a Program Transformation

$$\begin{aligned}
\ulcorner x|k \urcorner &\rightsquigarrow k\ x \\
\ulcorner (\lambda x.e)|k \urcorner &\rightsquigarrow k\ (\lambda k'\ x.\ulcorner e|k' \urcorner) \\
\ulcorner (e_1\ e_2)|k \urcorner &\rightsquigarrow \ulcorner e_1|(\lambda x_1.\ulcorner e_2|(\lambda x_2.(x_1\ k\ x_2))\urcorner)\urcorner \\
e_0 &\rightsquigarrow \ulcorner e_0|({\color{red}\lambda x.x})\urcorner
\end{aligned}$$

Implementation

1. Convert source program to CPS.
2. Thread step count and limit.
3. Translate CPS to C.
4. Combine with general-purpose interruption and resumption interface and $\sqrt{\mathcal{J}}$ written in C.
5. Compile to machine code.

Treading Step Counts and Limits in CPS Conversion

$$
\begin{aligned}
\ulcorner x|k \quad \urcorner &\leadsto k \quad x\\
\ulcorner(\lambda x.e)|k \quad \urcorner &\leadsto k \quad (\lambda k \quad x.\ulcorner e|k \quad \urcorner)\\
\ulcorner(e_1\ e_2)|k \quad \urcorner &\leadsto \ulcorner e_1|(\lambda \quad x_1.\\
&\qquad \ulcorner e_2|(\lambda \quad x_2.\\
&\qquad\qquad (x_1\ k \quad x_2)),\\
&\qquad\qquad \urcorner),\\
&\qquad\qquad \urcorner
\end{aligned}
$$

Treading Step Counts and Limits in CPS Conversion

$$\ulcorner x | k \urcorner \rightsquigarrow k\ x$$

$$\ulcorner (\lambda x.e) | k \urcorner \rightsquigarrow k\ (\lambda k\ x.\ulcorner e | k \urcorner)$$

$$\ulcorner (e_1\ e_2) | k \urcorner \rightsquigarrow \ulcorner e_1 | (\lambda\ x_1.\ \ulcorner e_2 | (\lambda\ x_2.\ (x_1\ k\ x_2)), \urcorner), \urcorner$$

Treading Step Counts and Limits in CPS Conversion

$$
\begin{aligned}
\ulcorner x | k, n \ \lrcorner &\rightsquigarrow k\ (n+1)\ \ x \\
\ulcorner (\lambda x.e) | k, n \ \lrcorner &\rightsquigarrow k\ (n+1)\ \ (\lambda k\ n\ \ x.\ulcorner e | k, n\ \lrcorner) \\
\ulcorner (e_1\ e_2) | k, n \ \lrcorner &\rightsquigarrow \ulcorner e_1 | (\lambda n\ \ x_1. \\
&\qquad \ulcorner e_2 | (\lambda n\ \ x_2. \\
&\qquad\qquad (x_1\ k\ n\ \ x_2)), \\
&\qquad n\ \lrcorner), \\
&\quad (n+1)\ \lrcorner
\end{aligned}
$$

Treading Step Counts and Limits in CPS Conversion

$$
\begin{aligned}
\ulcorner x|k,n,\color{red}{l}\urcorner &\rightsquigarrow\ k\ (n+1)\ \color{red}{l}\ x\\
\ulcorner(\lambda x.e)|k,n,\color{red}{l}\urcorner &\rightsquigarrow\ k\ (n+1)\ \color{red}{l}\ (\lambda k\ n\ \color{red}{l}\ x.\ulcorner e|k,n,\color{red}{l}\urcorner)\\
\ulcorner(e_1\ e_2)|k,n,\color{red}{l}\urcorner &\rightsquigarrow\ \ulcorner e_1|(\lambda n\ \color{red}{l}\ x_1.\\
&\qquad\qquad \ulcorner e_2|(\lambda n\ \color{red}{l}\ x_2.\\
&\qquad\qquad\qquad (x_1\ k\ n\ \color{red}{l}\ x_2)),\\
&\qquad\qquad\quad n,\color{red}{l}\urcorner),\\
&\qquad\qquad (n+1),\color{red}{l}\urcorner
\end{aligned}
$$

Treading Step Counts and Limits in CPS Conversion

$$
\begin{aligned}
\ulcorner x|k,n,l\urcorner &\leadsto k\;(n+1)\;l\;x\\
\ulcorner(\lambda x.e)|k,n,l\urcorner &\leadsto k\;(n+1)\;l\;(\lambda k\;n\;l\;x.\ulcorner e|k,n,l\urcorner)\\
\ulcorner(e_1\;e_2)|k,n,l\urcorner &\leadsto \ulcorner e_1|(\lambda n\;l\;x_1.\\
&\qquad\quad \ulcorner e_2|(\lambda n\;l\;x_2.\\
&\qquad\qquad\quad (x_1\;k\;n\;l\;x_2)),\\
&\qquad\qquad n,l\urcorner),\\
&\qquad (n+1),l\urcorner
\end{aligned}
$$

$$\langle\!\langle e\rangle\!\rangle_{k,n,l} \leadsto \textbf{if}\; n=l\;\textbf{then}\;[\![k,\lambda k\;n\;l\;\_.e]\!]\;\textbf{else}\;e$$

Treading Step Counts and Limits in CPS Conversion

$$
\begin{aligned}
\ulcorner x|k,n,l\urcorner &\leadsto \langle\!\langle k\ (n+1)\ l\ x\rangle\!\rangle_{k,n,l}\\
\ulcorner(\lambda x.e)|k,n,l\urcorner &\leadsto \langle\!\langle k\ (n+1)\ l\ (\lambda k\ n\ l\ x.\ulcorner e|k,n,l\urcorner)\rangle\!\rangle_{k,n,l}\\
\ulcorner(e_1\ e_2)|k,n,l\urcorner &\leadsto \langle\!\langle \ulcorner e_1|(\lambda n\ l\ x_1.\\
&\qquad\qquad \ulcorner e_2|(\lambda n\ l\ x_2.\\
&\qquad\qquad\qquad (x_1\ k\ n\ l\ x_2)),\\
&\qquad\qquad n,l\urcorner),\\
&\qquad (n+1),l\urcorner\rangle\!\rangle_{k,n,l}
\end{aligned}
$$

$$\langle\!\langle e\rangle\!\rangle_{k,n,l} \leadsto \textbf{if}\ n=l\ \textbf{then}\ [\![k,\lambda k\ n\ l\ \_.e]\!]\ \textbf{else}\ e$$

Treading Step Counts and Limits in CPS Conversion

$$\ulcorner x|k,n,l\urcorner \rightsquigarrow \langle\!\langle k\ (n+1)\ l\ x\rangle\!\rangle_{k,n,l}$$

$$\ulcorner(\lambda x.e)|k,n,l\urcorner \rightsquigarrow \langle\!\langle k\ (n+1)\ l\ (\lambda k\ n\ l\ x.\ulcorner e|k,n,l\urcorner)\rangle\!\rangle_{k,n,l}$$

$$\begin{aligned}\ulcorner(e_1\ e_2)|k,n,l\urcorner \rightsquigarrow \langle\!\langle \ulcorner e_1|(\lambda n\ l\ x_1.\\ \ulcorner e_2|(\lambda n\ l\ x_2.\\ (x_1\ k\ n\ l\ x_2)),\\ n,l\urcorner),\\ (n+1),l\urcorner\rangle\!\rangle_{k,n,l}\end{aligned}$$

$$\vdots$$

$$\langle\!\langle e\rangle\!\rangle_{k,n,l} \rightsquigarrow \textbf{if}\ n=l\ \textbf{then}\ [\![k,\lambda k\ n\ l\ \_.e]\!]\ \textbf{else}\ e$$

Implementation of the General-Purpose Interruption and Resumption Interface with Threaded CPS

$$\begin{aligned}
\text{PRIMOPS}\, f\ x &= \mathcal{A}\ (\lambda n\ l\ v.n)\ 0\ \infty\ f\ x\\
\text{INTERRUPT}\, f\ x\ l &= \mathcal{A}\ (\lambda n\ l\ v.v)\ 0\ l\, f\ x\\
\text{RESUME}\ [\![k, f]\!] &= \mathcal{A}\ k\ 0\ \infty\ f\ \bot
\end{aligned}$$

Implementation of the General-Purpose Interruption and Resumption Interface with Threaded CPS

$$\text{PRIMOPS}\ \textcolor{red}{f}\ x = \mathcal{A}\ (\lambda n\ l\ v.n)\ 0\ \infty\ \textcolor{red}{f}\ x$$
$$\text{INTERRUPT}\ f\ x\ l = \mathcal{A}\ (\lambda n\ l\ v.v)\ 0\ l\ f\ x$$
$$\text{RESUME}\ [\![k, f]\!] = \mathcal{A}\ k\ 0\ \infty\ f\ \bot$$

Implementation of the General-Purpose Interruption and Resumption Interface with Threaded CPS

$$\text{PRIMOPS}\, f\ \color{red}{x} = \mathcal{A}\ (\lambda n\ l\ v.n)\ 0\ \infty\ f\ \color{red}{x}$$

$$\text{INTERRUPT}\, f\ x\ l = \mathcal{A}\ (\lambda n\ l\ v.v)\ 0\ l\ f\ x$$

$$\text{RESUME}\ [\![k, f]\!] = \mathcal{A}\ k\ 0\ \infty\ f\ \bot$$

Implementation of the General-Purpose Interruption and Resumption Interface with Threaded CPS

$$\text{PRIMOPS}\, f\ x = \mathcal{A}\ (\lambda n\ l\ v.n)\ {\color{red}0}\ \infty\ f\ x$$
$$\text{INTERRUPT}\, f\ x\ l = \mathcal{A}\ (\lambda n\ l\ v.v)\ 0\ l\ f\ x$$
$$\text{RESUME}\ [\![k,f]\!] = \mathcal{A}\ k\ 0\ \infty\ f\ \bot$$

Implementation of the General-Purpose Interruption and Resumption Interface with Threaded CPS

$$\text{PRIMOPS}\, f\; x = \mathcal{A}\; (\lambda n\; l\; v.n)\; 0\; \infty\; f\; x$$

$$\text{INTERRUPT}\, f\; x\; l = \mathcal{A}\; (\lambda n\; l\; v.v)\; 0\; l\; f\; x$$

$$\text{RESUME}\; [\![k, f]\!] = \mathcal{A}\; k\; 0\; \infty\; f\; \bot$$

Implementation of the General-Purpose Interruption and Resumption Interface with Threaded CPS

$$\text{PRIMOPS}\, f\ x = \mathcal{A}\ (\lambda n\ l\ v.n)\ 0\ \infty\ f\ x$$
$$\text{INTERRUPT}\, f\ x\ l = \mathcal{A}\ (\lambda n\ l\ v.v)\ 0\ l\ f\ x$$
$$\text{RESUME}\ [\![k, f]\!] = \mathcal{A}\ k\ 0\ \infty\ f\ \bot$$

Implementation of the General-Purpose Interruption and Resumption Interface with Threaded CPS

$$\begin{aligned}
\text{PRIMOPS}\; f\; x &= \mathcal{A}\;(\lambda n\; l\; v.n)\; 0\; \infty\; f\; x\\
\text{INTERRUPT}\; f\; x\; l &= \mathcal{A}\;(\lambda n\; l\; v.v)\; 0\; l\; f\; x\\
\text{RESUME}\; [\![k, f]\!] &= \mathcal{A}\; k\; 0\; \infty\; f\; \bot
\end{aligned}$$

Implementation of the General-Purpose Interruption and Resumption Interface with Threaded CPS

$$\textsc{primops}\ f\ x = \mathcal{A}\ (\lambda n\ l\ v.n)\ 0\ \infty\ f\ x$$

$$\textsc{interrupt}\ f\ {\color{red}x}\ l = \mathcal{A}\ (\lambda n\ l\ v.v)\ 0\ l\ f\ {\color{red}x}$$

$$\textsc{resume}\ [\![k, f]\!] = \mathcal{A}\ k\ 0\ \infty\ f\ \bot$$

Implementation of the General-Purpose Interruption and Resumption Interface with Threaded CPS

$$
\begin{aligned}
\text{PRIMOPS}\, f\ x &= \mathcal{A}\ (\lambda n\ l\ v.n)\ 0\ \infty\ f\ x\\
\text{INTERRUPT}\, f\ x\ l &= \mathcal{A}\ (\lambda n\ l\ v.v)\ \color{red}{0}\ l\, f\ x\\
\text{RESUME}\ [\![k, f]\!] &= \mathcal{A}\ k\ 0\ \infty\ f\ \bot
\end{aligned}
$$

Implementation of the General-Purpose Interruption and Resumption Interface with Threaded CPS

$$\text{PRIMOPS}\, f\ x = \mathcal{A}\ (\lambda n\ l\ v.n)\ 0\ \infty\ f\ x$$
$$\text{INTERRUPT}\, f\ x\ {\color{red}l} = \mathcal{A}\ (\lambda n\ l\ v.v)\ 0\ {\color{red}l}\ f\ x$$
$$\text{RESUME}\ [\![k,f]\!] = \mathcal{A}\ k\ 0\ \infty\ f\ \bot$$

Implementation of the General-Purpose Interruption and Resumption Interface with Threaded CPS

$$\begin{aligned}
\text{PRIMOPS}\, f\ x &= \mathcal{A}\ (\lambda n\ l\ v.n)\ 0\ \infty\ f\ x\\
\text{INTERRUPT}\, f\ x\ l &= \mathcal{A}\ (\lambda n\ l\ v.v)\ 0\ l\ f\ x\\
\text{RESUME}\ [\![k,f]\!] &= \mathcal{A}\ k\ 0\ \infty\ f\ \bot
\end{aligned}$$

Implementation of the General-Purpose Interruption and Resumption Interface with Threaded CPS

$$\text{PRIMOPS}\, f\ x = \mathcal{A}\ (\lambda n\ l\ v.n)\ 0\ \infty\ f\ x$$
$$\text{INTERRUPT}\, f\ x\ l = \mathcal{A}\ (\lambda n\ l\ v.v)\ 0\ l\ f\ x$$
$$\text{RESUME}\ [\![k, f]\!] = \mathcal{A}\ k\ 0\ \infty\ f\ \bot$$

Implementation of the General-Purpose Interruption and Resumption Interface with Threaded CPS

$$\text{PRIMOPS}\, f\ x = \mathcal{A}\ (\lambda n\ l\ v.n)\ 0\ \infty\ f\ x$$
$$\text{INTERRUPT}\, f\ x\ l = \mathcal{A}\ (\lambda n\ l\ v.v)\ 0\ l\ f\ x$$
$$\text{RESUME}\ [\![k, f]\!] = \mathcal{A}\ k\ 0\ \infty\ f\ \bot$$

Implementation of the General-Purpose Interruption and Resumption Interface with Threaded CPS

$$\text{PRIMOPS}\,f\;x = \mathcal{A}\;(\lambda n\;l\;v.n)\;0\;\infty\;f\;x$$
$$\text{INTERRUPT}\,f\;x\;l = \mathcal{A}\;(\lambda n\;l\;v.v)\;0\;l\,f\;x$$
$$\text{RESUME}\;[\![k,f]\!] = \mathcal{A}\;k\;0\;\infty\;f\;\bot$$

Implementation of the General-Purpose Interruption and Resumption Interface with Threaded CPS

$$\text{PRIMOPS}\, f\ x = \mathcal{A}\ (\lambda n\ l\ v.n)\ 0\ \infty\ f\ x$$
$$\text{INTERRUPT}\, f\ x\ l = \mathcal{A}\ (\lambda n\ l\ v.v)\ 0\ l\ f\ x$$
$$\text{RESUME}\ [\![k, f]\!] = \mathcal{A}\ k\ {\color{red}0}\ \infty\ f\ \bot$$

Implementation of the General-Purpose Interruption and Resumption Interface with Threaded CPS

$$\text{PRIMOPS}\, f\; x = \mathcal{A}\; (\lambda n\; l\; v.n)\; 0\; \infty\; f\; x$$
$$\text{INTERRUPT}\, f\; x\; l = \mathcal{A}\; (\lambda n\; l\; v.v)\; 0\; l\; f\; x$$
$$\text{RESUME}\, [\![k, f]\!] = \mathcal{A}\; k\; 0\; \infty\; f\; \bot$$

Implementation of the General-Purpose Interruption and Resumption Interface with Threaded CPS

$$
\begin{aligned}
\text{PRIMOPS}\, f\ x &= \mathcal{A}\ (\lambda n\ l\ v.n)\ 0\ \infty\ f\ x\\
\text{INTERRUPT}\, f\ x\ l &= \mathcal{A}\ {\color{red}(\lambda n\ l\ v.v)}\ 0\ l\, f\ x\\
\text{RESUME}\ [\![k, f]\!] &= \mathcal{A}\ {\color{red}k}\ 0\ \infty\ f\ \bot
\end{aligned}
$$

Implementation of the General-Purpose Interruption and Resumption Interface with Threaded CPS

$$\begin{aligned}
\text{PRIMOPS}\, f\ x &= \mathcal{A}\ (\lambda n\ l\ v.n)\ 0\ \infty\ f\ x\\
\text{INTERRUPT}\, f\ x\ l &= \mathcal{A}\ (\lambda n\ l\ {\color{red}v}.{\color{red}v})\ 0\ l\, f\ x\\
\text{RESUME}\ [\![k, f]\!] &= \mathcal{A}\ k\ 0\ \infty\ f\ \bot
\end{aligned}$$

Implementation

1. Convert source program to CPS.
2. Thread step count and limit.
3. Translate CPS to C.
4. Combine with general-purpose interruption and resumption interface and $\sqrt{\mathcal{J}}$ written in C.
5. Compile to machine code.

Code Generation

$$
\begin{aligned}
\mathcal{S}\,\pi\,() &= \texttt{null\_constant}\\
\mathcal{S}\,\pi\,\textbf{true} &= \texttt{true\_constant}\\
\mathcal{S}\,\pi\,\textbf{false} &= \texttt{false\_constant}\\
\mathcal{S}\,\pi\,(c_1,c_2) &= \texttt{cons}\,\texttt{((}\mathcal{S}\,\pi\,c_1\texttt{),\ \ (}\mathcal{S}\,\pi\,c_1\texttt{))}\\
\mathcal{S}\,\pi\,n & \\
\mathcal{S}\,\pi\,\texttt{`k'} &= \texttt{continuation}\\
\mathcal{S}\,\pi\,\texttt{`n'} &= \texttt{count}\\
\mathcal{S}\,\pi\,\texttt{`l'} &= \texttt{limit}\\
\mathcal{S}\,\pi\,\texttt{`x'} &= \texttt{argument}\\
\mathcal{S}\,\pi\,x &= \texttt{as\_closure(target)->environment[}\pi\,x\texttt{]}
\end{aligned}
$$

Code Generation

$\mathcal{S}\,\pi\,(\lambda_3 n\, l\, x.e)$ = ({

```
    thing function(thing target,
                   thing count,
                   thing limit,
                   thing argument) {
    return (S (φ e) e);
    }
    thing lambda = (thing)GC_malloc(sizeof(struct {
      enum tag tag;
      struct {
        thing (*function)();
        unsigned n;
        thing environment[|φ e|];
      }))
    }
    set_closure(lambda);
    as_closure(lambda)->function = &function;
    as_closure(lambda)->n = |φ e|;
    as_closure(lambda)->environment[0] = S π (φ e)_0
    ⋮
    as_closure(lambda)->environment[|φ e| − 1] = S π (φ e)_{|φ e|−1}
    lambda;
  })
```

Code Generation

$\mathcal{S}\,\pi\,(\lambda_4 k\,n\,l\,x.e)$ = ({

```
              thing function(thing target,
                             thing continuation,
                             thing count,
                             thing limit,
                             thing argument) {
              return ($\mathcal{S}\,(\phi\,e)\,e$);
              }
              thing lambda = (thing)GC_malloc(sizeof(struct {
                enum tag tag;
                struct {
                  thing (*function)();
                  unsigned n;
                  thing environment[$|\phi\,e|$];
                }))
              }
              set_closure(lambda);
              as_closure(lambda)->function = &function;
              as_closure(lambda)->n = $|\phi\,e|$;
              as_closure(lambda)->environment[0] = $\mathcal{S}\,\pi\,(\phi\,e)_0$
              ⋮
              as_closure(lambda)->environment[$|\phi\,e|-1$] = $\mathcal{S}\,\pi\,(\phi\,e)_{|\phi\,e|-1}$
              lambda;
            })
```

Code Generation

$$\mathcal{S}\,\pi\,(e_1\;e_2\;e_3\;e_4) = \texttt{continuation\_apply}\texttt{(}(\mathcal{S}\,\pi\,e_1)\texttt{,}\;(\mathcal{S}\,\pi\,e_2)\texttt{,}\;(\mathcal{S}\,\pi\,e_3)\texttt{,}\;(\mathcal{S}\,\pi\,e_4)\texttt{)}$$

$$\mathcal{S}\,\pi\,(e_1\;e_2\;e_3\;e_4\;e_5) = \texttt{converted\_apply}\texttt{(}(\mathcal{S}\,\pi\,e_1)\texttt{,}\;(\mathcal{S}\,\pi\,e_2)\texttt{,}\;(\mathcal{S}\,\pi\,e_3)\texttt{,}\;(\mathcal{S}\,\pi\,e_4)\texttt{,}\;(\mathcal{S}\,\pi\,e_5)\texttt{)}$$

$$\mathcal{S}\,\pi\,(\textbf{if}\;e_1\;\textbf{then}\;e_2\;\textbf{else}\;e_3) = \texttt{(!is\_false(}(\mathcal{S}\,\pi\,e_1)\texttt{)?}(\mathcal{S}\,\pi\,e_2)\texttt{:}(\mathcal{S}\,\pi\,e_3)\texttt{)}$$

$$\mathcal{S}\,\pi\,(\diamond e) = (\mathcal{N}\,\diamond)\texttt{(}(\mathcal{S}\,\pi\,e)\texttt{)}$$

$$\mathcal{S}\,\pi\,(e_1 \bullet e_2) = (\mathcal{N}\,\bullet)\texttt{(}(\mathcal{S}\,\pi\,e_1)\texttt{,}\;(\mathcal{S}\,\pi\,e_2)\texttt{)}$$

$$\mathcal{S}\,\pi\,(\overrightarrow{\mathcal{J}}\;e_1\;e_2\;e_3) = (\mathcal{N}\,\overrightarrow{\mathcal{J}})\texttt{(}(\mathcal{S}\,\pi\,e_1)\texttt{,}\;(\mathcal{S}\,\pi\,e_2)\texttt{,}\;(\mathcal{S}\,\pi\,e_3)\texttt{)}$$

$$\mathcal{S}\,\pi\,(\overleftarrow{\mathcal{J}}\;e_1\;e_2\;e_3) = (\mathcal{N}\,\overleftarrow{\mathcal{J}})\texttt{(}(\mathcal{S}\,\pi\,e_1)\texttt{,}\;(\mathcal{S}\,\pi\,e_2)\texttt{,}\;(\mathcal{S}\,\pi\,e_3)\texttt{)}$$

$$\mathcal{S}\,\pi\,(\stackrel{\checkmark}{\mathcal{J}}\;e_1\;e_2\;e_3) = (\mathcal{N}\,\stackrel{\checkmark}{\mathcal{J}})\texttt{(}(\mathcal{S}\,\pi\,e_1)\texttt{,}\;(\mathcal{S}\,\pi\,e_2)\texttt{,}\;(\mathcal{S}\,\pi\,e_3)\texttt{)}$$

Implementation

1. Convert source program to CPS.
2. Thread step count and limit.
3. Translate CPS to C.
4. Combine with general-purpose interruption and resumption interface and $\checkmark{}$ $\mathcal{J}$ written in C.
5. Compile to machine code.

Implementation of the General-Purpose Interruption and Resumption Interface with Threaded CPS

$$
\begin{aligned}
\text{PRIMOPS}\, f\; x &= \mathcal{A}\; (\lambda n\; l\; v.n)\; 0\; \infty\; f\; x\\
\text{INTERRUPT}\, f\; x\; l &= \mathcal{A}\; (\lambda n\; l\; v.v)\; 0\; l\; f\; x\\
\text{RESUME}\, [\![k,f]\!] &= \mathcal{A}\; k\; 0\; \infty\; f\; \bot
\end{aligned}
$$

Implementation of the General-Purpose Interruption and Resumption Interface with Threaded CPS

written in C

```
static thing lambda_expression_that_returns_x
(thing f, thing n, thing l, thing x) {
  return x;
}

static thing lambda_expression_that_returns_n
(thing f, thing n, thing l, thing x) {
  return n;
}

static thing lambda_expression_that_resumes
(thing f, thing continuation, thing n, thing l, thing x) {
  if (!is_interrupt(x)) internal_error();
  return converted_apply(as_interrupt(x)->closure,
                         as_interrupt(x)->continuation,
                         make_real(0.0),
                         l,
                         null_constant);
}
```

Implementation of the General-Purpose Interruption and Resumption Interface with Threaded CPS

```
static unsigned long primops(thing f, thing x) {
  thing result = converted_apply(f,
                                 continuation_that_returns_n,
                                 make_real(0.0),
                                 make_real(HUGE_VAL),
                                 x);
  else if (is_real(result)) return (unsigned long)as_real(result);
}

static thing interrupt(thing f, thing x, thing l) {
  thing result = converted_apply(f,
                                 continuation_that_returns_x,
                                 make_real(0.0),
                                 l,
                                 x);
  if (!is_interrupt(result)) internal_error();
  return result;
}
```

Algorithm for Divide-and-Conquer Checkpointing

To compute $(y, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ f\ x\ \grave{y}$:

base case ($f\ x$ fast):	$(y, \grave{x}) = \overleftarrow{\mathcal{J}}\ f\ x\ \grave{y}$	(step 0)
inductive case:	$h \circ g = f$	(step 1)
	$z = g\ x$	(step 2)
	$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ h\ z\ \grave{y}$	(step 3)
	$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z}$	(step 4)

Algorithm for Divide-and-Conquer Checkpointing

written in C

```
static thing checkpoint_starj(thing f, thing x, thing y_cotangent)
{
  thing loop(thing f, thing x, thing y_cotangent, unsigned long l) {
    if (l<=base_case_duration) return ternary_starj(f, x, y_cotangent);
    else {
      thing u = interrupt(f, x, make_real(l/2));
      thing y_u_cotangent = loop(closure_that_resumes, u, y_cotangent, l-l/2);
      if (!is_pair(y_u_cotangent)) internal_error();
      thing u_x_cotangent =
        loop(make_closure_for_interrupt(f, l/2),
             x,
             as_pair(y_u_cotangent)->cdr,
             l/2);
      if (!is_pair(u_x_cotangent)) internal_error();
      return cons(as_pair(y_u_cotangent)->car,
                  as_pair(u_x_cotangent)->cdr);
    }
  }
  return loop(f, x, y_cotangent, primops(f, x));
}
```

Implementation

1. Convert source program to CPS.
2. Thread step count and limit.
3. Translate CPS to C.
4. Combine with general-purpose interruption and resumption interface and $\sqrt{\mathcal{J}}$ written in C.
5. Compile to machine code.

Determinant Example

```
(define (car (cons car cdr)) car)

(define (cdr (cons car cdr)) cdr)

(define (matrix-rows a)
 (if (null? a) 0 (+ (matrix-rows (cdr a)) 1)))

(define (list-ref l i)
 (if (zero? i) (car l) (list-ref (cdr l) (- i 1))))

(define (matrix-ref a i j) (list-ref (list-ref a i) j))

(define (list-set l i x)
 (if (zero? i)
     (cons x (cdr l))
     (cons (car l) (list-set (cdr l) (- i 1) x))))

(define (matrix-set a i j x)
 (list-set a i (list-set (list-ref a i) j x)))

(define (map-n f n)
 (if (zero? n) '() (cons (f (- n 1)) (map-n f (- n 1)))))

(define (identity-matrix n)
 (map-n (lambda (i) (map-n (lambda (j) (if (= i j) 1 0)) n)) n))
```

Determinant Example

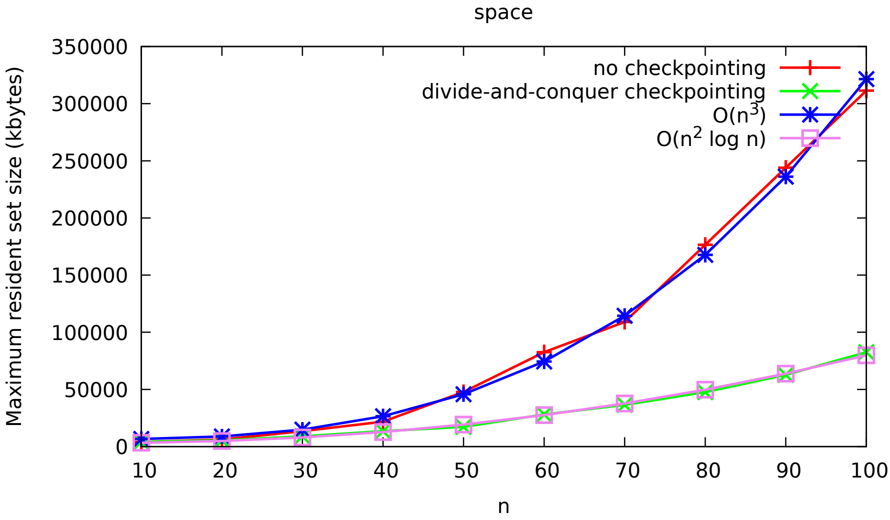
```
(define (determinant a)
 (let ((n (matrix-rows a)))
  (let loop ((i 0) (b a) (d 1))
   (if (= i n)
       d
       (let* ((c (matrix-ref b i i))
              (b (let loop ((j i) (b b))
                  (if (= j n)
                      b
                      (loop (+ j 1) (matrix-set b i j (/ (matrix-ref b i j) c)))))))
        (loop (+ i 1)
              (let loop ((j (+ i 1)) (b b))
               (if (= j n)
                   b
                   (loop (+ j 1)
                         (let ((e (matrix-ref b j i)))
                          (let loop ((k (+ i 1)) (b b))
                           (if (= k n)
                               b
                               (loop (+ k 1)
                                     (matrix-set b j k
                                                 (- (matrix-ref b j k)
                                                    (* e (matrix-ref b i k)))))))))))
              (* d c)))))))

(write-real (determinant (cdr (checkpoint-*j determinant (identity-matrix (read-real)) 1))))
```

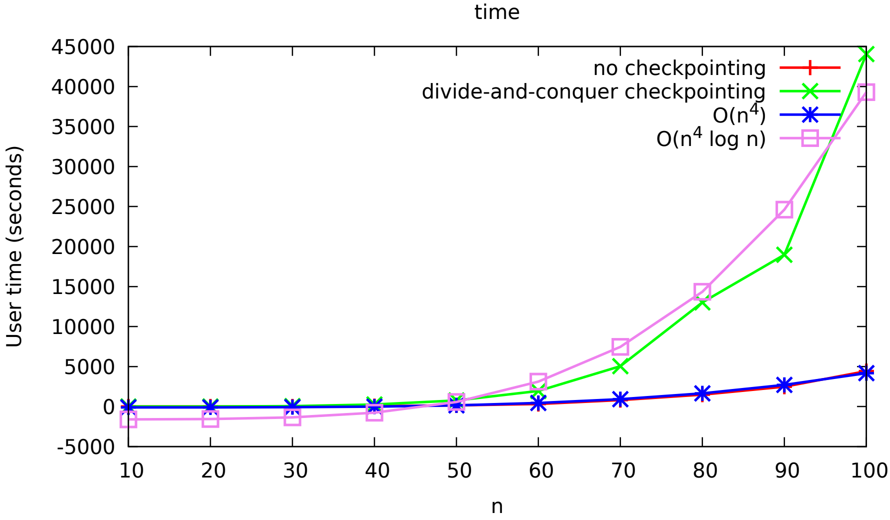
Complexity Analysis

	space	time
primal	$O(n^2)$	$O(n^4)$
no checkpointing	$O(n^3)$	$O(n^4)$
divide-and-conquer checkpointing	$O(n^2 \log n)$	$O(n^4 \log n)$

Space Usage of the Determinant Example



Time Usage of the Determinant Example



Three Reference Implementations

Three Reference Implementations

1. Interpreter using CPS evaluator

Three Reference Implementations

1. Interpreter using CPS evaluator
2. Hybrid compiler/interpreter using CPS conversion followed by direct-style evaluator

Three Reference Implementations

1. Interpreter using CPS evaluator
2. Hybrid compiler/interpreter using CPS conversion followed by direct-style evaluator
3. Compiler using CPS conversion followed by translation to C

Three Reference Implementations

1. Interpreter using CPS evaluator
2. Hybrid compiler/interpreter using CPS conversion followed by direct-style evaluator
3. Compiler using CPS conversion followed by translation to C

All three support exact same source language. No exceptions.
Same space and time complexity. Differ only in constant factors.

Three Reference Implementations

1. Interpreter using CPS evaluator
2. Hybrid compiler/interpreter using CPS conversion followed by direct-style evaluator
3. Compiler using CPS conversion followed by translation to C

All three support exact same source language. No exceptions.
Same space and time complexity. Differ only in constant factors.

Could add FFI bindings to GPU Tensor library.
Could serve as reference for rewrite to PYTHON,. . . .

Three Reference Implementations

1. Interpreter using CPS evaluator
2. Hybrid compiler/interpreter using CPS conversion followed by direct-style evaluator
3. Compiler using CPS conversion followed by translation to C

All three support exact same source language. No exceptions.
Same space and time complexity. Differ only in constant factors.

Could add FFI bindings to GPU Tensor library.
Could serve as reference for rewrite to PYTHON,. . . .

All three documented in detail in manuscript.
`http://arxiv.org/abs/1708.06799`

Three Reference Implementations

1. Interpreter using CPS evaluator
2. Hybrid compiler/interpreter using CPS conversion followed by direct-style evaluator
3. Compiler using CPS conversion followed by translation to C

All three support exact same source language. No exceptions.
Same space and time complexity. Differ only in constant factors.

Could add FFI bindings to GPU Tensor library.
Could serve as reference for rewrite to PYTHON,. . . .

All three documented in detail in manuscript.
`http://arxiv.org/abs/1708.06799`

Will release when manuscript is accepted.

Take-Home Message

Divide-and-Conquer checkpointing

Take-Home Message

Divide-and-Conquer checkpointing

- is traditionally formulated around loop iterations

Take-Home Message

Divide-and-Conquer checkpointing

- is traditionally formulated around loop iterations
- but can be extended to arbitrary code

Take-Home Message

Divide-and-Conquer checkpointing

- is traditionally formulated around loop iterations
- but can be extended to arbitrary code
- that doesn't have same-size iterations of a single loop

Take-Home Message

Divide-and-Conquer checkpointing

- is traditionally formulated around loop iterations
- but can be extended to arbitrary code
- that doesn't have same-size iterations of a single loop
- using CPS to make arbitrary code look like it does.

Take-Home Message

Divide-and-Conquer checkpointing

- is traditionally formulated around loop iterations
- but can be extended to arbitrary code
- that doesn't have same-size iterations of a single loop
- using CPS to make arbitrary code look like it does.

metaphor: a CPU is an instruction-execution loop

Thank You