What does CPS have to do with deep learning?

Jeffrey Mark Siskind, qobi@purdue.edu

PURDUE UNIVERSITY

Northeastern University, Monday 11 September 2017

Joint work with Barak Avrum Pearlmutter

Automatic Differentiation (AD)

$$f = f_t \circ f_{t-1} \circ \cdots \circ f_2 \circ f_1$$

Automatic Differentiation (AD)

$$f = f_t \circ f_{t-1} \circ \cdots \circ f_2 \circ f_1$$

$$f'(x_1) = f_t'(x_t) \times f_{t-1}'(x_{t-1}) \times \cdots \times f_2'(x_2) \times f_1'(x_1)$$

Automatic Differentiation (AD)

$$
\begin{aligned}
f &= f_t \circ f_{t-1} \circ \cdots \circ f_2 \circ f_1 \\
f'(x_1) &= f_t'(x_t) \times f_{t-1}'(x_{t-1}) \times \cdots \times f_2'(x_2) \times f_1'(x_1) \\
f'(x_1)^\top &= f_1'(x_1)^\top \times f_2(x_2)^\top \times \cdots \times f_{t-1}'(x_{t-1})^\top \times f_t'(x_t)^\top
\end{aligned}
$$

Automatic Differentiation (AD)

$$f = f_t \circ f_{t-1} \circ \cdots \circ f_2 \circ f_1$$

$$\acute{y} = f'(x_1) \times \acute{x}_1 = f'_t(x_t) \times f'_{t-1}(x_{t-1}) \times \cdots \times f'_2(x_2) \times f'_1(x_1) \times \acute{x}_1$$

$$\grave{x}_1 = f'(x_1)^\top \times \grave{y} = f'_1(x_1)^\top \times f_2(x_2)^\top \times \cdots \times f'_{t-1}(x_{t-1})^\top \times f'_t(x_t)^\top \times \grave{y}$$

Automatic Differentiation (AD)

$$f = f_t \circ f_{t-1} \circ \cdots \circ f_2 \circ f_1$$

$$x_2 = f_1(x_1)$$

Automatic Differentiation (AD)

$$f = f_t \circ f_{t-1} \circ \cdots \circ f_2 \circ f_1$$

$$x_3 = f_2(f_1(x_1)) = f_2(x_2)$$

Automatic Differentiation (AD)

$$f = f_t \circ f_{t-1} \circ \cdots \circ f_2 \circ f_1$$

$$x_t = f_{t-1}(\ldots f_2(f_1(x_1))\ldots) = f_{t-1}(x_{t-1})$$

Automatic Differentiation (AD)

$$f = f_t \circ f_{t-1} \circ \cdots \circ f_2 \circ f_1$$

$$y = f_t(f_{t-1}(\ldots f_2(f_1(x_1))\ldots)) = f_t(x_t)$$

Automatic Differentiation (AD)

$$\acute{x}_2 = f_1'(x_1) \times \acute{x}_1$$

$$x_1$$

Automatic Differentiation (AD)

$$\acute{x}_3 = f_2'(x_2) \times f_1'(x_1) \times \acute{x}_1$$

$$x_2 = f_1(x_1)$$

Automatic Differentiation (AD)

$$\acute{x}_t = f'_{t-1}(x_{t-1}) \times \cdots \times f'_2(x_2) \times f'_1(x_1) \times \acute{x}_1$$

$$x_{t-1} = f_{t-2}(\ldots f_2(f_1(x_1))\ldots) = f_{t-2}(x_{t-2})$$

Automatic Differentiation (AD)

$$\acute{y} = f'(x_1) \times \acute{x}_1 = f'_t(x_t) \times f'_{t-1}(x_{t-1}) \times \cdots \times f'_2(x_2) \times f'_1(x_1) \times \acute{x}_1$$

$$x_t = \quad f_{t-1}(\ldots f_2(f_1(x_1))\ldots) \quad = f_{t-1}(x_{t-1})$$

Automatic Differentiation (AD)

x_1

x_1

Automatic Differentiation (AD)

$$x_2 = f_1(x_1)$$

$$x_2, x_1$$

Automatic Differentiation (AD)

$$x_{t-1} = f_{t-2}(\ldots f_2(f_1(x_1))\ldots) = f_{t-2}(x_{t-2})$$

$$x_{t-1}, \ldots, x_2, x_1$$

Automatic Differentiation (AD)

$$x_t = f_{t-1}(\ldots f_2(f_1(x_1))\ldots) = f_{t-1}(x_{t-1})$$

$$x_t, x_{t-1}, \ldots, x_2, x_1$$

Automatic Differentiation (AD)

$$\grave{x}_t = f_t'(x_t)^\top \times \grave{y}$$

$$x_t, x_{t-1}, \ldots, x_2, x_1$$

Automatic Differentiation (AD)

$$\grave{x}_{t-1} = f'_{t-1}(x_{t-1})^\top \times f'_t(x_t)^\top \times \grave{y}$$

$$x_{t-1}, \ldots, x_2, x_1$$

Automatic Differentiation (AD)

$$\grave{x}_2 = \qquad\qquad f_2(x_2)^\top \times \cdots \times f'_{t-1}(x_{t-1})^\top \times f'_t(x_t)^\top \times \grave{y}$$

x_2, x_1

Automatic Differentiation (AD)

$$\dot{x}_1 = f'(x_1)^\top \times \dot{y} = f_1'(x_1)^\top \times f_2(x_2)^\top \times \cdots \times f_{t-1}'(x_{t-1})^\top \times f_t'(x_t)^\top \times \dot{y}$$

x_1

Automatic Differentiation (AD)

$$f = f_t \circ f_{t-1} \circ \cdots \circ f_2 \circ f_1$$

$$\acute{y} = f'(x_1) \times \acute{x}_1 = f'_t(x_t) \times f'_{t-1}(x_{t-1}) \times \cdots \times f'_2(x_2) \times f'_1(x_1) \times \acute{x}_1$$

$$\grave{x}_1 = f'(x_1)^\top \times \grave{y} = f'_1(x_1)^\top \times f_2(x_2)^\top \times \cdots \times f'_{t-1}(x_{t-1})^\top \times f'_t(x_t)^\top \times \grave{y}$$

Automatic Differentiation (AD)

$$\mathbf{f} = \mathbf{f}_t \circ \mathbf{f}_{t-1} \circ \cdots \circ \mathbf{f}_2 \circ \mathbf{f}_1$$

$$\acute{\mathbf{y}} = \mathbf{F}'(\mathbf{x}_1) \times \acute{\mathbf{x}}_1 = \mathbf{F}'_t(\mathbf{x}_t) \times \mathbf{F}'_{t-1}(\mathbf{x}_{t-1}) \times \cdots \times \mathbf{F}'_2(\mathbf{x}_2) \times \mathbf{F}'_1(\mathbf{x}_1) \times \acute{\mathbf{x}}_1$$

$$\grave{\mathbf{x}}_1 = \mathbf{F}'(\mathbf{x}_1)^\top \times \grave{\mathbf{y}} = \mathbf{F}'_1(\mathbf{x}_1)^\top \times \mathbf{F}'_2(\mathbf{x}_2)^\top \times \cdots \times \mathbf{F}'_{t-2}(\mathbf{x}_{t-2})^\top \times \mathbf{F}'_{t-1}(\mathbf{x}_{t-1})^\top \times \grave{\mathbf{y}}$$

Essence of Forward Mode in Scheme

```
(define-record-type dual-number (fields primal tangent))

(define old-f f)

(define old-g g)

(define (f x)
 (make-dual-number
  (old-f (dual-number-primal x))
  (old-* (df/dx (dual-number-primal x)) (dual-number-tangent x))))

(define (g x y)
 (make-dual-number
  (old-g (dual-number-primal x) (dual-number-primal y))
  (old-+ (old-* (dg/dx (dual-number-primal x) (dual-number-primal y))
                (dual-number-tangent x))
         (old-* (dg/dy (dual-number-primal x) (dual-number-primal y))
                (dual-number-tangent y)))))

(define (forward-mode f x x-tangent)
 (let* ((input (map* make-dual-number x x-tangent))
        (output (f input)))
  (list (map* dual-number-primal output) (map* dual-number-tangent output))))
```

Essence of Reverse Mode in Scheme

```
(define-record-type tape (fields primal factors tapes (mutable fanout) (mutable cotangent)))

(define old-f f)

(define old-g g)

(define (f x)
 (tape-fanout-set! x (old-+ (tape-fanout x) 1))
 (make-tape (old-f (tape-primal x)) (list (df/dx (tape-primal x))) (list x) 0 0))

(define (g x y)
 (tape-fanout-set! x (old-+ (tape-fanout x) 1))
 (tape-fanout-set! y (old-+ (tape-fanout y) 1))
 (make-tape (old-g (tape-primal x) (tape-primal y))
            (list (dg/dx (tape-primal x) (tape-primal y)) (dg/dy (tape-primal x) (tape-primal y)))
            (list x y)
            0
            0))

(define (reverse-phase! cotangent tape)
 (tape-cotangent-set! tape (old-+ (tape-cotangent tape) cotangent))
 (tape-fanout-set! tape (old-- (tape-fanout tape) 1))
 (when (old-zero? (tape-fanout tape))
  (let ((cotangent (tape-cotangent tape)))
   (for-each (lambda (factor tape) (reverse-phase! (old-* cotangent factor) tape))
             (tape-factors tape) (tape-tapes tape)))))

(define (reverse-mode f x y-cotangent)
 (let* ((input (map* (lambda (x) (make-tape x '() '() 0 0)) x))
        (ouput (f input)))
  (for-each* reverse-phase! y-cotangent ouput)
  (list (map* tape-primal ouput) (map* tape-cotangent input))))
```

Typical Case

- $f : \mathbb{R}^n \to \mathbb{R}$

Typical Case

- $f : \mathbb{R}^n \to \mathbb{R}$
- Jacobian is $1 \times n$, aka ∇f

Typical Case

- $f : \mathbb{R}^n \to \mathbb{R}$
- Jacobian is $1 \times n$, aka ∇f
- computing ∇f takes n calls to forward mode with $\acute{x}_1$ being basis vectors

Typical Case

- $f : \mathbb{R}^n \to \mathbb{R}$
- Jacobian is $1 \times n$, aka ∇f
- computing ∇f takes n calls to forward mode with $\acute{x}_1$ being basis vectors
- computing ∇f with reverse mode requires a tape with t entries

Typical Case

- $f : \mathbb{R}^n \to \mathbb{R}$
- Jacobian is $1 \times n$, aka ∇f
- computing ∇f takes n calls to forward mode with $\acute{x}_1$ being basis vectors
- computing ∇f with reverse mode requires a tape with t entries
- computing ∇f with forward mode entails an increase of $O(1)$ space but $O(n)$ time over computing f

Typical Case

- $f : \mathbb{R}^n \to \mathbb{R}$
- Jacobian is $1 \times n$, aka ∇f
- computing ∇f takes n calls to forward mode with $\acute{x}_1$ being basis vectors
- computing ∇f with reverse mode requires a tape with t entries
- computing ∇f with forward mode entails an increase of $O(1)$ space but $O(n)$ time over computing f
- computing ∇f with reverse mode entails an increase of $O(t)$ space but $O(1)$ time over computing f

Typical Case

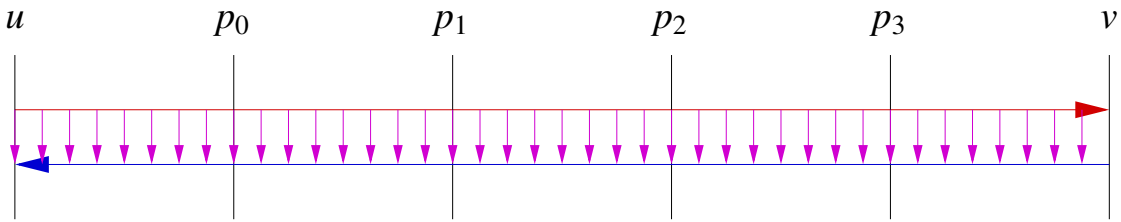
- $f : \mathbb{R}^n \to \mathbb{R}$
- Jacobian is $1 \times n$, aka ∇f
- computing ∇f takes n calls to forward mode with $\acute{x}_1$ being basis vectors
- computing ∇f with reverse mode requires a tape with t entries
- computing ∇f with forward mode entails an increase of $O(1)$ space but $O(n)$ time over computing f
- computing ∇f with reverse mode entails an increase of $O(t)$ space but $O(1)$ time over computing f
- both n and t are large

Typical Case

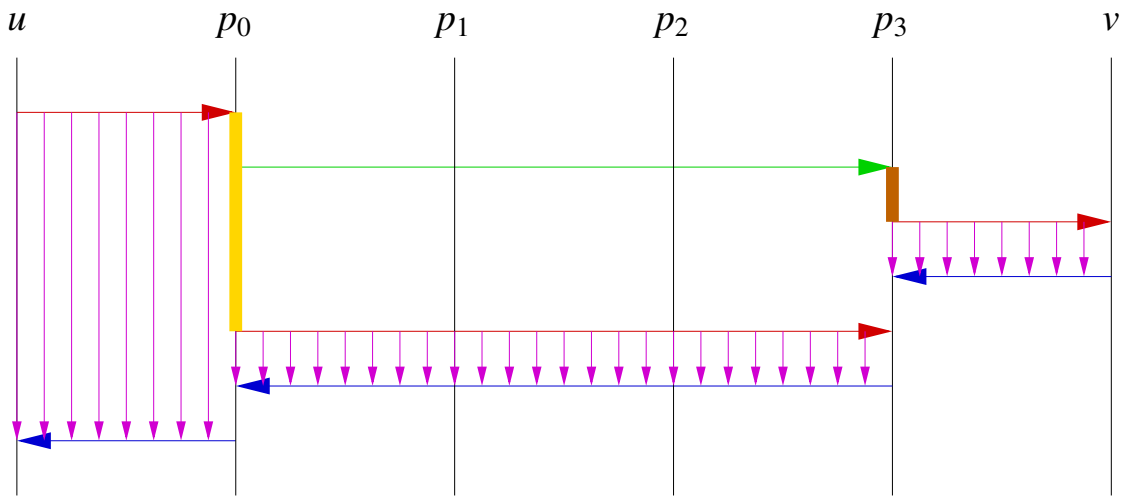
- $f : \mathbb{R}^n \to \mathbb{R}$
- Jacobian is $1 \times n$, aka ∇f
- computing ∇f takes n calls to forward mode with $\acute{x}_1$ being basis vectors
- computing ∇f with reverse mode requires a tape with t entries
- computing ∇f with forward mode entails an increase of $O(1)$ space but $O(n)$ time over computing f
- computing ∇f with reverse mode entails an increase of $O(t)$ space but $O(1)$ time over computing f
- both n and t are large
- today: computing ∇f with divide-and-conquer checkpointing reverse mode entails an increase of $O(\log t)$ space and $O(\log t)$ time over computing f

Checkpointing

Checkpointing

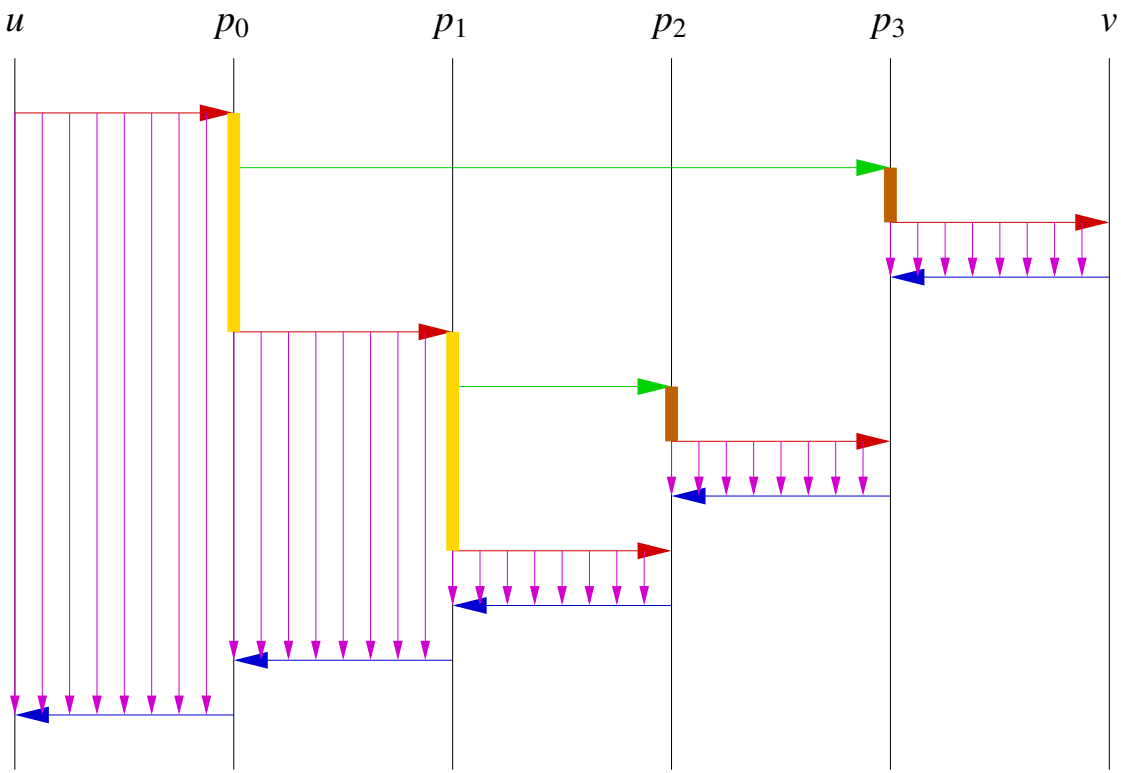


Checkpointing



$[p_0, p_3)$ and $[p_3, v)$ not simultaneously live
$[u, p_0)$ and $[p_0, p_3)$ simultaneously live
$[u, p_0)$ and $[p_3, v)$ simultaneously live

Checkpointing



$[p_1, p_2)$, $[p_2, p_3)$ and $[p_3, v)$ not simultaneously live
$[p_0, p_1)$ and either $[p1, p_2)$ or $[p_2, p_3)$ simultaneously live
$[u, p_0)$ and either $[p_0, p1)$, $[p_1, p_2)$, $[p_2, p_3)$, and $[p_3, v)$ simultaneously live

Specifying Checkpoints

program interval Execution intervals of specified program intervals constitute checkpoints.

subroutine call site Execution intervals of specified subroutine call sites constitute checkpoints.

subroutine body Execution intervals of specified subroutine bodies constitute checkpoints (Volin & Ostrovskii, 1985).

Specifying Checkpoints

program interval specified via pragma

```
subroutine p(a, b)
...
end

subroutine q(a, b)
...
end

subroutine r(a, b)
...
end

subroutine s(a, b)
...
end
```

```
      subroutine f(x, y)
      call p(x, u)
c$ad checkpoint-start
      call q(u, v)
      call r(v, w)
c$ad checkpoint-end
      call s (w, y)
      end
```

Specifying Checkpoints

subroutine call site specified via pragma

```
subroutine p(a, b)
...
end

subroutine q(a, b)
...
end

subroutine r(a, b)
...
end

subroutine s(a, b)
...
end
```

```
      subroutine f(x, y)
      call p(x, u)
c$ad nocheckpoint
      call q(u, v)
c$ad nocheckpoint
      call r(v, w)
      call s (w, y)
      end
```

Specifying Checkpoints

subroutine body specified via pragma

```
      subroutine p(a, b)
      ...
      end

c$ad nocheckpoint
      subroutine q(a, b)
      ...
      end

c$ad nocheckpoint
      subroutine r(a, b)
      ...
      end

      subroutine s(a, b)
      ...
      end
```

```
subroutine f(x, y)
call p(x, u)
call q(u, v)
call r(v, w)
call s (w, y)
end
```

Specifying Checkpoints

subroutine body specified via command line

```
subroutine p(a, b)
...
end

subroutine q(a, b)
...
end

subroutine r(a, b)
...
end

subroutine s(a, b)
...
end
```

```
subroutine f(x, y)
call p(x, u)
call q(u, v)
call r(v, w)
call s (w, y)
end
```

```
-nocheckpoint "q,r"
```

Specifying Checkpoints

program interval → subroutine call site

```
      subroutine f(x, y)
      call p(x, u)
c$ad checkpoint-start
      call q(u, v)
      call r(v, w)
c$ad checkpoint-end
      call s (w, y)
      end
```

```
subroutine g(u, v, w)
call q(u, v)
call r(v, w)
end

subroutine f(x, y)
call p(x, u)
call g(u, v, w)
call s (w, y)
end
```

Specifying Checkpoints

subroutine call site → subroutine body

```
      subroutine p(a, b)
      ...
      end

      subroutine f(x, y)
c$ad nocheckpoint
      call p(x, u)
      call p(u, v)
      end
```

```
      subroutine p(a, b)
      ...
      end

c$ad nocheckpoint
      subroutine pno(a, b)
      ...
      end

      subroutine f(x, y)
      call pno(x, u)
      call p(u, v)
      end
```

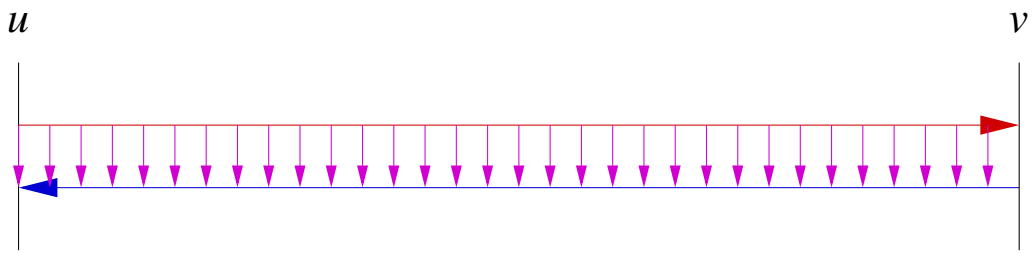
Specifying Checkpoints

subroutine body → program interval

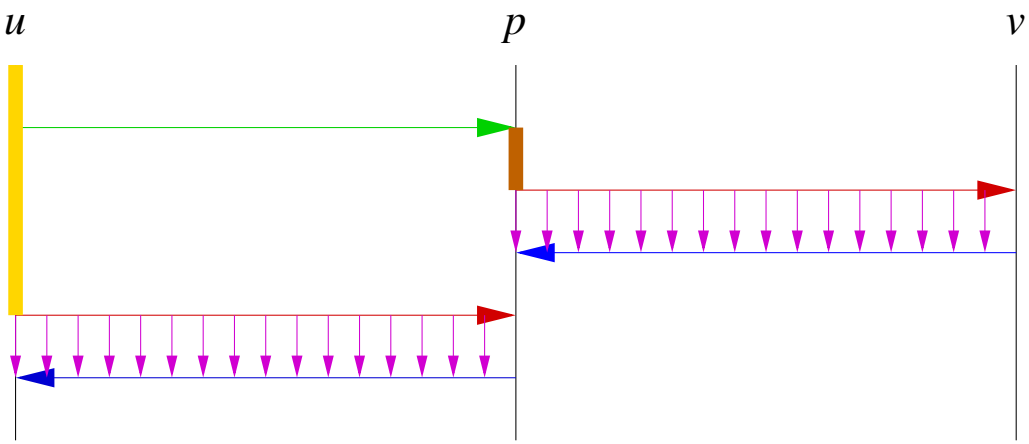
```
c checkpoint
      subroutine p(a, b)
      ...
      end
```

```
      subroutine p(a, b)
c$ad checkpoint-start
      ...
c$ad checkpoint-end
      end
```

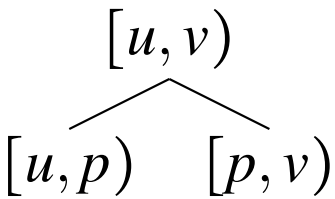
Divide-and-Conquer Checkpointing



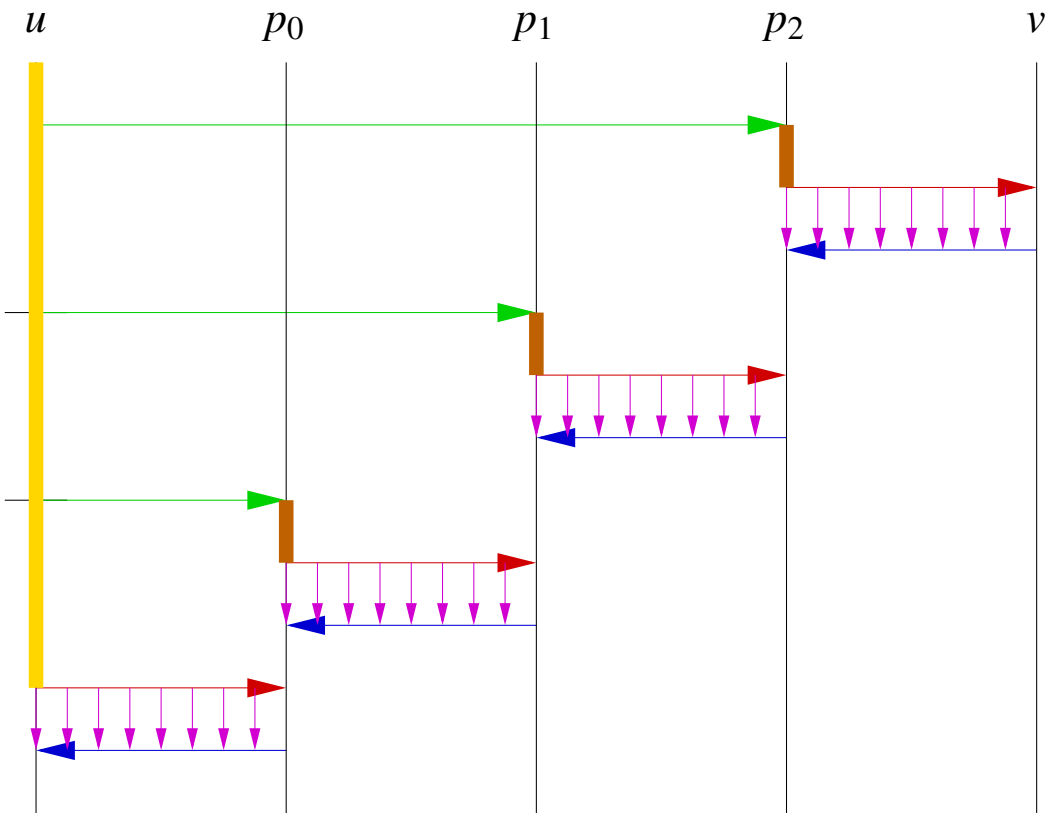
Divide-and-Conquer Checkpointing



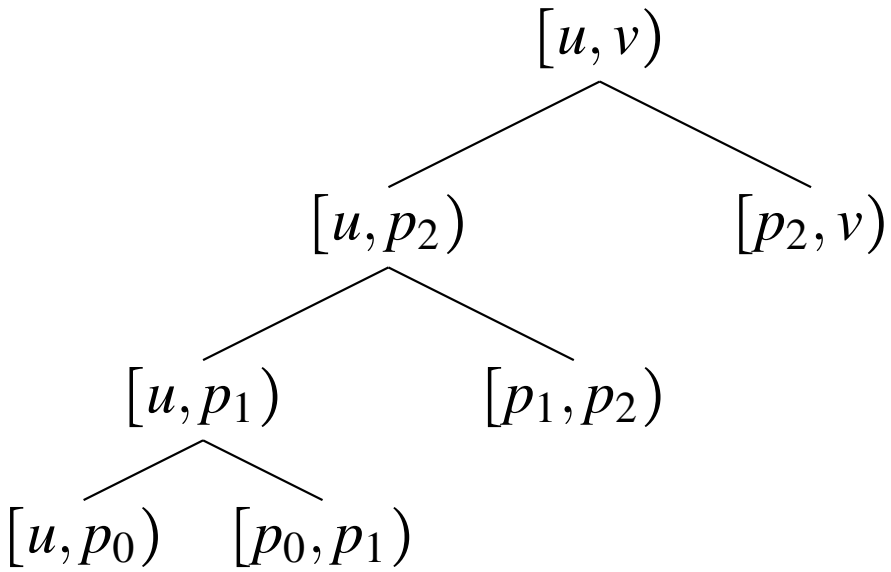
Divide-and-Conquer Checkpointing



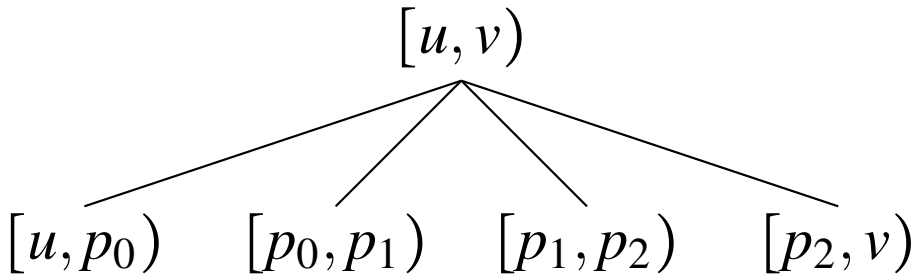
Divide-and-Conquer Checkpointing



Divide-and-Conquer Checkpointing



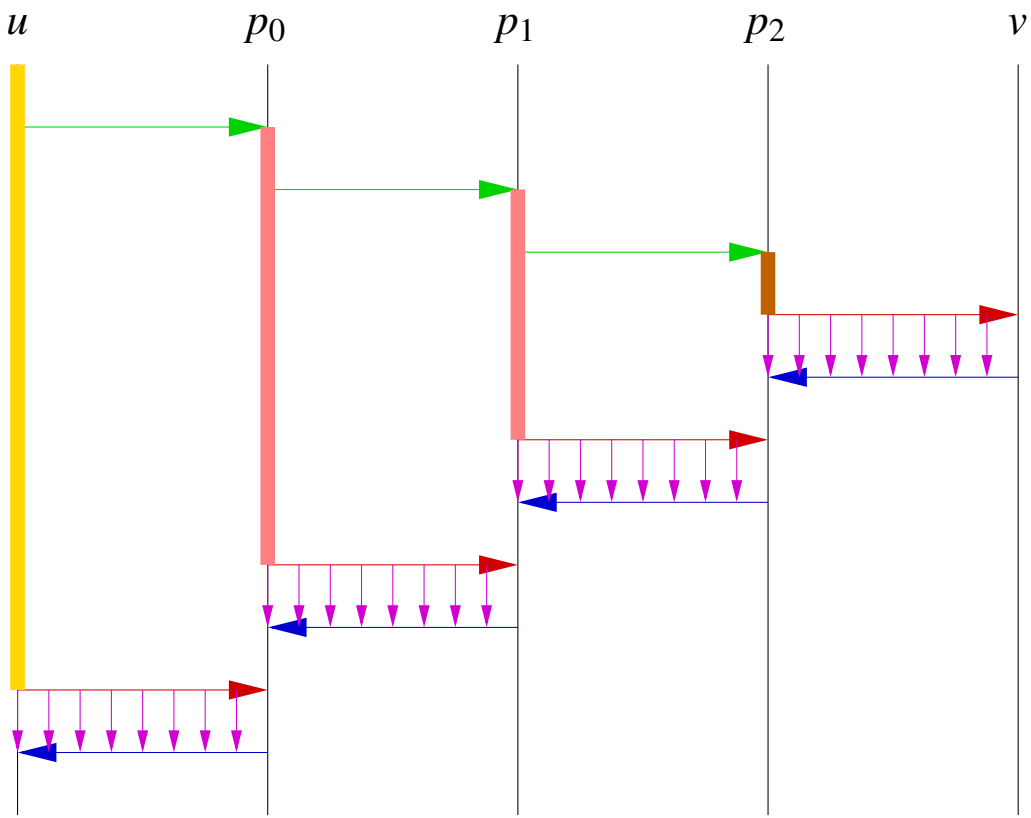
Divide-and-Conquer Checkpointing



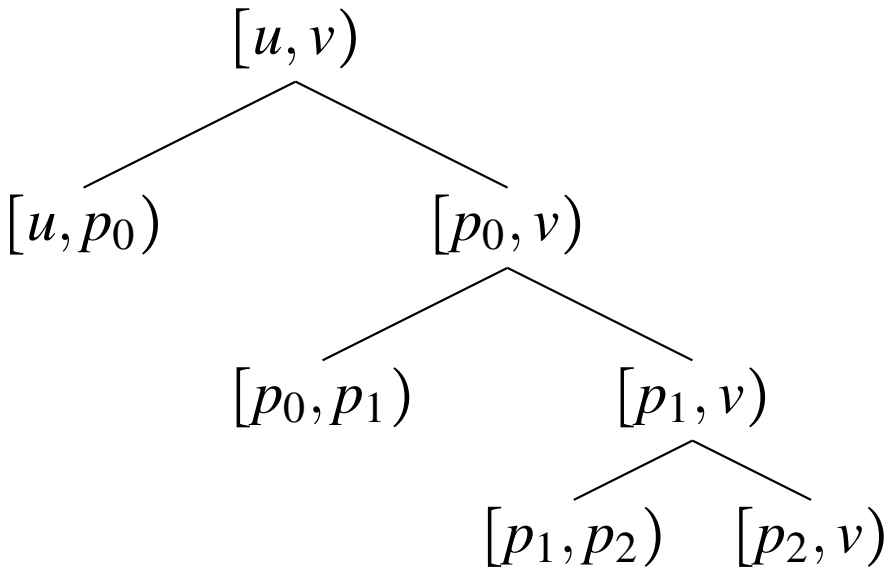
Divide-and-Conquer Checkpointing

more recomputation
quadratic
$O(t)$ green lines
each taking $O(t)$
but only one live snapshot
so $O(1)$ space

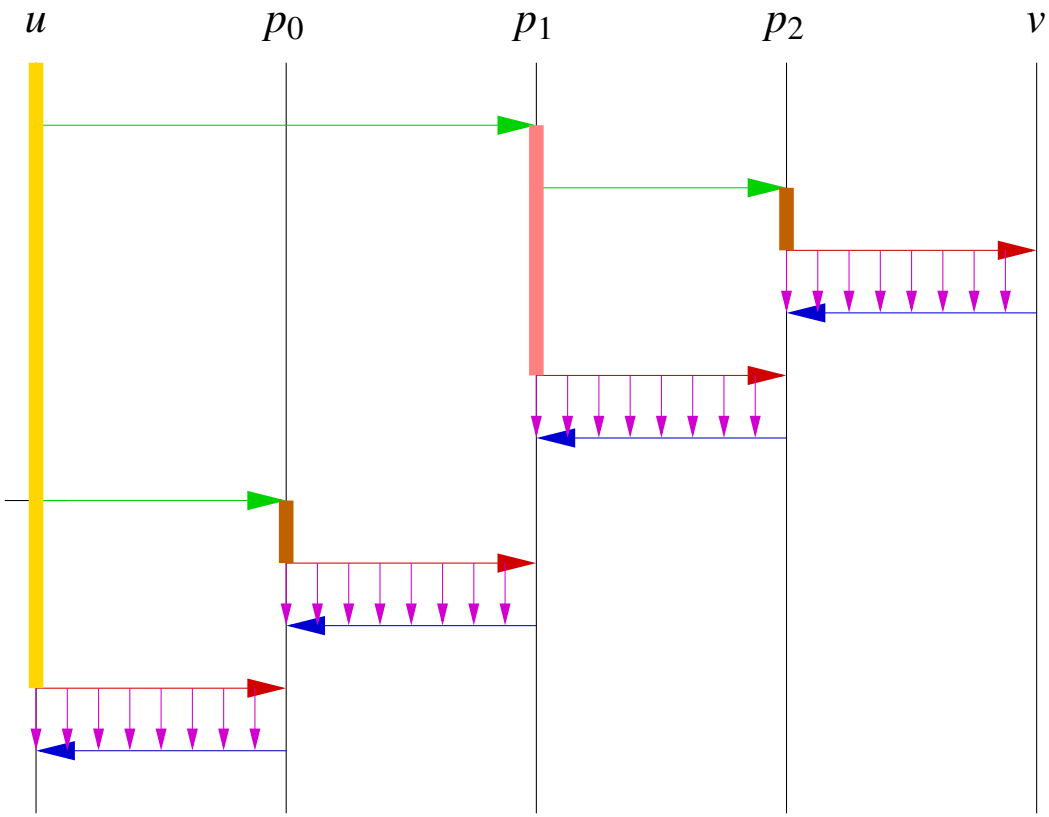
Divide-and-Conquer Checkpointing



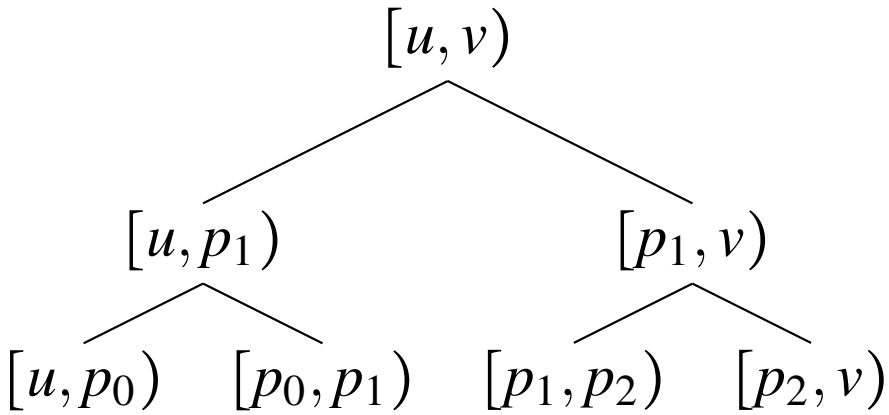
Divide-and-Conquer Checkpointing



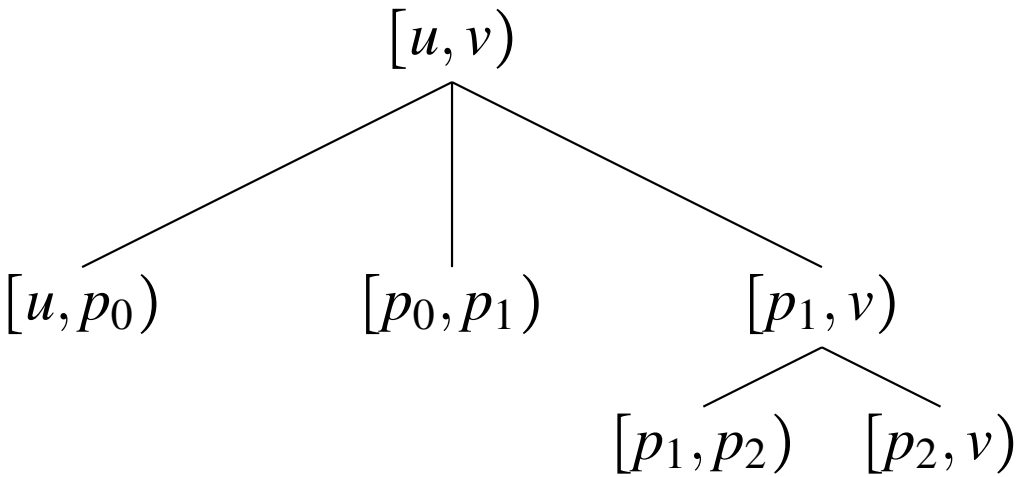
Divide-and-Conquer Checkpointing



Divide-and-Conquer Checkpointing



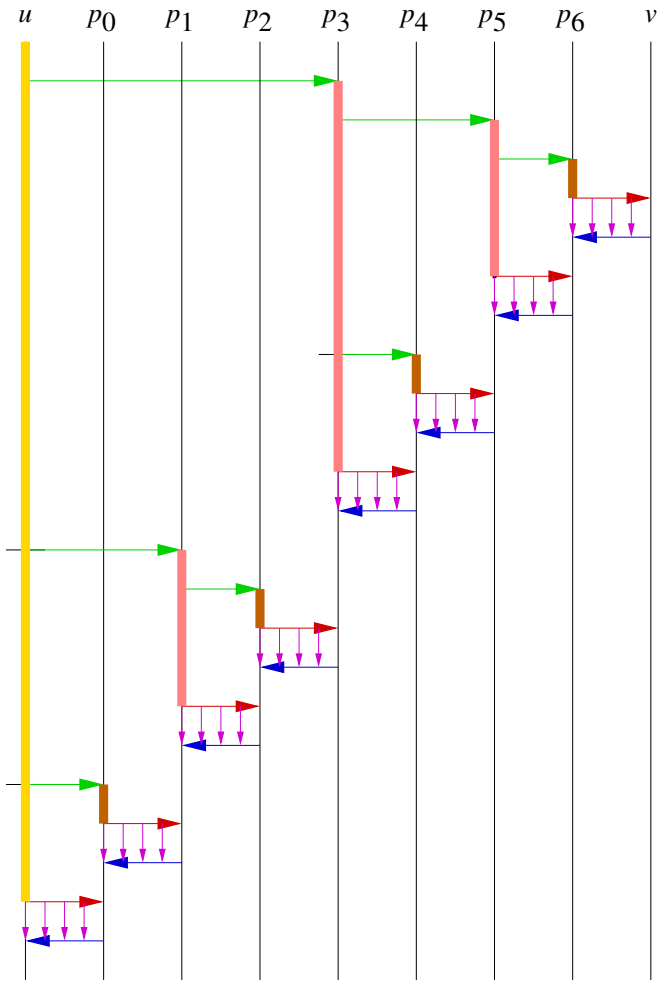
Divide-and-Conquer Checkpointing



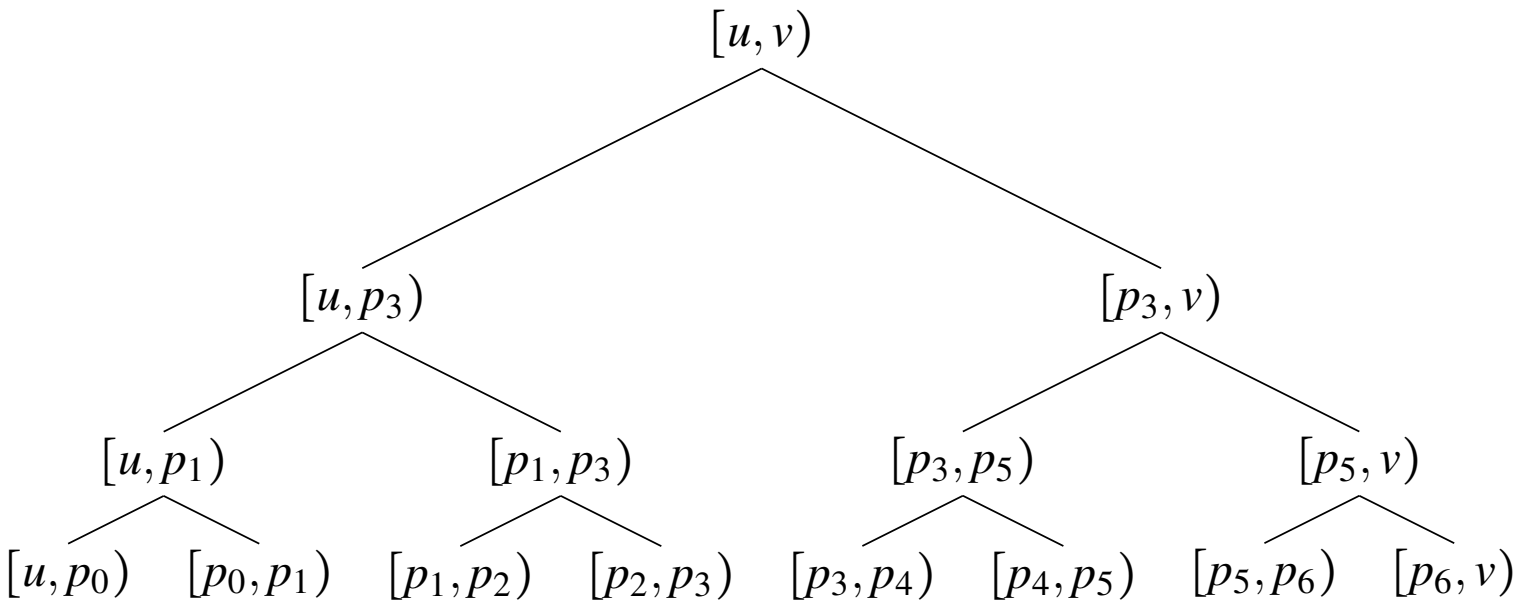
Divide-and-Conquer Checkpointing

more space
$O(t)$ live yellow/pink snapshots
but total length of green is $O(t)$

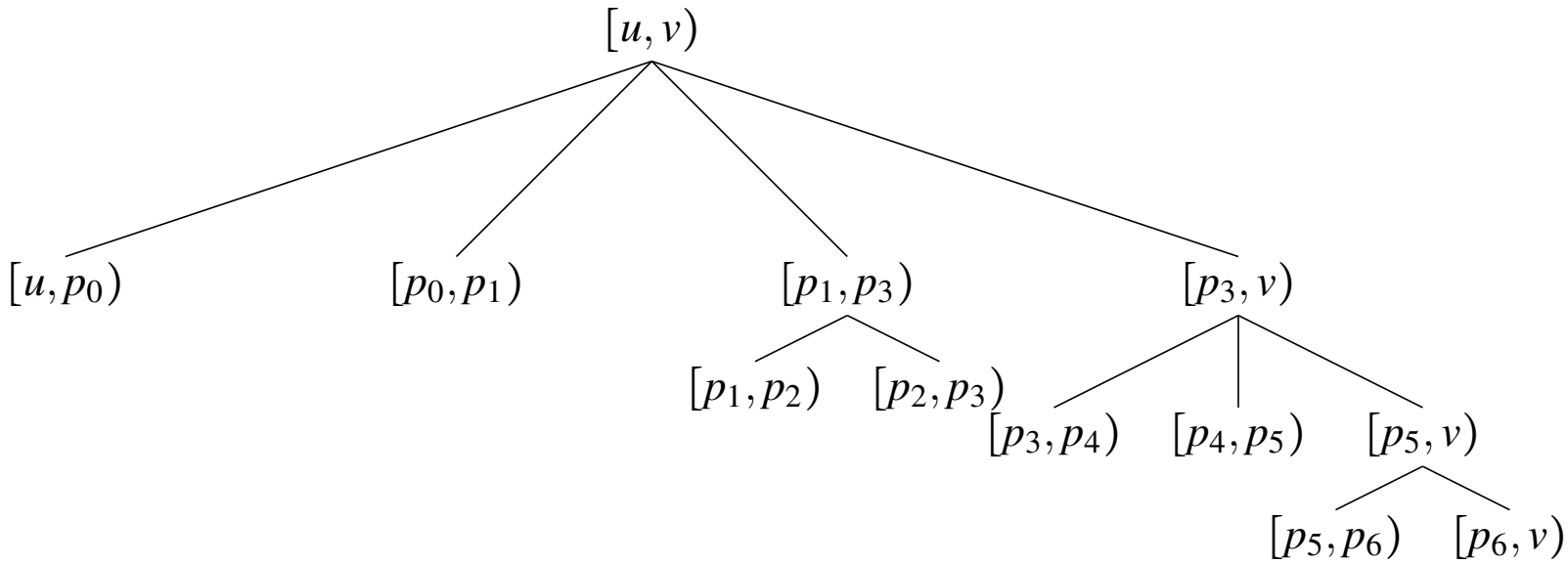
Divide-and-Conquer Checkpointing



Divide-and-Conquer Checkpointing



Divide-and-Conquer Checkpointing



Divide-and-Conquer Checkpointing

$O(\log t)$ live yellow/pink snapshots
$O(t \log t)$ total length of green

Two Divide-and-Conquer Algorithms

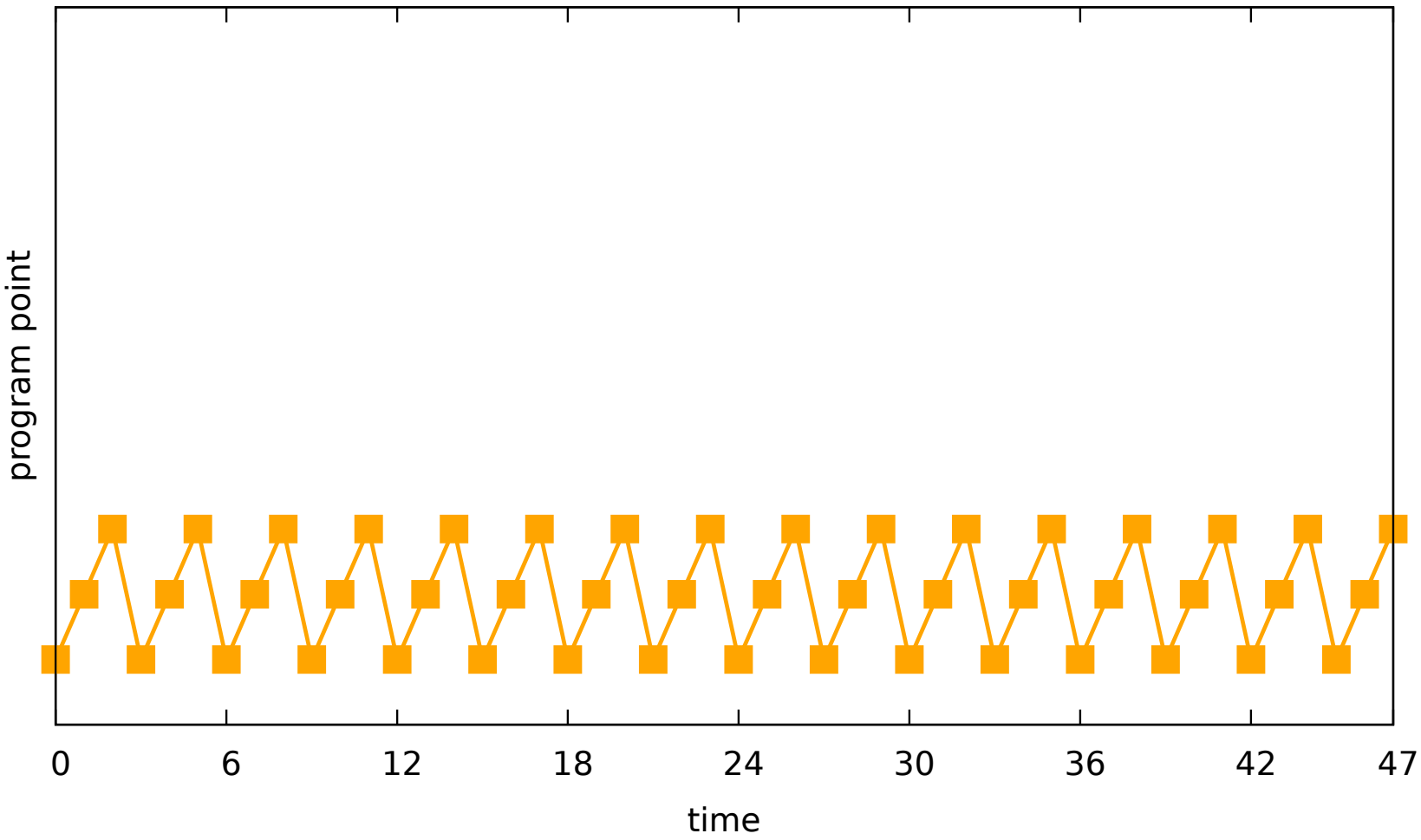
binary	An algorithm that constructs a binary checkpoint tree.
treeverse	The algorithm from Griewank (1992, Figs. 2 and 3) that constructs an n-ary checkpoint tree.

Specifying Divide-and-Conquer Checkpointing

```
      subroutine f(x, y)
      n = 100003
      y = x
c$ad binomial-ckp n+1 30 1
      do i = 1, n
         m = l(x, i)
         do j = 1, m
            y = y*y
            y = sqrt(y)
         end do
      end do
      end
```

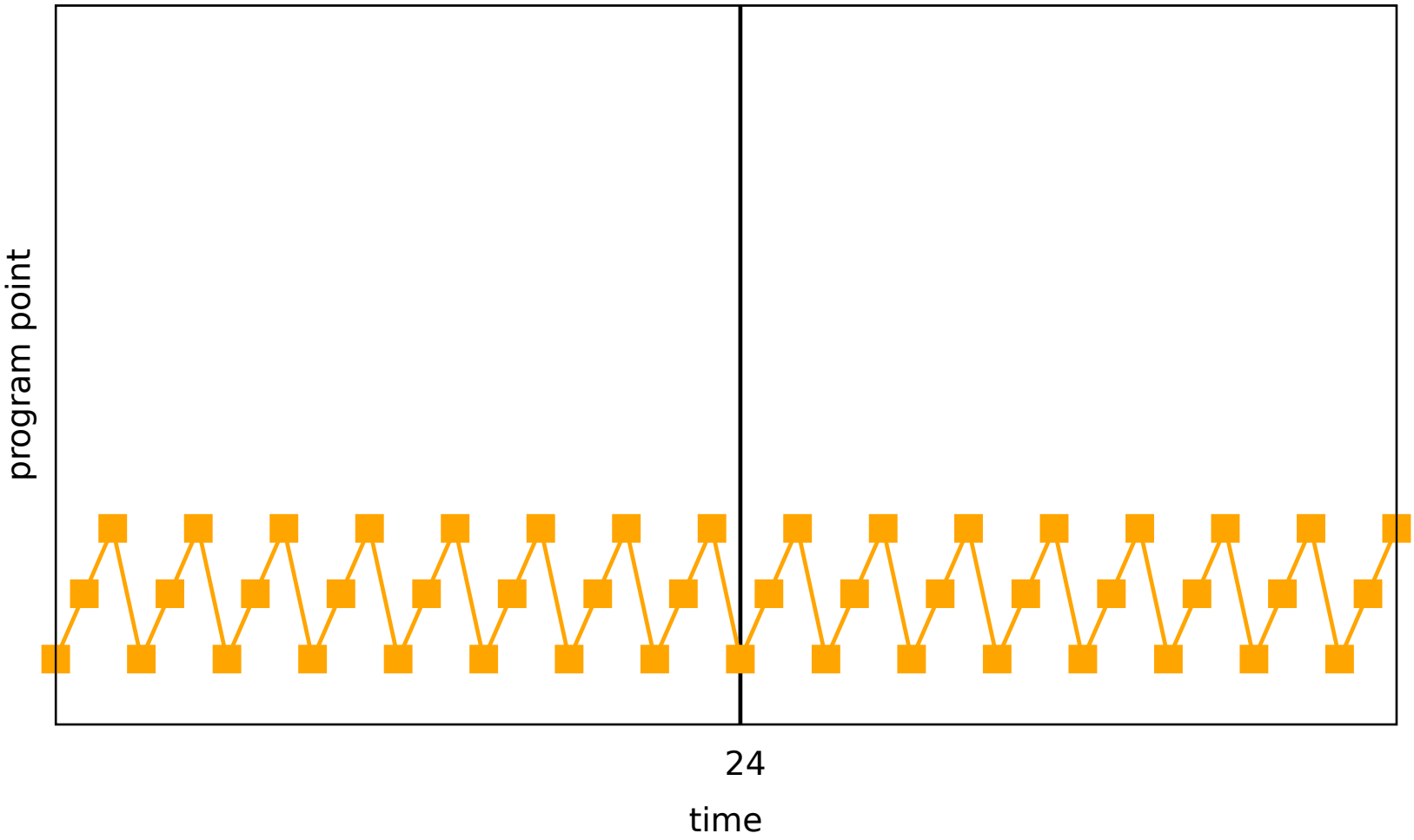
l is chosen to be data dependent on x and i so that on average l is $O(1)$ but in worst case is $O(n)$

Execution Trace of Loop



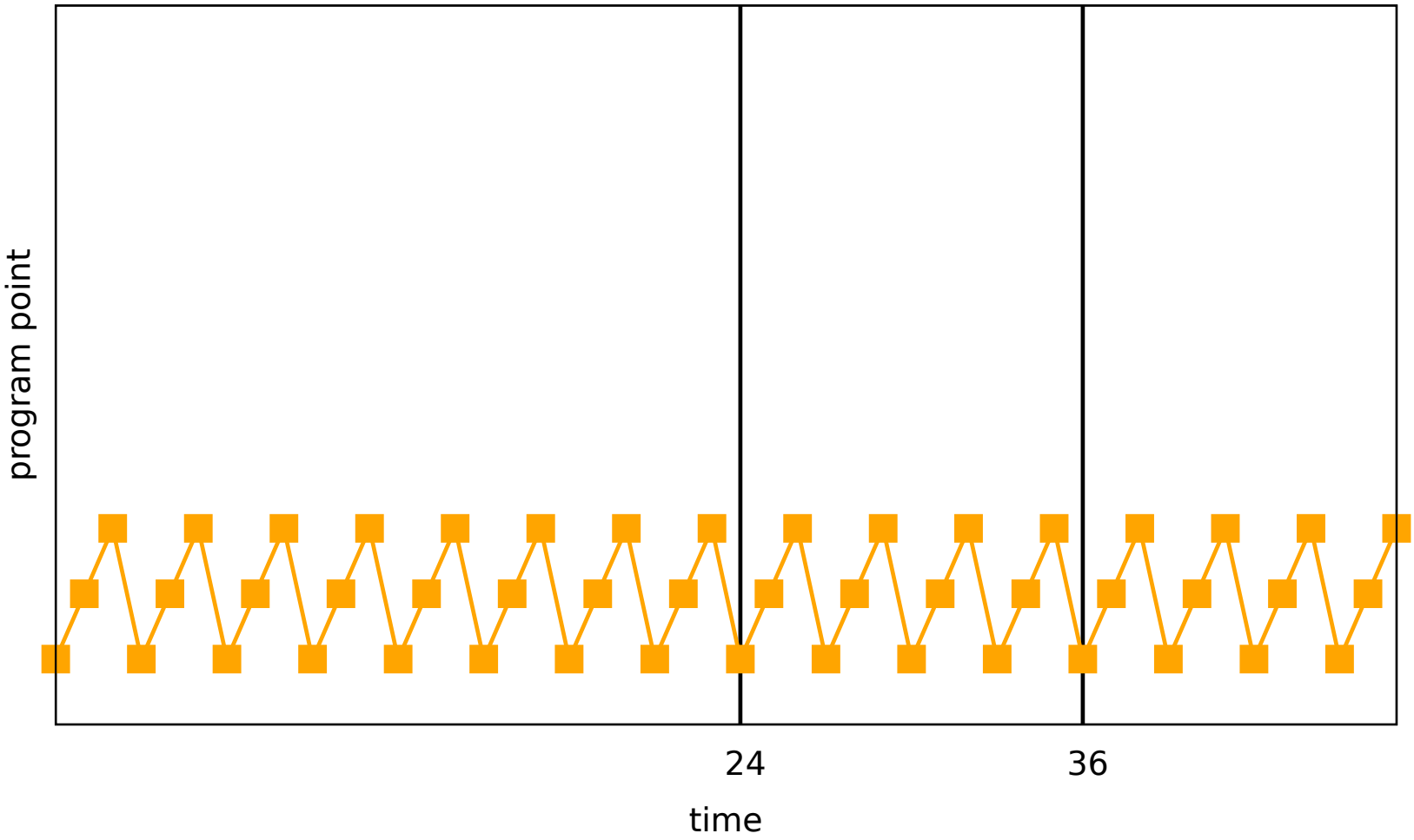
Execution Trace of Loop

Easy to make regular and uniform split points



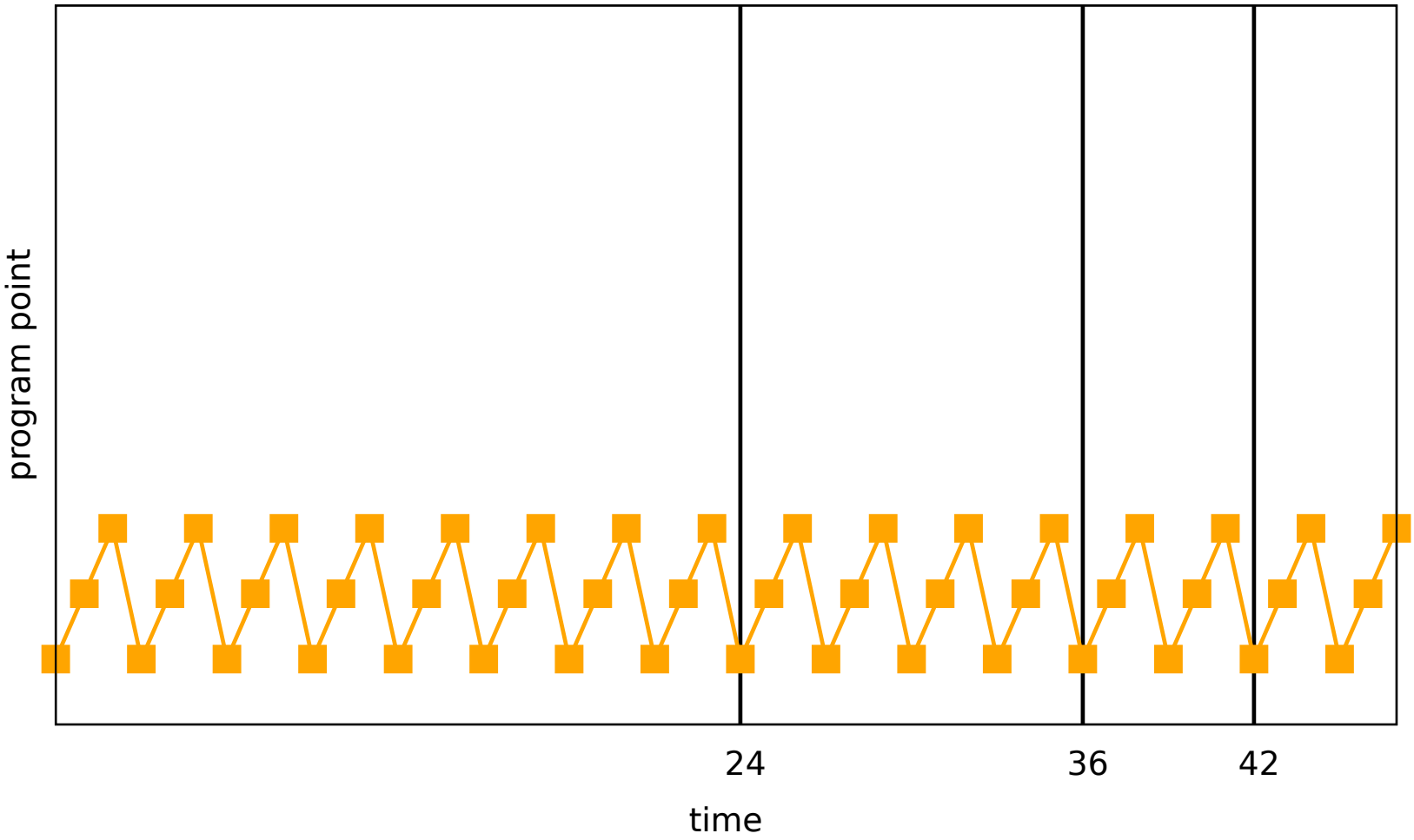
Execution Trace of Loop

Easy to make regular and uniform split points



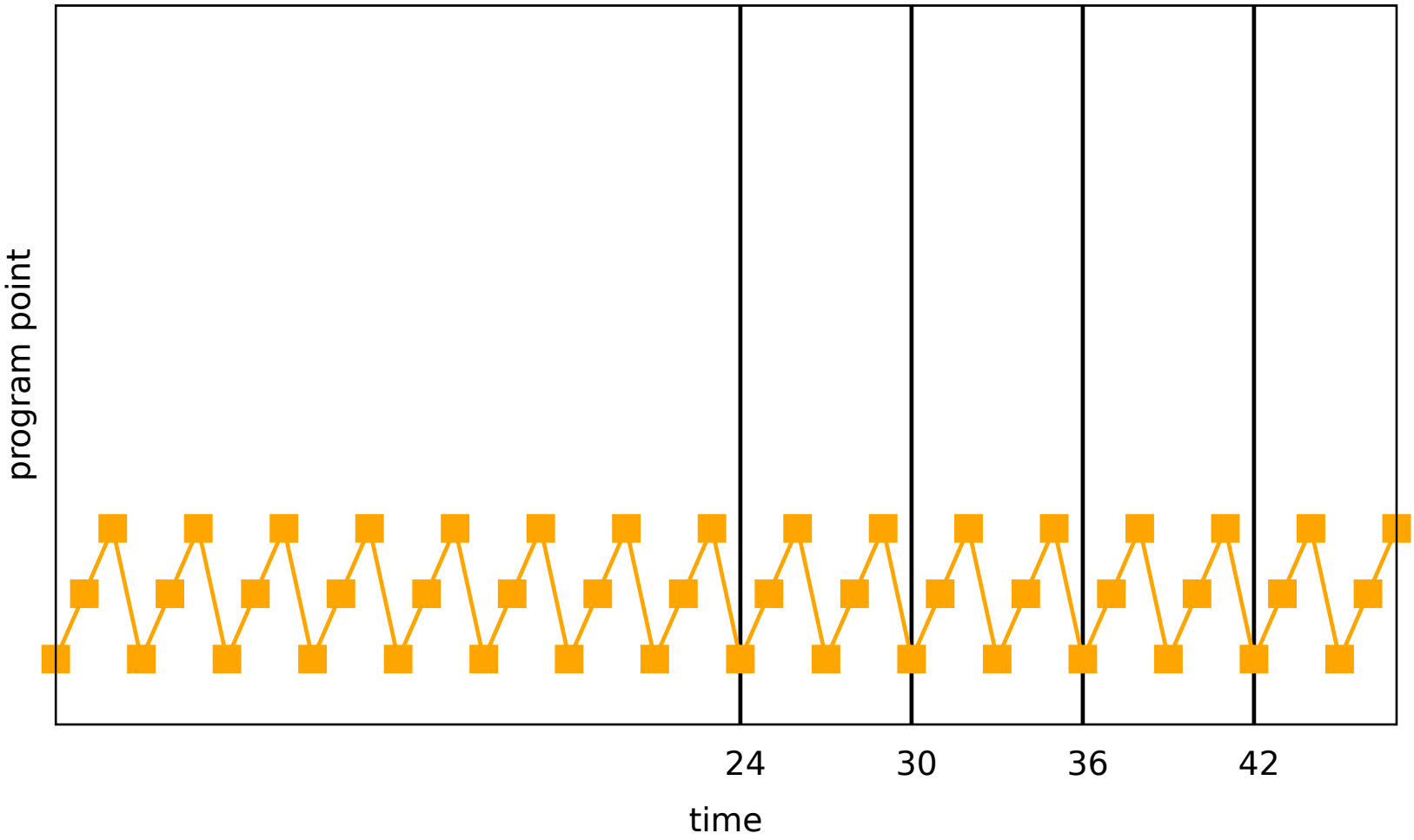
Execution Trace of Loop

Easy to make regular and uniform split points



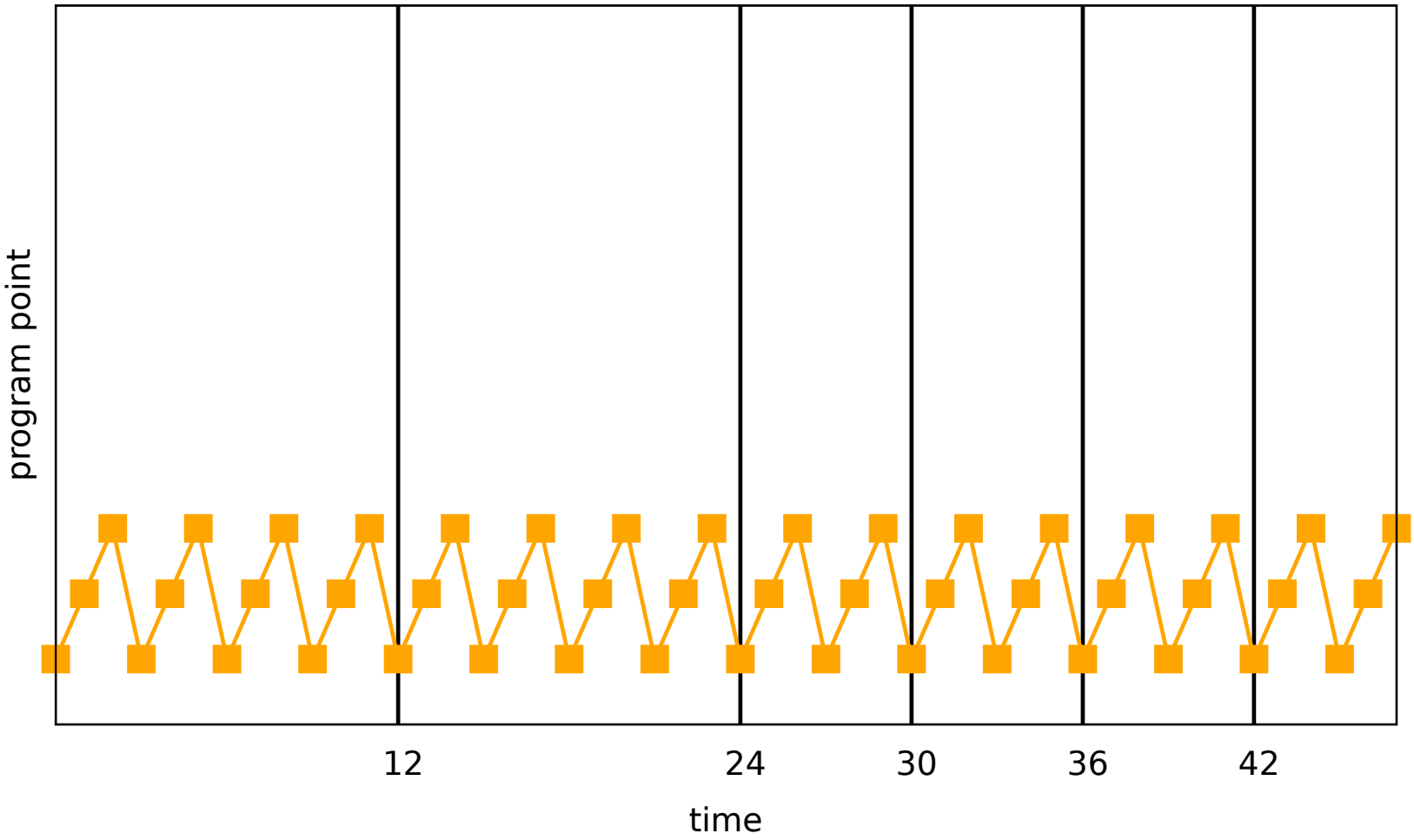
Execution Trace of Loop

Easy to make regular and uniform split points



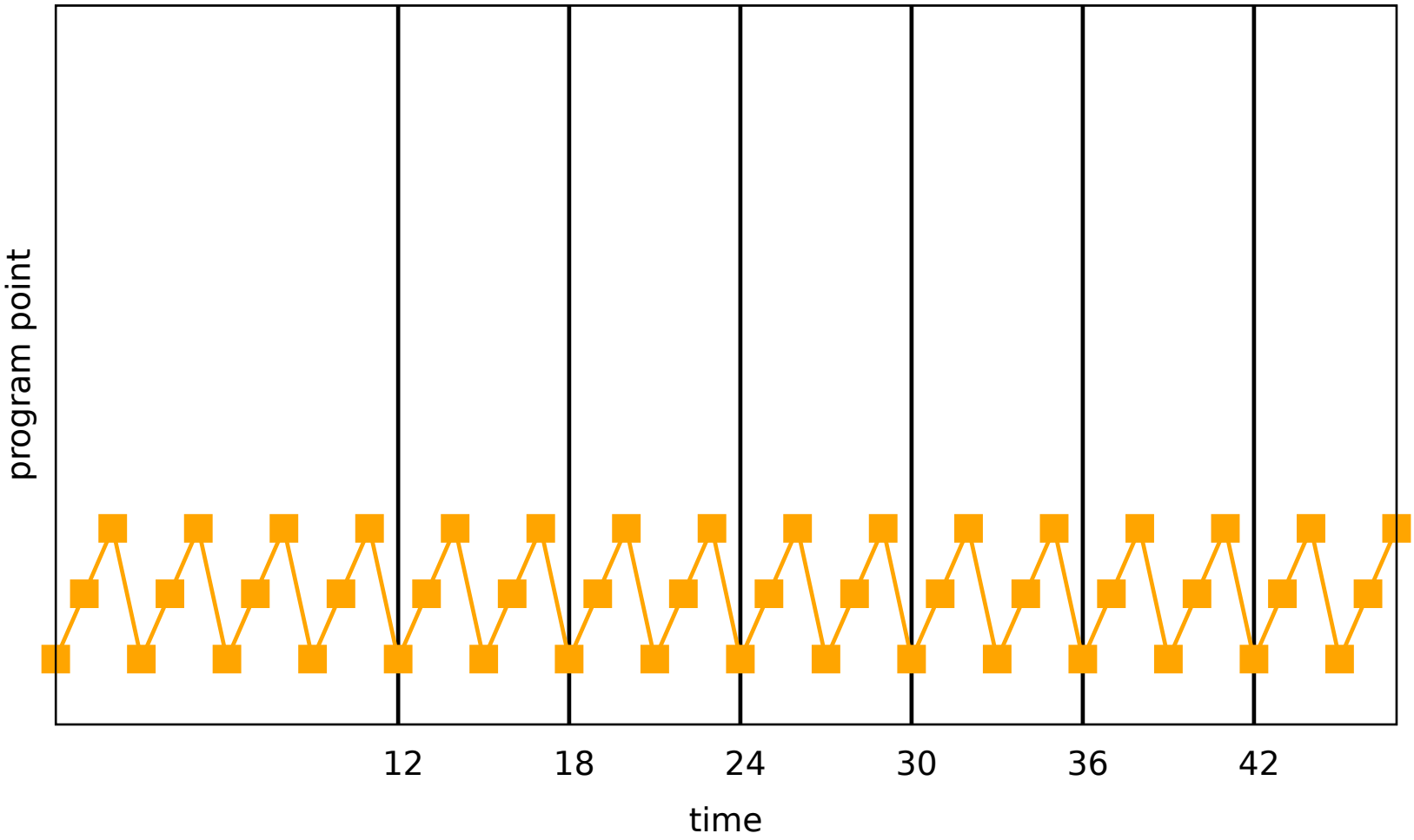
Execution Trace of Loop

Easy to make regular and uniform split points



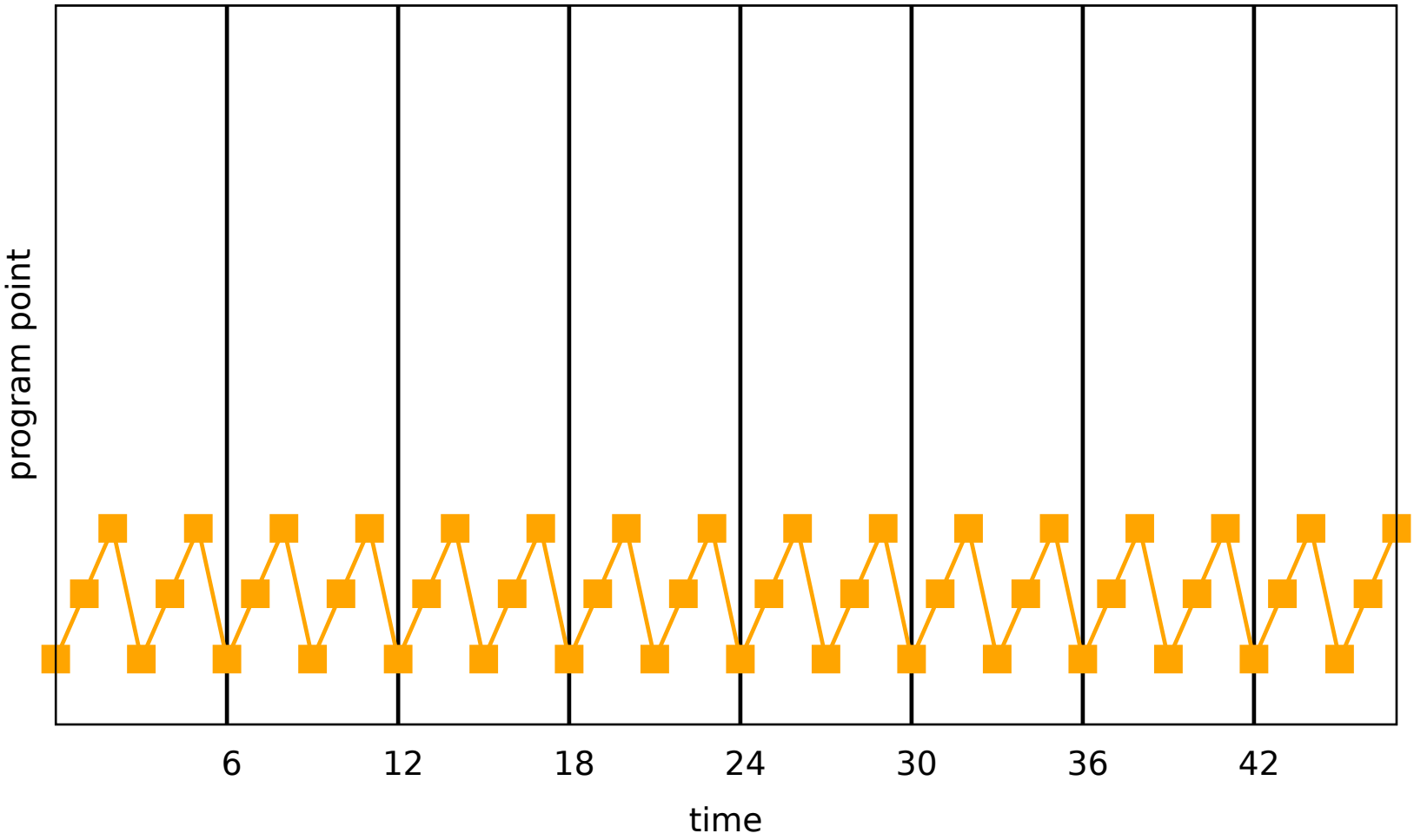
Execution Trace of Loop

Easy to make regular and uniform split points

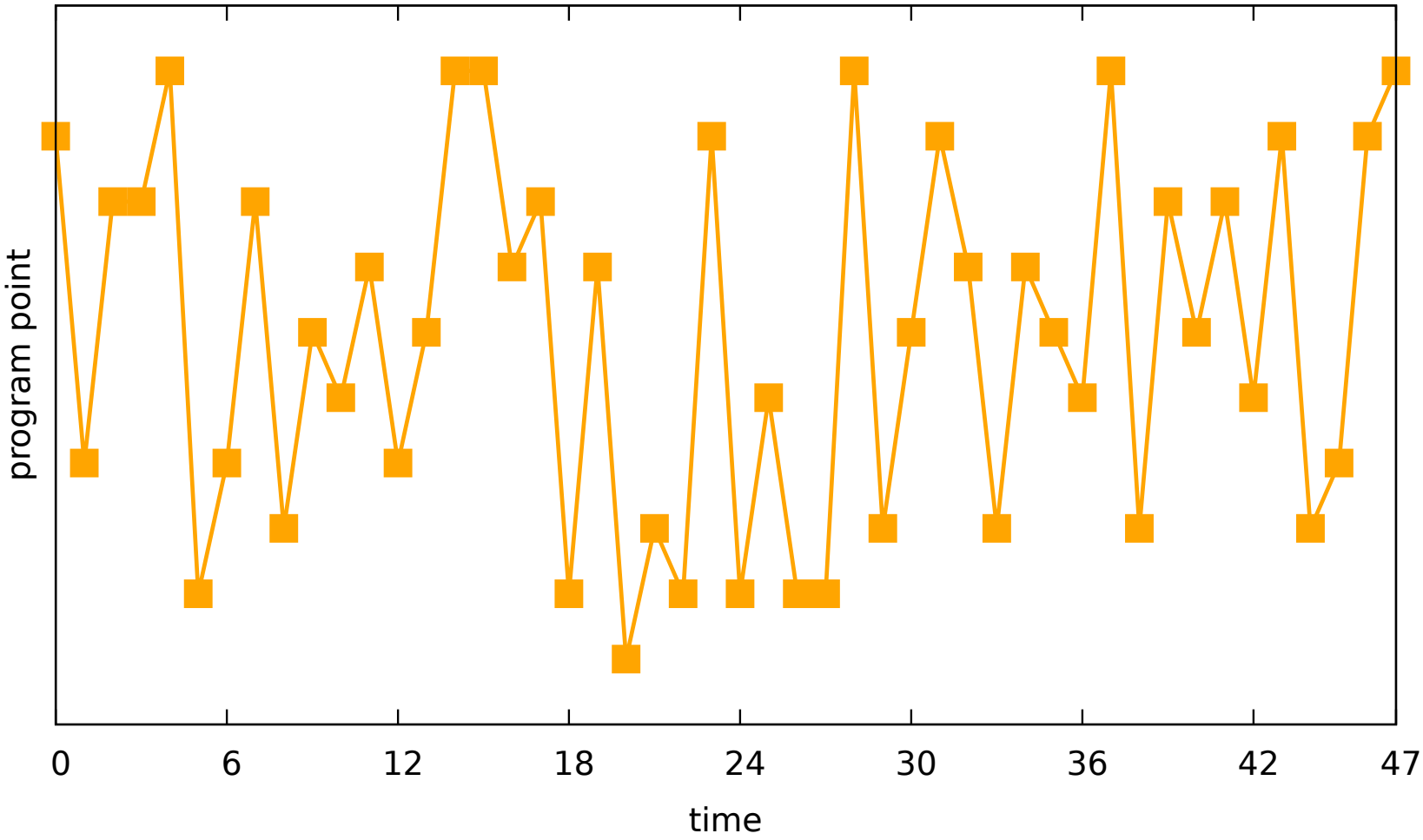


Execution Trace of Loop

Easy to make regular and uniform split points

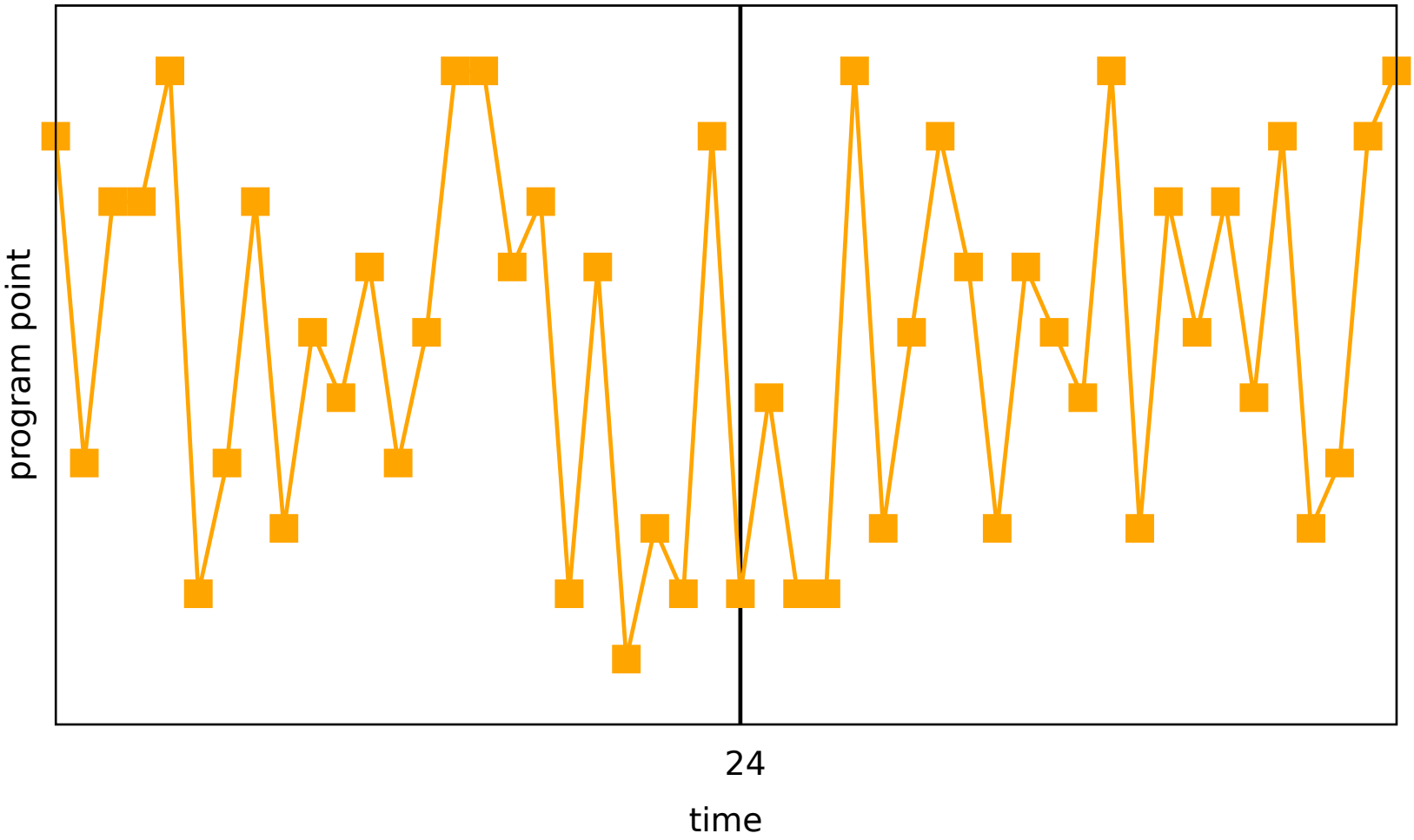


Execution Trace of Arbitrary Code



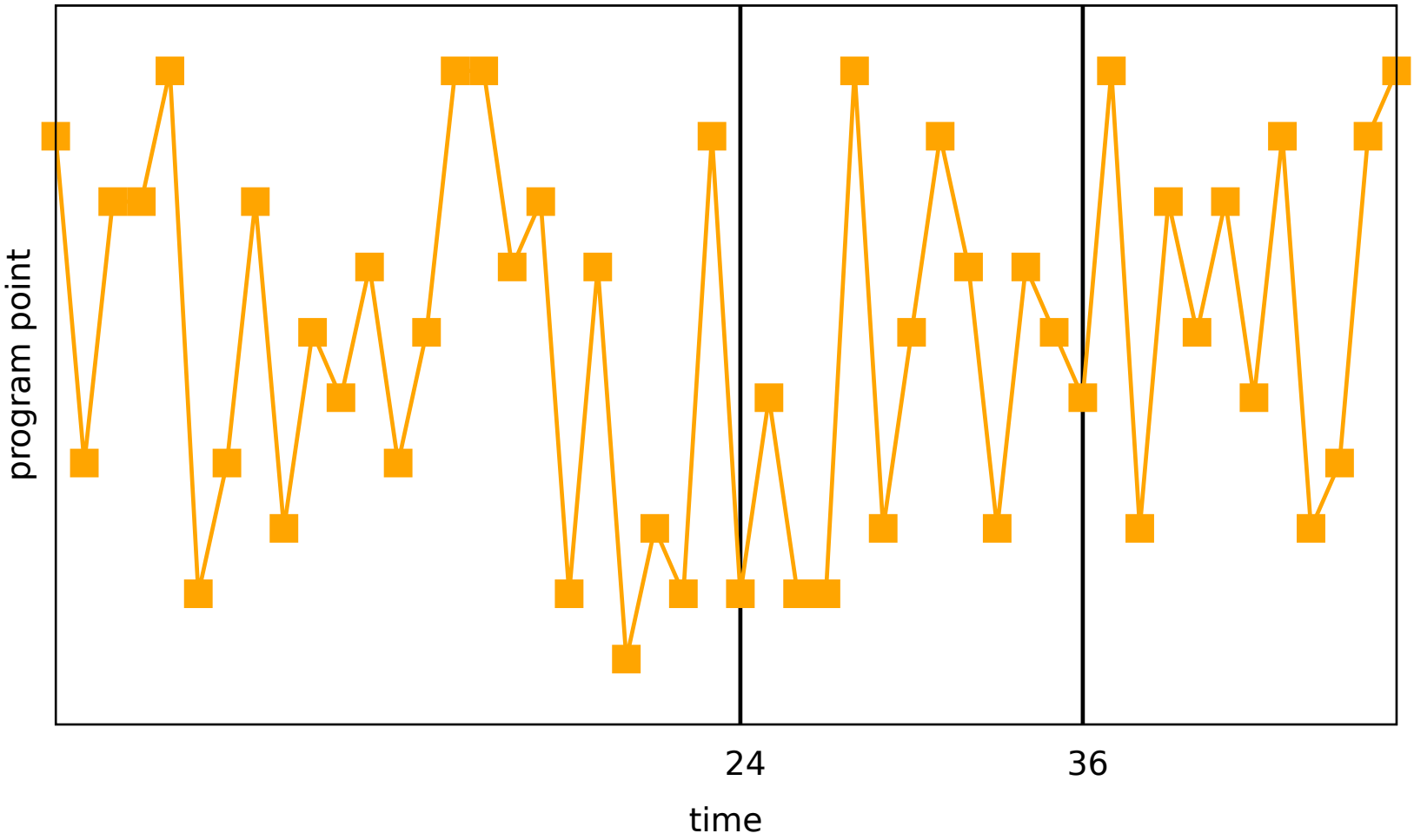
Execution Trace of Arbitrary Code

Difficult to make regular and uniform split points



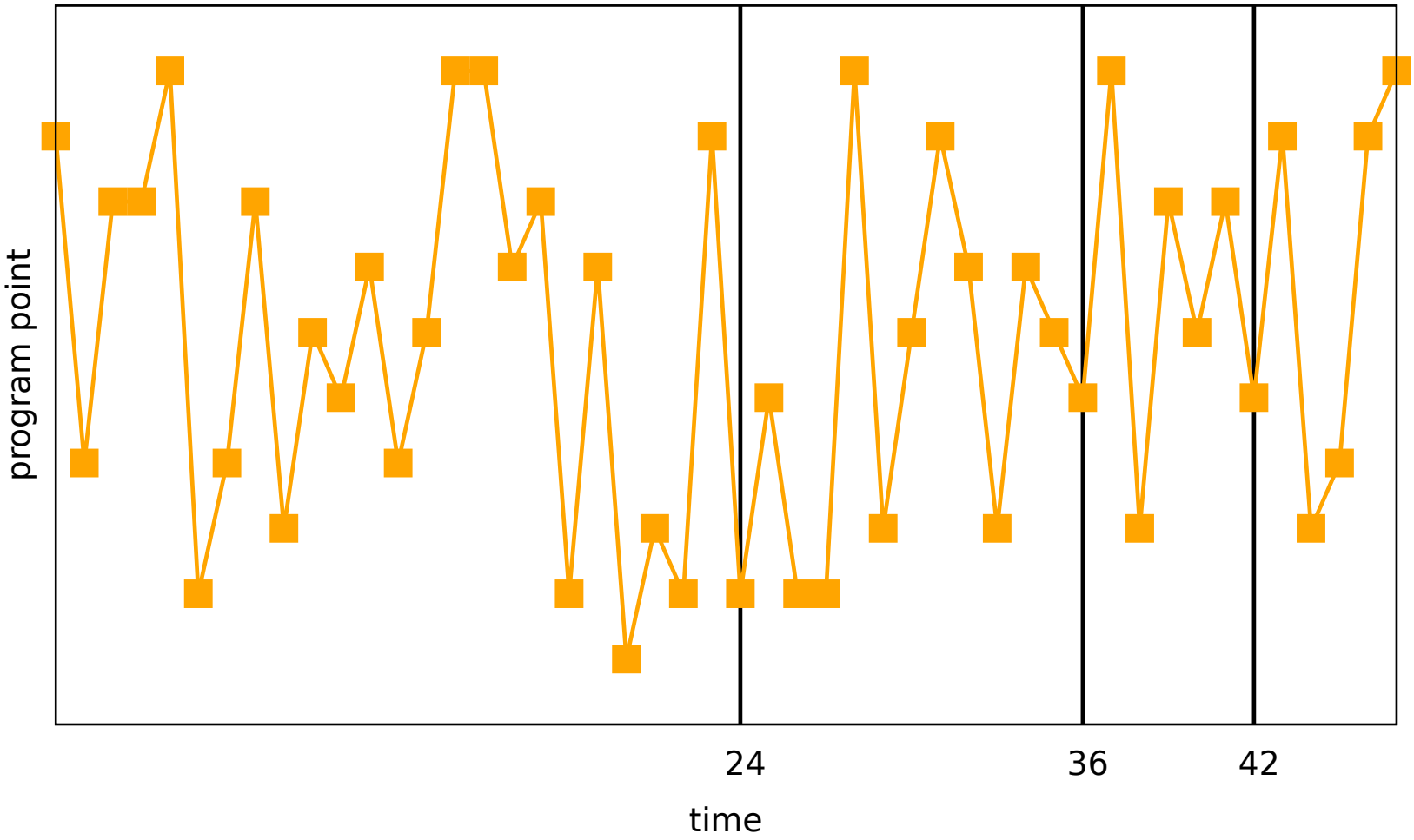
Execution Trace of Arbitrary Code

Difficult to make regular and uniform split points



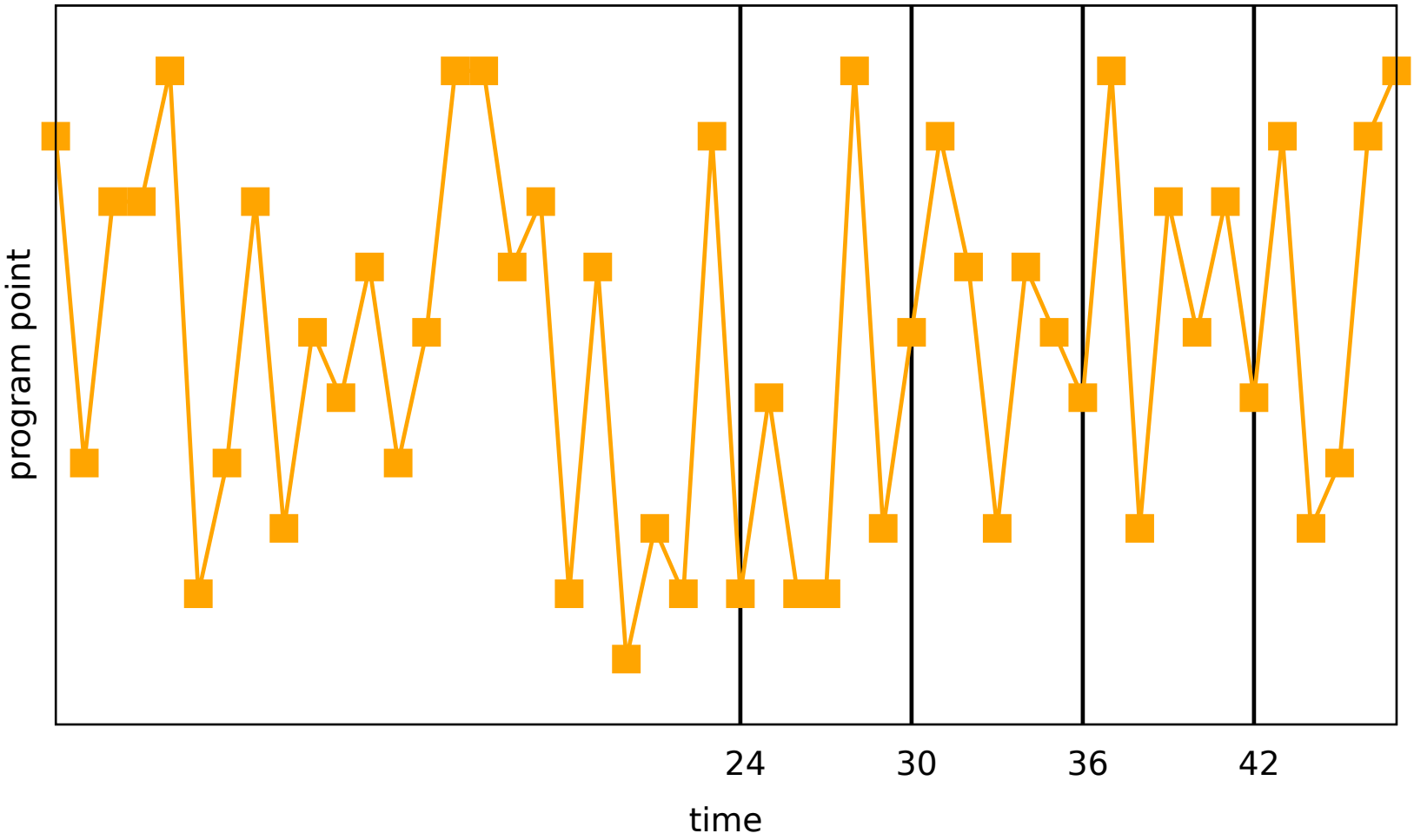
Execution Trace of Arbitrary Code

Difficult to make regular and uniform split points



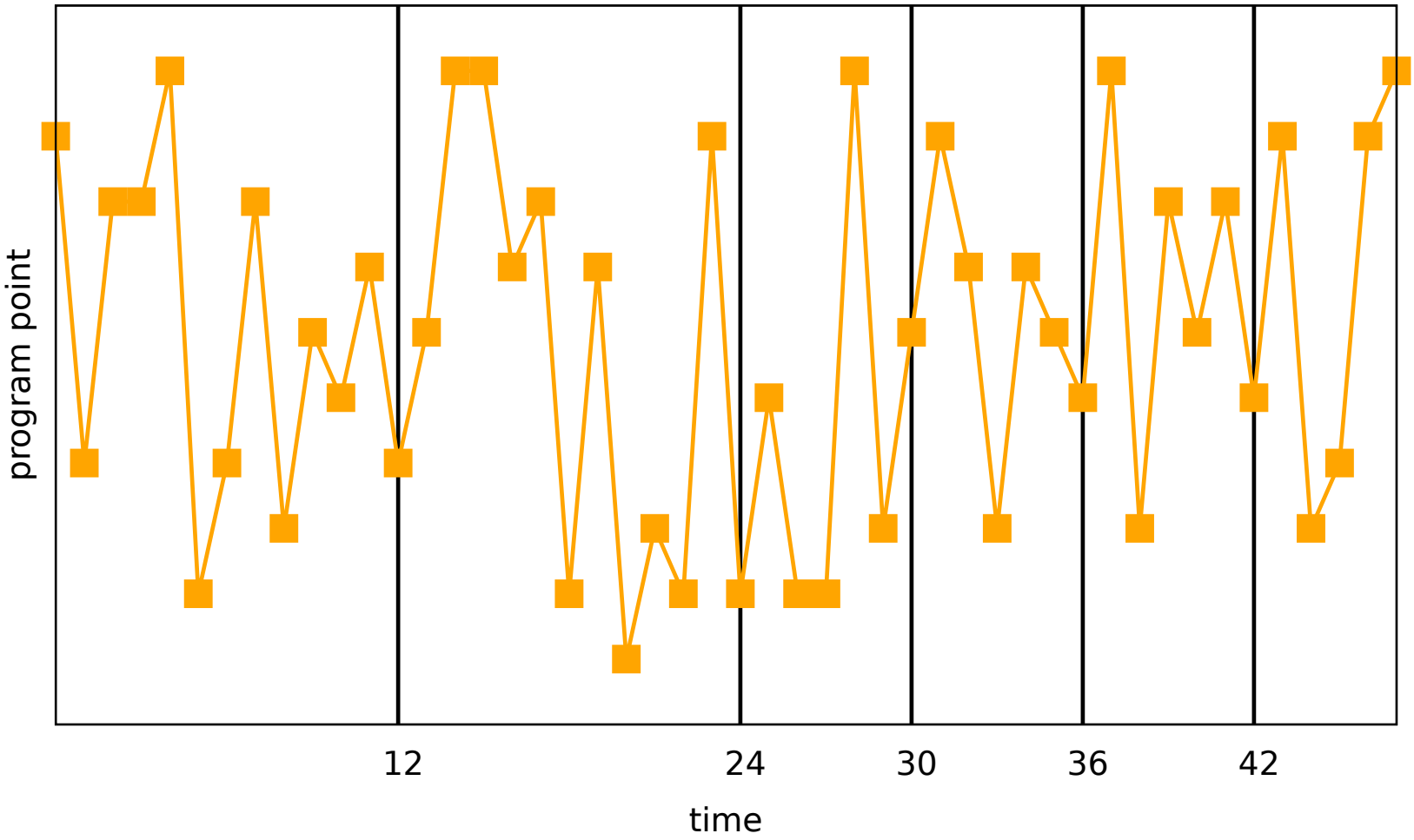
Execution Trace of Arbitrary Code

Difficult to make regular and uniform split points



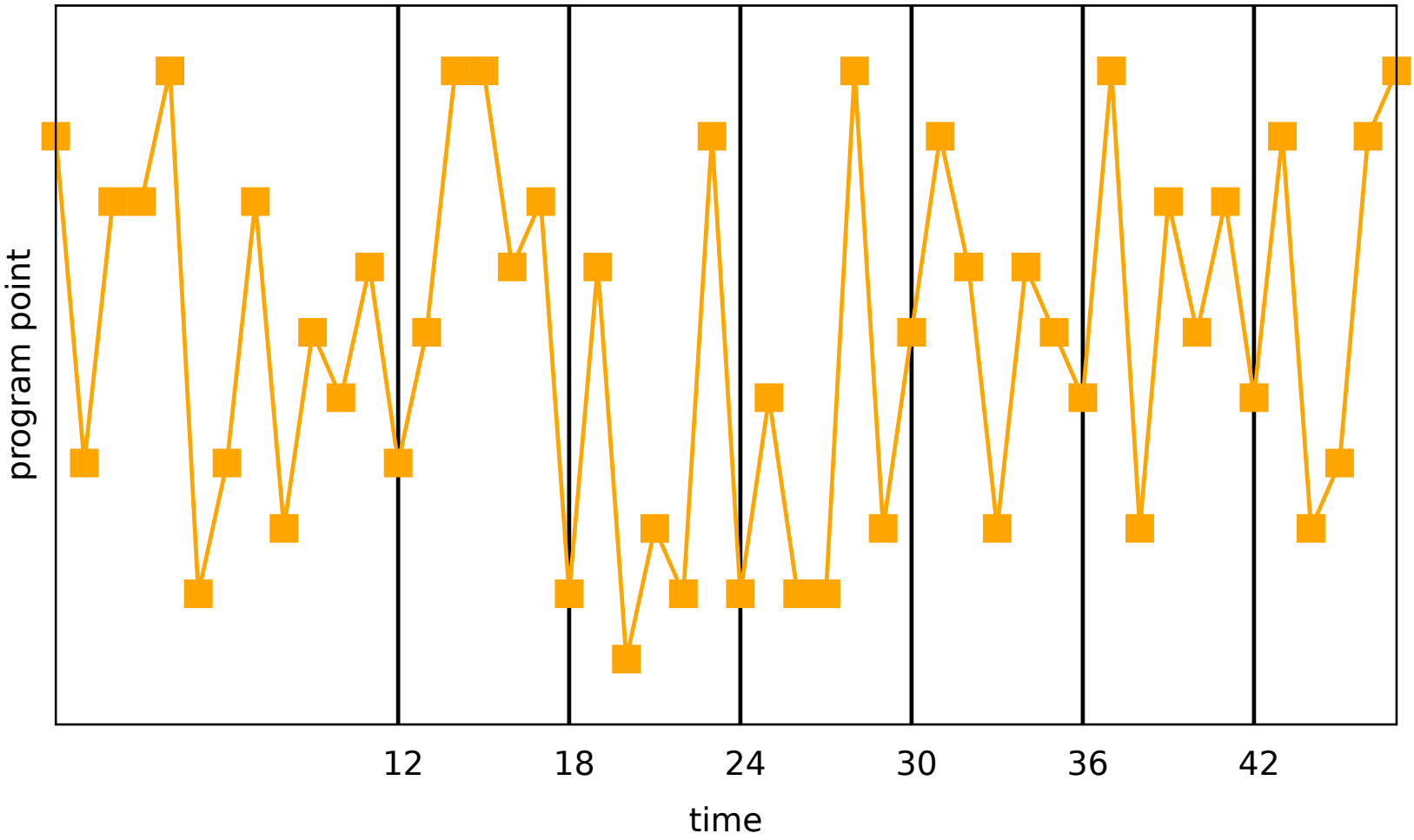
Execution Trace of Arbitrary Code

Difficult to make regular and uniform split points



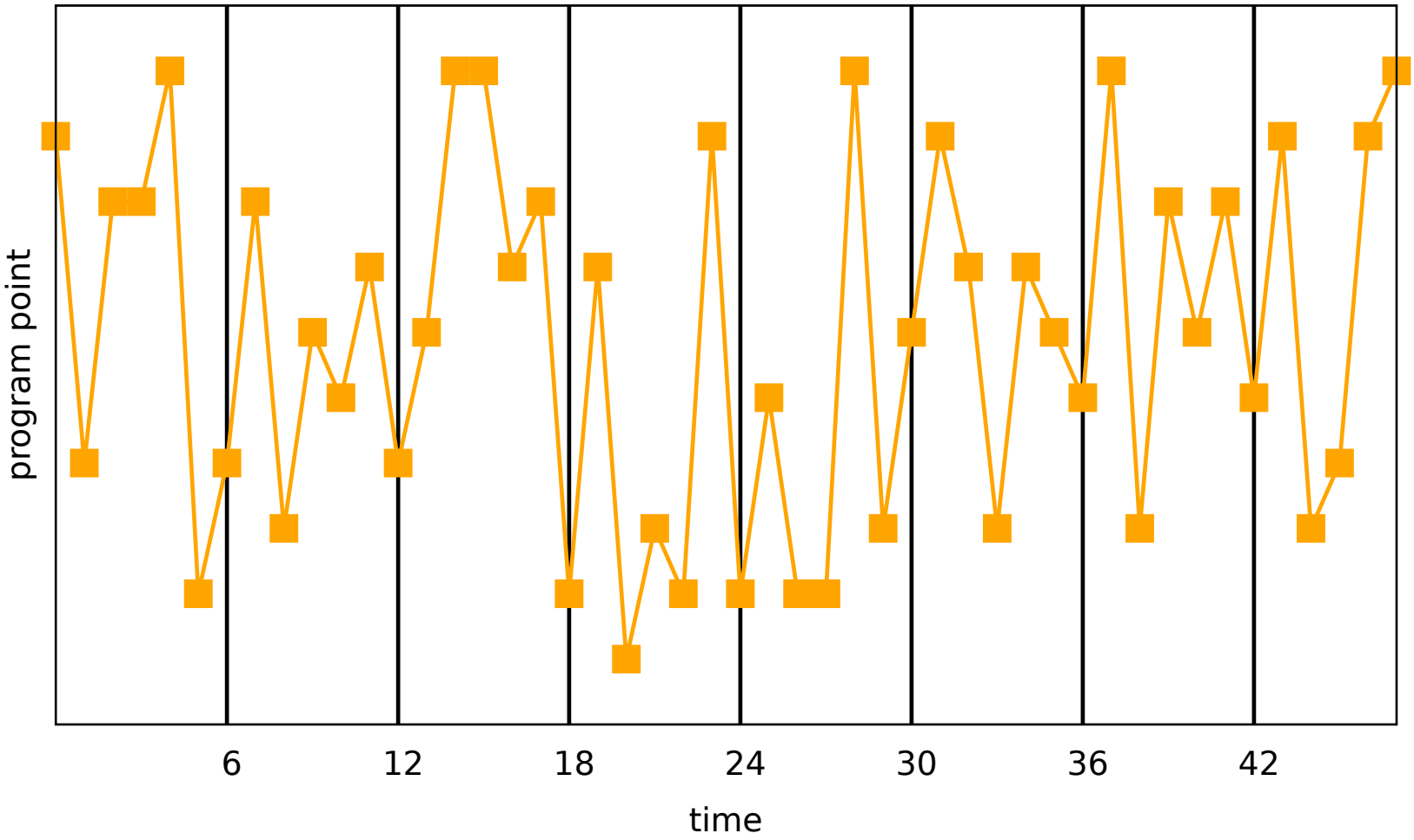
Execution Trace of Arbitrary Code

Difficult to make regular and uniform split points

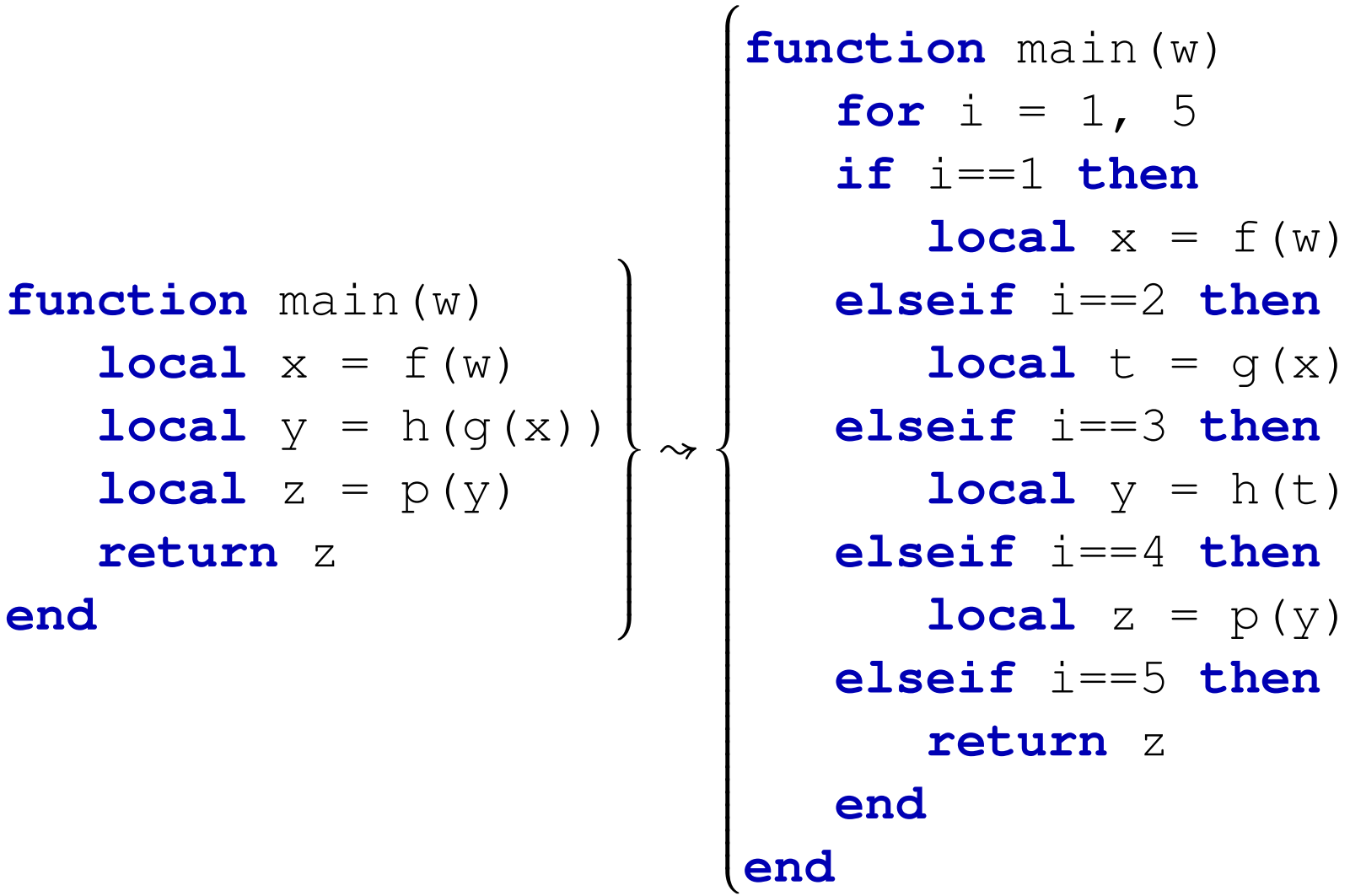


Execution Trace of Arbitrary Code

Difficult to make regular and uniform split points



Key Idea


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

Design Choices

- What root execution interval(s) should be subject to divide-and-conquer checkpointing?
- Which execution points are candidate split points?
- Which candidate split points are selected as actual split points?
- What is the shape or depth of the checkpoint tree, *i.e.*, what is the termination criterion for the divide-and-conquer process?

Design Choices

What root execution interval(s) should be subject to divide-and-conquer checkpointing?

loop Execution intervals resulting from invocations of specified DO loops are subject to divide-and-conquer checkpointing.

entire derivative calculation The execution interval for an entire specified derivative calculation is subject to divide-and-conquer checkpointing.

Design Choices

Which execution points are candidate split points?

iteration boundary Iteration boundaries of the DO loop specified as the root execution interval are taken as candidate split points.

arbitrary Any execution point inside the root execution interval can be taken as a candidate split point.

Design Choices

Which candidate split points are selected as actual split points?

bisection Split points are selected so as to divide the computation dominated by a node in half as one progresses successively from right to left among children (Griewank 1992, eq. 12). One can employ a variety of termination criteria, including that from Griewank (1992, p. 46). If the termination criterion is such that the total number of leaves is a power of two, one obtains a complete binary checkpoint tree. A termination criterion that bounds the number of evaluation steps in a leaf limits the size of the tape and achieves logarithmic overhead in both asymptotic space and time complexity compared with the primal.

binomial Split points are selected using the criterion from Griewank (1992, eq. 16). The termination criterion from Griewank (1992, p. 46) is usually adopted to achieve the desired properties discussed in Griewank (1992). Different termination criteria can be selected to control space-time tradeoffs.

Design Choices

What is the termination criterion for the divide-and-conquer process?

fixed space overhead One can bound the size of the tape and the number of snapshots to obtain sublinear but superlogarithmic overhead in asymptotic time complexity compared with the primal.

fixed time overhead One can bound the size of the tape and the (re)computation of the primal to obtain sublinear but superlogarithmic overhead in asymptotic space complexity compared with the primal.

logarithmic space and time overhead One can bound the size of the tape and obtain logarithmic overhead in both asymptotic space and time complexity compared with the primal. The constant factor is less than that of bisection checkpointing.

What is needed?

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

Core Language

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

Direct-Style Evaluator for the Core Language

$$\mathcal{A}\ \langle(\lambda x.e), \rho\rangle\ v = \mathcal{E}\ \rho[x \mapsto v]\ e \tag{1}$$

$$\mathcal{E}\ \rho\ c = c \tag{2}$$

$$\mathcal{E}\ \rho\ x = \rho\ x \tag{3}$$

$$\mathcal{E}\ \rho\ (\lambda x.e) = \langle(\lambda x.e), \rho\rangle \tag{4}$$

$$\mathcal{E}\ \rho\ (e_1\ e_2) = \mathcal{A}\ (\mathcal{E}\ \rho\ e_1)\ (\mathcal{E}\ \rho\ e_2) \tag{5}$$

$$\mathcal{E}\ \rho\ (\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3) = \textbf{if}\ (\mathcal{E}\ \rho\ e_1)\ \textbf{then}\ (\mathcal{E}\ \rho\ e_2)\ \textbf{else}\ (\mathcal{E}\ \rho\ e_3) \tag{6}$$

$$\mathcal{E}\ \rho\ (\diamond e) = \diamond(\mathcal{E}\ \rho\ e) \tag{7}$$

$$\mathcal{E}\ \rho\ (e_1 \bullet e_2) = (\mathcal{E}\ \rho\ e_1) \bullet (\mathcal{E}\ \rho\ e_2) \tag{8}$$

Adding AD Operators to the Core Language

$$\overrightarrow{\mathcal{J}} : f\ x\ \acute{x} \mapsto (y, \acute{y})$$

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

Adding AD Operators to the Core Language

$$e ::= \overrightarrow{\mathcal{J}}\ e_1\ e_2\ e_3 \mid \overleftarrow{\mathcal{J}}\ e_1\ e_2\ e_3$$

Adding AD Operators to the Core Language

$$\overrightarrow{\mathcal{J}}\ v_1\ v_2\ \acute{v}_3 = \textbf{let}\ (v_4 \triangleright \acute{v}_5) = (\mathcal{A}\ v_1\ (v_2 \triangleright \acute{v}_3))\ \textbf{in}\ (v_4, \acute{v}_5)$$

$$\overleftarrow{\mathcal{J}}\ v_1\ v_2\ \grave{v}_3 = \textbf{let}\ (v_4 \triangleleft \grave{v}_5) = ((\mathcal{A}\ v_1\ v_2) \triangleleft \grave{v}_3)\ \textbf{in}\ (v_4, \grave{v}_5)$$

Adding AD Operators to the Core Language

$$\overrightarrow{\mathcal{J}}\ v_1\ v_2\ \acute{v}_3 = \textbf{let}\ (v_4 \triangleright \acute{v}_5) = (\mathcal{A}\ v_1\ (v_2 \triangleright \acute{v}_3))\ \textbf{in}\ (v_4, \acute{v}_5)$$

$$\overleftarrow{\mathcal{J}}\ v_1\ v_2\ \grave{v}_3 = \textbf{let}\ (v_4 \triangleleft \grave{v}_5) = ((\mathcal{A}\ v_1\ v_2) \triangleleft \grave{v}_3)\ \textbf{in}\ (v_4, \grave{v}_5)$$

$$\mathcal{E}\ \rho\ (\overrightarrow{\mathcal{J}}\ e_1\ e_2\ e_3) = \overrightarrow{\mathcal{J}}\ (\mathcal{E}\ \rho\ e_1)\ (\mathcal{E}\ \rho\ e_2)\ (\mathcal{E}\ \rho\ e_3)$$

$$\mathcal{E}\ \rho\ (\overleftarrow{\mathcal{J}}\ e_1\ e_2\ e_3) = \overleftarrow{\mathcal{J}}\ (\mathcal{E}\ \rho\ e_1)\ (\mathcal{E}\ \rho\ e_2)\ (\mathcal{E}\ \rho\ e_3)$$

Algorithm for Binary Checkpointing

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

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

inductive case:

$h \circ g = f$ (1)

$z = g\ x$ (2)

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

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

Algorithm for Binary Checkpointing

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

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

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

$z = g\ x$ (2)

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

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

Algorithm for Binary Checkpointing

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

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

inductive case:

$h \circ g = f$ (1)

$z = g\ x$ (2)

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

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

Algorithm for Binary Checkpointing

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

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

inductive case:

$h \circ g = f$ (1)

$z = g\ x$ (2)

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

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

Algorithm for Binary Checkpointing

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

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

inductive case:

$h \circ g = f$ (1)

$z = g\ x$ (2)

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

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

Algorithm for Binary Checkpointing

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

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

inductive case:

$h \circ g = f$ (1)

$z = g\ x$ (2)

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

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

Algorithm for Binary Checkpointing

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

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

inductive case:

$h \circ g = f$ (1)

$z = g\ x$ (2)

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

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

Algorithm for Binary Checkpointing

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

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

inductive case:

$h \circ g = f$ (1)

$z = g\ x$ (2)

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

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

Algorithm for Binary Checkpointing

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

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

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

$z = g\ x$ (2)

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

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

Algorithm for Binary Checkpointing

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

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

inductive case:

$h \circ g = f$ (1)

$z = g\ x$ (2)

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

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

Algorithm for Binary Checkpointing

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

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

inductive case:

$h \circ g = f$ (1)

$z = g\ x$ (2)

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

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

Algorithm for Binary Checkpointing

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

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

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

$z = g\ x$ (2)

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

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

Algorithm for Binary Checkpointing

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

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

inductive case:

$h \circ g = f$ (1)

$z = g\ x$ (2)

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

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

Algorithm for Binary Checkpointing

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

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

inductive case:

$h \circ g = f$ (1)

$z = g\ x$ (2)

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

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

Algorithm for Binary Checkpointing

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

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

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

$z = g\ x$ (2)

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

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

Algorithm for Binary Checkpointing

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

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

inductive case:

$h \circ g = f$ (1)

$z = g\ x$ (2)

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

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

Algorithm for Binary Checkpointing

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

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

inductive case:

$h \circ g = f$ (1)

$z = g\ x$ (2)

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

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

Algorithm for Binary Checkpointing

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

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

inductive case:

$h \circ g = f$ (1)

$z = g\ x$ (2)

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

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

Algorithm for Binary Checkpointing

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

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

inductive case:

$h \circ g = f$ (1)

$z = g\ x$ (2)

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

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

Algorithm for Binary Checkpointing

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

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

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

$z = g\ x$ (2)

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

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

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

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

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 $y = f(x)$.

Algorithm for Binary Checkpointing

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

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

inductive case:

$h \circ g = f$ (1)

$z = g\ x$ (2)

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

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

Algorithm for Binary Checkpointing

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

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

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

$z = g\ x$ (2)

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

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

Algorithm for Binary Checkpointing

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

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

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

$z = g\ x$ (2)

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

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

Algorithm for Binary Checkpointing

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

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

inductive case:

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

$z = g\ x$ (2)

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

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

Algorithm for Binary Checkpointing

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

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

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

$z = g\ x$ (2)

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

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

Algorithm for Binary Checkpointing

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

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

inductive case:

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

$z = g\ x$ (2)

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

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

Algorithm for Binary Checkpointing

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

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

inductive case:

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

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

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

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

Algorithm for Binary Checkpointing

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

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

inductive case:

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

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

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

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

Algorithm for Binary Checkpointing

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

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

inductive case:

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

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

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

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

Algorithm for Binary Checkpointing

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

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

inductive case:

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

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

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

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

Algorithm for Binary Checkpointing

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

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

inductive case:

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

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

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

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

Algorithm for Binary Checkpointing

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

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

inductive case:
$$l = \text{PRIMOPS}\ f\ x \tag{1}$$
$$z = \text{INTERRUPT}\ f\ x\ \lfloor \tfrac{l}{2} \rfloor \tag{2}$$
$$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\ (\lambda z.\text{RESUME}\ z)\ z\ \grave{y} \tag{3}$$
$$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ g\ x\ \grave{z} \tag{4}$$

Algorithm for Binary Checkpointing

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

base case ($f\ x$ fast):

$$(y, \grave{x}) = \overset{\leftarrow}{\mathcal{J}}\, f\ x\ \grave{y} \tag{0}$$

inductive case:

$$l = \text{PRIMOPS}\ f\ x \tag{1}$$

$$z = \text{INTERRUPT}\ f\ x\ \lfloor \tfrac{l}{2} \rfloor \tag{2}$$

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

$$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ (\lambda x.\text{INTERRUPT}\ f\ x\ \lfloor \tfrac{l}{2} \rfloor)\ x\ \grave{z} \tag{4}$$

CPS Evaluator for the Core Language

$$
\begin{aligned}
\mathcal{A}\ k\ n\ l\ \langle(\lambda x.e), \rho\rangle\ v &= \mathcal{E}\ k\ n\ l\ \rho[x \mapsto v]\ e \\
\mathcal{E}\ k\ l\ l\ \rho\ e &= [\![k, \langle(\lambda\_.e), \rho\rangle]\!] \\
\mathcal{E}\ k\ n\ l\ \rho\ c &= k\ (n+1)\ l\ c \\
\mathcal{E}\ k\ n\ l\ \rho\ x &= k\ (n+1)\ l\ (\rho\ x) \\
\mathcal{E}\ k\ n\ l\ \rho\ (\lambda x.e) &= k\ (n+1)\ l\ \langle(\lambda x.e), \rho\rangle \\
\mathcal{E}\ k\ n\ l\ \rho\ (e_1\ e_2) &= \mathcal{E}\ (\lambda n\ l\ v_1.\ (\mathcal{E}\ (\lambda n\ l\ v_2.\ (\mathcal{A}\ k\ n\ l\ v_1\ v_2))\ n\ l\ \rho\ e_2))\ (n+1)\ l\ \rho\ e_1
\end{aligned}
$$

CPS Evaluator for the Core Language

$$\mathcal{A}\ k\ n\ l\ \langle(\lambda x.e), \rho\rangle\ v = \mathcal{E}\ k\ n\ l\ \rho[x \mapsto v]\ e$$

$$\mathcal{E}\ k\ l\ l\ \rho\ e = [\![k, \langle(\lambda\_.e), \rho\rangle]\!]$$

$$\mathcal{E}\ k\ n\ l\ \rho\ c = k\ (n+1)\ l\ c$$

$$\mathcal{E}\ k\ n\ l\ \rho\ x = k\ (n+1)\ l\ (\rho\ x)$$

$$\mathcal{E}\ k\ n\ l\ \rho\ (\lambda x.e) = k\ (n+1)\ l\ \langle(\lambda x.e), \rho\rangle$$

$$\begin{aligned}\mathcal{E}\ k\ n\ l\ \rho\ (e_1\ e_2) = \mathcal{E}\ (&\lambda n\ l\ v_1.\\ &(\mathcal{E}\ (\lambda n\ l\ v_2.\\ &\quad(\mathcal{A}\ k\ n\ l\ v_1\ v_2))\\ &\quad n\ l\ \rho\ e_2))\\ &(n+1)\ l\ \rho\ e_1\end{aligned}$$

CPS Evaluator for the Core Language

$$
\begin{aligned}
\mathcal{A}\ k\ n\ l\ \langle(\lambda x.e), \rho\rangle\ v &= \mathcal{E}\ k\ n\ l\ \rho[x \mapsto v]\ e \\
\mathcal{E}\ k\ l\ l\ \rho\ e &= [\![k, \langle(\lambda\_.e), \rho\rangle]\!] \\
\mathcal{E}\ k\ n\ l\ \rho\ c &= k\ (n+1)\ l\ c \\
\mathcal{E}\ k\ n\ l\ \rho\ x &= k\ (n+1)\ l\ (\rho\ x) \\
\mathcal{E}\ k\ n\ l\ \rho\ (\lambda x.e) &= k\ (n+1)\ l\ \langle(\lambda x.e), \rho\rangle \\
\mathcal{E}\ k\ n\ l\ \rho\ (e_1\ e_2) &= \mathcal{E}\ (\lambda n\ l\ v_1.\ (\mathcal{E}\ (\lambda n\ l\ v_2.\ (\mathcal{A}\ k\ n\ l\ v_1\ v_2))\ n\ l\ \rho\ e_2))\ (n+1)\ l\ \rho\ e_1
\end{aligned}
$$

CPS Evaluator for the Core Language

$$\mathcal{A}\ k\ n\ l\ \langle(\lambda x.e), \rho\rangle\ v = \mathcal{E}\ k\ n\ l\ \rho[x \mapsto v]\ e$$

$$\mathcal{E}\ k\ l\ l\ \rho\ e = [\![k, \langle(\lambda\_.e), \rho\rangle]\!]$$

$$\mathcal{E}\ k\ n\ l\ \rho\ c = k\ (n+1)\ l\ c$$

$$\mathcal{E}\ k\ n\ l\ \rho\ x = k\ (n+1)\ l\ (\rho\ x)$$

$$\mathcal{E}\ k\ n\ l\ \rho\ (\lambda x.e) = k\ (n+1)\ l\ \langle(\lambda x.e), \rho\rangle$$

$$\begin{aligned}\mathcal{E}\ k\ n\ l\ \rho\ (e_1\ e_2) = \mathcal{E}\ (&\lambda n\ l\ v_1.\\ &(\mathcal{E}\ (\lambda n\ l\ v_2.\\ &\quad(\mathcal{A}\ k\ n\ l\ v_1\ v_2))\\ &\quad n\ l\ \rho\ e_2))\\ &(n+1)\ l\ \rho\ e_1\end{aligned}$$

CPS Evaluator for the Core Language

$$
\begin{aligned}
\mathcal{A}\ k\ n\ l\ \langle(\lambda x.e), \rho\rangle\ v &= \mathcal{E}\ k\ n\ l\ \rho[x \mapsto v]\ e \\
\mathcal{E}\ k\ l\ l\ \rho\ e &= [\![k, \langle(\lambda\_.e), \rho\rangle]\!] \\
\mathcal{E}\ k\ n\ l\ \rho\ c &= k\ (n+1)\ l\ c \\
\mathcal{E}\ k\ n\ l\ \rho\ x &= k\ (n+1)\ l\ (\rho\ x) \\
\mathcal{E}\ k\ n\ l\ \rho\ (\lambda x.e) &= k\ (n+1)\ l\ \langle(\lambda x.e), \rho\rangle \\
\mathcal{E}\ k\ n\ l\ \rho\ (e_1\ e_2) &= \mathcal{E}\ (\lambda n\ l\ v_1.\ (\mathcal{E}\ (\lambda n\ l\ v_2.\ (\mathcal{A}\ k\ n\ l\ v_1\ v_2))\ n\ l\ \rho\ e_2))\ (n+1)\ l\ \rho\ e_1
\end{aligned}
$$

CPS Evaluator for the Core Language

$$\mathcal{A}\ k\ n\ l\ \langle(\lambda x.e), \rho\rangle\ v = \mathcal{E}\ k\ n\ l\ \rho[x \mapsto v]\ e$$

$$\mathcal{E}\ k\ l\ l\ \rho\ e = [\![k, \langle(\lambda\_.e), \rho\rangle]\!]$$

$$\mathcal{E}\ k\ n\ l\ \rho\ c = k\ (n+1)\ l\ c$$

$$\mathcal{E}\ k\ n\ l\ \rho\ x = k\ (n+1)\ l\ (\rho\ x)$$

$$\mathcal{E}\ k\ n\ l\ \rho\ (\lambda x.e) = k\ (n+1)\ l\ \langle(\lambda x.e), \rho\rangle$$

$$\begin{aligned}\mathcal{E}\ k\ n\ l\ \rho\ (e_1\ e_2) = \mathcal{E}\ (&\lambda n\ l\ v_1.\\ &(\mathcal{E}\ (\lambda n\ l\ v_2.\\ &\quad(\mathcal{A}\ k\ n\ l\ v_1\ v_2))\\ &\quad n\ l\ \rho\ e_2))\\ &(n+1)\ l\ \rho\ e_1\end{aligned}$$

CPS Evaluator for the Core Language

$$\mathcal{E}\ k\ n\ l\ \rho\ (\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3) = \mathcal{E}\ (\lambda n\ l\ v_1.\ (\textbf{if}\ v_1\ \textbf{then}\ (\mathcal{E}\ k\ n\ l\ \rho\ e_2)\ \textbf{else}\ (\mathcal{E}\ k\ n\ l\ \rho\ e_3)))\ (n+1)\ l\ \rho\ e_1$$

$$\mathcal{E}\ k\ n\ l\ \rho\ (\diamond e) = \mathcal{E}\ (\lambda n\ l\ v.\ (k\ n\ l\ (\diamond v)))\ (n+1)\ l\ \rho\ e$$

$$\mathcal{E}\ k\ n\ l\ \rho\ (e_1 \bullet e_2) = \mathcal{E}\ (\lambda n\ l\ v_1.\ (\mathcal{E}\ (\lambda n\ l\ v_2.\ (k\ n\ l\ (v_1 \bullet v_2)))\ n\ l\ \rho\ e_2))\ (n+1)\ l\ \rho\ e_1$$

CPS Evaluator for the Core Language

$$\mathcal{E}\ k\ n\ l\ \rho\ (\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3) = \mathcal{E}\ (\lambda n\ l\ v_1.\ (\textbf{if}\ v_1\ \textbf{then}\ (\mathcal{E}\ k\ n\ l\ \rho\ e_2)\ \textbf{else}\ (\mathcal{E}\ k\ n\ l\ \rho\ e_3)))\ (n+1)\ l\ \rho\ e_1$$

$$\mathcal{E}\ k\ n\ l\ \rho\ (\diamond e) = \mathcal{E}\ (\lambda n\ l\ v.\ (k\ n\ l\ (\diamond v)))\ (n+1)\ l\ \rho\ e$$

$$\mathcal{E}\ k\ n\ l\ \rho\ (e_1 \bullet e_2) = \mathcal{E}\ (\lambda n\ l\ v_1.\ (\mathcal{E}\ (\lambda n\ l\ v_2.\ (k\ n\ l\ (v_1 \bullet v_2)))\ n\ l\ \rho\ e_2))\ (n+1)\ l\ \rho\ e_1$$

CPS Evaluator for the Core Language

$$\begin{aligned}
\mathcal{E}\ k\ n\ l\ \rho\ (\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3) = \mathcal{E}\ (&\lambda n\ l\ v_1.\\
&(\textbf{if}\ v_1\\
&\ \ \textbf{then}\ (\mathcal{E}\ k\ n\ l\ \rho\ e_2)\\
&\ \ \textbf{else}\ (\mathcal{E}\ k\ n\ l\ \rho\ e_3)))\\
&(n+1)\ l\ \rho\ e_1
\end{aligned}$$

$$\begin{aligned}
\mathcal{E}\ k\ n\ l\ \rho\ (\diamond e) = \mathcal{E}\ (&\lambda n\ l\ v.\\
&(k\ n\ l\ (\diamond v)))\\
&(n+1)\ l\ \rho\ e
\end{aligned}$$

$$\begin{aligned}
\mathcal{E}\ k\ n\ l\ \rho\ (e_1 \bullet e_2) = \mathcal{E}\ (&\lambda n\ l\ v_1.\\
&(\mathcal{E}\ (\lambda n\ l\ v_2.\\
&\qquad (k\ n\ l\ (v_1 \bullet v_2)))\\
&\quad n\ l\ \rho\ e_2))\\
&(n+1)\ l\ \rho\ e_1
\end{aligned}$$

Additions to the CPS Evaluator to Support AD

$$\overrightarrow{\mathcal{J}}\ v_1\ v_2\ \acute{v}_3 = \mathcal{A}\ (\lambda n\ l\ (v_4 \triangleright \acute{v}_5).\ (v_4, \acute{v}_5))\ 0 \infty v_1\ (v_2 \triangleright \acute{v}_3)$$

$$\overleftarrow{\mathcal{J}}\ v_1\ v_2\ \grave{v}_3 = \mathcal{A}\ (\lambda n\ l\ v.\ \textbf{let}\ (v_4 \triangleleft \grave{v}_5) = v \triangleleft \grave{v}_3\ \textbf{in}\ (v_4, \grave{v}_5))\ 0 \infty v_1\ v_2$$

Additions to the CPS Evaluator to Support AD

$$\overrightarrow{\mathcal{J}}\ v_1\ v_2\ \acute{v}_3 = \mathcal{A}\ (\lambda n\ l\ (v_4 \triangleright \acute{v}_5).\ (v_4, \acute{v}_5))\ 0\ \infty\ v_1\ (v_2 \triangleright \acute{v}_3)$$

$$\overleftarrow{\mathcal{J}}\ v_1\ v_2\ \grave{v}_3 = \mathcal{A}\ (\lambda n\ l\ v.\ \textbf{let}\ (v_4 \triangleleft \grave{v}_5) = v \triangleleft \grave{v}_3\ \textbf{in}\ (v_4, \grave{v}_5))\ 0\ \infty\ v_1\ v_2$$

Additions to the CPS Evaluator to Support AD

$$\begin{aligned}\mathcal{E}\ k\ n\ l\ \rho\ (\overrightarrow{\mathcal{J}}\ e_1\ e_2\ e_3) = \mathcal{E}\ (&\lambda n\ l\ v_1.\\ &(\mathcal{E}\ (\lambda n\ l\ v_2.\\ &\quad(\mathcal{E}\ (\lambda n\ l\ v_3.\\ &\qquad(k\ n\ l\ (\overrightarrow{\mathcal{J}}\ v_1\ v_2\ v_3)))\\ &\quad n\ l\ \rho\ e_3))\\ &\ n\ l\ \rho\ e_2))\\ &(n+1)\ l\ \ \rho\ e_1\end{aligned}$$

$$\begin{aligned}\mathcal{E}\ k\ n\ l\ \rho\ (\overleftarrow{\mathcal{J}}\ e_1\ e_2\ e_3) = \mathcal{E}\ (&\lambda n\ l\ v_1.\\ &(\mathcal{E}\ (\lambda n\ l\ v_2.\\ &\quad(\mathcal{E}\ (\lambda n\ l\ v_3.\\ &\qquad(k\ n\ l\ (\overleftarrow{\mathcal{J}}\ v_1\ v_2\ v_3)))\\ &\quad n\ l\ \rho\ e_3))\\ &\ n\ l\ \rho\ e_2))\\ &(n+1)\ l\ \rho\ e_1\end{aligned}$$

Additions to the CPS Evaluator to Support AD

$$\begin{aligned}\mathcal{E}\ k\ n\ l\ \rho\ (\overrightarrow{\mathcal{J}}\ e_1\ e_2\ e_3) = \mathcal{E}\ (\lambda n\ l\ v_1.\\ (\mathcal{E}\ (\lambda n\ l\ v_2.\\ (\mathcal{E}\ (\lambda n\ l\ v_3.\\ (k\ n\ l\ (\overrightarrow{\mathcal{J}}\ v_1\ v_2\ v_3)))\\ n\ l\ \rho\ e_3))\\ n\ l\ \rho\ e_2))\\ (n+1)\ l\ \ \rho\ e_1\end{aligned}$$

$$\begin{aligned}\mathcal{E}\ k\ n\ l\ \rho\ (\overleftarrow{\mathcal{J}}\ e_1\ e_2\ e_3) = \mathcal{E}\ (\lambda n\ l\ v_1.\\ (\mathcal{E}\ (\lambda n\ l\ v_2.\\ (\mathcal{E}\ (\lambda n\ l\ v_3.\\ (k\ n\ l\ (\overleftarrow{\mathcal{J}}\ v_1\ v_2\ v_3)))\\ n\ l\ \rho\ e_3))\\ n\ l\ \rho\ e_2))\\ (n+1)\ l\ \rho\ e_1\end{aligned}$$

Additions to the CPS Evaluator to Support AD

$$\begin{aligned}\mathcal{E}\ k\ n\ l\ \rho\ (\overrightarrow{\mathcal{J}}\ e_1\ e_2\ e_3) = \mathcal{E}\ (\lambda n\ l\ v_1.\\ (\mathcal{E}\ (\lambda n\ l\ v_2.\\ (\mathcal{E}\ (\lambda n\ l\ v_3.\\ (k\ n\ l\ (\overrightarrow{\mathcal{J}}\ v_1\ v_2\ v_3)))\\ n\ l\ \rho\ e_3))\\ n\ l\ \rho\ e_2))\\ (n+1)\ l\ \ \rho\ e_1\end{aligned}$$

$$\begin{aligned}\mathcal{E}\ k\ n\ l\ \rho\ (\overleftarrow{\mathcal{J}}\ e_1\ e_2\ e_3) = \mathcal{E}\ (\lambda n\ l\ v_1.\\ (\mathcal{E}\ (\lambda n\ l\ v_2.\\ (\mathcal{E}\ (\lambda n\ l\ v_3.\\ (k\ n\ l\ (\overleftarrow{\mathcal{J}}\ v_1\ v_2\ v_3)))\\ n\ l\ \rho\ e_3))\\ n\ l\ \rho\ e_2))\\ (n+1)\ l\ \rho\ e_1\end{aligned}$$

Implementation of the API using the CPS Evaluator

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

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

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

Implementation of the API using the CPS Evaluator

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

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

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

Implementation of the API using the CPS Evaluator

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

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

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

Implementation of the API using the CPS Evaluator

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

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

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

Implementation of the API using the CPS Evaluator

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

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

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

Implementation of the API using the CPS Evaluator

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

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

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

Implementation of the API using the CPS Evaluator

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

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

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

Implementation of the API using the CPS Evaluator

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

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

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

Implementation of the API using the CPS Evaluator

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

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

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

Implementation of the API using the CPS Evaluator

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

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

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

Implementation of the API using the CPS Evaluator

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

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

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

Implementation of the API using the CPS Evaluator

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

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

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

Implementation of the API using the CPS Evaluator

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

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

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

Implementation of the API using the CPS Evaluator

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

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

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

Implementation of the API using the CPS Evaluator

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

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

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

Implementation of the API using the CPS Evaluator

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

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

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

Implementation of the API using the CPS Evaluator

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

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

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

Implementation of the API using the CPS Evaluator

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

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

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

Implementation of the API using the CPS Evaluator

$$e ::= \textbf{interrupt}\ e_1\ e_2\ e_3 \mid \textbf{resume}\ e$$

Implementation of the API using the CPS Evaluator

$$\mathcal{I}\, f\, l = \langle (\lambda x.(\textbf{interrupt}\, f\, x\, l)), \rho_0[f \mapsto f][l \mapsto l] \rangle$$
$$\mathcal{R} = \langle (\lambda z.(\textbf{resume}\, z)), \rho_0 \rangle$$

Implementation of the API using the CPS Evaluator

$$\begin{aligned}
\mathcal{E}\ k\ n\ l\ \rho\ (\textbf{interrupt}\ e_1\ e_2\ e_3) = \mathcal{E}\ (&\lambda n\ l\ v_1.\\
&(\mathcal{E}\ (\lambda n\ l\ v_2.\\
&\quad(\mathcal{E}\ (\lambda n\ l\ v_3.\\
&\qquad\textbf{if}\ \ l = \infty\\
&\qquad\textbf{then}\ (\mathcal{A}\ k\ 0\ v_3\ v_1\ v_2)\\
&\qquad\textbf{else let}\ [\![k,f]\!] = (\mathcal{A}\ k\ 0\ l\ v_1\ v_2)\\
&\qquad\quad\textbf{in}\ [\![k, (\mathcal{I}\ f\ (v_3 - l))]\!])\\
&\quad n\ l\ \rho\ e_3))\\
&n\ l\ \rho\ e_2))\\
&(n+1)\ l\ \rho\ e_1
\end{aligned}$$

$$\begin{aligned}
\mathcal{E}\ k\ n\ l\ \rho\ (\textbf{resume}\ e) = \mathcal{E}\ (&\lambda n\ l\ [\![k',f]\!].\\
&(\mathcal{A}\ k'\ 0\ l\ f\ \perp))\\
&(n+1)\ l\ \rho\ e
\end{aligned}$$

Algorithm for Binary Checkpointing

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

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

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

Algorithm for Binary Checkpointing

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

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

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

Algorithm for Binary Checkpointing

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

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

inductive case:
$$l = \text{PRIMOPS}\ f\ x \tag{1}$$
$$z = \text{INTERRUPT}\ f\ x\ \lfloor \tfrac{l}{2} \rfloor \tag{2}$$
$$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\, \mathcal{R}\ z\ \grave{y} \tag{3}$$
$$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\, (\lambda x.\text{INTERRUPT}\ f\ x\ \lfloor \tfrac{l}{2} \rfloor)\ x\ \grave{z} \tag{4}$$

Algorithm for Binary Checkpointing

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

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

inductive case:
$$l = \text{PRIMOPS}\ f\ x \tag{1}$$
$$z = \text{INTERRUPT}\ f\ x\ \lfloor \tfrac{l}{2} \rfloor \tag{2}$$
$$(y, \grave{z}) = \overset{\checkmark}{\mathcal{J}}\, \mathcal{R}\ z\ \grave{y} \tag{3}$$
$$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\, (\lambda x.\text{INTERRUPT}\ f\ x\ \lfloor \tfrac{l}{2} \rfloor)\ x\ \grave{z} \tag{4}$$

Algorithm for Binary Checkpointing

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

base case ($f\ x$ fast):

$$(y, \grave{x}) = \overset{\leftarrow}{\mathcal{J}}\, f\ x\ \grave{y} \tag{0}$$

inductive case:

$$l = \text{PRIMOPS}\ f\ x \tag{1}$$

$$z = \text{INTERRUPT}\ f\ x\ \lfloor \tfrac{l}{2} \rfloor \tag{2}$$

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

$$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\, (\lambda x.\text{INTERRUPT}\ f\ x\ \lfloor \tfrac{l}{2} \rfloor)\ x\ \grave{z} \tag{4}$$

Algorithm for Binary Checkpointing

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

base case ($f\ x$ fast):

$$(y, \grave{x}) = \overset{\leftarrow}{\mathcal{J}}\, f\ x\ \grave{y} \tag{0}$$

inductive case:

$$l = \text{PRIMOPS}\ f\ x \tag{1}$$

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

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

$$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\, \left(\mathcal{I}\, f\ \left\lfloor \frac{l}{x} \right\rfloor\right)\ x\ \grave{z} \tag{4}$$

Algorithm for Binary Checkpointing

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

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

inductive case:

$$l = \text{PRIMOPS}\ f\ x \tag{1}$$

$$z = \text{INTERRUPT}\ f\ x\ \lfloor \tfrac{l}{2} \rfloor \tag{2}$$

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

$$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\, (\mathcal{I} f\ \lfloor \tfrac{l}{x} \rfloor)\ x\ \grave{z} \tag{4}$$

Nested Interruptions and Resumptions

$$(\mathcal{I}\ (\mathcal{I}\ \ldots\ (\mathcal{I}\ (\mathcal{I}\ f\ l_0)\ l_1)\ \ldots\ l_{i-1})\ l_i)$$

Nested Interruptions and Resumptions

$$
\begin{array}{l}
\langle(\lambda x.(\textbf{interrupt}\, f\ x\ l)),\\
\ \rho_0[f \mapsto \langle(\lambda x.(\textbf{interrupt}\, f\ x\ l)),\\
\qquad \rho_0[f \mapsto \langle \ldots\\
\qquad\qquad (\lambda x.(\textbf{interrupt}\, f\ x\ l)),\\
\qquad\qquad \rho_0[f \mapsto \langle(\lambda x.(\textbf{interrupt}\, f\ x\ l)),\\
\qquad\qquad\qquad \rho_0[f \mapsto f]\\
\qquad\qquad\qquad\ [l \mapsto l_0]\rangle]\\
\qquad\qquad [l \mapsto l_1]\ldots\rangle]\\
\qquad [l \mapsto l_{i-1}]\rangle]\\
\ [l \mapsto l_i]\rangle
\end{array}
$$

Nested Interruptions and Resumptions

$$(\mathcal{I}\ (\mathcal{I}\ \ldots\ (\mathcal{I}\ (\mathcal{I}\ \mathcal{R}\ l_0)\ l_1)\ \ldots\ l_{i-1})\ l_i)$$

Nested Interruptions and Resumptions

$$
\begin{array}{l}
\langle(\lambda x.(\textbf{interrupt}\, f\ x\ l)),\\
\ \rho_0[f \mapsto \langle(\lambda x.(\textbf{interrupt}\, f\ x\ l)),\\
\qquad \rho_0[f \mapsto \langle \ldots\\
\qquad\qquad (\lambda x.(\textbf{interrupt}\, f\ x\ l)),\\
\qquad\qquad \rho_0[f \mapsto \langle(\lambda x.(\textbf{interrupt}\, f\ x\ l)),\\
\qquad\qquad\qquad \rho_0[f \mapsto \langle(\lambda z.(\textbf{resume}\ z)), \rho_0\rangle]\\
\qquad\qquad\qquad\ [l \mapsto l_0]\rangle]\\
\qquad\qquad [l \mapsto l_1]\ldots\rangle]\\
\qquad [l \mapsto l_{i-1}]\rangle]\\
\ [l \mapsto l_i]\rangle
\end{array}
$$

Nested Interruptions and Resumptions

$$[\![k, f]\!]$$

Nested Interruptions and Resumptions

$$[\![k, (\mathcal{I}\ \ldots\ (\mathcal{I}\ (\mathcal{I}\ f\ l_0)\ l_1)\ \ldots\ l_{i-1})]\!]$$

Nested Interruptions and Resumptions

$$
\begin{aligned}
[\![k, \langle (\lambda x.(\textbf{interrupt}\, f\ x\ l)),\ & \\
& \rho_0[f \mapsto \langle \ldots \\
& \qquad (\lambda x.(\textbf{interrupt}\, f\ x\ l)), \\
& \qquad \rho_0[f \mapsto \langle (\lambda x.(\textbf{interrupt}\, f\ x\ l)), \\
& \qquad\qquad \rho_0[f \mapsto f] \\
& \qquad\qquad\quad [l \mapsto l_0]\rangle] \\
& \qquad [l \mapsto l_1] \ldots \rangle] \\
& [l \mapsto l_{i-1}]\rangle]\!]
\end{aligned}
$$

Divide-and-Conquer Checkpointing with the CPS Evaluator

$$e ::= \overset{\checkmark}{\mathcal{J}}\ e_1\ e_2\ e_3$$

Divide-and-Conquer Checkpointing with the CPS Evaluator

$$
\begin{aligned}
\mathcal{E}\ k\ n\ l\ \rho\ (\checkmark{\mathcal{J}}\ e_1\ e_2\ e_3) = \mathcal{E}\ (\lambda n\ l\ v_1.\\
(\mathcal{E}\ (\lambda n\ l\ v_2.\\
(\mathcal{E}\ (\lambda n\ l\ v_3.\\
(k\ n\ l\ (\checkmark{\mathcal{J}}\ v_1\ v_2\ v_3)))\\
n\ l\ \rho\ e_3))\\
n\ l\ \rho\ e_2))\\
(n+1)\ l\ \rho\ e_1
\end{aligned}
$$

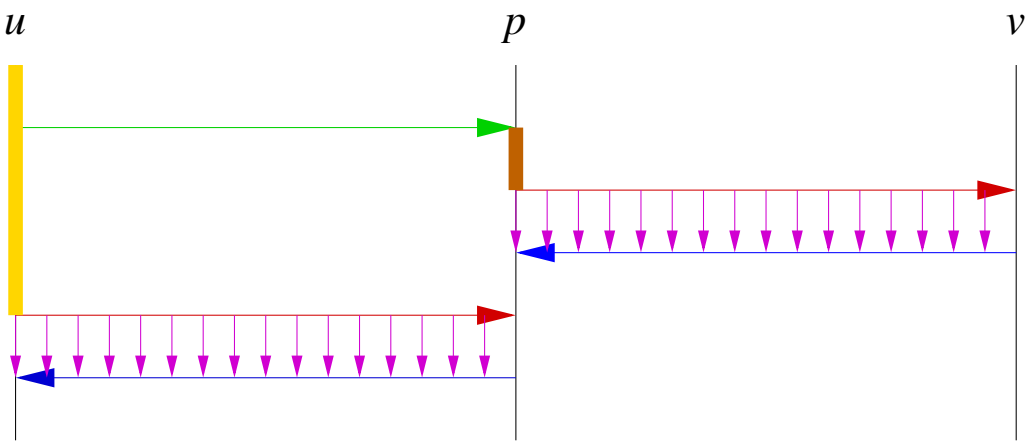
Divide-and-Conquer Checkpointing with the CPS Evaluator

$$
\begin{aligned}
\mathcal{E}\ k\ n\ l\ \rho\ (\checkmark{\mathcal{J}}\ e_1\ e_2\ e_3) = \mathcal{E}\ (\lambda n\ l\ v_1.\\
(\mathcal{E}\ (\lambda n\ l\ v_2.\\
(\mathcal{E}\ (\lambda n\ l\ v_3.\\
(k\ n\ l\ (\checkmark{\mathcal{J}}\ v_1\ v_2\ v_3)))\\
n\ l\ \rho\ e_3))\\
n\ l\ \rho\ e_2))\\
(n+1)\ l\ \rho\ e_1
\end{aligned}
$$

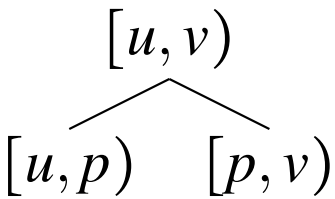
Divide-and-Conquer Checkpointing with the CPS Evaluator

$$
\begin{aligned}
\mathcal{E}\ k\ n\ l\ \rho\ (\checkmark{\mathcal{J}}\ e_1\ e_2\ e_3) = \mathcal{E}\ (\lambda n\ l\ v_1.\\
(\mathcal{E}\ (\lambda n\ l\ v_2.\\
(\mathcal{E}\ (\lambda n\ l\ v_3.\\
(k\ n\ l\ (\checkmark{\mathcal{J}}\ v_1\ v_2\ v_3)))\\
n\ l\ \rho\ e_3))\\
n\ l\ \rho\ e_2))\\
(n+1)\ l\ \rho\ e_1
\end{aligned}
$$

Some Intuition

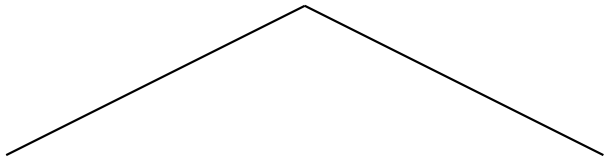


Some Intuition



Some Intuition

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



$$(z, \grave{x}) = (\checkmark\mathcal{J}\ (\mathcal{I} f\ \lfloor \tfrac{l}{2} \rfloor)\ x\ \grave{z}) \qquad (y, \grave{z}) = (\checkmark\mathcal{J}\ \mathcal{R}\ z\ \grave{y})$$

Some Intuition

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

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

inductive case:

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

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

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

$(z, \grave{x}) = \overset{\checkmark}{\mathcal{J}}\ (\mathcal{I}\ f\ \lfloor \frac{l}{x} \rfloor)\ x\ \grave{z}$ (4)

CPS Conversion

$$
\begin{aligned}
\lceil c|k \rfloor &\rightsquigarrow k\ c \\
\lceil x|k \rfloor &\rightsquigarrow k\ x \\
\lceil (\lambda x.e)|k \rfloor &\rightsquigarrow k\ (\lambda_4 k\ x.\lceil e|k \rfloor) \\
\lceil (e_1\ e_2)|k \rfloor &\rightsquigarrow \lceil e_1|(\lambda_3\ x_1. \\
&\qquad \lceil e_2|(\lambda_3\ x_2. \\
&\qquad\quad (x_1\ k\ x_2)), \\
&\qquad \rfloor), \\
&\quad \rfloor
\end{aligned}
$$

CPS Conversion

$$
\begin{aligned}
\lceil c|k \rceil &\rightsquigarrow k\ c \\
\lceil x|k \rceil &\rightsquigarrow k\ x \\
\lceil (\lambda x.e)|k \rceil &\rightsquigarrow k\ (\lambda_4 k\ x.\lceil e|k \rceil) \\
\lceil (e_1\ e_2)|k \rceil &\rightsquigarrow \lceil e_1|(\lambda_3\ x_1. \\
&\qquad \lceil e_2|(\lambda_3\ x_2. \\
&\qquad\quad (x_1\ k\ x_2)), \\
&\qquad \rceil), \\
&\quad \rceil
\end{aligned}
$$

CPS Conversion

$$
\begin{aligned}
\ulcorner c|k,n \urcorner &\rightsquigarrow k\ (n+1)\ c \\
\ulcorner x|k,n \urcorner &\rightsquigarrow k\ (n+1)\ x \\
\ulcorner (\lambda x.e)|k,n \urcorner &\rightsquigarrow k\ (n+1)\ (\lambda_4 k\ n\ x.\ulcorner e|k,n \urcorner) \\
\ulcorner (e_1\ e_2)|k,n \urcorner &\rightsquigarrow \ulcorner e_1|(\lambda_3 n\ x_1. \\
&\qquad \ulcorner e_2|(\lambda_3 n\ x_2. \\
&\qquad\quad (x_1\ k\ n\ x_2)), \\
&\qquad\quad n \urcorner), \\
&\qquad (n+1) \urcorner
\end{aligned}
$$

CPS Conversion

$$
\begin{aligned}
\ulcorner c|k,n,l\urcorner &\rightsquigarrow\ k\ (n+1)\ l\ c \\
\ulcorner x|k,n,l\urcorner &\rightsquigarrow\ k\ (n+1)\ l\ x \\
\ulcorner(\lambda x.e)|k,n,l\urcorner &\rightsquigarrow\ k\ (n+1)\ l\ (\lambda_4 k\ n\ l\ x.\ulcorner e|k,n,l\urcorner) \\
\ulcorner(e_1\ e_2)|k,n,l\urcorner &\rightsquigarrow\ \ulcorner e_1|(\lambda_3 n\ l\ x_1. \\
&\qquad\qquad \ulcorner e_2|(\lambda_3 n\ l\ x_2. \\
&\qquad\qquad\qquad (x_1\ k\ n\ l\ x_2)), \\
&\qquad\qquad n,l\urcorner), \\
&\qquad (n+1),l\urcorner
\end{aligned}
$$

CPS Conversion

$$
\begin{aligned}
\ulcorner c|k,n,l\urcorner &\rightsquigarrow\ k\ (n+1)\ l\ c \\
\ulcorner x|k,n,l\urcorner &\rightsquigarrow\ k\ (n+1)\ l\ x \\
\ulcorner(\lambda x.e)|k,n,l\urcorner &\rightsquigarrow\ k\ (n+1)\ l\ (\lambda_4 k\ n\ l\ x.\ulcorner e|k,n,l\urcorner) \\
\ulcorner(e_1\ e_2)|k,n,l\urcorner &\rightsquigarrow\ \ulcorner e_1|(\lambda_3 n\ l\ x_1. \\
&\qquad \ulcorner e_2|(\lambda_3 n\ l\ x_2. \\
&\qquad\qquad (x_1\ k\ n\ l\ x_2)), \\
&\qquad n,l\urcorner), \\
&\quad (n+1),l\urcorner
\end{aligned}
$$

$$\langle\!\langle e \rangle\!\rangle_{k,n,l} \rightsquigarrow \textbf{if } n = l \textbf{ then } [\![k, \lambda_4 k\ n\ l\ \_.e]\!] \textbf{ else } e$$

CPS Conversion

$$
\begin{aligned}
\ulcorner c|k,n,l\urcorner &\rightsquigarrow \langle\!\langle k\ (n+1)\ l\ c\rangle\!\rangle_{k,n,l} \\
\ulcorner x|k,n,l\urcorner &\rightsquigarrow \langle\!\langle k\ (n+1)\ l\ x\rangle\!\rangle_{k,n,l} \\
\ulcorner(\lambda x.e)|k,n,l\urcorner &\rightsquigarrow \langle\!\langle k\ (n+1)\ l\ (\lambda_4 k\ n\ l\ x.\ulcorner e|k,n,l\urcorner)\rangle\!\rangle_{k,n,l} \\
\ulcorner(e_1\ e_2)|k,n,l\urcorner &\rightsquigarrow \langle\!\langle \ulcorner e_1|(\lambda_3 n\ l\ x_1.\\
&\qquad \ulcorner e_2|(\lambda_3 n\ l\ x_2.\\
&\qquad\quad (x_1\ k\ n\ l\ x_2)),\\
&\qquad n,l\urcorner),\\
&\quad (n+1),l\urcorner\rangle\!\rangle_{k,n,l}
\end{aligned}
$$

$$
\langle\!\langle e\rangle\!\rangle_{k,n,l} \rightsquigarrow \textbf{if } n = l \textbf{ then } [\![k, \lambda_4 k\ n\ l\ \_.e]\!] \textbf{ else } e
$$

CPS Conversion

$$
\begin{aligned}
\ulcorner c|k,n,l\urcorner &\rightsquigarrow \langle\!\langle k\ (n+1)\ l\ c\rangle\!\rangle_{k,n,l} \\
\ulcorner x|k,n,l\urcorner &\rightsquigarrow \langle\!\langle k\ (n+1)\ l\ x\rangle\!\rangle_{k,n,l} \\
\ulcorner(\lambda x.e)|k,n,l\urcorner &\rightsquigarrow \langle\!\langle k\ (n+1)\ l\ (\lambda_4 k\ n\ l\ x.\ulcorner e|k,n,l\urcorner)\rangle\!\rangle_{k,n,l} \\
\ulcorner(e_1\ e_2)|k,n,l\urcorner &\rightsquigarrow \langle\!\langle \ulcorner e_1|(\lambda_3 n\ l\ x_1. \\
&\qquad \ulcorner e_2|(\lambda_3 n\ l\ x_2. \\
&\qquad\qquad (x_1\ k\ n\ l\ x_2)), \\
&\qquad n,l\urcorner), \\
&\quad (n+1),l\urcorner\rangle\!\rangle_{k,n,l}
\end{aligned}
$$

$$\vdots$$

$$\langle\!\langle e\rangle\!\rangle_{k,n,l} \rightsquigarrow \textbf{if } n = l \textbf{ then } [\![k, \lambda_4 k\ n\ l\ \_.e]\!] \textbf{ else } e$$

CPS Conversion then Direct-Style Evaluation

Instead of using the CPS evaluator on code in direct style, can use the direct-style evaluator on code in CPS (converted from code in direct style by CPS conversion).

With proper machinery, can support the general purpose interruption and resumption interface and $\overset{\checkmark}{\mathcal{J}}$.

CPS Conversion then Direct-Style Evaluation

Instead of using the CPS evaluator on code in direct style, can use the direct-style evaluator on code in CPS (converted from code in direct style by CPS conversion).

With proper machinery, can support the general purpose interruption and resumption interface and $\overset{\checkmark}{\mathcal{J}}$.

Code Generation

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

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 (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

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

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

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

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

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

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

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

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

Three 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.

Space Complexity

	primal	snapshots	tape	total	overhead
$\overleftarrow{\mathcal{J}}$	$O(w)$		$O(t)$	$O(w+t) = O(t)$	$O(t)$
$\overleftarrow{\mathcal{J}}$	$O(w)$	$O(w \log t)$	$O(1)$	$O(w \log t)$	$O(\log t)$

$$O(t) \geq O(w)$$

Time Complexity

	primal	recompute primal	forward sweep	reverse sweep	total	overhead
$\overleftarrow{\mathcal{J}}$	$O(t)$		$O(t)$	$O(t)$	$O(t)$	$O(1)$
$\overleftarrow{\mathcal{J}}$	$O(t)$	$O(t \log t)$	$O(t)$	$O(t)$	$O(t \log t)$	$O(\log t)$

FORTRAN Example

```
      function ilog2(n)
      ilog2 = dlog(real(n, 8))/dlog(2.0d0)
      end

      subroutine f(n, x, y)
      y = x
c$ad binomial-ckp n+1 30 1
      do i = 1, n
         m = 2**(ilog2(n)-
     +           ilog2(1+int(mod(real(x, 8)**3*real(i, 8)*
     +                                          1007.0d0,
     +                              real(n, 8)))))
         do j = 1, m
            y = y*y
            y = sqrt(y)
         end do
      end do
      end

      program main
      read *, n
      read *, x
      read *, yb
      call f(n, x, y)
      call f_b(n, x, xb, y, yb)
      print *, y
      print *, xb
      end
```

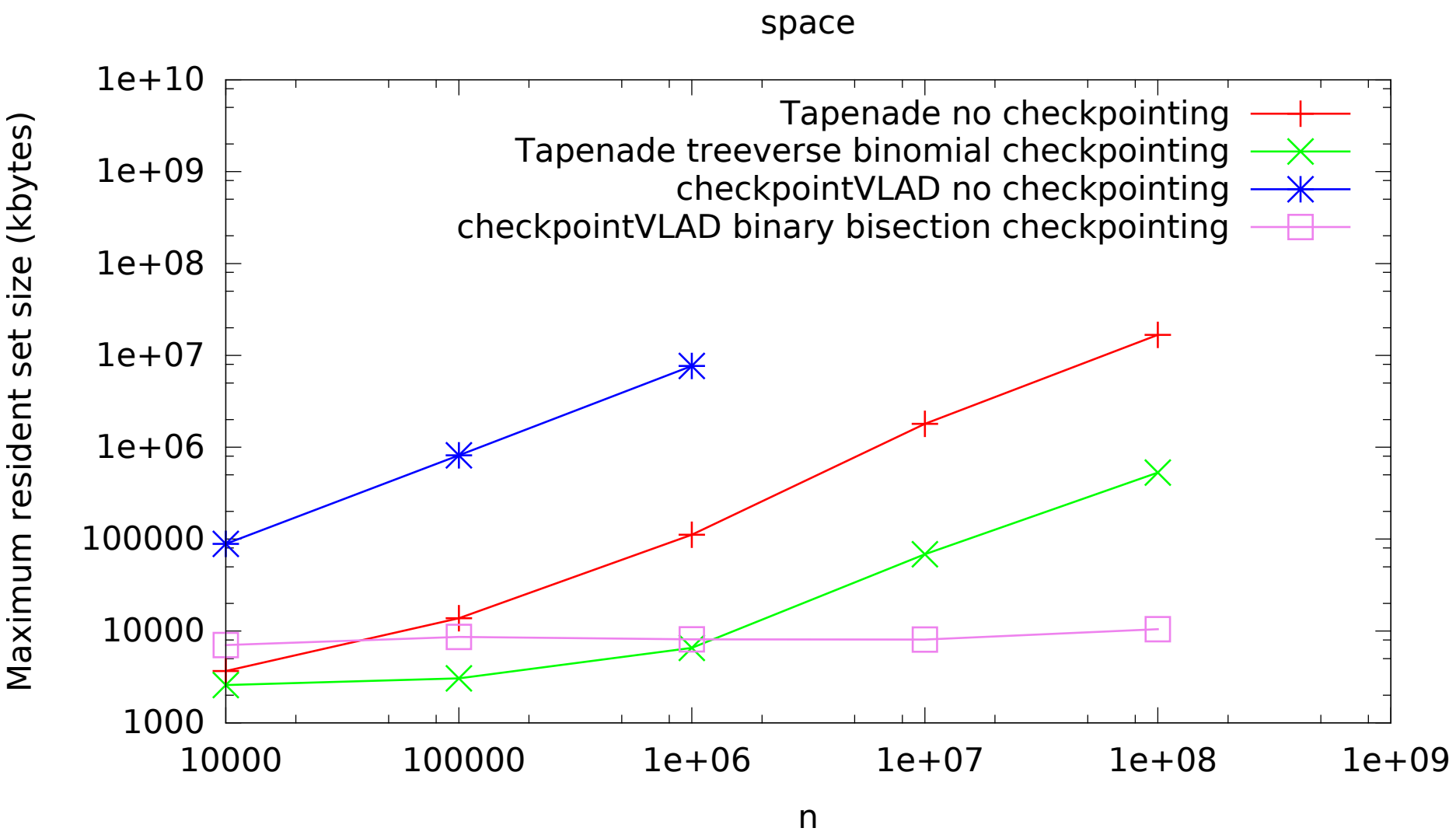
CHECKPOINTVLAD Example

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

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

(define (ilog2 n) (floor (/ (log n) (log 2))))

(define (f n x)
 (let outer ((i 1) (y x))
  (if (> i n)
      y
      (outer
       (+ i 1)
       (let ((m (expt
                 2
                 (- (ilog2 n)
                    (ilog2
                     (+ 1 (modulo (* (* (floor (expt 3 x)) i)
                                     1007)
                                  n)))))))
        (let inner ((j 1) (y y))
         (if (> j m)
             y
             (inner (+ j 1) (sqrt (* y y)))))))))))

(let* ((n (read-real))
       (x (read-real))
       (y-grave (read-real))
       (result (checkpoint-*j (lambda (x) (f n x)) x y-grave)))
 (cons (write-real (car result)) (write-real (cdr result))))
```

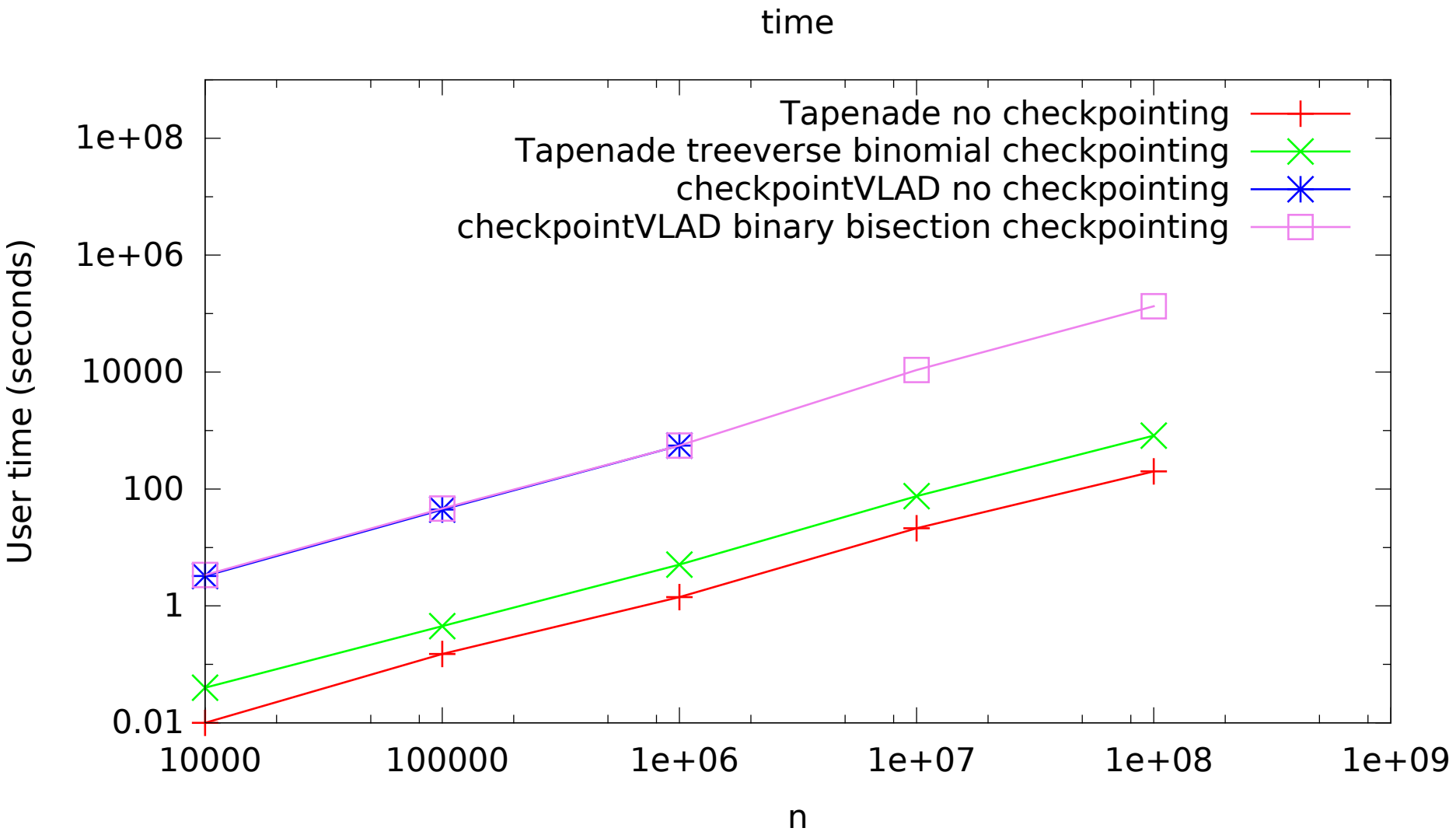
Complexity Analysis

	space	time
primal	$O(1)$	$O(n)$
TAPENADE no checkpointing	$O(n)$	$O(n)$
TAPENADE divide-and-conquer checkpointing	$O(1)$	$O\left(n\sqrt[d]{n}\right)$
CHECKPOINTVLAD no checkpointing	$O(n)$	$O(n)$
CHECKPOINTVLAD divide-and-conquer checkpointing	$O(\log n)$	$O(n \log n)$

Space Usage of Example



Time Usage of Example



FORTRAN Determinant Example

```
      subroutine ident(n, a)
      double precision a(n, n)
      do i = 1, n
         do j = 1, n
            a(i, j) = 0.0d0
            if (i.eq.j) a(i, j) = 1.0d0
         end do
      end do
      end

      subroutine det(n, a, d)
      include 'determinant.inc'
      double precision a(n, n), b(nn, nn), c, d, e
      do i = 1, n
         do j = 1, n
            b(i, j) = a(i, j)
         end do
      end do
      d = 1.0d0
c$ad binomial-ckp n+1 30 1
      do i = 1, n
         c = b(i, i)
         d = d*c
         do j = i, n
            b(i, j) = b(i, j)/c
         end do
         do j = i+1, n
            e = b(j, i)
            do k = i+1, n
               b(j, k) = b(j, k)-e*b(i, k)
            end do
         end do
      end do
      end

      program main
      include 'determinant.inc'
      double precision a(nn, nn), ab(nn, nn), d, db
      read *, n
      call ident(n, a)
      call det(n, a, d)
      db = 1.0d0
      call det_b(n, a, ab, d, db)
      call det(n, ab, d)
      print *, d
      end
```

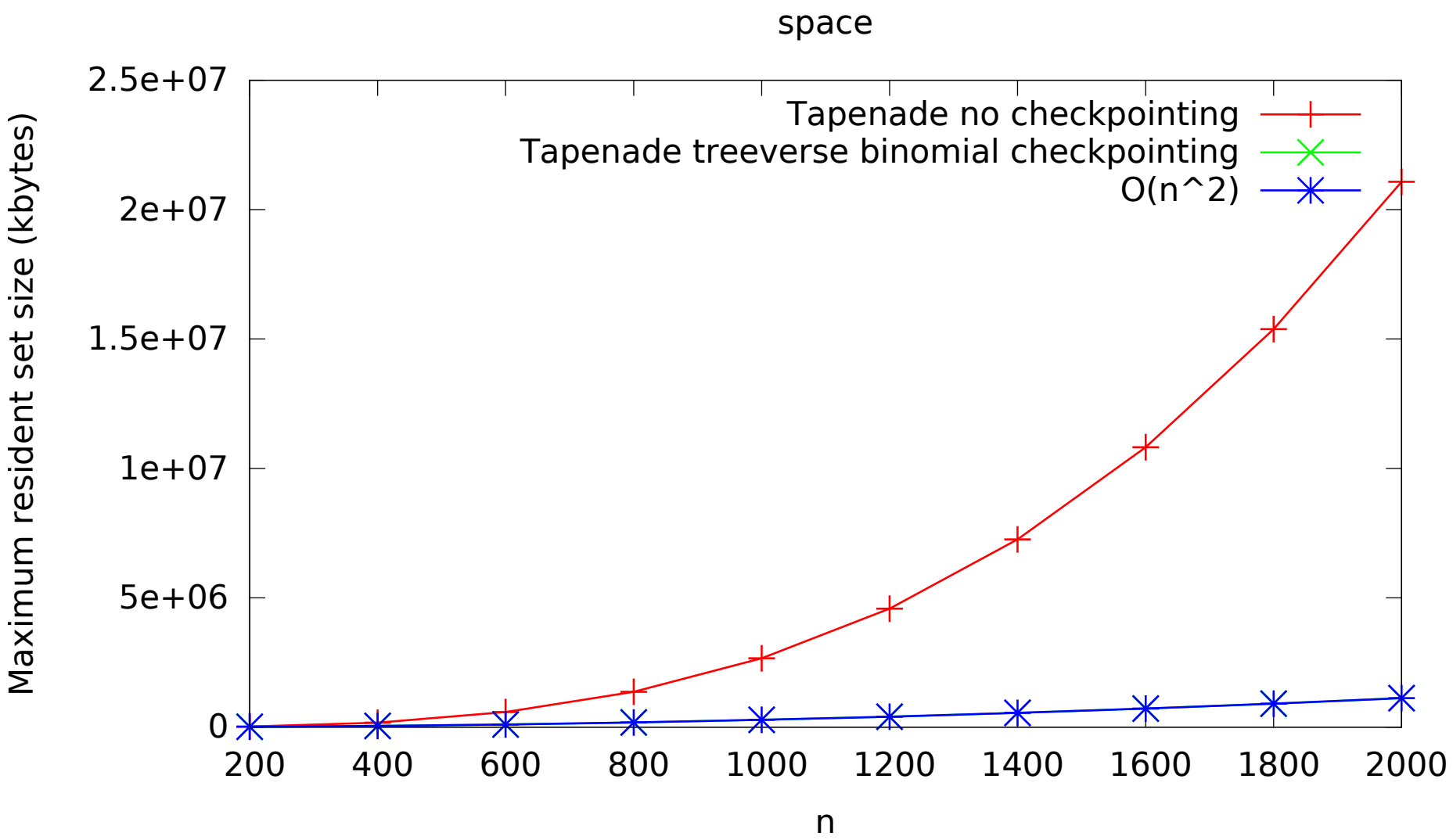
CHECKPOINTVLAD Determinant Example

```
(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))

(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))))
```

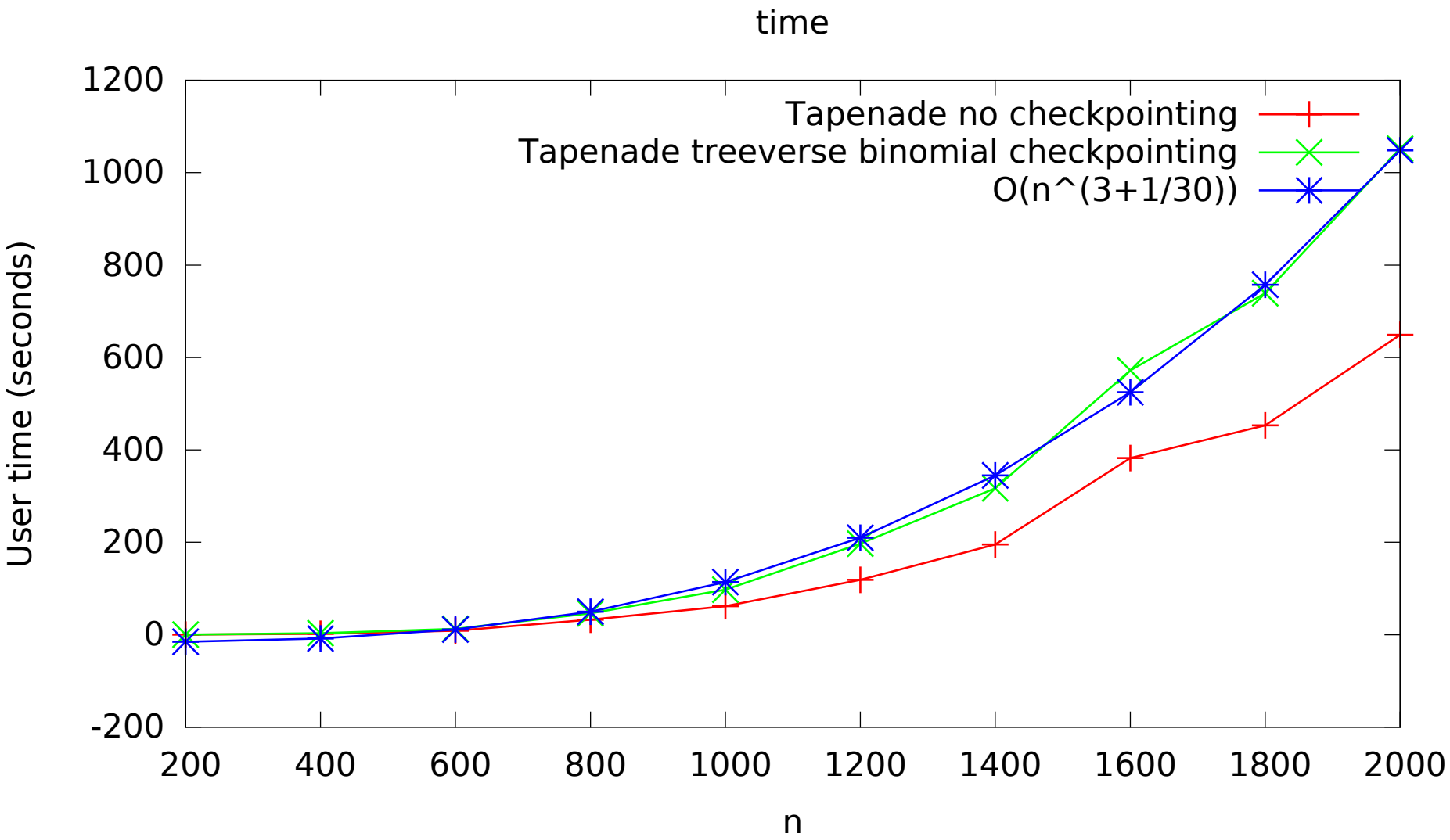
Complexity Analysis

	space	time
FORTRAN primal	$O(n^2)$	$O(n^3)$
TAPENADE no checkpointing	$O(n^3)$	$O(n^3)$
TAPENADE divide-and-conquer checkpointing	$O(n^2)$	$O\left(n^3 \sqrt[d]{n}\right)$
CHECKPOINTVLAD primal	$O(n^2)$	$O(n^4)$
CHECKPOINTVLAD no checkpointing	$O(n^3)$	$O(n^4)$
CHECKPOINTVLAD divide-and-conquer checkpointing	$O(n^2 \log n)$	$O(n^4 \log n)$

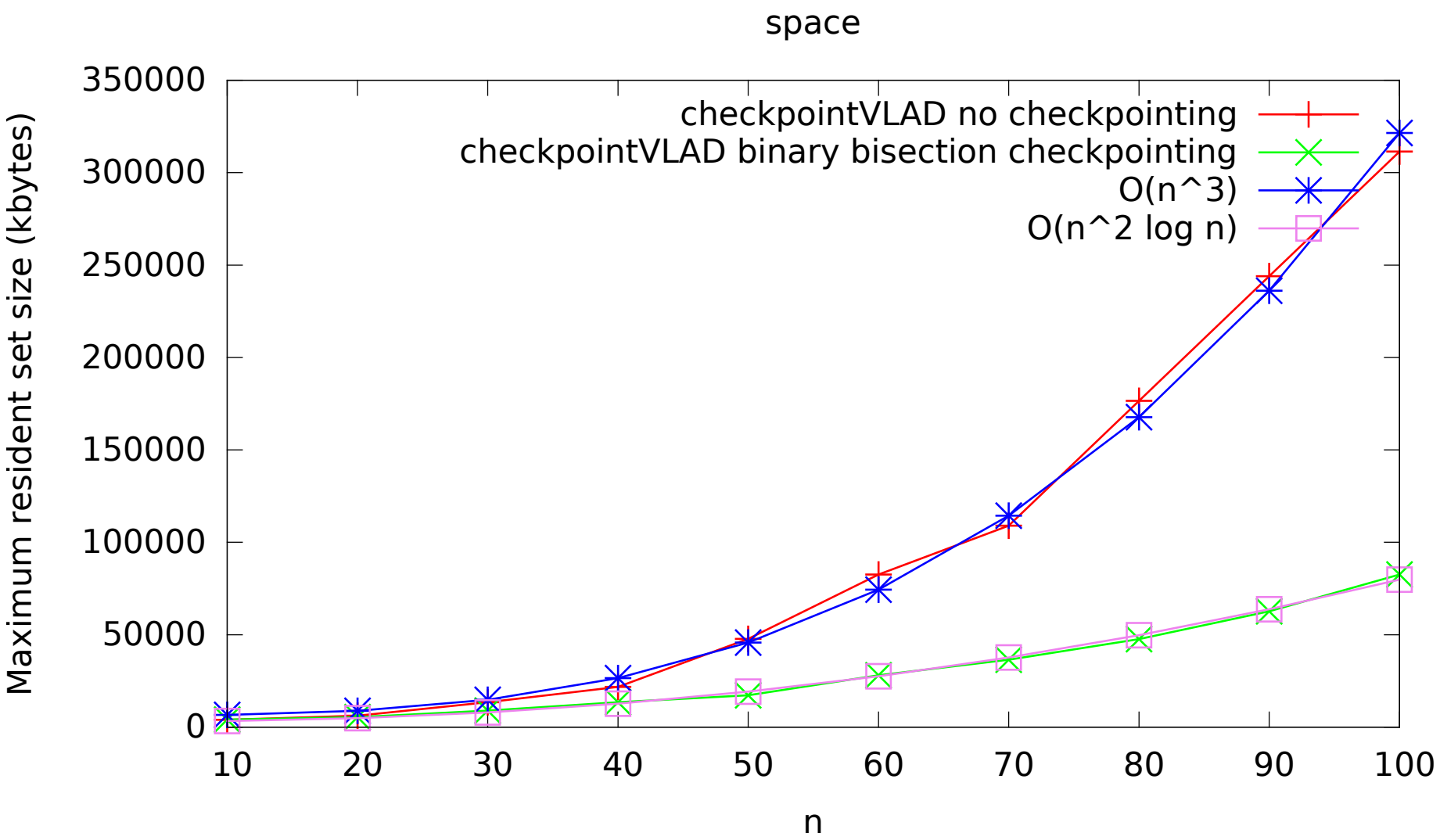
Space Usage of FORTRAN Determinant Example



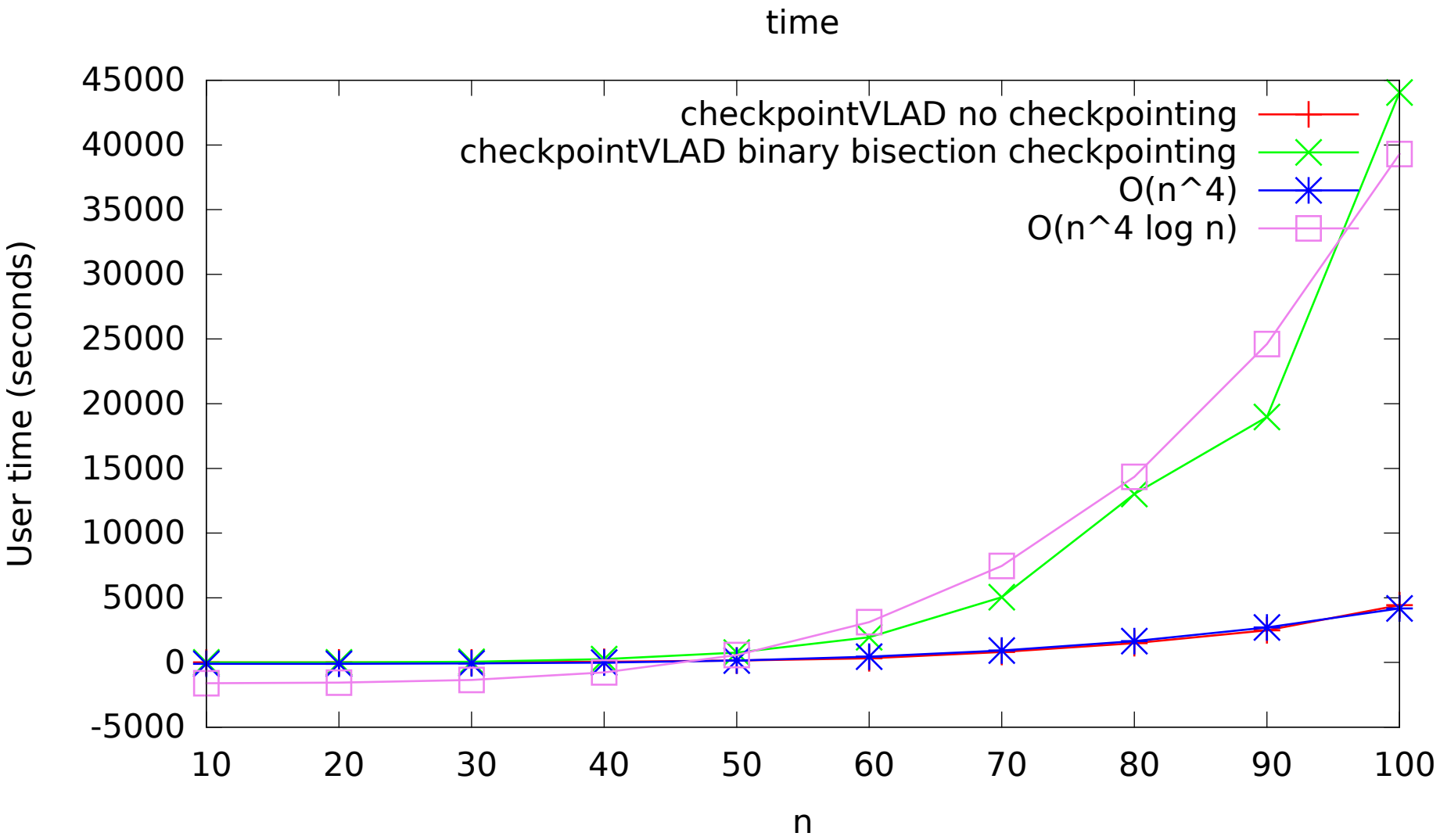
Time Usage of FORTRAN Determinant Example



Space Usage of CHECKPOINTVLAD Determinant Example



Time Usage of CHECKPOINTVLAD Determinant Example



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