Asymptotic Speedup via Effect Handlers

Daniel Hillerström

Laboratory for Foundations of Computer Science
The University of Edinburgh, Scotland, UK

December 13, 2022

Formal Analysis, Theory and Algorithms
School of Computing Science
University of Glasgow

(joint work with Sam Lindley and John Longley)
Effects for Efficiency
Asymptotic Speedup with First-Class Control

 DANIEL HILLERSTRÖM, The University of Edinburgh, UK
 SAM LINDLEY, The University of Edinburgh and Imperial College London and Heriot-Watt University, UK
 JOHN LONGLEY, The University of Edinburgh, UK

We study the fundamental efficiency of delimited control. Specifically, we show that effect handlers enable an asymptotic improvement in runtime complexity for a certain class of functions. We consider the generic count problem using a pure PCF-like base language $\lambda_b$ and its extension with effect handlers $\lambda_h$. We show that $\lambda_h$ admits an asymptotically more efficient implementation of generic count than any $\lambda_b$ implementation. We also show that this efficiency gap remains when $\lambda_h$ is extended with mutable state.

To our knowledge this result is the first of its kind for control operators.

CCS Concepts: • Theory of computation → Lambda calculus; Abstract machines; Control primitives.

Additional Key Words and Phrases: effect handlers, asymptotic complexity analysis, generic search

ACM Reference Format:
Asymptotic Speedup via Effect Handlers

DAVID HILLERSTROM
Laboratory for Foundations of Computer Science
The University of Edinburgh, Scotland, UK
(e-mail: david.hillerstrom@ed.ac.uk)

SAM LINDLEY
Laboratory for Foundations of Computer Science
The University of Edinburgh, Scotland, UK
(e-mail: sam.lindley@ed.ac.uk)

JOHN LONGLEY
Laboratory for Foundations of Computer Science
The University of Edinburgh, Scotland, UK
(e-mail: jrl@staffmail.ed.ac.uk)

Abstract

We study a fundamental efficiency benefit afforded by delimited control, showing that for certain higher-order functions, a language with advanced control features offers an asymptotic improvement in runtime over a language without them. Specifically, we consider the generic count problem in the context of a pure PCF-like base language \( \lambda_b \) and an extension \( \lambda_h \) with general effect handlers. We prove that \( \lambda_b \) admits an asymptotically more efficient implementation of generic count than any implementation in \( \lambda_h \). We also show that this gap remains even when \( \lambda_b \) is extended to a language \( \lambda_d \) with affine effect handlers, which is strong enough to encode exceptions, local state, coroutines and single-shot continuations. This locates the efficiency difference in the gap between ‘single-shot’ and
Effect handlers provide a request-response paradigm-style of programming

- Perform an abstract request: \( \text{do } \ell \ V \) (Plotkin and Power 2003)
- Respond to requests in some computation: \( \text{handle } M \text{ with } H \) (Plotkin and Pretnar 2009)
Effect handlers primer

Effect handlers provide a request-response paradigm-style of programming

- Perform an abstract request: do $\ell \ V$ (Plotkin and Power 2003)
- Respond to requests in some computation: handle $M$ with $H$ (Plotkin and Pretnar 2009)

Example: Count the number of true valuations.
One request operation Branch : Unit $\rightarrow$ Bool.

$$(\text{do Branch } \langle \rangle \ || \text{do Branch } \langle \rangle)$$
Effect handlers provide a request-response paradigm-style of programming

- Perform an abstract request: do $\ell V$ (Plotkin and Power 2003)
- Respond to requests in some computation: handle $M$ with $H$ (Plotkin and Pretnar 2009)

**Example:** Count the number of true valuations.

One request operation Branch : Unit $\rightarrow$ Bool.

$$(\text{do Branch} \langle \rangle || \text{do Branch} \langle \rangle)$$

**Computation tree model**
Effect handlers provide a request-response paradigm-style of programming

- Perform an abstract request: \( \ell V \) (Plotkin and Power 2003)
- Respond to requests in some computation: handle \( M \) with \( H \) (Plotkin and Pretnar 2009)

**Example:** Count the number of true valuations.

One request operation \( \text{Branch} : \text{Unit} \to \text{Bool} \).

\[
\text{handle} \ (\text{do} \ \text{Branch} \langle \rangle \ || \ \text{do} \ \text{Branch} \langle \rangle) \ \text{with}
\]

Computation tree model

- \( \text{Branch} \)
  - \( \text{true} \)
  - \( \text{Branch} \)
    - \( \text{true} \)
    - \( \text{false} \)
Effect handlers provide a request-response paradigm-style of programming

- Perform an abstract request: do \( \ell V \) (Plotkin and Power 2003)
- Respond to requests in some computation: handle \( M \) with \( H \) (Plotkin and Pretnar 2009)

**Example:** Count the number of true valuations.

One request operation `Branch : Unit \to\ Bool`.

```latex
\[
\text{handle (do Branch } \langle \rangle \text{ || do Branch } \langle \rangle \text{) with val x } \mapsto \text{if x then 1 else 0}
\]
```

**Computation tree model**

```
Branch
  /     \
true   Branch
  /   \
true   false
```
Effect handlers primer

Effect handlers provide a request-response paradigm-style of programming

- Perform an abstract request: do ℓ V (Plotkin and Power 2003)
- Respond to requests in some computation: handle M with H (Plotkin and Pretnar 2009)

**Example:** Count the number of true valuations.
One request operation Branch : Unit → Bool.

```plaintext
handle (do Branch ⟨⟩ || do Branch ⟨⟩) with
  val x  ↦ if x then 1 else 0
  Branch ⟨⟩ resume  ↦ resume true + resume false
```

**Computation tree model**
Motivation: space exploration
Motivation: space exploration

Expressivity of $\mathcal{L} \subset \mathcal{L}'$

- **Computability**: Can some things be done in $\mathcal{L}'$ but not in $\mathcal{L}$?
- **Complexity**: Can some things be done faster in $\mathcal{L}'$ than in $\mathcal{L}$?
- **Programmability**: Can some things be done more easily in $\mathcal{L}'$ than in $\mathcal{L}$?
Motivation: space exploration

Asymptotic speedup with effect handlers

The **generic count** problem

\[
\text{count}_n : ((\text{Nat}_n \rightarrow \text{Bool}) \rightarrow \text{Bool}) \rightarrow \text{Nat}
\]
Motivation: space exploration

Asymptotic speedup with effect handlers

The **generic count** problem

\[
\text{count}_n : ((\text{Nat}_n \rightarrow \text{Bool}) \rightarrow \text{Bool}) \rightarrow \text{Nat}
\]

- point: boolean-valued vector of size \( n \)
Motivation: space exploration

Asymptotic speedup with effect handlers

The **generic count** problem

\[
\text{count}_n : ((\text{Nat}_n \rightarrow \text{Bool}) \rightarrow \text{Bool}) \rightarrow \text{Nat}
\]

- point: boolean-valued vector of size \( n \)
- predicate: encodes some search problem (e.g. \( n \)-Queens)
Motivation: space exploration

Asymptotic speedup with effect handlers

The **generic count** problem

$$\text{count}_n : ((\text{Nat}_n \to \text{Bool}) \to \text{Bool}) \to \text{Nat}$$

- point: boolean-valued vector of size $n$
- predicate: encodes some search problem (e.g. $n$-Queens)
- $\text{count}_n P$: returns number of $n$-points satisfying $P$
Motivation: space exploration

Asymptotic speedup with effect handlers

The **generic count** problem

\[
\text{count}_n : ((\text{Nat}_n \to \text{Bool}) \to \text{Bool}) \to \text{Nat}
\]

- point: boolean-valued vector of size \(n\)
- predicate: encodes some search problem (e.g. \(n\)-Queens)
- \(\text{count}_n P\): returns number of \(n\)-points satisfying \(P\)

Fix \(\mathcal{L} := \text{PCF}\)
Motivation: space exploration

Asymptotic speedup with effect handlers

The **generic count** problem

\[ \text{count}_n : ((\text{Nat}_n \rightarrow \text{Bool}) \rightarrow \text{Bool}) \rightarrow \text{Nat} \]

- point: boolean-valued vector of size \( n \)
- predicate: encodes some search problem (e.g. \( n \)-Queens)
- \( \text{count}_n \ P \): returns number of \( n \)-points satisfying \( P \)

Fix \( \mathcal{L} := \text{PCF} \) and \( \mathcal{L}' := \text{PCF}_h \) with effect handlers
Motivation: space exploration

Asymptotic speedup with effect handlers

The **generic count** problem

\[
\text{count}_n : \left( (\text{Nat}_n \to \text{Bool}) \to \text{Bool} \right) \to \text{Nat}
\]

- **point**: boolean-valued vector of size \( n \)
- **predicate**: encodes some search problem (e.g. \( n \)-Queens)
- **count\(_n\), \( P \)**: returns number of \( n \)-points satisfying \( P \)

Fix \( \mathcal{L} := \text{PCF} \) and \( \mathcal{L}' := \text{PCF}_h \) with effect handlers

- There **exists** an implementation, \( \text{effcount} \in \text{PCF}_h \), of generic count such that \( \text{effcount} \in O(2^n) \)
Motivation: space exploration

Asymptotic speedup with effect handlers

The **generic count** problem

\[
\text{count}_n : ((\text{Nat}_n \to \text{Bool}) \to \text{Bool}) \to \text{Nat}
\]

- point: boolean-valued vector of size \( n \)
- predicate: encodes some search problem (e.g. \( n \)-Queens)
- \( \text{count}_n P \): returns number of \( n \)-points satisfying \( P \)

Fix \( \mathcal{L} := \text{PCF} \) and \( \mathcal{L}' := \text{PCF}_h \) with effect handlers

1. There **exists** an implementation, \( \text{effcount} \in \text{PCF}_h \), of generic count such that \( \text{effcount} \in \mathcal{O}(2^n) \)
2. For all implementations, \( \text{count} \in \text{PCF} \), of generic count it holds that \( \text{count} \in \Omega(n2^n) \)
Methodology

One ground rule:

No change of type signatures is allowed!

- Fixed signature count\(n\) : \(((\text{Nat}_n \rightarrow \text{Bool}) \rightarrow \text{Bool}) \rightarrow \text{Nat}\)
- Prohibits translation of PCF\(_h\) into PCF (interpreter / CPS)
- Programming against a fixed interface

(PCF due to Plotkin (1977))
A predicate and its model

\[ \text{ex} : (\text{Nat}_3 \rightarrow \text{Bool}) \rightarrow \text{Bool} \]
\[ \text{ex} = \lambda p. \text{if } p \ 0 \text{ then } p \ 1 \text{ xor } p \ 2 \]
\[ \text{else not (} p \ 2 \text{ xor } p \ 1 \text{)} \]

Behaviour of \( \text{ex} (\lambda j. \text{nth [true, false, true]} j) \):
Consider a constant predicate, e.g.,

\[ T_0 \doteq \lambda q. \text{true} \]

whose model is

\[ \text{true} \]

**Potential problem**: the runtime of the predicate doesn’t depend on the input \( q \).
Consider the following identity predicate over $\mathbb{B}^1 \rightarrow \mathbb{B}$

$$l_2 \doteq \lambda q. (q 0) \&\& (q 0)$$

whose model is

Potential problem: Repeated queries may yield imperfect binary tree models.
Consider the following false-only yielding predicate

\[ \infty \triangleq \text{rec } P \text{.if } q \text{ if } q \text{ then } P q \text{ else false} \]

whose model is infinite

**Potential problem**: Possibly infinite runtime.
Restriction to \( n \)-standard predicates

Properties of an \( n \)-standard model

- Perfect binary tree of height \( n > 0 \)
- Contains every query \(?j\) for \( j \in \{0, \ldots, n - 1\} \)
- No repeated queries along any path

Example: \( \text{ex} \) is 3-standard
Definition (untimed decision tree)

1. The address set $\text{Addr}$ is simply the set $\mathbb{B}^*$ of finite lists of booleans. If $bs, bs' \in \text{Addr}$, we write $bs \sqsubseteq bs'$ (resp. $bs \sqsubset bs'$) to mean that $bs$ is a prefix (resp. proper prefix) of $bs'$.

2. The label set $\text{Lab}$ consists of queries parameterised by a natural number and answers parameterised by a boolean:

$$\text{Lab} \doteq \{?k \mid k \in \mathbb{N}\} \cup \{!b \mid b \in \mathbb{B}\}$$

3. An (untimed) decision tree is a partial function $\tau : \text{Addr} \rightarrow \text{Lab}$ such that:
   - The domain of $\tau$ (written $\text{dom}(\tau)$) is prefix closed.
   - Answer nodes are always leaves: if $\tau(bs) = !b$ then $\tau(bs')$ is undefined whenever $bs \sqsubseteq bs'$.

Definition (timed decision tree)

A timed decision tree is a partial function $\tau : \text{Addr} \rightarrow \text{Lab} \times \mathbb{N}$ such that its first projection $bs \mapsto \tau(bs).1$ is a decision tree. We write $\text{labs}(\tau)$ for the first projection $(bs \mapsto \tau(bs).1)$ and $\text{steps}(\tau)$ for the second projection $(bs \mapsto \tau(bs).2)$ of a timed decision tree.
Restriction to \( n \)-standard trees and predicates, formally

**Definition (\( n \)-standard trees and predicates)**

An \( n \)-predicate tree \( \tau \) is said to be \( n \)-**standard** if the following hold:

- The domain of \( \tau \) is precisely \( \text{Addr}_n \), the set of bit vectors of length \( \leq n \).
- There are no repeated queries along any path in \( \tau \):
  \[
  \forall bs, bs' \in \text{dom}(\tau), \ k \in \mathbb{N}_n. \ bs \sqsubseteq bs' \land \tau(bs) = \tau(bs') = \bot_k \Rightarrow bs = bs'
  \]

A timed decision tree \( \tau \) is \( n \)-standard if its underlying untimed decision tree \( (bs \mapsto \tau(bs)).1 \) is so. An \( n \)-predicate \( P \) is \( n \)-standard if its model is \( n \)-standard.
Given an $n$-standard tree $\tau$, we may associate to each address $bs \in \text{dom}(\tau)$ a $\lambda_b$ term $T(\tau, bs)$ (with free variable $q : (\text{Nat}_n \rightarrow \text{Bool})$) by reverse induction on the length of $bs$:

$$T(\tau, bs) \overset{=} \equiv b \quad \text{if } \tau(bs) \neq b$$

$$T(\tau, bs) \overset{=} \equiv \text{if } q(k) \text{ then } T(\tau, bs ++ [\text{true}]) \text{ else } T(\tau, bs ++ [\text{false}]) \quad \text{if } \tau(bs) = ?k$$

We then define

$$P(\tau) \overset{=} \equiv \lambda q. \ T(\tau, [])$$

such that model of $P(\tau)$ is $\tau$, and call $P(\tau)$ the **canonical $n$-standard predicate** for $\tau$. 
Example: Canonicalising \( \text{ex} \)

\[
ex : (\text{Nat}_3 \to \text{Bool}) \to \text{Bool}
\]

\[
ex \equiv \lambda p. \text{if } p \mathsf{0} \text{ then } p \mathsf{1} \text{ xor } p \mathsf{2} \text{ else not } (p \mathsf{2} \text{ xor } p \mathsf{1})
\]

\[
= ex : (\text{Nat}_3 \to \text{Bool}) \to \text{Bool}
ex \equiv \lambda p. \text{if } p \mathsf{0} \text{ then }
\quad \text{if } p \mathsf{1} \text{ then }
\quad \quad \text{if } p \mathsf{2} \text{ then true xor true }
\quad \quad \text{else true xor false }
\quad \text{else }
\quad \quad \text{if } p \mathsf{2} \text{ then false xor true }
\quad \quad \text{else false xor false }
\quad \text{else }
\quad \quad \text{if } p \mathsf{2} \text{ then }
\quad \quad \quad \text{if } p \mathsf{1} \text{ then not (true xor true) }
\quad \quad \quad \text{else not (true xor false) }
\quad \quad \text{else }
\quad \quad \quad \text{if } p \mathsf{1} \text{ then not (false xor true) }
\quad \quad \quad \text{else not (false xor false) }
\]
Specification of generic counting

**Definition (n-points)**

A closed value \( Q : (\text{Nat}_n \to \text{Bool}) \) is said to be a **syntactic n-point** if:

\[
\forall k \in \text{N}_n. \exists b \in \mathbb{B}. \quad Q(k) \leadsto^* b
\]

A **semantic n-point** \( \pi \) is a mathematical function \( \pi : \text{N}_n \to \mathbb{B} \). Any syntactic n-point \( Q \) is said to **denote** the semantic n-point \( \mathbb{P}[Q] \) given by:

\[
\forall k \in \text{N}_n, \ b \in \mathbb{B}. \quad \mathbb{P}[Q](k) = b \iff Q(k) \leadsto^* b
\]

Any two syntactic n-points \( Q \) and \( Q' \) are said to be **distinct** if \( \mathbb{P}[Q] \neq \mathbb{P}[Q'] \).

**Definition (Generic count specification)**

1. The **count** of a semantic n-predicate \( \Pi \), written \( \#\Pi \), is simply the number of semantic n-points \( \pi \in \mathbb{B}^n \) for which \( \Pi(\pi) = \text{true} \).
2. If \( P \) is any n-predicate, we say that \( K \) correctly counts \( P \) if \( K P \leadsto^* m \), where \( m = \#\mathbb{P}[P] \).
Efficient generic count with effect handlers

\[
\text{effcount} : ((\text{Nat} \to \text{Bool}) \to \text{Bool}) \to \text{Nat}
\]

\[
\text{effcount} \doteq
\]

Behaviour of effcount ex:
Efficient generic count with effect handlers

effcount : ((Nat → Bool) → Bool) → Nat
effcount ≡ λP.  P (λj. do Branch ⟨⟩)

(where Branch : ⟨⟩ → Bool ∈ Σ)

Behaviour of effcount ex:

Steps: $O(2^n)$
Efficient generic count with effect handlers

\[
\text{effcount} : ((\text{Nat} \to \text{Bool}) \to \text{Bool}) \to \text{Nat}
\]
\[
\text{effcount} \doteq \lambda P. \; \text{handle } P \; (\lambda j.\; \text{do } \text{Branch} \; \langle \rangle ) \; \text{with}
\]

(where \( \text{Branch} : \langle \rangle \to \text{Bool} \in \Sigma \))

Behaviour of effcount ex:

```
0
\uparrow
\varepsilon
|   |
?1 ?2
|   |
?2 ?2 ?1 ?1
|   |   |   |
!false !true !true !false
|   |   |   |
\varepsilon FALSE TRUE FALSE
|   |   |   |
\varepsilon FALSE TRUE FALSE
```

Steps: \( O(2^n) \)
Efficient generic count with effect handlers

effcount : ((Nat → Bool) → Bool) → Nat

\[
effcount \triangleq \lambda P. \ handle P (\lambda j. do \ Branch \ \langle \rangle ) \ with \\
val \ ans \ \iff \ if \ ans \ then \ 1 \ else \ 0
\]

(where \ Branch : \ \langle \rangle \ → \ Bool \ ∈ \ \Sigma)

Behaviour of effcount ex:

![Diagram showing the behaviour of effcount with example inputs and outputs.](image-url)
Efficient generic count with effect handlers

\[
\text{effcount} : ((\text{Nat} \rightarrow \text{Bool}) \rightarrow \text{Bool}) \rightarrow \text{Nat}
\]
\[
\text{effcount} \doteq \lambda P. \text{handle } P (\lambda j. \text{do} \, \text{Branch} \langle \rangle) \text{ with }
\]
\[
\begin{align*}
\text{val ans} & \mapsto \text{if } \text{ans} \text{ then 1 else 0} \\
\text{Branch } \langle \rangle \, \text{resume} & \mapsto \text{resume true + resume false}
\end{align*}
\]
(\text{where } \text{Branch} : \langle \rangle \rightarrow \text{Bool} \in \Sigma)

Behaviour of effcount ex:

```
?0
\downarrow
?1
\downarrow
?2
\downarrow
?2
\downarrow
!false
?1
\downarrow
!true
?2
\downarrow
!true
?2
\downarrow
!false
?1
\downarrow
!true
?1
\downarrow
!false
?1
\downarrow
!false
?1
\downarrow
!true
```
Efficient generic count with effect handlers

\[
effcount : ((\text{Nat} \to \text{Bool}) \to \text{Bool}) \to \text{Nat}
\]
\[
effcount = \lambda P. \text{handle } P (\lambda j. \text{do } \text{Branch } \langle \rangle ) \text{ with }
\]
\[
\begin{align*}
\text{val } ans & \quad \mapsto \quad \text{if } ans \text{ then } 1 \text{ else } 0 \\
\text{Branch } \langle \rangle \text{ resume } & \quad \mapsto \quad \text{resume true + resume false}
\end{align*}
\]

(where \( \text{Branch} : \langle \rangle \to \text{Bool} \in \Sigma \))

Behaviour of effcount ex:

Steps: \( O(2^n) \)
Efficient generic count theorem

Theorem

The following hold for any \( n \in \mathbb{N} \) and any \( n \)-standard predicate \( P \) of \( \text{PCF}_h \):

1. \( \text{effcount} \) correctly counts \( P \).
2. The number of steps required to evaluate \( \text{effcount} \) \( P \) is

\[
\left( \sum_{bs \in \text{Addr}_n} \text{steps}(T(P))(bs) \right) + \mathcal{O}(2^n)
\]

Proof.

By labourious backwards induction on \( bs \)
Naïve count

The naïve approach applies $P$ to all $2^n$ possible points.

$\text{naivecount}_n : ((\text{Nat}_n \to \text{Bool}) \to \text{Bool}) \to \text{Nat}$

$\text{naivecount}_n = \lambda P. \text{count } n (\lambda i. \bot)$

where $\text{count } 0 p \doteq \text{if } P p \text{ then } 1 \text{ else } 0$

$\text{count } (1 + n) p \doteq \text{count } n (\lambda i. \text{if } i = n \text{ then } \text{true} \text{ else } p i) + \text{count } n (\lambda i. \text{if } i = n \text{ then } \text{false} \text{ else } p i)$

Here $(\lambda i. \bot)$ is the divergent point.
The naïve approach applies $P$ to all $2^n$ possible points.

\[
\text{naivecount}_n : ((\text{Nat}_n \rightarrow \text{Bool}) \rightarrow \text{Bool}) \rightarrow \text{Nat}
\]

\[
\text{naivecount}_n \equiv \lambda P. \text{count } n (\lambda i. \bot)
\]

where \(\text{count } 0\) if $P p$ then 1 else 0

\(\text{count } (1 + n)\) p = \text{count } n (\lambda i. \text{if } i = n \text{ then true else } p i)

+ \text{count } n (\lambda i. \text{if } i = n \text{ then false else } p i)

Here $(\lambda i. \bot)$ is the divergent point.

Iteration suffices to implement the naïve approach.

\[
\text{while}_A : (A \rightarrow \text{Bool}) \rightarrow A \rightarrow (A \rightarrow A) \rightarrow A
\]

\[
\text{while}_A \text{ test } x f \equiv \text{if } \text{test } x \text{ then while}_A \text{ test } (f x) f \text{ else } x
\]
Berger count

Counter-intuitively, nested calls to a given predicate, $P$, can vastly improve the performance.

$$\text{bestshot}_n : ((\text{Nat}_n \to \text{Bool}) \to \text{Bool}) \to (\text{Nat}_n \to \text{Bool})$$

Returns a point $Q$ such that $P Q$ evaluates to true.
Counter-intuitively, nested calls to a given predicate, \( P \), can vastly improve the performance.

\[
\text{bestshot}_n : ((\text{Nat}_n \to \text{Bool}) \to \text{Bool}) \to (\text{Nat}_n \to \text{Bool})
\]

Returns a point \( Q \) such that \( P \ Q \) evaluates to true.

For example, we can implement a ‘fail-fast’ variation of naivecount.

\[
\text{lazycount}_n \triangleq \lambda P. \text{if } P (\text{bestshot}_n P) \text{ then naivecount}_n P \text{ else } 0
\]
Counter-intuitively, nested calls to a given predicate, \( P \), can vastly improve the performance.

\[
\text{bestshot}_n : ((\text{Nat}_n \to \text{Bool}) \to \text{Bool}) \to (\text{Nat}_n \to \text{Bool})
\]

Returns a point \( Q \) such that \( P Q \) evaluates to true.

For example, we can implement a ‘fail-fast’ variation of naive\( \text{count} \).

\[
\text{lazycount}_n \triangleq \lambda P. \text{if } P (\text{bestshot}_n P) \text{ then naive\( \text{count}_n P \) else 0}
\]

One can take this idea further and do better than \( \text{naive\( \text{count} \)\) to implement Berger\( \text{count} \) (Berger 1990).}

\[
\text{Berger\( \text{count} \) : ((\text{Nat}_n \to \text{Bool}) \to \text{Bool}) \to \text{Nat}}
\]

(see Escardó (2007) for mind-boggling uses of this trick)
We can do better!

**Idea:** remember which components of the point a given predicate inspects (Longley 1999).

\[
\text{modulus} : ((\text{Nat}_n \to \text{Bool}) \to \text{Bool}) \to (\text{Nat}_n \to \text{Bool}) \to (\text{Bool} \times \text{List}_{\text{Nat}})
\]

\[
\text{modulus } P \ q \ \equiv \ \text{let } log \ \leftarrow \ \text{ref}([], \text{List}_{\text{Nat}}) \ \text{in}
\]

\[
\text{let } \text{wrap} \ \leftarrow \ \lambda i. (log := i :: !log; \ q \ i) \ \text{in}
\]

\[
\text{let } b \ \leftarrow \ P \ \text{wrap} \ \text{in}
\]

\[
\langle b, !log \rangle
\]

If \( \text{modulus } P \ q = \langle b, xs \rangle \), then \( P \ q' = b \) for every \( q' \) that agrees component-wise with \( q \) at \( xs \).

We can use this to effectively ‘prune’ the search space to either ‘fail-fast’ or ‘succeed-fast’.

\[
\text{prunedcount}_n : ((\text{Nat}_n \to \text{Bool}) \to \text{Bool}) \to \text{Nat}
\]
Generic count without effect handlers

\[ \text{count}_n : ((\text{Nat}_n \rightarrow \text{Bool}) \rightarrow \text{Bool}) \rightarrow \text{Nat} \]

Every \( \text{count}_n \in \text{PCF} \) must restart computation for every point, e.g. \( \text{count}_n \) ex:
Generic count without effect handlers

\[\text{count}_n : ((\text{Nat}_n \rightarrow \text{Bool}) \rightarrow \text{Bool}) \rightarrow \text{Nat}\]

Every \(\text{count}_n \in \text{PCF}\) must restart computation for every point, e.g. \(\text{count}_n\) ex:

Steps: \(\Omega(n2^n)\)
Theorem

If $K$ is a PCF program that correctly counts all canonical $n$-standard PCF predicates, and $P$ is any canonical $n$-standard PCF predicate, then the evaluation of $K \ P$ must take time $\Omega(n^{2^n})$.

Proof.

The proof involves tracking of reduction sequences and setting things up such that one can appeal to Milner (1977)'s Context Lemma.
Experiments

The efficiency gap can be observed in practice.

**Benchmarks**
- Queens: enumerating solutions to the \(n\)-Queens problem
- Integration: exact real integration (Simpson 1998)

**Methodology**
- Implementations: naïve, Berger, pruned, effectful, and bespoke
- Implemented in OCaml 5 using the multicont package
- Ran each program 11 times. Given 3 minutes to complete
- Reporting the median speedup (or slowdown) of the effectful implementation

The source code and data are available via

## Queens experiments

<table>
<thead>
<tr>
<th>Parameter</th>
<th>First solution</th>
<th>All solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>20</td>
<td>24</td>
</tr>
<tr>
<td>Naïve</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Berger</td>
<td>13.89</td>
<td>21.72</td>
</tr>
<tr>
<td>Pruned</td>
<td>3.75</td>
<td>4.90</td>
</tr>
<tr>
<td>Bespoke</td>
<td>0.24</td>
<td>0.28</td>
</tr>
</tbody>
</table>

**Table:** Runtime of the \( n \)-Queens procedures relative to the effectful implementation
## Integration experiments

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Id</th>
<th>Squaring</th>
<th>Logistic</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>20</td>
<td>14</td>
<td>17</td>
</tr>
<tr>
<td>Naïve</td>
<td>6.58</td>
<td>18.22</td>
<td>22.38</td>
</tr>
<tr>
<td>Berger</td>
<td>3.62</td>
<td>7.67</td>
<td>7.83</td>
</tr>
<tr>
<td>Pruned</td>
<td>1.25</td>
<td>1.67</td>
<td>1.54</td>
</tr>
</tbody>
</table>

**Table:** Runtime of exact real integration procedures relative to the effectful implementation
The puzzle so far

PCF

Effect handlers
The puzzle so far

Exceptions
Affine handlers
PCF
Effect handlers
State
The puzzle so far

- Exceptions
- PCF
- Effect handlers
- State
The puzzle so far

- Exceptions
- Affine handlers
- PCF
- Effect handlers
- State
Summary

- Take away: effect handlers admit asymptotically more efficient implementations
- Intuition: effect handlers enable computation to be shared via backtracking
- See the papers for rigorous mathematical analyses of this phenomenon

Future work

- What about the expressive power relative to McCarthy’s amb operator?
- What about the asymptotic space characteristics of effect handlers?
References


