This work is supported by

THE UNIVERSITY of EDINBURGH

School of informatics

EPSRC Centre for Doctoral Training in Pervasive Parallelism

EPSRC Engineering and Physical Sciences Research Council

Efficient Generic Search with Effect Handlers

Daniel Hillerström

Laboratory for Foundations of Computer Science
School of Informatics
The University of Edinburgh, UK

July 18, 2018

Programming Language Interest Group

(Joint work with Sam Lindley and John Longley)

A new complexity result for control operators

The crux of this work is to establish a new complexity result for control operators

Lay person's version of the result

There is a class of problems for which a language with control operators provides asymptotically more efficient solutions than a language without control operators ($\mathcal{O}(2^n)$ vs $\Omega(n2^n)$).

To establish the existence of this class, we use *generic search* as an example program and effect handlers as our control operator.

This talk is high-level walk-through of how we establish this result

(The possibility of the existence of this result can be traced back to Longley (2009))

Methodology

The plan of attack

- Define a pure functional language $\mathcal{L}$, and an extension thereof $\mathcal{L}_{\mathit{eff}}$ with effect handlers.
- Provide a specification (type signature) of *generic search* problem
- Implement an efficient version of generic search in $\mathcal{L}_{\mathit{eff}}$
- ...and prove that it is indeed efficient
- Finally show that *any* implementation of generic search in $\mathcal{L}$ has worse complexity

There is a single rule of engagement:

No change of types is allowed! (Longley and Normann 2015)

This rules out tricks such as

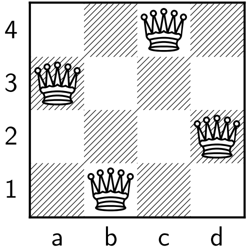
- CPS conversion (Hillerström et al. 2017)
- Implementing an interpreter for $\mathcal{L}_{\mathit{eff}}$ in $\mathcal{L}$

Generic search

Given a search problem P, a generic search algorithm finds solutions of P.

Applications include (Daniels 2016)

- n-Queens
- Sudoku
- Finding Nash equilibria
- Graph-colouring
- Exact real number integration



A solution to 4-Queens problem

Rather than finding solutions of P, we *count* the number of solutions of P

First instance of efficient generic search with effect handlers

Somewhat related is work on exhaustive search on infinite spaces

- Berger (1990): exhaustive search on the Cantor space $2^{\mathbb{N}}$
- Escardó (2007): characterisation of searchable infinite sets
- Bauer (2011): efficient search on infinite sets with effect handlers

Fine-grain call-by-value PCF (Levy et al. 2003)

The core of a "pure" functional programming language $\mathcal{L}$

$$
\begin{array}{lrcl}
\text{Types} & A, B, C, D & ::= & \langle\rangle \mid \mathsf{Bool} \mid \mathsf{Nat} \mid A \times B \mid A + B \mid A \to B \\[1em]
\text{Values} & V, W \in \mathrm{Val} & ::= & x \mid b \in \mathbb{B} \mid n \in \mathbb{N} \mid \mathsf{Plus} \mid \langle\rangle \mid \langle V; W\rangle \\
& & \mid & (\mathbf{inl}\, V)^B \mid (\mathbf{inr}\, W)^A \mid \lambda x^A.\, M \mid \mathbf{rec}\, f^A\, x.M \\[1em]
\text{Computations} & M, N \in \mathrm{Comp} & ::= & V\, W \\
& & \mid & \mathbf{let}\ \langle x; y\rangle = V\ \mathbf{in}\ N \\
& & \mid & \mathbf{if}\ V\ \mathbf{then}\ M\ \mathbf{else}\ N \\
& & \mid & \mathbf{case}\ V\ \{\mathbf{inl}\ x \mapsto M; \mathbf{inr}\ y \mapsto N\} \\
& & \mid & \mathbf{return}\ V \\
& & \mid & \mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N \\[1em]
\text{Eval. contexts} & \mathcal{E} \in \mathsf{Eval} & ::= & [\,] \mid \mathbf{let}\ x \leftarrow \mathcal{E}\ \mathbf{in}\ N
\end{array}
$$

The static and dynamic semantics are completely standard.

Service announcement: Syntactic sugar

I shall permit myself to use regular call-by-value syntax, e.g. for $f, g, h, a \in \mathrm{Val}$

$$f\ (h\, a) + g\ \langle\rangle$$

Service announcement: Syntactic sugar

I shall permit myself to use regular call-by-value syntax, e.g. for $f, g, h, a \in \mathrm{Val}$

$$\begin{aligned}[\![f\ (h\, a) + g\ \langle\rangle]\!] = {} & \textbf{let}\ x \leftarrow h\, a\ \textbf{in} \\ & \textbf{let}\ y \leftarrow f\ x\ \textbf{in} \\ & \textbf{let}\ z \leftarrow g\ \langle\rangle\ \textbf{in} \\ & \mathsf{Plus}\, \langle y; z\rangle \end{aligned}$$

FGCB PCF with effect handlers

The language $\mathcal{L}_{eff}$

Handler types	$F ::= C \Rightarrow D$
Signatures	$\Sigma ::= \cdot \mid \{\ell : A \to B\} \uplus \Sigma$
Labels	$\ell \in \mathcal{L}$
Computations	$M, N \in \mathrm{Comp} ::= \cdots \mid \textbf{do}\ \ell\ V \mid \textbf{handle}\ M\ \textbf{with}\ H$
Handlers	$H ::= \{\textbf{val}\ x \mapsto M\} \mid \{\ell\ p\ r \mapsto N\} \uplus H$
Eval. contexts	$\mathcal{E} \in \mathsf{Eval} ::= \cdots \mid \textbf{handle}\ \mathcal{E}\ \textbf{with}\ H$

FGCB PCF with effect handlers (dynamic semantics)

S-Ret $\quad$ $\textbf{handle}\ (\textbf{return}\ V)\ \textbf{with}\ H$

$\quad\quad \rightsquigarrow N[V/x], \qquad$ where $H^{\mathrm{val}} = \{\textbf{val}\ x \mapsto N\}$

S-Op $\quad$ $\textbf{handle}\ \mathcal{E}[\textbf{do}\ \ell\ V]\ \textbf{with}\ H$

$\quad\quad \rightsquigarrow N[V/p, \lambda y.\, \textbf{handle}\ \mathcal{E}[\textbf{return}\ y]\ \textbf{with}\ H/r]$, where $H^{\ell} = \{\ell\ p\ r \mapsto N\}$

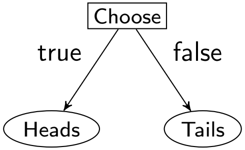
Example: Coin tossing (nondeterminism)

Fix $\Sigma = \{\mathsf{Choose} : \mathsf{Bool}\}$

A coin toss model

$$\begin{aligned} &\mathit{toss} : \langle\rangle \to \mathsf{Toss} \\ &\mathit{toss} = \textbf{if do } \mathsf{Choose} \textbf{ then } \mathsf{Heads} \\ &\qquad\quad \textbf{else } \mathsf{Tails} \end{aligned}$$

Computation tree



A possible handler for Choose

$$\begin{aligned} &\mathit{allChoices} : (\langle\rangle \to \mathsf{Toss}) \to [\mathsf{Toss}] \\ &\mathit{allChoices} = \lambda m.\, \textbf{handle } m\,\langle\rangle \textbf{ with} \\ &\qquad\qquad\qquad \textbf{val } x \mapsto [x] \\ &\qquad\qquad\qquad \mathsf{Choose}\ r \mapsto r\ \mathsf{true} +\!\!+\ r\ \mathsf{false} \end{aligned}$$

Enumerating all possible outcomes

$$\mathit{allChoices}\ \mathit{toss} \leadsto^{+} ??$$

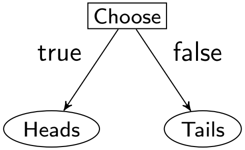
Example: Coin tossing (nondeterminism)

Fix $\Sigma = \{\mathsf{Choose} : \mathsf{Bool}\}$

A coin toss model

$$\begin{aligned} &\mathit{toss} : \langle\rangle \to \mathsf{Toss} \\ &\mathit{toss} = \textbf{if do } \mathsf{Choose} \textbf{ then } \mathsf{Heads} \\ &\qquad\quad \textbf{else } \mathsf{Tails} \end{aligned}$$

Computation tree



A possible handler for Choose

$$\begin{aligned} &\mathit{allChoices} : (\langle\rangle \to \mathsf{Toss}) \to [\mathsf{Toss}] \\ &\mathit{allChoices} = \lambda m.\, \textbf{handle } m\,\langle\rangle \textbf{ with} \\ &\qquad\qquad\qquad \textbf{val } x \mapsto [x] \\ &\qquad\qquad\qquad \mathsf{Choose}\ r \mapsto r\,\mathsf{true} \mathbin{+\!\!+} r\,\mathsf{false} \end{aligned}$$

Enumerating all possible outcomes

$$\mathit{allChoices}\ \mathit{toss} \leadsto^{+} [\mathsf{Heads}, \mathsf{Tails}]$$

Setting up generic search

The secret of generic search is *higher-order* functions

$$\mathsf{Predicate} \doteq (\mathsf{Nat} \to \mathsf{Bool}) \to \mathsf{Bool}$$

Setting up generic search

The secret of generic search is *higher-order* functions

$$\begin{aligned} \mathsf{Point} &\doteq \mathsf{Nat} \to \mathsf{Bool} \\ \mathsf{Predicate} &\doteq \mathsf{Point} \to \mathsf{Bool} \end{aligned}$$

Setting up generic search

The secret of generic search is *higher-order* functions

$$\begin{aligned} \mathsf{Point} &\doteq \mathsf{Nat} \to \mathsf{Bool} \\ \mathsf{Predicate} &\doteq \mathsf{Point} \to \mathsf{Bool} \\ \mathsf{Counter} &\doteq \mathsf{Predicate} \to \mathsf{Nat} \end{aligned}$$

Setting up generic search

The secret of generic search is *higher-order* functions

$$\begin{aligned} \mathsf{Point} &\doteq \mathsf{Nat} \to \mathsf{Bool} \\ \mathsf{Predicate} &\doteq \mathsf{Point} \to \mathsf{Bool} \\ \mathsf{Counter} &\doteq \mathsf{Predicate} \to \mathsf{Nat} \end{aligned}$$

Some (silly) example predicates

$$\begin{aligned} \mathsf{tt}_n &\doteq \lambda p. p\,0; \cdots ; p\,(n-1); \textbf{return}\ \mathsf{true} \\ \mathsf{div}_n &\doteq \textbf{rec}\, d\ p. \textbf{if}\ p\,(n-1)\ \textbf{then}\ d\ p\ \textbf{else}\ \textbf{return}\ \mathsf{false} \\ \mathsf{odd}_n &\doteq \lambda p. \mathsf{reduce\ xor\ false}\,[p\,0, \ldots, p\,(n-1)] \end{aligned}$$

A pure generic search procedure

A possible implementation of generic search in $\mathcal{L}$

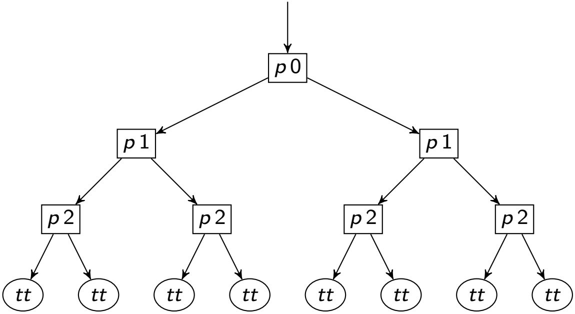
$$
\begin{aligned}
&\mathit{count}_n : (\mathsf{Predicate} \to \mathsf{Bool}) \to \mathsf{Nat} \\
&\mathit{count}_n \doteq \lambda \mathit{pred}.\mathit{count}'\, n\, (\lambda i.\bot) \\
&\qquad\qquad \mathbf{where} \\
&\qquad\qquad\qquad \mathit{count}'\, 0 \qquad\quad\;\; p \doteq \mathbf{if}\ \mathit{pred}\ p\ \mathbf{then}\ 1\ \mathbf{else}\ 0 \\
&\qquad\qquad\qquad \mathit{count}'\, (n+1)\ p \doteq \quad \mathit{count}'\, n\, (\lambda i.\mathbf{if}\ i = n\ \mathbf{then}\ \mathsf{true}\ \mathbf{else}\ p\, i) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\;\; + \mathit{count}'\, n\, (\lambda i.\mathbf{if}\ i = n\ \mathbf{then}\ \mathsf{false}\ \mathbf{else}\ p\, i)
\end{aligned}
$$

A pure generic search procedure

A possible implementation of generic search in $\mathcal{L}$

$$
\begin{aligned}
&\mathit{count}_n : (\text{Predicate} \rightarrow \text{Bool}) \rightarrow \text{Nat} \\
&\mathit{count}_n \doteq \lambda \mathit{pred}.\mathit{count}'\, n\, (\lambda i.\bot) \\
&\qquad\qquad \textbf{where} \\
&\qquad\qquad\qquad \mathit{count}'\, 0 \qquad\quad\;\; p \doteq \textbf{if}\ \mathit{pred}\ p\ \textbf{then}\ 1\ \textbf{else}\ 0 \\
&\qquad\qquad\qquad \mathit{count}'\, (n+1)\; p \doteq \quad \mathit{count}'\, n\, (\lambda i.\textbf{if}\ i = n\ \textbf{then}\ \text{true}\ \textbf{else}\ p\, i) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\;\; + \mathit{count}'\, n\, (\lambda i.\textbf{if}\ i = n\ \textbf{then}\ \text{false}\ \textbf{else}\ p\, i)
\end{aligned}
$$

Example $\mathit{count}_3\ \mathit{tt}_3$:

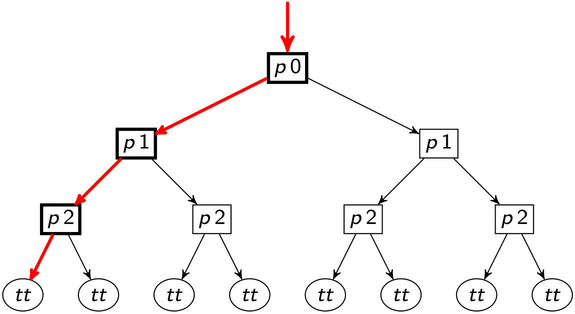


A pure generic search procedure

A possible implementation of generic search in $\mathcal{L}$

$$
\begin{array}{l}
count_n : (\text{Predicate} \to \text{Bool}) \to \text{Nat} \\
count_n \doteq \lambda pred.count'\, n\, (\lambda i.\bot) \\
\qquad\qquad \textbf{where} \\
\qquad\qquad\qquad \begin{array}{lcl}
count'\, 0 \qquad\quad p & \doteq & \textbf{if}\ pred\ p\ \textbf{then}\ 1\ \textbf{else}\ 0 \\
count'\, (n+1)\ p & \doteq & \quad count'\, n\, (\lambda i.\textbf{if}\ i = n\ \textbf{then}\ \text{true}\ \textbf{else}\ p\, i) \\
 & & +\ count'\, n\, (\lambda i.\textbf{if}\ i = n\ \textbf{then}\ \text{false}\ \textbf{else}\ p\, i)
\end{array}
\end{array}
$$

Example $count_3\ tt_3$: reaches the first leaf

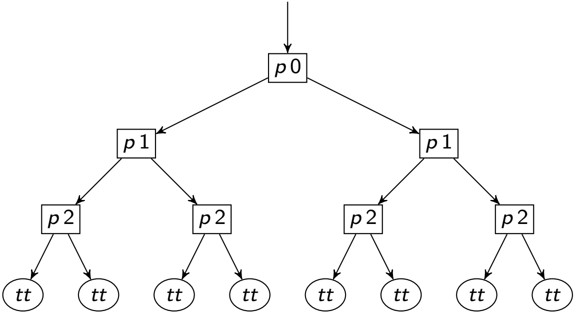


A pure generic search procedure

A possible implementation of generic search in $\mathcal{L}$

$$
\begin{array}{l}
count_n : (\mathsf{Predicate} \to \mathsf{Bool}) \to \mathsf{Nat} \\
count_n \doteq \lambda pred.count'\, n\, (\lambda i.\bot) \\
\qquad\qquad \textbf{where} \\
\qquad\qquad\qquad \begin{array}{lll}
count'\, 0 & p \doteq & \textbf{if } pred\ p \textbf{ then } 1 \textbf{ else } 0 \\
count'\, (n+1) & p \doteq & \quad count'\, n\, (\lambda i.\textbf{if } i = n \textbf{ then } \mathsf{true} \textbf{ else } p\, i) \\
 & & + \ count'\, n\, (\lambda i.\textbf{if } i = n \textbf{ then } \mathsf{false} \textbf{ else } p\, i)
\end{array}
\end{array}
$$

Example $count_3\ tt_3$: computation restarts

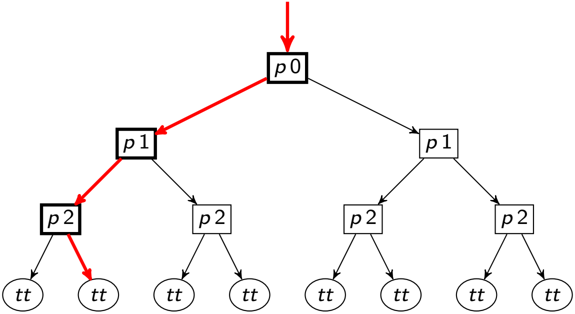


A pure generic search procedure

A possible implementation of generic search in $\mathcal{L}$

$$
\begin{array}{l}
count_n : (\text{Predicate} \to \text{Bool}) \to \text{Nat} \\
count_n \doteq \lambda pred.count'\, n\, (\lambda i.\bot) \\
\qquad\qquad \textbf{where} \\
\qquad\qquad\qquad \begin{array}{lll}
count'\, 0 & p \doteq & \textbf{if}\ pred\ p\ \textbf{then}\ 1\ \textbf{else}\ 0 \\
count'\, (n+1) & p \doteq & \quad count'\, n\, (\lambda i.\textbf{if}\ i = n\ \textbf{then}\ \text{true}\ \textbf{else}\ p\, i) \\
 & & +\ count'\, n\, (\lambda i.\textbf{if}\ i = n\ \textbf{then}\ \text{false}\ \textbf{else}\ p\, i)
\end{array}
\end{array}
$$

Example $count_3\ tt_3$: reaches the second leaf

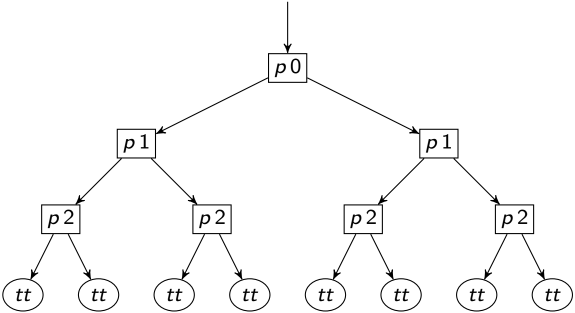


A pure generic search procedure

A possible implementation of generic search in $\mathcal{L}$

$$
\begin{array}{l}
count_n : (\mathsf{Predicate} \to \mathsf{Bool}) \to \mathsf{Nat} \\
count_n \doteq \lambda pred.count'\, n\, (\lambda i.\bot) \\
\qquad\qquad \textbf{where} \\
\qquad\qquad\qquad \begin{array}{lll}
count'\, 0 & p \doteq & \textbf{if } pred\ p \textbf{ then } 1 \textbf{ else } 0 \\
count'\, (n+1) & p \doteq & \quad count'\, n\, (\lambda i.\textbf{if } i = n \textbf{ then } \mathsf{true} \textbf{ else } p\ i) \\
 & & + \ count'\, n\, (\lambda i.\textbf{if } i = n \textbf{ then } \mathsf{false} \textbf{ else } p\ i)
\end{array}
\end{array}
$$

Example $count_3\ tt_3$: computation restarts

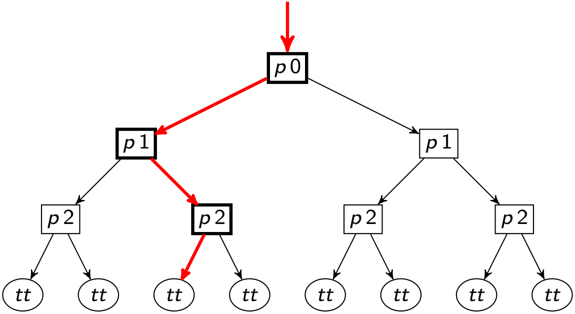


A pure generic search procedure

A possible implementation of generic search in $\mathcal{L}$

$$
\begin{array}{l}
count_n : (\text{Predicate} \to \text{Bool}) \to \text{Nat} \\
count_n \doteq \lambda pred.count'\, n\, (\lambda i.\bot) \\
\qquad\qquad \textbf{where} \\
\qquad\qquad\qquad \begin{array}{lll}
count'\, 0 & p \doteq & \textbf{if } pred\, p \textbf{ then } 1 \textbf{ else } 0 \\
count'\, (n+1) & p \doteq & \quad count'\, n\, (\lambda i.\textbf{if } i = n \textbf{ then } \text{true} \textbf{ else } p\, i) \\
 & & + \; count'\, n\, (\lambda i.\textbf{if } i = n \textbf{ then } \text{false} \textbf{ else } p\, i)
\end{array}
\end{array}
$$

Example $count_3\, tt_3$: reaches the third leaf, etc...



The effectful generic search procedure

For the efficient implementation of generic search in $\mathcal{L}_{\mathit{eff}}$, we require one operation; fix $\Sigma \doteq \{\mathsf{Branch} : \mathsf{Bool}\}$

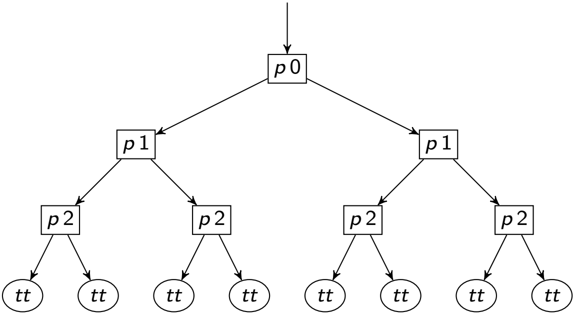
$$
\begin{array}{l}
\mathsf{genericSearch} : (\mathsf{Predicate} \to \mathsf{Bool}) \to \mathsf{Nat} \\
\mathsf{genericSearch} \doteq \lambda \mathit{pred}.\mathbf{handle}\ \ (\mathbf{if}\ \mathit{pred}\ (\lambda n.\mathbf{do}\ \mathsf{Branch})\ \mathbf{then}\ 1\ \mathbf{else}\ 0)\ \ \mathbf{with} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{val}\, x \mapsto x \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{Branch}\ r \mapsto r\,\mathsf{true} + r\,\mathsf{false}
\end{array}
$$

The effectful generic search procedure

For the efficient implementation of generic search in $\mathcal{L}_{eff}$, we require one operation; fix $\Sigma \doteq \{\mathsf{Branch} : \mathsf{Bool}\}$

$$\begin{aligned}
&\mathsf{genericSearch} : (\mathsf{Predicate} \to \mathsf{Bool}) \to \mathsf{Nat} \\
&\mathsf{genericSearch} \doteq \lambda \mathit{pred}.\mathbf{handle}\ \ (\mathbf{if}\ \mathit{pred}\ (\lambda n.\mathbf{do}\ \mathsf{Branch})\ \mathbf{then}\ 1\ \mathbf{else}\ 0)\ \ \mathbf{with} \\
&\qquad\qquad\qquad\qquad\quad \mathbf{val}\, x \mapsto x \\
&\qquad\qquad\qquad\qquad\quad \mathsf{Branch}\, r \mapsto r\, \mathsf{true} + r\, \mathsf{false}
\end{aligned}$$

Example genericSearch tt_3:

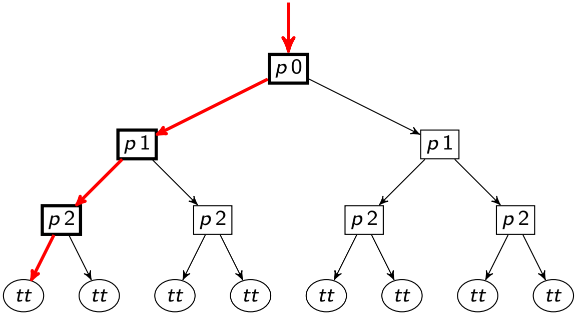


The effectful generic search procedure

For the efficient implementation of generic search in $\mathcal{L}_{\mathit{eff}}$, we require one operation; fix $\Sigma \doteq \{\mathsf{Branch} : \mathsf{Bool}\}$

$$\begin{aligned}
&\mathsf{genericSearch} : (\mathsf{Predicate} \to \mathsf{Bool}) \to \mathsf{Nat} \\
&\mathsf{genericSearch} \doteq \lambda \mathit{pred}.\textbf{handle}\ \ (\textbf{if}\ \mathit{pred}\,(\lambda n.\textbf{do}\ \mathsf{Branch})\ \textbf{then}\ 1\ \textbf{else}\ 0)\ \ \textbf{with} \\
&\qquad\qquad\qquad\qquad\qquad \textbf{val}\, x \mapsto x \\
&\qquad\qquad\qquad\qquad\qquad \mathsf{Branch}\, r \mapsto r\,\mathsf{true} + r\,\mathsf{false}
\end{aligned}$$

Example genericSearch tt_3: reaches the first leaf

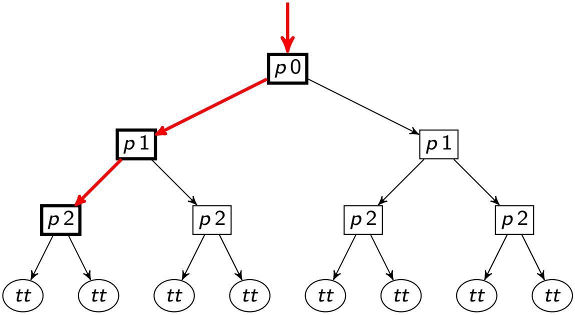


The effectful generic search procedure

For the efficient implementation of generic search in $\mathcal{L}_{eff}$, we require one operation; fix $\Sigma \doteq \{\mathsf{Branch} : \mathsf{Bool}\}$

$$\begin{aligned}
&\mathsf{genericSearch} : (\mathsf{Predicate} \to \mathsf{Bool}) \to \mathsf{Nat}\\
&\mathsf{genericSearch} \doteq \lambda \mathit{pred}.\mathbf{handle}\ \ (\mathbf{if}\ \mathit{pred}\ (\lambda n.\mathbf{do}\ \mathsf{Branch})\ \mathbf{then}\ 1\ \mathbf{else}\ 0)\ \ \mathbf{with}\\
&\qquad\qquad\qquad\qquad\qquad \mathbf{val}\, x \mapsto x\\
&\qquad\qquad\qquad\qquad\qquad \mathsf{Branch}\ r \mapsto r\,\mathsf{true} + r\,\mathsf{false}
\end{aligned}$$

Example genericSearch tt_3: computation backtracks

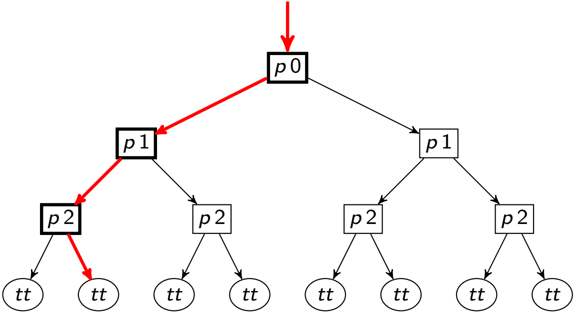


The effectful generic search procedure

For the efficient implementation of generic search in $\mathcal{L}_{\mathit{eff}}$, we require one operation; fix $\Sigma \doteq \{\mathsf{Branch} : \mathsf{Bool}\}$

$$\begin{aligned}
&\mathsf{genericSearch} : (\mathsf{Predicate} \to \mathsf{Bool}) \to \mathsf{Nat}\\
&\mathsf{genericSearch} \doteq \lambda \mathit{pred}.\mathbf{handle}\ \ (\mathbf{if}\ \mathit{pred}\ (\lambda n.\mathbf{do}\ \mathsf{Branch})\ \mathbf{then}\ 1\ \mathbf{else}\ 0)\ \ \mathbf{with}\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathbf{val}\, x \mapsto x\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathsf{Branch}\, r \mapsto r\,\mathsf{true} + r\,\mathsf{false}
\end{aligned}$$

Example genericSearch tt_3: reaches the second leaf

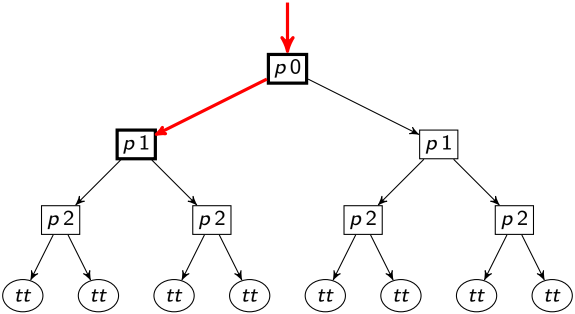


The effectful generic search procedure

For the efficient implementation of generic search in $\mathcal{L}_{eff}$, we require one operation; fix $\Sigma \doteq \{\mathsf{Branch} : \mathsf{Bool}\}$

$$
\begin{aligned}
&\mathsf{genericSearch} : (\mathsf{Predicate} \to \mathsf{Bool}) \to \mathsf{Nat} \\
&\mathsf{genericSearch} \doteq \lambda pred.\mathbf{handle}\ (\mathbf{if}\ pred\ (\lambda n.\mathbf{do}\ \mathsf{Branch})\ \mathbf{then}\ 1\ \mathbf{else}\ 0)\ \mathbf{with} \\
&\qquad\qquad\qquad\qquad \mathbf{val}\ x \mapsto x \\
&\qquad\qquad\qquad\qquad \mathsf{Branch}\ r \mapsto r\ \mathsf{true} + r\ \mathsf{false}
\end{aligned}
$$

Example $\mathsf{genericSearch}\ tt_3$: computation backtracks

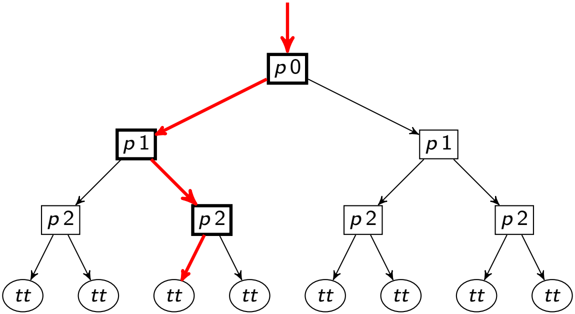


The effectful generic search procedure

For the efficient implementation of generic search in $\mathcal{L}_{\mathit{eff}}$, we require one operation; fix $\Sigma \doteq \{\mathsf{Branch} : \mathsf{Bool}\}$

$$\begin{aligned}
&\mathsf{genericSearch} : (\mathsf{Predicate} \to \mathsf{Bool}) \to \mathsf{Nat} \\
&\mathsf{genericSearch} \doteq \lambda \mathit{pred}.\mathbf{handle}\ \ (\mathbf{if}\ \mathit{pred}\,(\lambda n.\mathbf{do}\ \mathsf{Branch})\ \mathbf{then}\ 1\ \mathbf{else}\ 0)\ \ \mathbf{with} \\
&\qquad\qquad\qquad\qquad \mathbf{val}\,x \mapsto x \\
&\qquad\qquad\qquad\qquad \mathsf{Branch}\,r \mapsto r\,\mathsf{true} + r\,\mathsf{false}
\end{aligned}$$

Example genericSearch tt_3: reaches the third leaf, etc...



Semantics for predicates

Definition (The label set)

The set Lab consists of queries parameterised by a natural number and answers parameterised by a boolean, i.e. $\mathsf{Lab} \doteq \{!tt, !f\!f\} \cup \{?n \mid n \in \mathbb{N}\}$

Definition (Decision tree)

A decision tree is a partial function $t : \mathbb{B}^* \to \mathsf{Lab} \times \mathsf{Eval} \times \mathbb{N}$ from lists of booleans to node labels with the following properties:

- The domain of t, $dom(t)$, is prefix closed.
- For any boolean, $b \in \mathbb{B}$, and list, $bs \in \mathbb{B}^*$, of booleans, if $t_\ell(bs) = \ !b$ is an answer node then bs is a leaf of t.

Notation: write t_ℓ and t_s for the projection of the first and third components of $t(-)$, respectively.

Model construction

Definition

We implement the decision tree semantics as a partial function parameterised by an abstract point p, $\mathcal{T}_p : \mathrm{Comp} \rightharpoonup (\mathbb{B}^* \rightharpoonup \mathsf{Lab} \times \mathsf{Eval} \times \mathbb{N})$, that given a predicate, *pred*, constructs a function, that given a list of booleans, *bs*, returns the corresponding node label in model of *pred* p, where p is an "abstract point".

$$
\begin{aligned}
\mathcal{T}_p(\textbf{return}\ \mathsf{true})\,[\,] &= (!\mathsf{true}, [\,], 0)\\
\mathcal{T}_p(\textbf{return}\ \mathsf{false})\,[\,] &= (!\mathsf{false}, [\,], 0)\\
\mathcal{T}_p(\mathcal{E}[p\ n])\,[\,] &= (?n, \mathcal{E}, 0)\\
\mathcal{T}_p(\mathcal{E}[p\ n])(b :: bs) &\simeq \mathcal{T}_p(\mathcal{E}[\textbf{return}\ b])(bs)\\
\text{If } M \rightsquigarrow N \text{ then } \mathcal{T}_p(M)(bs) &\simeq \mathcal{I}(\mathcal{T}_p(N)(bs))\\
&\quad \text{where } \mathcal{I}(\ell, \mathcal{E}, i) = (\ell, \mathcal{E}, i+1)
\end{aligned}
$$

Define Model $\doteq \mathrm{Comp} \rightharpoonup (\mathbb{B}^* \times \mathsf{Eval} \times \mathbb{N})$.

Standard decision trees

We are interested in predicates whose models are complete binary trees, and query each component of a provided point exactly once.

Definition (n-standard trees)

For any $n > 0$ a decision tree t is said to be n-standard whenever

- The domain of t consists of all the lists whose length is at most n, i.e.

$$dom(t) = \{bs : \mathbb{B}^* \mid |bs| \leq n\}$$

- Every leaf node in t is an answer node, i.e. for all $bs \in dom(t)$

$$\text{if } t_\ell(bs) = \; !b \text{ then } |bs| = n$$

- There are no repeated queries in t, i.e. for all $bs, bs' \in dom(t), j \in \mathbb{N}$

$$\text{if } bs \sqsubseteq bs' \text{ and } t_\ell(bs) = t_\ell(bs') = ?j \text{ then } bs = bs'$$

  where $bs \sqsubseteq bs'$ means bs is a prefix of bs'.

Main theorem

Theorem

1. *For every n-standard predicate pred, the generic search procedure has at most time complexity*

$$\mathrm{Time}(\mathrm{genericSearch}\ pred) = \sum_{bs \in \mathbb{B}^*, |bs| \leq n} t_s(bs) + \mathcal{O}(2^n)$$

2. *Every generic counting function count $\in \mathcal{L}$ has for every n-standard predicate pred at least time complexity*

$$\mathrm{Time}(count\ pred) = \sum_{bs \in \mathbb{B}^*, |bs| \leq n} 2^{n-|bs|} t_s(bs) + \mathcal{O}(n2^n)$$

Proving the positive result

Define suitable evaluation state computing functions

$$\mathit{start}, \mathit{end} : \mathbb{B}^* \times \mathsf{Model} \to \mathsf{Comp}$$

Lemma

Suppose t is a model of a n-standard predicate, then for every boolean list bs $\in \mathbb{B}^*$

$$\begin{aligned}
& \mathit{start}(bs, t) \\
\longrightarrow^+ \ & \mathit{start}(\mathsf{true} :: bs, t) \longrightarrow^{\sum_{|bs|+1 \le n} t_s(\mathsf{true}::bs) + 2^{n-(|bs|+1)}} \mathit{end}(\mathsf{true} :: bs, t) \\
\longrightarrow^+ \ & \mathit{start}(\mathsf{false} :: bs, t) \longrightarrow^{\sum_{|bs|+1 \le n} t_s(\mathsf{false}::bs) + 2^{n-(|bs|+1)}} \mathit{end}(\mathsf{false} :: bs, t) \\
\longrightarrow^+ \ & \mathit{end}(bs, t)
\end{aligned}$$

Proof.

Proof by downward induction on the list of booleans *bs*. □

Proving the negative result

Suppose that we have an arbitrary implementation of generic search $\mathit{count} \in \mathcal{L}$. Pick any n-standard predicate *pred* and look at the computation arising from *count pred*. Now we need to show that

Lemma (Every leaf is visited (A))

The computation $(\mathit{count}\ \mathit{pred})$ visits every leaf in the model of pred.

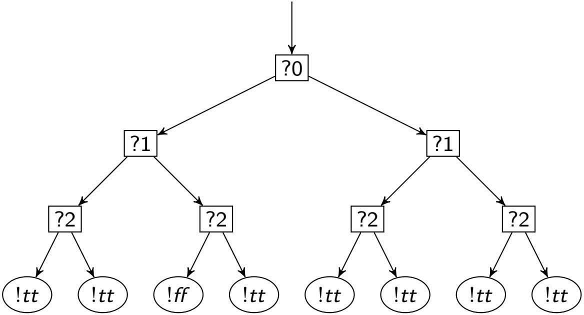
Lemma (No shared computation (B))

If p and p' are distinct points then their subcomputations are disjoint.

Since each subcomputation has length at least $\Omega(n)$ the entire computation must have at least length $\Omega(n2^n)$.

Threads and sections

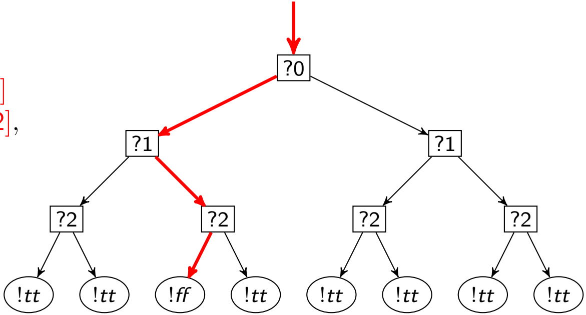
Consider a 3-standard predicate *seven* (has seven true leaves)



Threads and sections

Consider a 3-standard predicate *seven* (has seven true leaves)

$$\mathit{Thread} \doteq \{\ \mathit{pred}\ p \leadsto^* \mathcal{E}_0[p\,0],\ \mathcal{E}_0[\text{true}] \leadsto^* \mathcal{E}_1[p\,1]\ \mathcal{E}_1[\text{false}] \leadsto^* \mathcal{E}_2[p\,2],\ \mathcal{E}_2[\text{true}] \to \text{false}\ \}$$

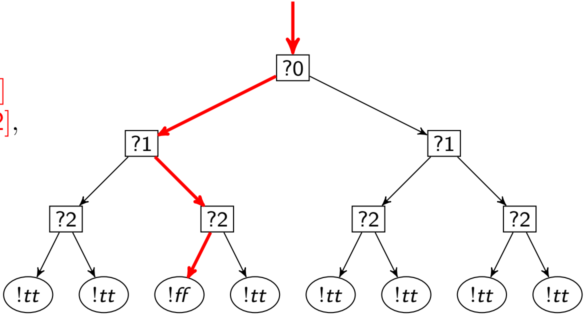


Any n-standard predicate has 2^n threads, and every thread consists of $n+1$ sections.

Threads and sections

Consider a 3-standard predicate *seven* (has seven true leaves)

$$\mathit{Thread} \doteq \{\ \mathit{pred}\ p \leadsto^* \mathcal{E}_0[p\,0],\ \mathcal{E}_0[\mathsf{true}] \leadsto^* \mathcal{E}_1[p\,1]\ \mathcal{E}_1[\mathsf{false}] \leadsto^* \mathcal{E}_2[p\,2],\ \mathcal{E}_2[\mathsf{true}] \to \mathsf{false}\ \}$$



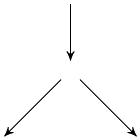
Any n-standard predicate has 2^n threads, and every thread consists of $n+1$ sections.

Proof of Lemma A.

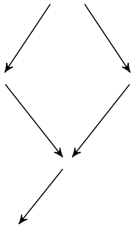
By contradiction: pick a leaf that has no thread; negate the value at the leaf; tweak the predicate accordingly; observe a wrong result. □

No shared computation

Every section has a unique successor



Every section has a single predecessor



Proof.

Follows by definition of section and the semantics being deterministic. □

Proof.

By direct calculation on the reduction sequence induced by a section. □

Summary and future work

In summary

- We have defined two languages $\mathcal{L}$ and $\mathcal{L}_{\mathit{eff}}$
- We have demonstrated that $\mathcal{L}_{\mathit{eff}}$ provides strictly more efficient implementations of generic search than $\mathcal{L}$ ($\mathcal{O}(2^n)$ vs $\Omega(n2^n)$)
- ...which establish a new complexity result for control operators

Future considerations

- Perform empirical experiments to observe the result in practice (Daniels 2016)
- Study the robustness of the result, i.e. what feature(s) can we add to $\mathcal{L}$ whilst retaining an efficiency gap between $\mathcal{L}$ and $\mathcal{L}_{\mathit{eff}}$?
- Generalise the result to all conceivable effective models of computations

References I

Bauer, Andrej (2011). *How make the "impossible" functionals run even faster*. Mathematics, Algorithms and Proofs, Leiden, the Netherlands. url: http://math.andrej.com/2011/12/06/how-to-make-the-impossible-functionals-run-even-faster/.

Berger, Ulrich (1990). "Totale Objekte und Mengen in der Bereichstheorie". PhD thesis. Munich: Ludwig Maximillians-Universtität.

Daniels, Robbie (2016). "Efficient Generic Searches and Programming Language Expressivity". MA thesis. Scotland: School of Informatics, the University of Edinburgh.

Escardó, Martín Hötzel (2007). "Infinite sets that admit fast exhaustive search". In: *LICS*. IEEE Computer Society, pp. 443–452.

Hillerström, Daniel et al. (2017). "Continuation Passing Style for Effect Handlers". In: *FSCD*. Vol. 84. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 18:1–18:19.

Levy, Paul Blain, John Power, and Hayo Thielecke (2003). "Modelling environments in call-by-value programming languages". In: *Inf. Comput.* 185.2, pp. 182–210.

References II

Longley, John (2009). "Some Programming Languages Suggested by Game Models (Extended Abstract)". In: *Electr. Notes Theor. Comput. Sci.* 249, pp. 117–134.

Longley, John and Dag Normann (2015). *Higher-Order Computability*. Theory and Applications of Computability. Springer.