Effects for efficiency

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 λ_{b} and its extension with effect handlers λ_{h}. We show that λ_{h} admits an asymptotically more efficient implementation of generic count than any λ_{b} implementation. We also show that this efficiency gap remains when λ_{b} 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:
Daniel Hillerström, Sam Lindley, and John Longley. 2020. Effects for Efficiency: Asymptotic Speedup with First-Class Control. *Proc. ACM Program. Lang.* 4, ICFP, Article 100 (August 2020), 29 pages. https://doi.org/10.1145/3408982

1 INTRODUCTION

In today's programming languages we find a wealth of powerful constructs and features — exceptions, higher-order store, dynamic method dispatch, coroutines, explicit continuations, concurrency features, Lisp-style 'quote' and so on — which may be present or absent in various combinations in any given language. There are of course many important pragmatic and stylistic differences between languages, but here we are concerned with whether languages may differ more essentially in their expressive power, according to the selection of features they contain.

One can interpret this question in various ways. For instance, Felleisen [1991] considers the question of whether a language $\mathcal{L}$ admits a translation into a sublanguage $\mathcal{L}'$ in a way which respects not only the behaviour of programs but also aspects of their (global or local) syntactic structure. If the translation of some $\mathcal{L}$-program into $\mathcal{L}'$ requires a complete global restructuring, we may say that $\mathcal{L}'$ is in some way less expressive than $\mathcal{L}$. In the present paper, however, we have in mind even more fundamental expressivity differences that would not be bridged even if whole-program translations were admitted. These fall under two headings.

(1) *Computability*: Are there operations of a given type that are programmable in $\mathcal{L}$ but not expressible at all in $\mathcal{L}'$?

Authors' addresses: Daniel Hillerström, The University of Edinburgh, UK, daniel.hillerstrom@ed.ac.uk; Sam Lindley, The University of Edinburgh and Imperial College London and Heriot-Watt University, UK, sam.lindley@ed.ac.uk; John Longley, The University of Edinburgh, UK, jrl@staffmail.ed.ac.uk.

2475-1421/2020/8-ART100
https://doi.org/10.1145/3408982

(2) *Complexity*: Are there operations programmable in $\mathcal{L}$ with some asymptotic runtime bound (e.g. '$O(n^2)$') that cannot be achieved in $\mathcal{L}'$?

We may also ask: are there examples of *natural, practically useful* operations that manifest such differences? If so, this might be considered as a significant advantage of $\mathcal{L}$ over $\mathcal{L}'$.

If the 'operations' we are asking about are ordinary first-order functions — that is, both their inputs and outputs are of ground type (strings, arbitrary-size integers etc.) — then the situation is easily summarised. At such types, all reasonable languages give rise to the same class of programmable functions, namely the Church-Turing computable ones. As for complexity, the runtime of a program is typically analysed with respect to some cost model for basic instructions (e.g. one unit of time per array access). Although the realism of such cost models in the asymptotic limit can be questioned (see, e.g., [Knuth 1997, Section 2.6]), it is broadly taken as read that such models are equally applicable whatever programming language we are working with, and moreover that all respectable languages can represent all algorithms of interest; thus, one does not expect the best achievable asymptotic run-time for a typical algorithm (say in number theory or graph theory) to be sensitive to the choice of programming language, except perhaps in marginal cases.

The situation changes radically, however, if we consider *higher-order* operations: programmable operations whose inputs may themselves be programmable operations. Here it turns out that both what is computable and the efficiency with which it can be computed can be highly sensitive to the selection of language features present. This is in fact true more widely for *abstract data types*, of which higher-order types can be seen as a special case: a higher-order value will be represented within the machine as ground data, but a program within the language typically has no access to this internal representation, and can interact with the value only by applying it to an argument.

Most work in this area to date has focused on computability differences. One of the best known examples is the *parallel if* operation which is computable in a language with parallel evaluation but not in a typical 'sequential' programming language [Plotkin 1977]. It is also well known that the presence of control features or local state enables observational distinctions that cannot be made in a purely functional setting: for instance, there are programs involving 'call/cc' that detect the order in which a (call-by-name) '+' operation evaluates its arguments [Cartwright and Felleisen 1992]. Such operations are 'non-functional' in the sense that their output is not determined solely by the extension of their input (seen as a mathematical function $\mathbb{N}_\perp \times \mathbb{N}_\perp \to \mathbb{N}_\perp$); however, there are also programs with 'functional' behaviour that can be implemented with control or local state but not without them [Longley 1999]. More recent results have exhibited differences lower down in the language expressivity spectrum: for instance, in a purely functional setting *à la* Haskell, the expressive power of *recursion* increases strictly with its type level [Longley 2018], and there are natural operations computable by low-order recursion but not by high-order iteration [Longley 2019]. Much of this territory, including the mathematical theory of some of the natural notions of higher-order computability that arise in this way, is mapped out by Longley and Normann [2015].

Relatively few results of this character have so far been established on the complexity side. Pippenger [1996] gives an example of an 'online' operation on infinite sequences of atomic symbols (essentially a function from streams to streams) such that the first n output symbols can be produced within time $O(n)$ if one is working in an 'impure' version of Lisp (in which mutation of 'cons' pairs is admitted), but with a worst-case runtime no better than $\Omega(n \log n)$ for any implementation in pure Lisp (without such mutation). This example was reconsidered by Bird et al. [1997] who showed that the same speedup can be achieved in a pure language by using lazy evaluation. Another candidate is the familiar $\log n$ overhead involved in implementing maps (supporting lookup and extension) in a pure functional language [Okasaki 1999], although to our knowledge this situation has not yet been subjected to theoretical scrutiny. Jones [2001] explores the approach of manifesting

expressivity and efficiency differences between certain languages by artificially restricting attention to 'cons-free' programs; in this setting, the classes of representable first-order functions for the various languages are found to coincide with some well-known complexity classes.

The purpose of the present paper is to give a clear example of such an inherent complexity difference higher up in the expressivity spectrum. Specifically, we consider the following *generic count* problem, parametric in n: given a boolean-valued predicate P on the space $\mathbb{B}^n$ of boolean vectors of length n, return the number of such vectors q for which $P\,q = \mathsf{true}$. We shall consider boolean vectors of any length to be represented by the type $\mathsf{Nat} \to \mathsf{Bool}$; thus for each n, we are asking for an implementation of a certain third-order operation

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

A naïve implementation strategy, supported by any reasonable language, is simply to apply P to each of the 2^n vectors in turn. A much less obvious, but still purely 'functional', approach due to Berger [1990] achieves the effect of 'pruned search' where the predicate allows it (serving as a warning that counter-intuitive phenomena can arise in this territory). Nonetheless, under a mild condition on P (namely that it must inspect all n components of the given vector before returning), both these approaches will have a $\Omega(n2^n)$ runtime. Moreover, we shall show that in a typical call-by-value language without advanced control features, one cannot improve on this: *any* implementation of count_n must necessarily take time $\Omega(n2^n)$ on *any* predicate P. On the other hand, if we extend our language with a feature such as *effect handlers* (see Section 2 below), it becomes possible to bring the runtime down to $O(2^n)$: an asymptotic gain of a factor of n.

The *generic search* problem is just like the generic count problem, except rather than counting the vectors q such that $P\,q = \mathsf{true}$, it returns the list of all such vectors. The $\Omega(n2^n)$ runtime for purely functional implementations transfers directly to generic search, as generic count reduces to generic search composed with computing the length of the resulting list. In Section 7.2 we illustrate that the $O(2^n)$ runtime for generic count with effect handlers also transfers to generic search.

The idea behind the speedup is easily explained and will already be familiar, at least informally, to programmers who have worked with multi-shot continuations. Suppose for example $n = 3$, and suppose that the predicate P always inspects the components of its argument in the order $0, 1, 2$. A naïve implementation of count_3 might start by applying the given P to $q_0 = (\mathsf{true}, \mathsf{true}, \mathsf{true})$, and then to $q_1 = (\mathsf{true}, \mathsf{true}, \mathsf{false})$. Clearly there is some duplication here: the computations of $P\,q_0$ and $P\,q_1$ will proceed identically up to the point where the value of the final component is requested. What we would like to do, then, is to record the state of the computation of $P\,q_0$ at just this point, so that we can later resume this computation with false supplied as the final component value in order to obtain the value of $P\,q_1$. (Similarly for all other internal nodes in the evident binary tree of boolean vectors.) Of course, this 'backup' approach would be standardly applied if one were implementing a bespoke search operation for some *particular* choice of P (corresponding, say, to the n-queens problem); but to apply this idea of resuming previous subcomputations in the *generic* setting (that is, uniformly in P) requires some special language feature such as effect handlers or multi-shot continuations. One could also obviate the need for such a feature by choosing to present the predicate P in some other way, but from our present perspective this would be to move the goalposts: our intention is precisely to show that our languages differ in an essential way *as regards their power to manipulate data of type* $(\mathsf{Nat} \to \mathsf{Bool}) \to \mathsf{Bool}$.

This idea of using first-class control to achieve 'backtracking' has been exploited before and is fairly widely known (see e.g. [Kiselyov et al. 2005]), and there is a clear programming intuition that this yields a speedup unattainable in languages without such control features. Our main contribution in this paper is to provide, for the first time, a precise mathematical theorem that pins down this fundamental efficiency difference, thus giving formal substance to this intuition. Since

our goal is to give a realistic analysis of the efficiency achievable in various settings without getting bogged down in inessential implementation details, we shall work concretely and operationally with the languages in question, using a CEK-style abstract machine semantics as our basic model of execution time, and with some specific programs in these languages. In the first instance, we formulate our results as a comparison between a purely functional base language (a version of call-by-value PCF) and an extension with first-class control; we then indicate how these results can be extended to base languages with other features such as mutable state.

In summary, our purpose is to exhibit an efficiency gap which, in our view, manifests a fundamental feature of the programming language landscape, challenging a common assumption that all real-world programming languages are essentially 'equivalent' from an asymptotic point of view. We believe that such results are important not only for a rounded understanding of the relative merits of existing languages, but also for informing future language design.

For their convenience as structured delimited control operators we adopt effect handlers as our universal control abstraction of choice, but our results adapt mutatis mutandis to other first-class control abstractions such as 'call/cc' [Sperber et al. 2009], 'control' ($\mathcal{F}$) and 'prompt' (#) [Felleisen 1988], or 'shift' and 'reset' [Danvy and Filinski 1990].

The rest of the paper is structured as follows.

- Section 2 provides an introduction to effect handlers as a programming abstraction.
- Section 3 presents a PCF-like language λ_{b} and its extension λ_{h} with effect handlers.
- Section 4 defines abstract machines for λ_{b} and λ_{h}, yielding a runtime cost model.
- Section 5 introduces generic count and some associated machinery, and presents an implementation in λ_{h} with runtime $O(2^n)$.
- Section 6 establishes that any generic count implementation in λ_{b} must have runtime $\Omega(n2^n)$.
- Section 7 shows that our results scale to richer settings including support for a wider class of predicates, the adaptation from generic count to generic search, and an extension of the base language with state.
- Section 8 evaluates implementations of generic search based on λ_{b} and λ_{h} in Standard ML.
- Section 9 concludes.

The languages λ_{b} and λ_{h} are rather minimal versions of previously studied systems — we only include the machinery needed for illustrating the generic search efficiency phenomenon. Auxiliary results are included in the appendices of the extended version of the paper [Hillerström et al. 2020b].

2 EFFECT HANDLERS PRIMER

Effect handlers were originally studied as a theoretical means to provide a semantics for exception handling in the setting of algebraic effects [Plotkin and Power 2001; Plotkin and Pretnar 2013]. Subsequently they have emerged as a practical programming abstraction for modular effectful programming [Bauer and Pretnar 2015; Convent et al. 2020; Dolan et al. 2015; Hillerström et al. 2020a; Kammar et al. 2013; Kiselyov et al. 2013; Leijen 2017]. In this section we give a short introduction to effect handlers. For a thorough introduction to programming with effect handlers, we recommend the tutorial by Pretnar [2015], and as an introduction to the mathematical foundations of handlers, we refer the reader to the founding paper by Plotkin and Pretnar [2013] and the excellent tutorial paper by Bauer [2018].

Viewed through the lens of universal algebra, an algebraic effect is given by a signature Σ of typed *operation symbols* along with an equational theory that describes the properties of the operations [Plotkin and Power 2001]. An example of an algebraic effect is *nondeterminism*, whose signature consists of a single nondeterministic choice operation: $\Sigma := \{\mathsf{Branch} : \mathsf{Unit} \to \mathsf{Bool}\}$. The operation takes a single parameter of type unit and ultimately produces a boolean value. The

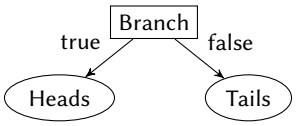
pragmatic programmatic view of algebraic effects differs from the original development as no implementation accounts for equations over operations yet.

As a simple example, let us use the operation Branch to model a coin toss. Suppose we have a data type Toss := Heads | Tails, then we may implement a coin toss as follows.

$$\begin{array}{l}\mathsf{toss} : \mathsf{Unit} \longrightarrow \mathsf{Toss} \\ \mathsf{toss}\ \langle\rangle = \textbf{if do}\ \mathsf{Branch}\ \langle\rangle\ \textbf{then}\ \mathsf{Heads}\ \textbf{else}\ \mathsf{Tails}\end{array}$$

From the type signature it is clear that the computation returns a value of type Toss. It is not clear from the signature of toss whether it performs an effect. However, from the definition, it evidently performs the operation Branch with argument $\langle\rangle$ using the **do**-invocation form. The result of the operation determines whether the computation returns either Heads or Tails. Systems such as Frank [Convent et al. 2020; Lindley et al. 2017], Helium [Biernacki et al. 2019, 2020], Koka [Leijen 2017], and Links [Hillerström and Lindley 2016; Hillerström et al. 2020a] include type-and-effect systems which track the use of effectful operations, whilst current iterations of systems such as Eff [Bauer and Pretnar 2015] and Multicore OCaml [Dolan et al. 2015] elect not to track effects in the type system. Our language is closer to the latter two.

We may view an effectful computation as a tree, where the interior nodes correspond to operation invocations and the leaves correspond to return values. The computation tree for toss is as follows.



It models interaction with the environment. The operation Branch can be viewed as a *query* for which the *response* is either true or false. The response is provided by an effect handler. As an example, consider the following handler which enumerates the possible outcomes of a coin toss.

$$\begin{array}{l}\textbf{handle}\ \mathsf{toss}\ \langle\rangle\ \textbf{with} \\ \quad \textbf{val}\ x \qquad\quad \mapsto [x] \\ \quad \mathsf{Branch}\ \langle\rangle\ r \mapsto r\ \mathsf{true} +\!\!+ r\ \mathsf{false}\end{array}$$

The **handle**-construct generalises the exceptional syntax of Benton and Kennedy [2001]. This handler has a *success* clause and an *operation* clauses. The success clause determines how to interpret the return value of toss, or equivalently how to interpret the leaves of its computation tree. It lifts the return value into a singleton list. The operation clause determines how to interpret occurrences of Branch in toss. It provides access to the argument of Branch (which is unit) and its resumption, r. The resumption is a first-class delimited continuation which captures the remainder of the toss computation from the invocation of Branch up to its nearest enclosing handler.

Applying r to true resumes evaluation of toss via the true branch, returning Heads and causing the success clause of the handler to be invoked; thus the result of r true is [Heads]. Evaluation continues in the operation clause, meaning that r is applied again, but this time to false, which causes evaluation to resume in toss via the false branch. By the same reasoning, the value of r false is [Tails], which is concatenated with the result of the true branch; hence the handler ultimately returns [Heads, Tails].

3 CALCULI

In this section, we present our base language λ_{b} and its extension with effect handlers λ_{h}.

3.1 Base Calculus

The base calculus λ_{b} is a fine-grain call-by-value [Levy et al. 2003] variation of PCF [Plotkin 1977]. Fine-grain call-by-value is similar to A-normal form [Flanagan et al. 1993] in that every intermediate computation is named, but unlike A-normal form is closed under reduction.

The syntax of λ_{b} is as follows.

$$
\begin{array}{lrl}
\text{Types} & A, B, C, D \in \mathsf{Type} & ::= \mathsf{Nat} \mid \mathsf{Unit} \mid A \to B \mid A \times B \mid A + B \\
\text{Type Environments} & \Gamma \in \mathsf{Ctx} & ::= \cdot \mid \Gamma, x : A \\
\text{Values} & V, W \in \mathsf{Val} & ::= x \mid k \mid c \mid \lambda x^{A}.\, M \mid \mathbf{rec}\, f^{A \to B}\, x.M \\
 & & \mid \langle\rangle \mid \langle V, W \rangle \mid (\mathbf{inl}\ V)^{B} \mid (\mathbf{inr}\ W)^{A} \\
\text{Computations} & M, N \in \mathsf{Comp} & ::= V\ W \mid \mathbf{let}\ \langle x, y \rangle = V\ \mathbf{in}\ N \\
 & & \mid \mathbf{case}\ V\ \{\mathbf{inl}\ x \mapsto M; \mathbf{inr}\ y \mapsto N\} \\
 & & \mid \mathbf{return}\ V \mid \mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N
\end{array}
$$

The ground types are Nat and Unit which classify natural number values and the unit value, respectively. The function type $A \to B$ classifies functions that map values of type A to values of type B. The binary product type $A \times B$ classifies pairs of values whose first and second components have types A and B respectively. The sum type $A + B$ classifies tagged values of either type A or B. Type environments Γ map term variables to their types.

We let k range over natural numbers and c range over primitive operations on natural numbers $(+, -, =)$. We let x, y, z range over term variables. For convenience, we also use f, g, and h for variables of function type, i and j for variables of type Nat, and r to denote resumptions. The value terms are standard.

We will occasionally blur the distinction between object and meta language by writing A for the meta level type of closed value terms of type A. All elimination forms are computation terms. Abstraction is eliminated using application $(V\ W)$. The product eliminator $(\mathbf{let}\ \langle x, y \rangle = V\ \mathbf{in}\ N)$ splits a pair V into its constituents and binds them to x and y, respectively. Sums are eliminated by a case split $(\mathbf{case}\ V\ \{\mathbf{inl}\ x \mapsto M; \mathbf{inr}\ y \mapsto N\})$. A trivial computation $(\mathbf{return}\ V)$ returns value V. The sequencing expression $(\mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N)$ evaluates M and binds the result value to x in N.

The typing rules are given in Figure 1. We require two typing judgements: one for values and the other for computations. The judgement $\Gamma \vdash \square : A$ states that a $\square$-term has type A under type environment Γ, where $\square$ is either a value term (V) or a computation term (M). The constants have the following types.

$$
\{(+), (-)\} : \mathsf{Nat} \times \mathsf{Nat} \to \mathsf{Nat} \qquad\qquad (=) : \mathsf{Nat} \times \mathsf{Nat} \to \mathsf{Unit} + \mathsf{Unit}
$$

We give a small-step operational semantics for λ_{b} with *evaluation contexts* in the style of Felleisen [1987]. The reduction rules are given in Figure 2. We write $M[V/x]$ for M with V substituted for x and $\ulcorner c \urcorner$ for the usual interpretation of constant c as a meta-level function on closed values. The reduction relation $\rightsquigarrow$ is defined on computation terms. The statement $M \rightsquigarrow N$ reads: term M reduces to term N in one step. We write R^{+} for the transitive closure of relation R and R^{*} for the reflexive, transitive closure of relation R.

Notation. We elide type annotations when clear from context. For convenience we often write code in direct-style assuming the standard left-to-right call-by-value elaboration into fine-grain call-by-value [Flanagan et al. 1993; Moggi 1991]. For example, the expression $f\,(h\,w) + g\,\langle\rangle$ is syntactic sugar for:

$$
\mathbf{let}\ x \leftarrow h\,w\ \mathbf{in}\ \mathbf{let}\ y \leftarrow f\,x\ \mathbf{in}\ \mathbf{let}\ z \leftarrow g\,\langle\rangle\ \mathbf{in}\ y + z
$$

We define sequencing of computations in the standard way.

$$
M; N := \mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N, \quad \text{where } x \notin FV(N)
$$

Values

$$\text{T-Var}\ \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \text{T-Unit}\ \frac{}{\Gamma \vdash \langle\rangle : \mathsf{Unit}} \qquad \text{T-Nat}\ \frac{k \in \mathbb{N}}{\Gamma \vdash k : \mathsf{Nat}} \qquad \text{T-Const}\ \frac{c : A \to B}{\Gamma \vdash c : A \to B}$$

$$\text{T-Lam}\ \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x^{A}.\, M : A \to B} \qquad \text{T-Rec}\ \frac{\Gamma, f : A \to B, x : A \vdash M : B}{\Gamma \vdash \mathbf{rec}\ f^{A \to B}\, x.\, M : A \to B}$$

$$\text{T-Prod}\ \frac{\Gamma \vdash V : A \qquad \Gamma \vdash W : B}{\Gamma \vdash \langle V, W\rangle : A \times B} \qquad \text{T-Inl}\ \frac{\Gamma \vdash V : A}{\Gamma \vdash (\mathbf{inl}\ V)^{B} : A + B} \qquad \text{T-Inr}\ \frac{\Gamma \vdash W : B}{\Gamma \vdash (\mathbf{inr}\ W)^{A} : A + B}$$

Computations

$$\text{T-App}\ \frac{\Gamma \vdash V : A \to B \qquad \Gamma \vdash W : A}{\Gamma \vdash V\ W : B} \qquad \text{T-Split}\ \frac{\Gamma \vdash V : A \times B \qquad \Gamma, x : A, y : B \vdash N : C}{\Gamma \vdash \mathbf{let}\ \langle x, y\rangle = V\ \mathbf{in}\ N : C}$$

$$\text{T-Case}\ \frac{\Gamma \vdash V : A + B \qquad \Gamma, x : A \vdash M : C \qquad \Gamma, y : B \vdash N : C}{\Gamma \vdash \mathbf{case}\ V\ \{\mathbf{inl}\ x \mapsto M; \mathbf{inr}\ y \mapsto N\} : C}$$

$$\text{T-Return}\ \frac{\Gamma \vdash V : A}{\Gamma \vdash \mathbf{return}\ V : A} \qquad \text{T-Let}\ \frac{\Gamma \vdash M : A \qquad \Gamma, x : A \vdash N : C}{\Gamma \vdash \mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N : C}$$

Fig. 1. Typing Rules for λ_{b}

$$\begin{array}{lll}
\text{S-App} & & (\lambda x^{A}.\, M)V \rightsquigarrow M[V/x] \\
\text{S-App-Rec} & & (\mathbf{rec}\ f^{A}\, x.\, M)V \rightsquigarrow M[(\mathbf{rec}\ f^{A}\, x.\, M)/f, V/x] \\
\text{S-Const} & & c\ V \rightsquigarrow \mathbf{return}\ (\ulcorner c \urcorner (V)) \\
\text{S-Split} & & \mathbf{let}\ \langle x, y\rangle = \langle V, W\rangle\ \mathbf{in}\ N \rightsquigarrow N[V/x, W/y] \\
\text{S-Case-inl} & \mathbf{case}\ (\mathbf{inl}\ V)^{B}\ \{\mathbf{inl}\ x \mapsto M; \mathbf{inr}\ y \mapsto N\} & \rightsquigarrow M[V/x] \\
\text{S-Case-inr} & \mathbf{case}\ (\mathbf{inr}\ V)^{A}\ \{\mathbf{inl}\ x \mapsto M; \mathbf{inr}\ y \mapsto N\} & \rightsquigarrow N[V/y] \\
\text{S-Let} & & \mathbf{let}\ x \leftarrow \mathbf{return}\ V\ \mathbf{in}\ N \rightsquigarrow N[V/x] \\
\text{S-Lift} & & \mathcal{E}[M] \rightsquigarrow \mathcal{E}[N], \qquad \text{if } M \rightsquigarrow N
\end{array}$$

$$\text{Evaluation contexts} \quad \mathcal{E} ::= [\,] \mid \mathbf{let}\ x \leftarrow \mathcal{E}\ \mathbf{in}\ N$$

Fig. 2. Contextual Small-Step Operational Semantics

We make use of standard syntactic sugar for pattern matching. For instance, we write

$$\lambda\langle\rangle.M := \lambda x^{\mathsf{Unit}}.M, \quad \text{where } x \notin FV(M)$$

for suspended computations, and if the binder has a type other than Unit, we write:

$$\lambda\_^{A}.M := \lambda x^{A}.M, \quad \text{where } x \notin FV(M)$$

We use the standard encoding of booleans as a sum:

$$\mathsf{Bool} := \mathsf{Unit} + \mathsf{Unit} \qquad \mathsf{true} := \mathbf{inl}\ \langle\rangle \qquad \mathsf{false} := \mathbf{inr}\ \langle\rangle$$

$$\mathbf{if}\ V\ \mathbf{then}\ M\ \mathbf{else}\ N := \mathbf{case}\ V\ \{\mathbf{inl}\ \langle\rangle \mapsto M; \mathbf{inr}\ \langle\rangle \mapsto N\}$$

Computations

$$\textsc{T-Do}\ \frac{(\ell : A \to B) \in \Sigma \qquad \Gamma \vdash V : A}{\Gamma \vdash \mathbf{do}\ \ell\ V : B} \qquad \textsc{T-Handle}\ \frac{\Gamma \vdash M : C \qquad \Gamma \vdash H : C \Rightarrow D}{\Gamma \vdash \mathbf{handle}\ M\ \mathbf{with}\ H : D}$$

Handlers

$$\textsc{T-Handler}\ \frac{H^{\mathrm{val}} = \{\mathbf{val}\ x \mapsto M\} \qquad [H^{\ell} = \{\ell\ p\ r \mapsto N_{\ell}\}]_{\ell \in dom(\Sigma)} \qquad \Gamma, x : C \vdash M : D \qquad [\Gamma, p : A_{\ell}, r : B_{\ell} \to D \vdash N_{\ell} : D]_{(\ell : A_{\ell} \to B_{\ell}) \in \Sigma}}{\Gamma \vdash H : C \Rightarrow D}$$

Fig. 3. Additional Typing Rules for λ_{h}

3.2 Handler Calculus

We now define λ_{h} as an extension of λ_{b}.

Operation symbols	$\ell \in \mathcal{L}$
Signatures	$\Sigma ::= \cdot \mid \{\ell : A \to B\} \cup \Sigma$
Handler types	$F ::= C \Rightarrow D$
Computations	$M, N ::= \cdots \mid \mathbf{do}\ \ell\ V \mid \mathbf{handle}\ M\ \mathbf{with}\ H$
Handlers	$H ::= \{\mathbf{val}\ x \mapsto M\} \mid \{\ell\ p\ r \mapsto N\} \uplus H$

We assume a countably infinite set $\mathcal{L}$ of operation symbols ℓ. An effect signature Σ is a map from operation symbols to their types, thus we assume that each operation symbol in a signature is distinct. An operation type $A \to B$ classifies operations that take an argument of type A and return a result of type B. We write $dom(\Sigma) \subseteq \mathcal{L}$ for the set of operation symbols in a signature Σ. A handler type $C \Rightarrow D$ classifies effect handlers that transform computations of type C into computations of type D. Following Pretnar [2015], we assume a global signature for every program. Computations are extended with operation invocation ($\mathbf{do}\ \ell\ V$) and effect handling ($\mathbf{handle}\ M\ \mathbf{with}\ H$). Handlers are constructed from one success clause ($\{\mathbf{val}\ x \mapsto M\}$) and one operation clause ($\{\ell\ p\ r \mapsto N\}$) for each operation ℓ in Σ. Following Plotkin and Pretnar [2013], we adopt the convention that a handler with missing operation clauses (with respect to Σ) is syntactic sugar for one in which all missing clauses perform explicit forwarding:

$$\{\ell\ p\ r \mapsto \mathbf{let}\ x \leftarrow \mathbf{do}\ \ell\ p\ \mathbf{in}\ r\ x\}$$

The typing rules for λ_{h} are those of λ_{b} (Figure 1) plus three additional rules for operations, handling, and handlers given in Figure 3. The T-Do rule ensures that an operation invocation is only well-typed if the operation ℓ appears in the effect signature Σ and the argument type A matches the type of the provided argument V. The result type B determines the type of the invocation. The T-Handle rule types handler application. The T-Handler rule ensures that the bodies of the success clause and the operation clauses all have the output type D. The type of x in the success clause must match the input type C. The type of the parameter p (A_{ℓ}) and resumption r ($B_{\ell} \to D$) in operation clause H^{ℓ} is determined by the type of ℓ; the return type of r is D, as the body of the resumption will itself be handled by H. We write H^{ℓ} and H^{val} for projecting success and operation clauses.

$$H^{\mathrm{val}} := \{\mathbf{val}\ x \mapsto M\}, \quad \text{where } \{\mathbf{val}\ x \mapsto M\} \in H$$
$$H^{\ell} := \{\ell\ p\ r \mapsto M\}, \quad \text{where } \{\ell\ p\ r \mapsto M\} \in H$$

We extend the operational semantics to λ_{h}. Specifically, we add two new reduction rules: one for handling return values and another for handling operation invocations.

$$\text{S-Ret} \quad \textbf{handle}\ (\textbf{return}\ V)\ \textbf{with}\ H \rightsquigarrow N[V/x], \qquad \text{where } H^{\mathrm{val}} = \{\textbf{val}\ x \mapsto N\}$$
$$\text{S-Op} \quad \textbf{handle}\ \mathcal{E}[\textbf{do}\ \ell\ V]\ \textbf{with}\ H \rightsquigarrow N[V/p,\ (\lambda y.\textbf{handle}\ \mathcal{E}[\textbf{return}\ y]\ \textbf{with}\ H)/r], \quad \text{where } H^{\ell} = \{\ell\ p\ r \mapsto N\}$$

The first rule invokes the success clause. The second rule handles an operation via the corresponding operation clause. If we were naïvely to extend evaluation contexts with the handle construct then our semantics would become nondeterministic, as it may pick an arbitrary handler in scope. In order to ensure that the semantics is deterministic, we instead add a distinct form of evaluation context for effectful computation, which we call handler contexts.

$$\text{Handler contexts} \quad \mathcal{H} ::= [\,] \mid \textbf{handle}\ \mathcal{H}\ \textbf{with}\ H \mid \textbf{let}\ x \leftarrow \mathcal{H}\ \textbf{in}\ N$$

We replace the S-Lift rule with a corresponding rule for handler contexts.

$$\mathcal{H}[M] \rightsquigarrow \mathcal{H}[N], \qquad \text{if } M \rightsquigarrow N$$

The separation between pure evaluation contexts $\mathcal{E}$ and handler contexts $\mathcal{H}$ ensures that the S-Op rule always selects the innermost handler.

We now characterise normal forms and state the standard type soundness property of λ_{h}.

Definition 3.1 (Computation normal forms). A computation term N is normal with respect to Σ, if $N = \textbf{return}\ V$ for some V or $N = \mathcal{E}[\textbf{do}\ \ell\ W]$ for some $\ell \in \mathit{dom}(\Sigma)$, $\mathcal{E}$, and W.

Theorem 3.2 (Type Soundness). *If $\vdash M : C$, then either there exists $\vdash N : C$ such that $M \rightsquigarrow^* N$ and N is normal with respect to Σ, or M diverges.*

3.3 The Role of Types

Readers familiar with backtracking search algorithms may wonder where types come into the expressiveness picture. Types will not play a direct role in our proofs but rather in the characterisation of which programs can be meaningfully compared. In particular, types are used to rule out global approaches such as continuation passing style (CPS): without types one could obtain an efficient pure generic count program by CPS transforming the entire program.

Readers familiar with effect handlers may wonder why our handler calculus does not include an effect type system. As types frame the comparison of programs between languages, we require that types be fixed across languages; hence λ_{h} does not include effect types. Future work includes reconciling effect typing with our approach to expressiveness.

4 ABSTRACT MACHINE SEMANTICS

Thus far we have introduced the base calculus λ_{b} and its extension with effect handlers λ_{h}. For each calculus we have given a *small-step operational semantics* which uses a substitution model for evaluation. Whilst this model is semantically pleasing, it falls short of providing a realistic account of practical computation as substitution is an expensive operation. We now develop a more practical model of computation based on an *abstract machine semantics*.

4.1 Base Machine

We choose a *CEK*-style abstract machine semantics [Felleisen and Friedman 1987] for λ_{b} based on that of Hillerström et al. [2020a]. The CEK machine operates on configurations which are triples of the form $\langle M \mid \gamma \mid \sigma \rangle$. The first component contains the computation currently being evaluated. The second component contains the environment γ which binds free variables. The third component

Transition relation

$$\begin{array}{lll}
\text{M-App} & \langle V\ W \mid \gamma \mid \sigma\rangle \longrightarrow \langle M \mid \gamma'[x \mapsto [\![W]\!]\gamma] \mid \sigma\rangle, & \text{if } [\![V]\!]\gamma = (\gamma', \lambda x^A.\, M) \\
\text{M-Rec} & \langle V\ W \mid \gamma \mid \sigma\rangle \longrightarrow \langle M \mid \gamma'[f \mapsto (\gamma', \mathbf{rec}\, f^{A\to B}\, x.M), x \mapsto [\![W]\!]\gamma] \mid \sigma\rangle, & \text{if } [\![V]\!]\gamma = (\gamma', \mathbf{rec}\, f^{A\to B}\, x.M) \\
\text{M-Const} & \langle V\ W \mid \gamma \mid \sigma\rangle \longrightarrow \langle \mathbf{return}\ (\ulcorner c \urcorner ([\![W]\!]\gamma)) \mid \gamma \mid \sigma\rangle, & \text{if } [\![V]\!]\gamma = c \\
\text{M-Split} & \langle \mathbf{let}\ \langle x, y\rangle = V\ \mathbf{in}\ N \mid \gamma \mid \sigma\rangle \longrightarrow \langle N \mid \gamma[x \mapsto v, y \mapsto w] \mid \sigma\rangle, & \text{if } [\![V]\!]\gamma = \langle v; w\rangle \\
\text{M-CaseL} & \langle \mathbf{case}\ V\ \{\mathbf{inl}\ x \mapsto M; \mathbf{inr}\ y \mapsto N\} \mid \gamma \mid \sigma\rangle \longrightarrow \langle M \mid \gamma[x \mapsto v] \mid \sigma\rangle, & \text{if } [\![V]\!]\gamma = \mathbf{inl}\ v \\
\text{M-CaseR} & \langle \mathbf{case}\ V\ \{\mathbf{inl}\ x \mapsto M; \mathbf{inr}\ y \mapsto N\} \mid \gamma \mid \sigma\rangle \longrightarrow \langle N \mid \gamma[y \mapsto v] \mid \sigma\rangle, & \text{if } [\![V]\!]\gamma = \mathbf{inr}\ v \\
\text{M-Let} & \langle \mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N \mid \gamma \mid \sigma\rangle \longrightarrow \langle M \mid \gamma \mid (\gamma, x, N) :: \sigma\rangle & \\
\text{M-RetCont} & \langle \mathbf{return}\ V \mid \gamma \mid (\gamma', x, N) :: \sigma\rangle \longrightarrow \langle N \mid \gamma'[x \mapsto [\![V]\!]\gamma] \mid \sigma\rangle &
\end{array}$$

Value interpretation

$$\begin{array}{lll}
[\![x]\!]\gamma = \gamma(x) & [\![n]\!]\gamma = n & [\![\lambda x^A.M]\!]\gamma = (\gamma, \lambda x^A.M) \\
[\![\langle\rangle]\!]\gamma = \langle\rangle & [\![c]\!]\gamma = c & [\![\mathbf{rec}\, f^{A\to B}\, x.M]\!]\gamma = (\gamma, \mathbf{rec}\, f^{A\to B}\, x.M) \\
[\![\langle V, W\rangle]\!]\gamma = \langle [\![V]\!]\gamma, [\![W]\!]\gamma\rangle & & [\![(\mathbf{inl}\ V)^B]\!]\gamma = (\mathbf{inl}\ [\![V]\!]\gamma)^B \\
 & & [\![(\mathbf{inr}\ V)^A]\!]\gamma = (\mathbf{inr}\ [\![V]\!]\gamma)^A
\end{array}$$

Fig. 4. Abstract Machine Semantics for λ_{b}

contains the continuation which instructs the machine how to proceed once evaluation of the current computation is complete. The syntax of abstract machine states is as follows.

$$\begin{array}{lll}
\text{Configurations} & C \in \mathsf{Conf} & ::= \langle M \mid \gamma \mid \sigma\rangle \\
\text{Environments} & \gamma \in \mathsf{Env} & ::= \emptyset \mid \gamma[x \mapsto v] \\
\text{Machine values} & v, w \in \mathsf{MVal} & ::= x \mid n \mid c \mid \langle\rangle \mid \langle v, w\rangle \\
 & & \quad \mid\ (\gamma, \lambda x^A.\, M) \mid (\gamma, \mathbf{rec}\, f^{A\to B}\, x.\, M) \mid (\mathbf{inl}\ v)^B \mid (\mathbf{inr}\ w)^A \\
\text{Pure continuations} & \sigma \in \mathsf{PureCont} & ::= [] \mid (\gamma, x, N) :: \sigma
\end{array}$$

Values consist of function closures, constants, pairs, and left or right tagged values. We refer to continuations of the base machine as *pure*. A pure continuation is a stack of pure continuation frames. A pure continuation frame (γ, x, N) closes a let-binding $\mathbf{let}\ x \leftarrow [\ ]\ \mathbf{in}\ N$ over environment γ. We write $[]$ for an empty pure continuation and $\phi :: \sigma$ for the result of pushing the frame ϕ onto σ. We use pattern matching to deconstruct pure continuations.

The abstract machine semantics is given in Figure 4. The transition relation ($\longrightarrow$) makes use of the value interpretation ($[\![-]\!]$) from value terms to machine values. The machine is initialised by placing a term in a configuration alongside the empty environment ($\emptyset$) and identity pure continuation ($[]$). The rules (M-App), (M-Rec), (M-Const), (M-Split), (M-CaseL), and (M-CaseR) eliminate values. The (M-Let) rule extends the current pure continuation with let bindings. The (M-RetCont) rule extends the environment in the top frame of the pure continuation with a returned value. Given an input of a well-typed closed computation term $\vdash M : A$, the machine will either diverge or return a value of type A. A final state is given by a configuration of the form $\langle \mathbf{return}\ V \mid \gamma \mid []\rangle$ in which case the final return value is given by the denotation $[\![V]\!]\gamma$ of V under environment γ.

Transition relation

$$
\begin{array}{lrl}
\textsc{M-Resume} & \langle V\ W \mid \gamma \mid \kappa \rangle & \longrightarrow \langle \mathbf{return}\ W \mid \gamma \mid (\sigma, \chi) :: \kappa \rangle, \\
 & & \quad \text{if } [\![V]\!]\gamma = (\sigma, \chi) \\
\textsc{M-Let} & \langle \mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N \mid \gamma \mid (\sigma, \chi) :: \kappa \rangle & \longrightarrow \langle M \mid \gamma \mid ((\gamma, x, N) :: \sigma, \chi) :: \kappa \rangle \\
\textsc{M-RetCont} & \langle \mathbf{return}\ V \mid \gamma \mid ((\gamma', x, N) :: \sigma, \chi) :: \kappa \rangle & \longrightarrow \langle N \mid \gamma'[x \mapsto [\![V]\!]\gamma] \mid (\sigma, \chi) :: \kappa \rangle \\
\textsc{M-Handle} & \langle \mathbf{handle}\ M\ \mathbf{with}\ H \mid \gamma \mid \kappa \rangle & \longrightarrow \langle M \mid \gamma \mid ([], (\gamma, H)) :: \kappa \rangle \\
\textsc{M-RetHandler} & \langle \mathbf{return}\ V \mid \gamma \mid ([], (\gamma', H)) :: \kappa \rangle & \longrightarrow \langle M \mid \gamma'[x \mapsto [\![V]\!]\gamma] \mid \kappa \rangle, \\
 & & \quad \text{if } H^{\mathrm{val}} = \{\mathbf{val}\ x \mapsto M\} \\
\textsc{M-Handle-Op} & \langle \mathbf{do}\ \ell\ V \mid \gamma \mid (\sigma, (\gamma', H)) :: \kappa \rangle & \longrightarrow \langle M \mid \gamma'[p \mapsto [\![V]\!]\gamma, \\
 & & \qquad r \mapsto (\sigma, (\gamma', H))] \mid \kappa \rangle, \\
 & & \quad \text{if } \ell : A \to B \in \Sigma \\
 & & \quad \text{and } H^{\ell} = \{\ell\ p\ r \mapsto M\}
\end{array}
$$

Fig. 5. Abstract Machine Semantics for λ_{h}

Correctness. The base machine faithfully simulates the operational semantics for λ_{b}; most transitions correspond directly to β-reductions, but M-Let performs an administrative step to bring the computation M into evaluation position. We formally state and prove the correspondence in Appendix A, relying on an inverse map $(\!|-|\!)$ from configurations to terms [Hillerström et al. 2020a].

4.2 Handler Machine

We now enrich the λ_{b} machine to a λ_{h} machine. We extend the syntax as follows.

$$
\begin{array}{ll}
\text{Configurations} & C \in \mathsf{Conf} ::= \langle M \mid \gamma \mid \kappa \rangle \\
\text{Resumptions} & \rho \in \mathsf{Res} ::= (\sigma, \chi) \\
\text{Continuations} & \kappa \in \mathsf{Cont} ::= [] \mid \rho :: \kappa \\
\text{Handler closures} & \chi \in \mathsf{HClo} ::= (\gamma, H) \\
\text{Machine values} & v, w \in \mathsf{MVal} ::= \cdots \mid \rho
\end{array}
$$

The notion of configurations changes slightly in that the continuation component is replaced by a generalised continuation $\kappa \in \mathsf{Cont}$ [Hillerström et al. 2020a]; a continuation is now a list of resumptions. A resumption is a pair of a pure continuation (as in the base machine) and a handler closure (χ). A handler closure consists of an environment and a handler definition, where the former binds the free variables that occur in the latter. The identity continuation is a singleton list containing the identity resumption, which is an empty pure continuation paired with the identity handler closure:

$$\kappa_0 := [([], (\emptyset, \{\mathbf{val}\ x \mapsto x\}))]$$

Machine values are augmented to include resumptions as an operation invocation causes the topmost frame of the machine continuation to be reified (and bound to the resumption parameter in the operation clause).

The handler machine adds transition rules for handlers, and modifies (M-Let) and (M-RetCont) from the base machine to account for the richer continuation structure. Figure 5 depicts the new and modified rules. The (M-Handle) rule pushes a handler closure along with an empty pure continuation onto the continuation stack. The (M-RetHandler) rule transfers control to the success clause of the current handler once the pure continuation is empty. The (M-Handle-Op) rule transfers control to the matching operation clause on the topmost handler, and during the process it reifies the handler closure. Finally, the (M-Resume) rule applies a reified handler closure, by pushing it onto the continuation stack. The handler machine has two possible final states: either it yields a value or it gets stuck on an unhandled operation.

Correctness. The handler machine faithfully simulates the operational semantics of λ_{h}. Extending the result for the base machine, we formally state and prove the correspondence in Appendix B.

4.3 Realisability and Asymptotic Complexity

As witnessed by the work of Hillerström and Lindley [2018] the machine structures are readily realisable using standard persistent functional data structures. Pure continuations on the base machine and generalised continuations on the handler machine can be implemented using linked lists with a time complexity of $O(1)$ for the extension operation $(\_ :: \_)$. The topmost pure continuation on the handler machine may also be extended in time $O(1)$, as extending it only requires reaching under the topmost handler closure. Environments, γ, can be realised using a map, with a time complexity of $O(\log|\gamma|)$ for extension and lookup [Okasaki 1999].

The worst-case time complexity of a single machine transition is exhibited by rules which involve operations on the environment, since any other operation is constant time, hence the worst-time complexity of a transition is $O(\log|\gamma|)$. The value interpretation function $[\![-]\!]\gamma$ is defined structurally on values. Its worst-time complexity is exhibited by a nesting of pairs of variables $[\![\langle x_1, \ldots, x_n\rangle]\!]\gamma$ which has complexity $O(n\log|\gamma|)$.

Continuation copying. On the handler machine the topmost continuation frame can be copied in constant time due to the persistent runtime and the layout of machine continuations. An alternative design would be to make the runtime non-persistent in which case copying a continuation frame $((\sigma, \_) :: \_)$ would be a $O(|\sigma|)$ time operation.

Primitive operations on naturals. Our model assumes that arithmetic operations on arbitrary natural numbers take $O(1)$ time. This is common practice in the study of algorithms when the main interest lies elsewhere [Cormen et al. 2009, Section 2.2]. If desired, one could adopt a more refined cost model that accounted for the bit-level complexity of arithmetic operations; however, doing so would have the same impact on both of the situations we are wishing to compare, and thus would add nothing but noise to the overall analysis.

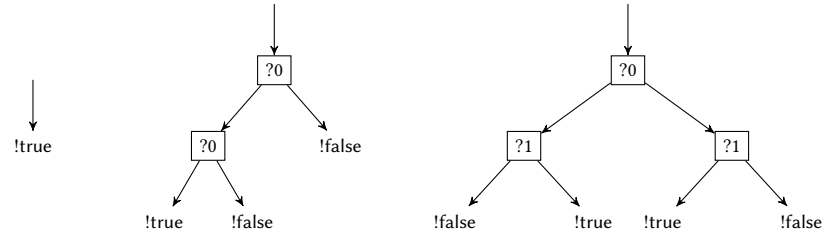
5 PREDICATES, DECISION TREES AND GENERIC COUNT

We now come to the crux of the paper. In this section and the next, we prove that λ_{h} supports implementations of certain operations with an asymptotic runtime bound that cannot be achieved in λ_{b} (Section 6). While the positive half of this claim essentially consolidates a known piece of folklore, the negative half appears to be new. To establish our result, it will suffice to exhibit a single 'efficient' program in λ_{h}, then show that no equivalent program in λ_{b} can achieve the same asymptotic efficiency. We take *generic search* as our example.

Generic search is a modular search procedure that takes as input a predicate P on some multi-dimensional search space, and finds all points of the space satisfying P. Generic search is agnostic to the specific instantiation of P, and as a result is applicable across a wide spectrum of domains. Classic examples such as Sudoku solving [Bird 2006], the n-queens problem [Bell and Stevens 2009] and graph colouring can be cast as instances of generic search, and similar ideas have been explored in connection with Nash equilibria and exact real integration [Daniels 2016; Simpson 1998].

For simplicity, we will restrict attention to search spaces of the form $\mathbb{B}^n$, the set of bit vectors of length n. To exhibit our phenomenon in the simplest possible setting, we shall actually focus on the *generic count* problem: given a predicate P on some $\mathbb{B}^n$, return the *number of* points of $\mathbb{B}^n$ satisfying P. However, we shall explain why our results are also applicable to generic search proper.

We shall view $\mathbb{B}^n$ as the set of functions $\mathbb{N}_n \to \mathbb{B}$, where $\mathbb{N}_n := \{0, \ldots, n-1\}$. In both λ_{b} and λ_{h} we may represent such functions by terms of type Nat $\to$ Bool. We will often informally write



(a) T_0 (b) I_2 (c) Odd_2

Fig. 6. Examples of Decision Trees

Nat_n in place of Nat to indicate that only the values $0, \ldots, n-1$ are relevant, but this convention has no formal status since our setup does not support dependent types.

To summarise, in both λ_b and λ_h we will be working with the types

$$\begin{array}{rr} \mathsf{Point} := \mathsf{Nat} \to \mathsf{Bool} & \mathsf{Point}_n := \mathsf{Nat}_n \to \mathsf{Bool} \\ \mathsf{Predicate} := \mathsf{Point} \to \mathsf{Bool} & \mathsf{Predicate}_n := \mathsf{Point}_n \to \mathsf{Bool} \end{array}$$

and will be looking for programs

$$\mathsf{count}_n : \mathsf{Predicate}_n \to \mathsf{Nat}$$

such that for suitable terms P representing semantic predicates $\Pi : \mathbb{B}^n \to \mathbb{B}$, $\mathsf{count}_n\ P$ finds the number of points of $\mathbb{B}^n$ satisfying Π.

Before formalising these ideas more closely, let us look at some examples, which will also illustrate the machinery of *decision trees* that we will be using.

5.1 Examples of Points, Predicates and Trees

Consider first the following terms of type Point:

$$\mathsf{q}_0 := \lambda\_.\mathsf{true} \qquad \mathsf{q}_1 := \lambda i.i = 0 \qquad \mathsf{q}_2 := \lambda i.\ \mathbf{if}\ i = 0\ \mathbf{then}\ \mathsf{true}\ \mathbf{else\ if}\ i = 1\ \mathbf{then}\ \mathsf{false}\ \mathbf{else}\ \bot$$

(Here $\bot$ is the diverging term $(\mathbf{rec}\ f\ i.f\ i)\ \langle\rangle$.) Then q_0 represents $\langle \mathsf{true}, \ldots, \mathsf{true}\rangle \in \mathbb{B}^n$ for any n; q_1 represents $\langle \mathsf{true}, \mathsf{false}, \ldots, \mathsf{false}\rangle \in \mathbb{B}^n$ for any $n \geq 1$; and q_2 represents $\langle \mathsf{true}, \mathsf{false}\rangle \in \mathbb{B}^2$.

Next some predicates. First, the following terms all represent the constant true predicate $\mathbb{B}^2 \to \mathbb{B}$:

$$\mathsf{T}_0 := \lambda q.\mathsf{true} \qquad \mathsf{T}_1 := \lambda q.(q\,1; q\,0; \mathsf{true}) \qquad \mathsf{T}_2 := \lambda q.(q\,0; q\,0; \mathsf{true})$$

These illustrate that in the course of evaluating a predicate term P at a point q, for each $i < n$ the value of q at i may be inspected zero, one or many times.

Likewise, the following all represent the 'identity' predicate $\mathbb{B}^1 \to \mathbb{B}$ (here && is shortcut 'and'):

$$\mathsf{I}_0 := \lambda q.q\,0 \qquad \mathsf{I}_1 := \lambda q.\ \mathbf{if}\ q\,0\ \mathbf{then}\ \mathsf{true}\ \mathbf{else}\ \mathsf{false} \qquad \mathsf{I}_2 := \lambda q.(q\,0)\ \&\&\ (q\,0)$$

Slightly more interestingly, for each n we have the following program which determines whether a point contains an odd number of true components:

$$\mathsf{Odd}_n := \lambda q.\ \mathsf{fold} \otimes \mathsf{false}\ (\mathsf{map}\ q\ [0, \ldots, n-1])$$

Here fold and map are the standard combinators on lists, and $\otimes$ is exclusive-or. Applying Odd_2 to q_0 yields false; applying it to q_1 or q_2 yields true.

We can think of a predicate term P as participating in a 'dialogue' with a given point $Q : \mathsf{Point}_n$. The predicate may *query* Q at some coordinate k; Q may *respond* with true or false and this returned value may influence the future course of the dialogue. After zero or more such query/response pairs, the predicate may return a final *answer* (true or false).

The set of possible dialogues with a given term P may be organised in an obvious way into an unrooted binary *decision tree*, in which each internal node is labelled with a query $?k$ (with $k < n$), and with left and right branches corresponding to the responses true, false respectively. Any point will thus determine a path through the tree, and each leaf is labelled with an answer !true or !false according to whether the corresponding point or points satisfy the predicate.

Decision trees for a sample of the above predicate terms are depicted in Figure 6; the relevant formal definitions are given in the next subsection. In the case of I_2, one of the !false leaves will be 'unreachable' if we are working in λ_{b} (but reachable in a language supporting mutable state).

We think of the edges in the tree as corresponding to portions of computation undertaken by P between queries, or before delivering the final answer. The tree is unrooted (i.e. starts with an edge rather than a node) because in the evaluation of $P\,Q$ there is potentially some 'thinking' done by P even before the first query or answer is reached. For the purpose of our runtime analysis, we will also consider *timed* variants of these decision trees, in which each edge is labelled with the number of computation steps involved.

It is possible that for a given P the construction of a decision tree may hit trouble, because at some stage P either goes undefined or gets stuck at an unhandled operation. It is also possible that the decision tree is infinite because P can keep asking queries forever. However, we shall be restricting our attention to terms representing *total* predicates: those with finite decision trees in which every path leads to a leaf.

In order to present our complexity results in a simple and clear form, we will give special prominence to certain well-behaved decision trees. For $n \in \mathbb{N}$, we shall say a tree is *n-standard* if it is total (i.e. every maximal path leads to a leaf labelled with an answer) and along any path to a leaf, each coordinate $k < n$ is queried once and only once. Thus, an n-standard decision tree is a complete binary tree of depth $n + 1$, with $2^n - 1$ internal nodes and 2^n leaves. However, there is no constraint on the order of the queries, which indeed may vary from one path to another. One pleasing property of this notion is that for a predicate term with an n-standard decision tree, the number of points in $\mathbb{B}^n$ satisfying the predicate is precisely the number of !true leaves in the tree.

Of the examples we have given, the tree for T_0 is 0-standard; those for I_0 and I_1 are 1-standard; that for Odd_n is n-standard; and the rest are not n-standard for any n.

5.2 Formal Definitions

We now formalise the above notions. We will present our definitions in the setting of λ_{h}, but everything can clearly be relativised to λ_{b} with no change to the meaning in the case of λ_{b} terms. For the purpose of this subsection we fix $n \in \mathbb{N}$, set $\mathbb{N}_n := \{0, \ldots, n-1\}$, and use k to range over $\mathbb{N}_n$. We write $\mathbb{B}$ for the set of booleans, which we shall identify with the (encoded) boolean values of λ_{h}, and use b to range over $\mathbb{B}$.

As suggested by the foregoing discussion, we will need to work with both syntax and semantics. For points, the relevant definitions are as follows.

Definition 5.1 (n-points). A closed value Q : Point is said to be a *syntactic n-point* if:

$$\forall k \in \mathbb{N}_n.\, \exists b \in \mathbb{B}.\; Q\; k \rightsquigarrow^* \mathbf{return}\; b$$

A *semantic n-point* π is simply a mathematical function $\pi : \mathbb{N}_n \to \mathbb{B}$. (We shall also write $\pi \in \mathbb{B}^n$.) Any syntactic n-point Q is said to *denote* the semantic n-point $[\![Q]\!]$ given by:

$$\forall k \in \mathbb{N}_n,\; b \in \mathbb{B}.\; [\![Q]\!](k) = b \;\Leftrightarrow\; Q\; k \rightsquigarrow^* \mathbf{return}\; b$$

Any two syntactic n-points Q and Q' are said to be *distinct* if $[\![Q]\!] \neq [\![Q']\!]$.

By default, the unqualified term *n-point* will from now on refer to syntactic n-points.

Likewise, we wish to work with predicates both syntactically and semantically. By a *semantic n-predicate* we shall mean simply a mathematical function $\Pi : \mathbb{B}^n \to \mathbb{B}$. One slick way to define syntactic n-predicates would be as closed terms P : Predicate such that for every n-point Q, $P\,Q$ evaluates to either **return** true or **return** false. For our purposes, however, we shall favour an approach to n-predicates via *decision trees*, which will yield more information on their behaviour.

We will model decision trees as certain partial functions from *addresses* to *labels*. An address will specify the position of a node in the tree via the path that leads to it, while a label will represent the information present at a node. Formally:

Definition 5.2 (untimed decision tree). (i) The address set Addr is simply the set $\mathbb{B}^*$ of finite lists of booleans. If $bs, bs' \in \mathsf{Addr}$, we write $bs \sqsubseteq bs'$ (resp. $bs \sqsubset bs'$) to mean that bs is a prefix (resp. proper prefix) of bs'.

(ii) The label set Lab consists of *queries* parameterised by a natural number and *answers* parameterised by a boolean:

$$\mathsf{Lab} := \{?k \mid k \in \mathbb{N}\} \cup \{!b \mid b \in \mathbb{B}\}$$

(iii) An (untimed) decision tree is a partial function $\tau : \mathsf{Addr} \rightharpoonup \mathsf{Lab}$ such that:

- The domain of τ (written $dom(\tau)$) is prefix closed.
- Answer nodes are always leaves: if $\tau(bs) = !b$ then $\tau(bs')$ is undefined whenever $bs \sqsubset bs'$.

As our goal is to reason about the time complexity of generic count programs and their predicates, it is also helpful to decorate decision trees with timing data that records the number of machine steps taken for each piece of computation performed by a predicate:

Definition 5.3 (timed decision tree). A timed decision tree is a partial function $\tau : \mathsf{Addr} \rightharpoonup \mathsf{Lab} \times \mathbb{N}$ such that its first projection $bs \mapsto \tau(bs).1$ is a decision tree. We write $\mathsf{labs}(\tau)$ for the first projection $(bs \mapsto \tau(bs).1)$ and $\mathsf{steps}(\tau)$ for the second projection $(bs \mapsto \tau(bs).2)$ of a timed decision tree.

Here we think of $\mathsf{steps}(\tau)(bs)$ as the computation time associated with the edge whose *target* is the node addressed by bs.

We now come to the method for associating a specific tree with a given term P. One may think of this as a kind of denotational semantics, but here we shall extract a tree from a term by purely operational means using our abstract machine model. The key idea is to try applying P to a distinguished free variable q : Point, which we think of as an 'abstract point'. Whenever P wants to interrogate its argument at some index i, the computation will get stuck at some term $q\,i$: this both flags up the presence of a query node in the decision tree, and allows us to explore the subsequent behaviour under both possible responses to this query.

The core of our definition is couched in terms of abstract machine configurations. We write Conf_q for the set of λ_h configurations possibly involving q (but no other free variables). We write $a \simeq b$ for Kleene equality: either both a and b are undefined or both are defined and $a = b$.

It is convenient to define the timed tree and then extract the untimed one from it:

Definition 5.4. (i) Define $\mathcal{T} : \mathsf{Conf}_q \to \mathsf{Addr} \rightharpoonup (\mathsf{Lab} \times \mathbb{N})$ to be the minimal family of partial functions satisfying the following equations:

$$\begin{aligned}
\mathcal{T}(\langle \mathbf{return}\ W \mid \gamma \mid [] \rangle)\,[] &= (!b, 0), && \text{if } [\![W]\!]\gamma = b \\
\mathcal{T}(\langle z\ V \mid \gamma \mid \kappa \rangle)\,[] &= (?[\![V]\!]\gamma, 0), && \text{if } \gamma(z) = q \\
\mathcal{T}(\langle z\ V \mid \gamma \mid \kappa \rangle)\,(b :: bs) &\simeq \mathcal{T}(\langle \mathbf{return}\ b \mid \gamma \mid \kappa \rangle)\,bs, && \text{if } \gamma(z) = q \\
\mathcal{T}(\langle M \mid \gamma \mid \kappa \rangle)\,bs &\simeq \mathsf{inc}\,(\mathcal{T}(\langle M' \mid \gamma' \mid \kappa' \rangle)\,bs), && \text{if } \langle M \mid \gamma \mid \kappa \rangle \longrightarrow \langle M' \mid \gamma' \mid \kappa' \rangle
\end{aligned}$$

Here $\mathsf{inc}(\ell, s) = (\ell, s + 1)$, and in all of the above equations $\gamma(q) = \gamma'(q) = q$. Clearly $\mathcal{T}(C)$ is a timed decision tree for any $C \in \mathsf{Conf}_q$.

(ii) The timed decision tree of a computation term is obtained by placing it in the initial configuration: $\mathcal{T}(M) := \mathcal{T}(\langle M, \emptyset[q \mapsto q], \kappa_0\rangle)$.
(iii) The timed decision tree of a closed value P : Predicate is $\mathcal{T}(P\,q)$. Since q plays the role of a dummy argument, we will usually omit it and write $\mathcal{T}(P)$ for $\mathcal{T}(P\,q)$.
(iv) The untimed decision tree $\mathcal{U}(P)$ is obtained from $\mathcal{T}(P)$ via first projection: $\mathcal{U}(P) = \mathsf{labs}(\mathcal{T}(P))$.

If the execution of a configuration C runs forever or gets stuck at an unhandled operation, then $\mathcal{T}(C)(bs)$ will be undefined for all bs. Although this is admitted by our definition of decision tree, we wish to exclude such behaviours for the terms we accept as valid predicates. Specifically, we frame the following definition:

Definition 5.5. A decision tree τ is an *n-predicate tree* if it satisfies the following:

- For every query $?k$ appearing in τ, we have $k \in \mathbb{N}_n$.
- Every query node has both children present:

$$\forall bs \in \mathsf{Addr},\ k \in \mathbb{N}_n,\ b \in \mathbb{B}.\ \tau(bs) = ?k \Rightarrow bs +\!\!+ [b] \in dom(\tau)$$

- All paths in τ are finite (so every maximal path terminates in an answer node).

A closed term P : Predicate is a *(syntactic) n-predicate* if $\mathcal{U}(P)$ is an n-predicate tree.

If τ is an n-predicate tree, clearly any semantic n-point π gives rise to a path $b_0 b_1 \ldots$ through τ, given inductively by:

$$\forall j.\ \text{if } \tau(b_0 \ldots b_{j-1}) = ?k_j \text{ then } b_j = \pi(k_j)$$

This path will terminate at some answer node $b_0 b_1 \ldots b_{r-1}$ of τ, and we may write $\tau \bullet \pi \in \mathbb{B}$ for the answer at this leaf.

Proposition 5.6. *If P is an n-predicate and Q is an n-point, then $P\,Q \rightsquigarrow^*$ **return** b where $b = \mathcal{U}(P) \bullet [\![Q]\!]$.*

Proof. By interleaving the computation for the relevant path through $\mathcal{U}(P)$ with computations for queries to Q, and appealing to the correspondence between the small-step reduction and abstract machine semantics. We omit the routine details. □

It is thus natural to define the *denotation* of an n-predicate P to be the semantic n-predicate $[\![P]\!]$ given by $[\![P]\!](\pi) = \mathcal{U}(P) \bullet \pi$.

As mentioned earlier, we shall also be interested in a more constrained class of trees and predicates:

Definition 5.7 (n-standard trees and predicates). An n-predicate tree τ is said to be n-standard if the following hold:

- The domain of τ is precisely Addr_n, the set of bit vectors of length $\leq n$.
- There are no repeated queries along any path in τ:

$$\forall bs, bs' \in dom(\tau),\ k \in \mathbb{N}_n.\ bs \sqsubseteq bs' \wedge \tau(bs) = \tau(bs') = ?k \Rightarrow bs = bs'$$

A timed decision tree τ is n-standard if its underlying untimed decision tree $(bs \mapsto \tau(bs).1)$ is so. An n-predicate P is n-standard if $\mathcal{T}(P)$ is n-standard.

Clearly, in an n-standard tree, each of the n queries $?0, \ldots, ?(n-1)$ appears exactly once on the path to any leaf, and there are 2^n leaves, all of them answer nodes.

5.3 Specification of Counting Programs

We can now specify what it means for a program K : Predicate $\to$ Nat to implement counting.

Definition 5.8. (i) The *count* of a semantic n-predicate Π, written $\sharp\Pi$, is simply the number of semantic n-points $\pi \in \mathbb{B}^n$ for which $\Pi(\pi) = \mathsf{true}$.

(ii) If P is any n-predicate, we say that K *correctly counts* P if $K\ P \rightsquigarrow^* \mathbf{return}\ m$, where $m = \sharp[\![P]\!]$.

This definition gives us the flexibility to talk about counting programs that operate on various classes of predicates, allowing us to state our results in their strongest natural form. On the positive side, we shall shortly see that there is a single 'efficient' program in λ_{h} that correctly counts all n-standard λ_h predicates for every n; in Section 7.1 we improve this to one that correctly counts *all* n-predicates of λ_h. On the negative side, we shall show that an n-indexed family of counting programs written in λ_{b}, even if only required to work correctly on n-standard λ_b predicates, can never compete with our λ_{h} program for asymptotic efficiency even in the most favourable cases.

5.4 Efficient Generic Count with Effects

We now present the simplest version of our effectful implementation of counting: one that works on n-standard predicates.

Our program uses a variation of the handler for nondeterministic computation that we gave in Section 2. The main idea is to implement points as 'nondeterministic computations' using the Branch operation such that the handler may respond to every query twice, by invoking the provided resumption with true and subsequently false. The key insight is that the resumption restarts computation at the invocation site of Branch, which means that prior computation need not be repeated. In other words, the resumption ensures that common portions of computations prior to any query are shared between both branches.

We assert that Branch : Unit $\to$ Bool $\in \Sigma$ is a distinguished operation that may not be handled in the definition of any input predicate (it has to be forwarded according to the default convention). The algorithm is then as follows.

$$
\begin{array}{l}
\mathsf{effcount} : ((\mathsf{Nat} \to \mathsf{Bool}) \to \mathsf{Bool}) \to \mathsf{Nat} \\
\mathsf{effcount}\ \mathit{pred} := \mathbf{handle}\ \mathit{pred}\ (\lambda\_.\mathbf{do}\ \mathsf{Branch}\ \langle\rangle)\ \mathbf{with} \\
\qquad\qquad \mathbf{val}\ x \qquad\quad \mapsto\ \mathbf{if}\ x\ \mathbf{then return}\ 1\ \mathbf{else return}\ 0 \\
\qquad\qquad \mathsf{Branch}\ \langle\rangle\ r\ \mapsto\ \mathbf{let}\ x_{\mathsf{true}} \leftarrow r\,\mathsf{true}\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\quad \mathbf{let}\ x_{\mathsf{false}} \leftarrow r\,\mathsf{false}\ \mathbf{in}\ x_{\mathsf{true}} + x_{\mathsf{false}}
\end{array}
$$

The handler applies predicate *pred* to a single 'generic point' defined using Branch. The boolean return value is interpreted as a single solution, whilst Branch is interpreted by alternately supplying true and false to the resumption and summing the results. The sharing enabled by the use of the resumption is exactly the 'magic' we need to make it possible to implement generic count more efficiently in λ_{h} than in λ_{b}. A curious feature of effcount is that it works for all n-standard predicates without having to know the value of n. This is because the generic point ($\lambda\_.\mathbf{do}$ Branch $\langle\rangle$) informally serves as a 'superposition' of all possible points.

We may now articulate the crucial correctness and efficiency properties of effcount.

Theorem 5.9. *The following hold for any $n \in \mathbb{N}$ and any n-standard predicate P of λ_{h}:*

(1) effcount *correctly counts P.*
(2) *The number of machine steps required to evaluate* effcount P *is*

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

Proof Outline. Suppose $bs \in \mathsf{Addr}_n$, with $|bs| = j$. From the construction of $\mathcal{T}(P)$, one may easily read off a configuration C_{bs} whose execution is expected to compute the count for the subtree below node bs, and we can explicitly describe the form C_{bs} will have. We write $\mathsf{Hyp}(bs)$ for the claim that C_{bs} correctly counts this subtree, and does so within the following number of steps:

$$\left(\sum_{bs' \in \mathsf{Addr}_n,\ bs' \sqsupset bs} \mathsf{steps}(\mathcal{T}(P))(bs') \right) + 9 * (2^{n-j} - 1) + 2 * 2^{n-j}$$

The $9 * (2^{n-j} - 1)$ expression is the number of machine steps contributed by the Branch-case inside the handler, whilst the $2 * 2^{n-j}$ expression is the number of machine steps contributed by the **val**-case. We prove $\mathsf{Hyp}(bs)$ by a laborious but routine downwards induction on the length of bs. The proof combines counting of explicit machine steps with 'oracular' appeals to the assumed behaviour of P as modelled by $\mathcal{T}(P)$. Once $\mathsf{Hyp}([])$ is established, both halves of the theorem follow easily. Full details are given in Appendix C. □

The above formula can clearly be simplified for certain reasonable classes of predicates. For instance, suppose we fix some constant $c \in \mathbb{N}$, and let $\mathcal{P}_{n,c}$ be the class of all n-standard predicates P for which all the edge times $\mathsf{steps}(\mathcal{T}(P))(bs)$ are bounded by c. (Clearly, many reasonable predicates will belong to $\mathcal{P}_{n,c}$ for some modest value of c.) Since the number of sequences bs in question is less than 2^{n+1}, we may read off from the above formula that for predicates in $\mathcal{P}_{n,c}$, the runtime of effcount is $O(c2^n)$.

Alternatively, should we wish to use the finer-grained cost model that assigns an $O(\log |\gamma|)$ runtime to each abstract machine step (see Section 4.3), we may note that any environment γ arising in the computation contains at most n entries introduced by the let-bindings in effcount, and (if $P \in \mathcal{P}_{n,c}$) at most $O(cn)$ entries introduced by P. Thus, the time for each step in the computation remains $O(\log c + \log n)$, and the total runtime for effcount is $O(c2^n(\log c + \log n))$.

One might also ask about the execution time for an implementation of λ_{h} that performs genuine copying of continuations, as in systems such as MLton [2020]. As MLton copies the entire continuation (stack), whose size is $O(n)$, at each of the 2^n branches, continuation copying alone takes time $O(n2^n)$ and the effectful implementation offers no performance benefit (Table 2). More refined implementations [Farvardin and Reppy 2020; Flatt and Dybvig 2020] that are able to take advantage of delimited control operators or sharing in copies of the stack can bring the complexity of continuation copying back down to $O(2^n)$.

Finally, one might consider another dimension of cost, namely the space used by effcount. Consider a class $\mathcal{Q}_{n,c,d}$ of n-standard predicates P for which the edge times in $\mathcal{T}(P)$ never exceed c and the sizes of pure continuations never exceed d. If we consider any $P \in \mathcal{Q}_{n,c,d}$ then the total number of environment entries is bounded by cn, taking up space $O(cn(\log cn))$. We must also account for the pure continuations. There are n of these, each taking at most d space. Thus the total space is $O(n(d + c(\log c + \log n)))$.

6 PURE GENERIC COUNT: A LOWER BOUND

We have shown that there is an implementation of generic count in λ_{h} with a runtime bound of $O(2^n)$ for certain well-behaved predicates. We now prove that no implementation in λ_{b} can match this: in fact, we establish a lower bound of $\Omega(n2^n)$ for the runtime of any counting program on *any* n-standard predicate. This mathematically rigorous characterisation of the efficiency gap between languages with and without first-class control constructs is the central contribution of the paper.

One might ask at this point whether the claimed lower bound could not be obviated by means of some known continuation passing style (CPS) or monadic transform of effect handlers [Hillerström et al. 2017; Leijen 2017]. This can indeed be done, but only by dint of changing the type of our

predicates P — which, as noted in the introduction, would defeat the purpose of our enquiry. Our intention is precisely to investigate the relative power of various languages for manipulating predicates that are given to us in a certain way which we do not have the luxury of choosing.

To get a feel for the issues that our proof must address, let us consider how one might construct a counting program in λ_b. The naïve approach, of course, would be simply to apply the given predicate P to all 2^n possible n-points in turn, keeping a count of those on which P yields true. It is a routine exercise to implement this approach in λ_b, yielding (parametrically in n) a program

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

Since the evaluation of an n-standard predicate on an individual n-point must clearly take time $\Omega(n)$, we have that the evaluation of $\mathsf{naivecount}_n$ on any n-standard predicate P must take time $\Omega(n2^n)$. If P is not n-standard, the $\Omega(n)$ lower bound need not apply, but we may still say that the evaluation of $\mathsf{naivecount}_n$ on *any* predicate P (at level n) must take time $\Omega(2^n)$.

One might at first suppose that these properties are inevitable for any implementation of generic count within λ_b, or indeed any purely functional language: surely, the only way to learn something about the behaviour of P on every possible n-point is to apply P to each of these points in turn? It turns out, however, that the $\Omega(2^n)$ lower bound can sometimes be circumvented by implementations that cleverly exploit *nesting* of calls to P. The germ of the idea may be illustrated within λ_b itself. Suppose that we first construct some program

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

which, given a predicate P, returns some n-point Q such that $P\ Q$ evaluates to true, if such a point exists, and any point at all if no such point exists. (In other words, $\mathsf{bestshot}_n$ embodies Hilbert's choice operator ε on predicates.) It is once again routine to construct such a program by naïve means; and we may moreover assume that for any P, the evaluation of $\mathsf{bestshot}_n\ P$ takes only constant time, all the real work being deferred until the argument of type Nat_n is supplied.

Now consider the following program:

$$\mathsf{lazycount}_n := \lambda \mathit{pred}.\ \textbf{if}\ \mathit{pred}\ (\mathsf{bestshot}_n\ \mathit{pred})\ \textbf{then}\ \mathsf{naivecount}_n\ \mathit{pred}\ \textbf{else return}\ 0$$

Here the term $\mathit{pred}\ (\mathsf{bestshot}_n\ \mathit{pred})$ serves to test whether there exists an n-point satisfying pred: if there is not, our count program may return 0 straightaway. It is thus clear that $\mathsf{lazycount}_n$ is a correct implementation of generic count, and also that if pred is the predicate $\lambda q.\mathsf{false}$ then $\mathsf{lazycount}_n\ \mathit{pred}$ returns 0 within $O(1)$ time, thus violating the $\Omega(2^n)$ lower bound suggested above.

This might seem like a footling point, as $\mathsf{lazycount}_n$ offers this efficiency gain *only* on (certain implementations of) the constantly false predicate. However, it turns out that by a recursive application of this nesting trick, we may arrive at a generic count program that spectacularly defies the $\Omega(2^n)$ lower bound for an interesting class of (non-n-standard) predicates, and indeed proves quite viable for counting solutions to 'n-queens' and similar problems. We shall refer to this program as BergerCount, as it is modelled largely on Berger's PCF implementation of the so-called *fan functional* [Berger 1990; Longley and Normann 2015]. This program is of interest in its own right and is briefly presented in Appendix D. It actually requires a mild extension of λ_b with a 'memoisation' primitive to achieve the effect of call-by-need evaluation; but such a language can still be seen as purely 'functional' in the same sense as Haskell.

In the meantime, however, the moral is that the use of *nesting* can lead to surprising phenomena which sometimes defy intuition (Escardó [2007] gives some striking further examples). What we now wish to show is that for *n-standard* predicates, the naïve lower bound of $\Omega(n2^n)$ cannot in fact be circumvented. The example of BergerCount both highlights the need for a rigorous proof of this and tells us that such a proof will need to pay particular attention to the possibility of nesting.

We now proceed to the proof itself. We here present the argument in the basic setting of λ_b; later we will see how a more delicate argument applies to languages with mutable state (Section 7.3).

As a first step, we note that where lower bounds are concerned, it will suffice to work with the small-step operational semantics of λ_b rather than the more elaborate abstract machine model employed in Section 4.1. This is because, as observed in Section 4.1, there is a tight correspondence between these two execution models such that for the evaluation of any closed term, the number of abstract machine steps is always at least the number of small-step reductions. Thus, if we are able to show that the number of small-step reductions for any generic program program in λ_b on any n-standard predicate is $\Omega(n2^n)$, this will establish the desired lower bound on the runtime.

Let us suppose, then, that K is a program of λ_b that correctly counts all n-standard predicates of λ_b for some specific n. We now establish a key lemma, which vindicates the naïve intuition that if P is n-standard, the only way for K to discover the correct value for $\sharp[\![P]\!]$ is to perform 2^n separate applications $P\ Q$ (allowing for the possibility that these applications need not be performed 'in turn' but might be nested in some complex way).

Lemma 6.1 (No shortcuts). *Suppose K correctly counts all n-standard predicates of λ_b. If P is an n-standard predicate, then K applies P to at least 2^n distinct n-points. More formally, for any of the 2^n possible semantic n-points $\pi : \mathbb{N}_n \to \mathbb{B}$, there is a term $\mathcal{E}[P\ Q]$ appearing in the small-step reduction of $K\ P$ such that Q is an n-point and $[\![Q]\!] = \pi$.*

Proof. Suppose for a contradiction that π is some semantic n-point such that no application $P\ Q$ with $[\![Q]\!] = \pi$ ever arises in the course of computing $K\ P$. Let τ be the untimed decision tree for P. Let l be the maximal path through τ associated with π: that is, the one we construct by responding to each query $?k$ with $\pi(k)$. Then l is a leaf node such that $\tau(l) = !(\tau \bullet \pi)$. We now let τ' be the tree obtained from τ by simply negating this answer value at l.

It is a simple matter to construct a λ_b n-standard predicate P' whose decision tree is τ'. This may be done just by mirroring the structure of τ' by nested **if** statements; we omit the easy details.

Since the numbers of true-leaves in τ and τ' differ by 1, it is clear that if K indeed correctly counts all n-standard predicates, then the values returned by $K\ P$ and $K\ P'$ will have an absolute difference of 1. On the other hand, we shall argue that if the computation of $K\ P$ never actually 'visits' the leaf l in question, then K will be unable to detect any difference between P and P'.

The situation is reminiscent of Milner's *context lemma* [Milner 1977], which (loosely) says that essentially the only way to observe a difference between two programs is to apply them to some argument on which they differ. Traditional proofs of the context lemma reason by induction on length of reduction sequences, and our present proof is closely modelled on these.

We shall make frequent use of term contexts $M[-]$ with a hole of type Predicate (which may appear zero, one or more times in M) in order to highlight particular occurrences of P within a term. The following definition enables us to talk about computations that avoid the critical point π:

Definition 6.2 (Safe terms). If $M[-]$ is such a context of ground type, let us say $M[-]$ is *safe* if

- $M[P]$ is closed, and $M[P] \rightsquigarrow^*$ **return** W for some closed ground type value W;
- For any term $\mathcal{E}[P\ Q]$ appearing in the reduction of $M[P]$, where the applicand P in $P\ Q$ is a residual of one of the abstracted occurrences in $M[P]$, we have that $[\![Q]\!] \neq \pi$.

We may express this as '$M[P]$ is safe' when it is clear which occurrences of P we intend to abstract.

For example, our current hypotheses imply that $K\ P$ is safe (formally, $K'[-] := K\ -$ is safe).

We may now prove the following:

Lemma 6.3. *(i) Suppose $Q[-]$: Point and k : Nat are values such that $Q[P]\ k$ is safe, and suppose $Q[P]\ k \rightsquigarrow^m$ **return** b where $m \in \mathbb{N}$. Then also $Q[P']\ k \rightsquigarrow^*$ **return** b.*

*(ii) Suppose $P\ Q[P]$ is safe and $P\ Q[P] \rightsquigarrow^m$ **return** b. Then also $P'\ Q[P'] \rightsquigarrow^*$ **return** b.*

We prove these claims by simultaneous induction on the computation length m. Both claims are vacuous when $m = 0$ as neither $Q[P]\ k$ nor $P\ Q[P]$ is a **return** term. We therefore assume $m > 0$ where both claims hold for all $m' < m$.

(i) Let p : Predicate be a distinguished free variable, and consider the behaviour of $Q[p]\ k$. If this reduces to a value **return** W, then also $Q[P]\ k \rightsquigarrow^* \mathbf{return}\ W$, whence $W = b$ and also $Q[P']\ k \rightsquigarrow \mathbf{return}\ b$ as required. Otherwise, the reduction of $Q[p]\ k$ will get stuck at some term $M_0 = \mathcal{E}_0[p\ Q_0[p], p]$. Here the first hole in $\mathcal{E}_0[-,-]$ is in the evaluation position, and the second hole abstracts all remaining occurrences of p within M_0. We may also assume that $Q_0[-]$ abstracts all occurrences of p in $Q_0[p]$.

Correspondingly, the reduction of $Q[P]\ k$ will reach $\mathcal{E}_0[P\ Q_0[P], P]$ and then proceed with the embedded reduction of $P\ Q_0[P]$. Note that $P\ Q_0[P]$ will be safe because $Q[P]\ k$ is. So let us suppose that $P\ Q_0[P] \rightsquigarrow^* \mathbf{return}\ b_0$, whence $Q[P]\ k \rightsquigarrow^* \mathcal{E}_0[\mathbf{return}\ b_0, P]$.

We may now investigate the subsequent reduction behaviour of $Q[P]\ k$ by considering the reduction of $\mathcal{E}_0[\mathbf{return}\ b_0, p]$. Once again, this may reduce to a value **return** W, in which case $W = b$ and our computation is complete. Otherwise, the reduction of $\mathcal{E}_0[\mathbf{return}\ b_0, p]$ will get stuck at some $M_1 = \mathcal{E}_1[p\ Q_1[p], p]$, and we may again proceed as above.

By continuing in this way, we may analyse the reduction of $Q[P]\ k$ as follows.

$$\begin{array}{c} Q[P]\ k \rightsquigarrow^* \mathcal{E}_0[P\ Q_0[P], P] \rightsquigarrow^* \mathcal{E}_0[\mathbf{return}\ b_0, P] \rightsquigarrow^* \mathcal{E}_1[P\ Q_1[P], P] \rightsquigarrow^* \mathcal{E}_1[\mathbf{return}\ b_1, P] \\ \rightsquigarrow^* \ldots \rightsquigarrow^* \mathcal{E}_{r-1}[P\ Q_{r-1}[P], P] \rightsquigarrow^* \mathcal{E}_{r-1}[\mathbf{return}\ b_{r-1}, P] \rightsquigarrow \mathbf{return}\ b \end{array}$$

Here the terms $P\ Q_j[P]$ will be safe, and the reductions $P\ Q_j[P] \rightsquigarrow^* \mathbf{return}\ b_j$ each have length $< m$. We may therefore apply part (ii) of the induction hypothesis and conclude that also $P'\ Q_j[P'] \rightsquigarrow^* \mathbf{return}\ b_j$. Furthermore, the remaining segments of the above computation are all obtained as instantiations of 'generic' reduction sequences involving p, so these segments will remain valid if p is instantiated to P'. Reassembling everything, we have a valid reduction sequence:

$$\begin{array}{c} Q[P']\ k \rightsquigarrow^* \mathcal{E}_0[P'\ Q_0[P'], P'] \rightsquigarrow^* \mathcal{E}_0[\mathbf{return}\ b_0, P'] \rightsquigarrow^* \mathcal{E}_1[P'\ Q_1[P'], P'] \rightsquigarrow^* \mathcal{E}_1[\mathbf{return}\ b_1, P'] \\ \rightsquigarrow^* \ldots \rightsquigarrow^* \mathcal{E}_{r-1}[P'\ Q_{r-1}[P'], P'] \rightsquigarrow^* \mathcal{E}_{r-1}[\mathbf{return}\ b_{r-1}, P'] \rightsquigarrow \mathbf{return}\ b \end{array}$$

This establishes the induction step for part (i).

(ii) We may apply a similar analysis to the computation of $P\ Q[P]$ to detect the places where $Q[P]$ is applied to an argument. We do this by considering the reduction behaviour of $P\ q$, where q : Point is the distinguished variable that featured in Definition 5.4. In this way we may analyse the computation of $P\ Q[P]$ as:

$$\begin{array}{c} P\ Q[P] \rightsquigarrow^* \mathcal{E}_0[Q[P]\ k_0, Q[P]] \rightsquigarrow^* \mathcal{E}_0[\mathbf{return}\ b_0, Q[P]] \rightsquigarrow^* \mathcal{E}_1[Q[P]\ k_1, Q[P]] \rightsquigarrow^* \ldots \\ \rightsquigarrow^* \mathcal{E}_{r-1}[Q[P]\ k_{r-1}, Q[P]] \rightsquigarrow^* \mathcal{E}_{r-1}[\mathbf{return}\ b_{r-1}, Q[P]] \rightsquigarrow \mathbf{return}\ b \end{array}$$

where for each j, the first hole in $\mathcal{E}_j[-,-]$ is in evaluation position, the term $Q[P]\ k_j$ is safe, the reduction $Q[P]\ k_j \rightsquigarrow^* \mathbf{return}\ b_j$ has length $< m$, and the remaining portions of computation are instantiations of generic reductions involving q. By part (i) of the induction hypothesis we may conclude that also $Q[P']\ k_j \rightsquigarrow^* \mathbf{return}\ b_j$ for each j, and for the remaining segments of computation we may instantiate q to $Q[P']$. We thus obtain a computation exhibiting that $P\ Q[P'] \rightsquigarrow^* \mathbf{return}\ b$.

It remains to show that the applicand P may be replaced by P' here without affecting the result. The idea here is that the booleans $b_0, \ldots, b_{r-1}$ trace out a path through the decision tree for P; but since $P\ Q[P]$ is safe, we have that $[\![Q[P]]\!] \neq \pi$, and so this path does *not* lead to the critical leaf l. We now have everything we need to establish that $P'\ Q[P'] \rightsquigarrow^* \mathbf{return}\ b$ as required.

More formally, in view of the correspondence between small-step reduction and abstract machine semantics, we may readily correlate the above computation of $P\ Q[P]$ with an exploration of the path $bs = b_0 \ldots b_{r-1}$ in $\tau = \mathcal{U}(P)$, leading to a leaf with label $!b$. Since P is n-standard, this correlation shows that $r = n$, that for each j we have $\tau(b_0 \ldots b_{j-1}) = ?k_j$, and that $\{k_0, \ldots, k_{r-1}\} = \{0, \ldots, n-1\}$. Furthermore, we have already ascertained that the values of $Q[P]$ and $Q[P']$ at k_j are both b_j, whence

$[\![Q[P]]\!] = [\![Q[P']]\!] = \pi'$ where $\pi'(k_j) = b_j$ for all j. But $P\ Q[P]$ is safe, so in particular $\pi' = [\![Q[P]]\!] \neq \pi$. We therefore also have $\tau'(b_0 \ldots b_{j-1}) = ?k_j$ for each $j \leq r$ and $\tau'(b_0 \ldots b_{r-1}) = b$. Since $\tau' = \mathcal{U}(P')$ and $[\![Q[P']]\!] = \pi'$, we may conclude by Proposition 5.6 that $P'\ Q[P'] \rightsquigarrow^*$ **return** b. This completes the proof of Lemma 6.3.

To finish off the proof of Lemma 6.1, we apply the same analysis one last time to the reduction of $K\ P$ itself. This will have the form

$$\begin{aligned} K\ P &\rightsquigarrow^*\ \mathcal{E}_0[P\ Q_0[P], P] \rightsquigarrow^* \mathcal{E}_0[\mathbf{return}\ b_0, P] \rightsquigarrow^*\ \ldots \\ &\rightsquigarrow^*\ \mathcal{E}_{r-1}[P\ Q_{r-1}[P], P] \rightsquigarrow^* \mathcal{E}_{r-1}[\mathbf{return}\ b_{r-1}, P] \rightsquigarrow^*\ \mathbf{return}\ c \end{aligned}$$

where, by hypothesis, each $P\ Q_j[P]$ is safe. Using Lemma 6.3 we may replace each subcomputation $P\ Q_j[P] \rightsquigarrow^*$ **return** b_j with $P'\ Q_j[P'] \rightsquigarrow^*$ **return** b_j, and so construct a computation exhibiting that $K\ P' \rightsquigarrow^*$ **return** c.

This gives our contradiction, as the values of $K\ P$ and $K\ P'$ are supposed to differ by 1. □

Corollary 6.4. *Suppose K and P are as in Lemma 6.1. For any semantic n-point π and any natural number $k < n$, the reduction sequence for $K\ P$ contains a term $\mathcal{F}[Q\ k]$, where $\mathcal{F}$ is an evaluation context and $[\![Q]\!] = \pi$.*

Proof. Suppose $\pi \in \mathbb{B}^n$. By Lemma 6.1, the computation of $K\ P$ contains some $\mathcal{E}[P\ Q]$ where $[\![Q]\!] = \pi$, and the above analysis of the computation of $P\ Q$ shows that it contains a term $\mathcal{E}'[Q\ k]$ for each $k < n$. The corollary follows, taking $\mathcal{F}[-] := \mathcal{E}[\mathcal{E}'[-]]$. □

This gives our desired lower bound. Since our n-points Q are values, it is clearly impossible that $\mathcal{F}[Q\ k] = \mathcal{F}'[Q'\ k']$ (where $\mathcal{F}, \mathcal{F}'$ are evaluation contexts) unless $Q = Q'$ and $k = k'$. We may therefore read off π from $\mathcal{F}[Q\ k]$ as $[\![Q]\!]$. There are thus at least $n2^n$ distinct terms in the reduction sequence for $K\ P$, so the reduction has length $\geq n2^n$. We have thus proved:

Theorem 6.5. *If K is a λ_{b} program that correctly counts all n-standard λ_{b} predicates, and P is any n-standard λ_{b} predicate, then the evaluation of $K\ P$ must take time $\Omega(n2^n)$.* □

Although we shall not go into details, it is not too hard to apply our proof strategy with minor adjustments to certain richer languages: for instance, an extension of λ_{b} with exceptions, or one containing the memoisation primitive required for BergerCount (Appendix D). A deeper adaptation is required for languages with state: we will return to this in Section 7.

It is worth noting where the above argument breaks down if applied to λ_{h}. In λ_{b}, in the course of computing $K\ P$, every Q to which P is applied will be a self-contained closed term denoting some specific point π. This is intuitively why we may only learn about one point at a time. In λ_{h}, this is not the case, because of the presence of operation symbols. For instance, our effcount program from Section 5.4 will apply P to the 'generic point' $\lambda\_.$**do** Branch $\langle\rangle$. Thus, for example, in our treatment of Lemma 6.3(i), it need no longer be the case that the reduction of $Q[p]\ k$ either yields a value or gets stuck at some $\mathcal{E}_0[p\ Q_0[p], p]$: a third possibility is that it gets stuck at some invocation of ℓ, so that control will then pass to the effect handler.

7 EXTENSIONS AND VARIATIONS

Our complexity result is robust in that it continues to hold in more general settings. We outline here how it generalises: beyond n-standard predicates, from generic count to generic search, and from pure λ_{b} to stateful λ_{s}.

7.1 Beyond n-Standard Predicates

The n-standard restriction on predicates serves to make the efficiency phenomenon stand out as clearly as possible. However, we can relax the restriction by tweaking effcount to handle repeated

queries and missing queries. The trade off is that the analysis of effcount becomes more involved. The key to relaxing the n-standard restriction is the use of state to keep track of which queries have been computed. We can give stateful implementations of effcount without changing its type signature by using *parameter-passing* [Kammar et al. 2013; Pretnar 2015] to internalise state within a handler. Parameter-passing abstracts every handler clause such that the current state is supplied before the evaluation of a clause continues and the state is threaded through resumptions: a resumption becomes a two-argument curried function $r : B \to S \to D$, where the first argument of type B is the return type of the operation and the second argument is the updated state of type S.

Repeated queries. We can generalise effcount to handle repeated queries by memoising previous answers. First, we generalise the type of Branch such that it carries an index of a query.

$$\mathsf{Branch} : \mathsf{Nat} \to \mathsf{Bool}$$

We assume a family of natural number to boolean maps, Map_n with the following interface.

$$\begin{aligned} \mathsf{empty}_n &: \mathsf{Map}_n \\ \mathsf{add}_n &: (\mathsf{Nat}_n \times \mathsf{Bool}) \to \mathsf{Map}_n \to \mathsf{Map}_n \\ \mathsf{lookup}_n &: \mathsf{Nat}_n \to \mathsf{Map}_n \to (\mathsf{Unit} + \mathsf{Bool}) \end{aligned}$$

Invoking lookup i *map* returns **inl** $\langle\rangle$ if i is not present in *map*, and **inr** *ans* if i is associated by *map* with the value *ans* : Bool. Allowing ourselves a few extra constant-time arithmetic operations, we can realise suitable maps in λ_{b} such that the time complexity of add_n and lookup_n is $O(\log n)$ [Okasaki 1999]. We can then use parameter-passing to support repeated queries as follows.

$$\begin{aligned} &\mathsf{effcount}'_n : ((\mathsf{Nat}_n \to \mathsf{Bool}) \to \mathsf{Bool}) \to \mathsf{Nat} \\ &\mathsf{effcount}'_n\ \mathit{pred} := \mathbf{let}\ h \leftarrow \mathbf{handle}\ \mathit{pred}\,(\lambda i.\mathbf{do}\ \mathsf{Branch}\ i)\ \mathbf{with} \\ &\qquad\qquad \mathbf{val}\ x \quad\quad \mapsto\ \lambda s.\mathbf{if}\ x\ \mathbf{then}\ 1\ \mathbf{else}\ 0 \\ &\qquad\qquad \mathsf{Branch}\ i\ r\ \mapsto\ \lambda s.\mathbf{case}\ \mathsf{lookup}_n\ i\ s\ \{ \\ &\qquad\qquad\qquad\qquad \mathbf{inl}\ \langle\rangle \mapsto \mathbf{let}\ x_{\mathsf{true}} \leftarrow r\ \mathsf{true}\ (\mathsf{add}_n\ \langle i, \mathsf{true}\rangle\ s)\ \mathbf{in} \\ &\qquad\qquad\qquad\qquad\qquad\quad \mathbf{let}\ x_{\mathsf{false}} \leftarrow r\ \mathsf{false}\ (\mathsf{add}_n\ \langle i, \mathsf{false}\rangle\ s)\ \mathbf{in} \\ &\qquad\qquad\qquad\qquad\qquad\quad (x_{\mathsf{true}} + x_{\mathsf{false}}); \\ &\qquad\qquad\qquad\qquad \mathbf{inr}\ x \mapsto r\ x\ s\ \} \\ &\qquad \mathbf{in}\ h\ \mathsf{empty}_n \end{aligned}$$

The state parameter s memoises query results, thus avoiding double-counting and enabling $\mathsf{effcount}'_n$ to work correctly for predicates performing the same query multiple times.

Missing queries. Similarly, we can use parameter-passing to support missing queries.

$$\begin{aligned} &\mathsf{effcount}''_n : ((\mathsf{Nat}_n \to \mathsf{Bool}) \to \mathsf{Bool}) \to \mathsf{Nat} \\ &\mathsf{effcount}''_n\ \mathit{pred} := \mathbf{let}\ h \leftarrow \mathbf{handle}\ \mathit{pred}\,(\lambda i.\mathbf{do}\ \mathsf{Branch}\ \langle\rangle)\ \mathbf{with} \\ &\qquad\qquad\qquad \mathbf{val}\ x \quad\quad \mapsto\ \lambda d.\mathbf{let}\ \mathit{result} \leftarrow \mathbf{if}\ x\ \mathbf{then}\ 1\ \mathbf{else}\ 0\ \mathbf{in}\ \mathit{result} \times 2^{n-d} \\ &\qquad\qquad\qquad \mathsf{Branch}\ \langle\rangle\ r\ \mapsto\ \lambda d.\mathbf{let}\ x_{\mathsf{true}} \leftarrow r\ \mathsf{true}\ (d+1)\ \mathbf{in} \\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{let}\ x_{\mathsf{false}} \leftarrow r\ \mathsf{false}\ (d+1)\ \mathbf{in} \\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad (x_{\mathsf{true}} + x_{\mathsf{false}}) \\ &\qquad\qquad \mathbf{in}\ h\ 0 \end{aligned}$$

The parameter d tracks the depth and the returned result is scaled by 2^{n-d} accounting for the unexplored part of the current subtree. This enables $\mathsf{effcount}''_n$ to operate correctly on predicates that inspect n points at most once. We leave it as an exercise for the reader to combine $\mathsf{effcount}'_n$ and $\mathsf{effcount}''_n$ in order to handle both repeated queries and missing queries.

7.2 From Generic Count to Generic Search

We can generalise the problem of generic counting to generic searching. The main operational difference is that a generic search procedure must materialise a list of solutions, thus its type is

$$\mathsf{search}_n : ((\mathsf{Nat}_n \to \mathsf{Bool}) \to \mathsf{Bool}) \to \mathsf{List}_{\mathsf{Nat}_n \to \mathsf{Bool}}$$

where List_A is the type of cons-lists whose elements have type A. We modify effcount to return a list of solutions rather than the number of solutions by lifting each result into a singleton list and using list concatenation instead of addition to combine partial results xs_{true} and xs_{false} as follows.

$$\begin{array}{l}
\mathsf{effsearch}_n : ((\mathsf{Nat}_n \to \mathsf{Bool}) \to \mathsf{Bool}) \to \mathsf{List}_{\mathsf{Nat}_n \to \mathsf{Bool}} \\
\mathsf{effsearch}_n\ \mathit{pred} := \mathbf{let}\ f \leftarrow \mathbf{handle}\ \mathit{pred}\ (\lambda i.\mathbf{do}\ \mathsf{Branch}\ i)\ \mathbf{with} \\
\qquad\qquad \mathbf{val}\ x \quad\ \mapsto\ \lambda q.\mathbf{if}\ x\ \mathbf{then}\ \mathsf{singleton}\ q\ \mathbf{else}\ \mathsf{nil} \\
\qquad\qquad \mathsf{Branch}\ i\ r\ \mapsto\ \lambda q.\mathbf{let}\ xs_{\mathsf{true}} \leftarrow r\ \mathsf{true}\ (\lambda j.\mathbf{if}\ i = j\ \mathbf{then}\ \mathsf{true}\ \mathbf{else}\ q\ j)\ \mathbf{in} \\
\qquad\qquad\qquad\qquad \mathbf{let}\ xs_{\mathsf{false}} \leftarrow r\ \mathsf{false}\ (\lambda j.\mathbf{if}\ i = j\ \mathbf{then}\ \mathsf{false}\ \mathbf{else}\ q\ j)\ \mathbf{in} \\
\qquad\qquad\qquad\qquad \mathsf{concat}\ \langle xs_{\mathsf{true}}, xs_{\mathsf{false}}\rangle \\
\qquad \mathbf{in}\ \mathsf{toConsList}\ (f\ (\lambda j.\bot))
\end{array}$$

The Branch operation is now parameterised by an index i. The handler is now parameterised by the current path as a point q, which is output at a leaf iff it is in the predicate. A little care is required to ensure that $\mathsf{effsearch}_n$ has runtime $O(2^n)$; naïve use of cons-list concatenation would result in $O(n2^n)$ runtime, as cons-list concatenation is linear in its first operand. In place of cons-lists we use Hughes lists [Hughes 1986], which admit constant time concatenation: $\mathsf{HList}_A := \mathsf{List}_A \to \mathsf{List}_A$. The empty Hughes list $\mathsf{nil} : \mathsf{HList}_A$ is defined as the identity function: $\mathsf{nil} := \lambda xs.xs$.

$$\begin{array}{lll}
\mathsf{singleton}_A : A \to \mathsf{HList}_A & \mathsf{concat}_A : \mathsf{HList}_A \times \mathsf{HList}_A \to \mathsf{HList}_A & \mathsf{toConsList}_A : \mathsf{HList} \to \mathsf{List}_A \\
\mathsf{singleton}_A\ x := \lambda xs.x :: xs & \mathsf{concat}_A\ f\ g := \lambda xs.g\ (f\ xs) & \mathsf{toConsList}_A\ f := f\ [\,]
\end{array}$$

We use the function toConsList to convert the final Hughes list to a standard cons-list at the end; this conversion has linear time complexity (it just conses all of the elements of the list together).

7.3 From Pure λ_{b} to Stateful λ_{s}

Mutable state is a staple ingredient of many practical programming languages. We now outline how our main lower bound result can be extended to a language with state. We will not give full details, but merely point out the respects in which our earlier treatment needs to be modified.

We have in mind an extension λ_{s} of λ_{b} with ML-style reference cells: we extend our grammar for types with a reference type ($\mathsf{Ref}\ A$), and that for computation terms with forms for creating references ($\mathbf{letref}\ x = V\ \mathbf{in}\ N$), dereferencing ($!x$), and destructive update ($x := V$), with the familiar typing rules. We also add a new kind of value, namely *locations* l^A, of type $\mathsf{Ref}\ A$. We adopt a basic Scott-Strachey [1971] model of store: a location is a natural number decorated with a type, and the execution of a stateful program allocates locations in the order $0, 1, 2, \ldots$, assigning types to them as it does so. A *store* s is a type-respecting mapping from some set of locations $\{0, \ldots, l-1\}$ to values. For the purposes of small-step operational semantics, a *configuration* will be a triple (M, l, s), where M is a computation, l is a 'location counter', and s is a store with domain $\{0, \ldots, l-1\}$. A reduction relation $\rightsquigarrow$ on configurations is defined in a familiar way (again we omit the details).

Certain aspects of our setup require care in the presence of state. For instance, there is in general no unique way to assign an (untimed) decision tree to a closed value $P : \mathsf{Predicate}_n$, since the behaviour of P on a value $q : \mathsf{Point}_n$ may depend both on the initial state when P is invoked, and on the ways in which the associated computations $q\ V \rightsquigarrow^* \mathbf{return}\ W$ modify the state. In this situation, there is not even a clear specification for what an n-count program ought to do.

The simplest way to circumvent this difficulty is to restrict attention to predicates P *within the sublanguage* λ_{b}. For such predicates, the notions of decision tree, counting and n-standardness are unproblematic. Our result will establish a runtime lower bound of $\Omega(n2^n)$ for programs $K \in \lambda_{\mathrm{s}}$

Table 1. SML/NJ: Runtime Relative to Effectful Implementation

	Queens						Integration								
	First solution			All solutions			Id	Squaring			Logistic				
Parameter	20	24	28	8	10	12	20	14	17	20	1	2	3	4	5
Naïve	–	–	–	217.74	–	–	12.89	45.04	57.80	69.86	–	–	–	–	–
Berger	11.24	15.70	–	2.06	2.86	3.64	5.18	20.62	22.37	23.46	22.51	28.97	30.14	29.30	27.94
Pruned	2.13	2.54	2.91	1.04	1.24	1.39	2.07	3.78	4.05	4.24	4.10	5.44	6.42	7.26	7.94
Bespoke	0.12	0.12	0.12	0.13	0.13	0.12									

that correctly count predicates P of this kind. On the other hand, since K itself may be stateful, we cannot exclude the possibility that $K\ P$ will apply P to a term Q that is itself stateful. Such a Q will no longer unambiguously denote a semantic point π, hence the proof of Section 6 must be adapted.

To adapt our proof to the setting of λ_s, some more machinery is needed. If K is an n-count program and P an n-standard predicate, we expect that the evaluation of $K\ P$ will feature terms $\mathcal{E}[P\ Q]$ which are then reduced to some $\mathcal{E}[\mathbf{return}\ b]$, via a reduction sequence which, modulo $\mathcal{E}[-]$, has the following form:

$$P\ Q \rightsquigarrow^* \mathcal{E}_0[Q\ k_0] \rightsquigarrow^* \mathcal{E}_0[\mathbf{return}\ b_0] \rightsquigarrow^* \cdots \rightsquigarrow^* \mathcal{E}_{n-1}[Q\ k_{n-1}] \rightsquigarrow^* \mathcal{E}_{n-1}[\mathbf{return}\ b_{n-1}] \rightsquigarrow^* \mathbf{return}\ b$$

(For notational clarity, we suppress mention of the location and store components here.) Informally we think of this as a dialogue in which control passes back and forth between P and Q. We shall refer to the portions $\mathcal{E}_j[Q\ k_j] \rightsquigarrow^* \mathcal{E}_j[\mathbf{return}\ b_j]$ of the above reduction as *Q-sections*, and to the remaining portions (including the first and the last) as *P-sections*. We refer to the totality of these P-sections and Q-sections as the *thread* arising from the given occurrence of the application $P\ Q$. An important point to note is that since Q may contain other occurrences of P, it is quite possible for the Q-sections above to contain further threads corresponding to other applications $P\ Q'$.

Since P is n-standard, we know that each thread will consist of $n+1$ P-sections separated by n Q-sections. Indeed, it is clear that this computation traces the path $b_0 \ldots b_{n-1}$ through the decision tree for P, with $k_0, \ldots, k_{n-1}$ the corresponding internal node labels. We may now, 'with hindsight', construe this as a semantic point $\pi : \mathbb{N}_n \to \mathbb{B}$ (where $\pi(k_j) = b_j$ for each j), and call it the semantic point *associated with* (the thread arising from) the application occurrence $P\ p$.

The following lemma now serves as a surrogate for Lemma 6.1:

Lemma 7.1. *Let P be an n-standard predicate. For any semantic point $\pi \in \mathbb{B}^n$, the evaluation of $K\ P$ involves an application occurrence $P\ Q$ with which π is associated.*

The proof of this lemma is not too different from that of Lemma 6.1: if π were a point with no associated thread, there would be an unvisited leaf in the decision tree, and we could manufacture an n-standard predicate P' whose tree differed from that of P only at this leaf. We can then show, by induction on length of reductions, that any portion of the evaluation of $K\ P$ can be suitably mimicked with P replaced by P'. Naturally, this idea now needs to be formulated at the level of *configurations* rather than plain terms: in the course of reducing $(K\ P, 0, [])$, we may encounter configurations (M, l, s) in which residual occurrences of P have found their way into s as well as M, so in order to replace P by P' we must abstract on all these occurrences via an evident notion of *configuration context*. With this adjustment, however, the argument of Lemma 6.1 goes through.

A further argument is then needed to show that any two threads are indeed 'disjoint' as regards their P-sections, so that there must be at least $n2^n$ steps in the overall reduction sequence.

8 EXPERIMENTS

The theoretical efficiency gap between realisations of λ_b and λ_h manifests in practice. We observe it empirically on instantiations of n-queens and exact real number integration, which can be cast as

Table 2. MLton: Runtime Relative to Effectful Implementation

	Queens						Integration								
	First solution			All solutions			Id	Squaring			Logistic				
Parameter	20	24	28	8	10	12	20	14	17	20	1	2	3	4	5
Naïve	–	–	–	17.31	–	–	1.45	4.51	5.13	5.82	–	–	–	–	–
Berger	0.52	0.66	–	0.19	0.22	0.20	0.43	2.02	1.95	1.92	2.17	3.59	4.24	4.34	4.28
Pruned	0.11	0.11	0.13	0.10	0.10	0.08	0.14	0.39	0.35	0.35	0.39	0.63	0.86	1.03	1.21
Bespoke	0.005	0.004	0.004	0.01	0.009	0.006									

generic search. Table 1 shows the speedup of using an effectful implementation of generic search over various pure implementations. We discuss the benchmarks and results in further detail below.

Methodology. We evaluated an effectful implementation of generic search against three "pure" implementations which are realisable in λ_b extended with mutable state:

- Naïve: a simple, and rather naïve, functional implementation;
- Pruned: a generic search procedure with space pruning based on Longley's technique [Longley 1999] (uses local state);
- Berger: a lazy pure functional generic search procedure based on Berger's algorithm.

Each benchmark was run 11 times. The reported figure is the median runtime ratio between the particular implementation and the baseline effectful implementation. Benchmarks that failed to terminate within a threshold (1 minute for single solution, 8 minutes for enumerations), are reported as –. The experiments were conducted in SML/NJ [2020] v110.97 64-bit with factory settings on an Intel Xeon CPU E5-1620 v2 @ 3.70GHz powered workstation running Ubuntu 16.04. The effectful implementation uses an encoding of delimited control akin to effect handlers based on top of SML/NJ's call/cc. The complete source code for the benchmarks is available at:

https://github.com/dhil/effects-for-efficiency-code

Queens. We phrase the n-queens problem as a generic search problem. As a control we include a bespoke implementation hand-optimised for the problem. We perform two experiments: finding the first solution for $n \in \{20, 24, 28\}$ and enumerating all solutions for $n \in \{8, 10, 12\}$. The speedup over the naïve implementation is dramatic, but less so over the Berger procedure. The pruned procedure is more competitive, but still slower than the baseline. Unsurprisingly, the baseline is slower than the bespoke implementation.

Exact Real Integration. The integration benchmarks are adapted from Simpson [1998]. We integrate three different functions with varying precision in the interval $[0, 1]$. For the identity function (Id) at precision 20 the speedup relative to Berger is 5.18×. For the squaring function the speedups are larger at higher precisions: at precision 14 the speedup is 3.78× over the pruned integrator, whilst it is 4.24× at precision 20. The speedups are more extreme against the naïve and Berger integrators. We also integrate the logistic map $x \mapsto 1 - 2x^2$ at a fixed precision of 15. We make the function harder to compute by iterating it up to 5 times. Between the pruned and effectful integrator the speedup ratio increases as the function becomes harder to compute.

MLton. SML/NJ is compiled into CPS, thus providing a particularly efficient implementation of call/cc. MLton [2020], a whole program compiler for SML, implements call/cc by copying the stack. We repeated our experiments using MLton 20180207. Table 2 shows the results. The effectful implementation performs much worse under MLton than SML/NJ, being surpassed in nearly every case by the pruned search procedure and in some cases by the Berger search procedure. Table 3 summarises the runtime of MLton relative to SML/NJ. Berger, Pruned, and Bespoke run between 1 and 3 times as fast with MLton compared to SML/NJ. However, the effectful implementation runs between 2 and 14 times as fast with SML/NJ compared with MLton.

Table 3. MLton: Runtime Relative to SML/NJ

	Queens						Integration								
	First solution			All solutions			Id	Squaring			Logistic				
Parameter	20	24	28	8	10	12	20	14	17	20	1	2	3	4	5
Naïve	–	–	–	0.49	–	–	0.55	0.35	0.35	0.35	–	–	–	–	–
Berger	0.62	0.64	–	0.73	0.65	0.68	0.41	0.35	0.34	0.34	0.37	0.37	0.37	0.37	0.37
Pruned	0.70	0.68	0.71	0.74	0.70	0.71	0.34	0.36	0.35	0.35	0.36	0.35	0.35	0.35	0.36
Effectful	12.87	13.99	14.90	8.00	8.60	12.19	4.93	3.53	3.95	4.20	3.80	3.00	2.62	2.46	2.37
Bespoke	0.56	0.56	0.56	0.69	0.63	0.59									

9 CONCLUSIONS AND FUTURE WORK

We presented a PCF-inspired language λ_{b} and its extension with effect handlers λ_{h}. We proved that λ_{h} supports an asymptotically more efficient implementation of generic search than any possible implementation in λ_{b}. We observed its effect in practice on several benchmarks. We also proved that our $\Omega(n2^n)$ lower bound applies to a language λ_{s} which extends λ_{b} with state.

Our positive result for λ_{h} extends to other control operators by appeal to existing results on interdefinability of handlers and other control operators [Forster et al. 2019; Piróg et al. 2019]. The result no longer applies directly if we add an effect type system to λ_{h}, as the implementation of the counting program would require a change of type for predicates to reflect the ability to perform effectful operations. In future we plan to investigate how to account for effect type systems.

We have verified that our $\Omega(n2^n)$ lower bound also applies to a language λ_{e} with (Benton-Kennedy style [Benton and Kennedy 2001]) *exceptions* and handlers. The lower bound also applies to the combined language λ_{se} with both state and exceptions — this seems to bring us close to the expressive power of real languages such as Standard ML, Java, and Python, strongly suggesting that the speedup we have discussed is unattainable in these languages.

In future work, we hope to establish the more general result that our $\Omega(n2^n)$ applies to a language with *affine effect handlers* (handlers which invoke the resumption r at most once). This would not only subsume our present results (since state and exceptions are examples of affine effects), but would also apply e.g. to a richer language with *coroutines*. However, it appears that our present methods do not immediately adapt to this more general situation, as our arguments depend at various points on an orderly nesting of subcomputations which coroutining would break.

One might object that the efficiency gap we have analysed is of merely theoretical interest, since an $\Omega(2^n)$ runtime is already 'infeasible'. We claim, however, that what we have presented is an example of a much more pervasive phenomenon, and our generic count example serves merely as a convenient way to bring this phenomenon into sharp formal focus. Suppose, for example, that our programming task was not to count all solutions to P, but to find just one of them. It is informally clear that for many kinds of predicates this would in practice be a feasible task, and also that we could still gain our factor n speedup here by working in a language with first-class control. However, such an observation appears less amenable to a clean mathematical formulation, as the runtimes in question are highly sensitive to both the particular choice of predicate and the search order employed.

ACKNOWLEDGMENTS

We would like to thank James McKinna and Maciej Piróg for insightful discussions, and Danel Ahman and the anonymous reviewers for helpful feedback and suggestions for improvement. Daniel Hillerström was supported by EPSRC grant EP/L01503X/1 and by ERC Consolidator Grant Skye (grant number 682315). Sam Lindley was supported by EPSRC grant EP/K034413/1 (From Data Types to Session Types—A Basis for Concurrency and Distribution).

REFERENCES

Andrej Bauer. 2018. What is algebraic about algebraic effects and handlers? *CoRR* abs/1807.05923 (2018).

Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *J. Log. Algebr. Meth. Program.* 84, 1 (2015), 108–123.

Jordan Bell and Brett Stevens. 2009. A survey of known results and research areas for n-queens. *Discret. Math.* 309, 1 (2009), 1–31.

Nick Benton and Andrew Kennedy. 2001. Exceptional Syntax Journal of Functional Programming. *J. Funct. Program.* 11, 4 (2001), 395–410.

Ulrich Berger. 1990. *Totale Objekte und Mengen in der Bereichstheorie.* Ph.D. Dissertation. Ludwig Maximillians-Universtität, Munich.

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2019. Abstracting algebraic effects. *PACMPL* 3, POPL (2019), 6:1–6:28.

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2020. Binders by day, labels by night: effect instances via lexically scoped handlers. *PACMPL* 4, POPL (2020), 48:1–48:29.

Richard Bird, Geraint Jones, and Oege de Moor. 1997. More haste less speed: lazy versus eager evaluation. *J. Funct. Program.* 7, 5 (1997), 541–547.

Richard S. Bird. 2006. Functional Pearl: A program to solve Sudoku. *J. Funct. Program.* 16, 6 (2006), 671–679.

Robert Cartwright and Matthias Felleisen. 1992. Observable Sequentiality and Full Abstraction. In *POPL*. ACM Press, 328–342.

Lukas Convent, Sam Lindley, Conor McBride, and Craig McLaughlin. 2020. Doo bee doo bee doo. *J. Funct. Program.* 30 (2020). To appear.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). MIT Press.

Robbie Daniels. 2016. *Efficient Generic Searches and Programming Language Expressivity.* Master's thesis. School of Informatics, the University of Edinburgh, Scotland. http://homepages.inf.ed.ac.uk/jrl/Research/Robbie_Daniels_MSc_dissertation.pdf

Olivier Danvy and Andrzej Filinski. 1990. Abstracting Control. In *LISP and Functional Programming*. ACM, 151–160.

Stephen Dolan, Leo White, KC Sivaramakrishnan, Jeremy Yallop, and Anil Madhavapeddy. 2015. Effective Concurrency through Algebraic Effects. OCaml Workshop.

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

Kavon Farvardin and John H. Reppy. 2020. From folklore to fact: comparing implementations of stacks and continuations. In *PLDI*. ACM, 75–90.

Matthias Felleisen. 1987. *The Calculi of Lambda-nu-cs Conversion: A Syntactic Theory of Control and State in Imperative Higher-order Programming Languages.* Ph.D. Dissertation. Indianapolis, IN, USA. AAI8727494.

Matthias Felleisen. 1988. The Theory and Practice of First-Class Prompts. In *POPL*. ACM Press, 180–190.

Matthias Felleisen. 1991. On the expressive power of programming languages. *Sci. Comput. Prog.* 17, 1–3 (1991), 35–75.

Matthias Felleisen and Daniel P. Friedman. 1987. Control Operators, the SECD-machine, and the λ-Calculus. In *The Proceedings of the Conference on Formal Description of Programming Concepts III, Ebberup, Denmark*. Elsevier, 193–217.

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *PLDI*. ACM, 237–247.

Matthew Flatt and R. Kent Dybvig. 2020. Compiler and runtime support for continuation marks. In *PLDI*. ACM, 45–58.

Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. 2019. On the expressive power of user-defined effects: Effect handlers, monadic reflection, delimited control. *J. Funct. Program.* 29 (2019), e15.

Daniel Hillerström and Sam Lindley. 2016. Liberating effects with rows and handlers. In *TyDe@ICFP*. ACM, 15–27.

Daniel Hillerström and Sam Lindley. 2018. Shallow Effect Handlers. In *APLAS (Lecture Notes in Computer Science, Vol. 11275)*. Springer, 415–435.

Daniel Hillerström, Sam Lindley, and Robert Atkey. 2020a. Effect handlers via generalised continuations. *J. Funct. Program.* 30 (2020), e5.

Daniel Hillerström, Sam Lindley, Robert Atkey, and K. C. Sivaramakrishnan. 2017. Continuation Passing Style for Effect Handlers. In *FSCD (LIPIcs, Vol. 84)*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 18:1–18:19.

Daniel Hillerström, Sam Lindley, and John Longley. 2020b. Effects for Efficiency: Asymptotic Speedup with First-Class Control (extended version). arXiv:2007.00605 [cs.PL]

John Hughes. 1986. A Novel Representation of Lists and its Application to the Function "reverse". *Inf. Process. Lett.* 22, 3 (1986), 141–144.

Neil Jones. 2001. The expressive power of higher-order types, or, life without CONS. *J. Funct. Program.* 11 (2001), 5–94.

Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in action. In *ICFP*. ACM, 145–158.

Oleg Kiselyov, Amr Sabry, and Cameron Swords. 2013. Extensible effects: an alternative to monad transformers. In *Haskell*. ACM, 59–70.

Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. 2005. Backtracking, Interleaving, and Terminating Monad Transformers: (Functional Pearl). (2005), 192–203.

Donald Knuth. 1997. *The Art of Computer Programming, Volume 1: Fundamental Algorithms (third edition).* Addison-Wesley.

Daan Leijen. 2017. Type directed compilation of row-typed algebraic effects. In *POPL*. ACM, 486–499.

Paul Blain Levy, John Power, and Hayo Thielecke. 2003. Modelling environments in call-by-value programming languages. *Inf. Comput.* 185, 2 (2003), 182–210.

Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do be do be do. In *POPL*. ACM, 500–514.

John Longley. 1999. When is a functional program not a functional program?. In *ICFP*. ACM, 1–7.

John Longley. 2018. The recursion hierarchy for PCF is strict. *Logical Methods in Comput. Sci.* 14, 3:8 (2018), 1–51.

John Longley. 2019. Bar recursion is not computable via iteration. *Computability* 8, 2 (2019), 119–153.

John Longley and Dag Normann. 2015. *Higher-Order Computability.* Springer.

Robin Milner. 1977. Fully Abstract Models of Typed λ-Calculi. *Theor. Comput. Sci.* 4, 1 (1977), 1–22.

MLton. 2020. MLton website. http://www.mlton.org

Eugenio Moggi. 1991. Notions of Computation and Monads. *Inf. Comput.* 93, 1 (1991), 55–92.

Chris Okasaki. 1999. *Purely functional data structures.* Cambridge University Press.

Nicholas Pippenger. 1996. Pure versus impure Lisp. In *POPL*. ACM, 104–109.

Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2019. Typed Equivalence of Effect Handlers and Delimited Control. In *FSCD (LIPIcs, Vol. 131)*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 30:1–30:16.

Gordon Plotkin. 1977. LCF considered as a programming language. *Theor. Comput. Sci.* 5, 3 (1977), 223–255.

Gordon D. Plotkin and John Power. 2001. Adequacy for Algebraic Effects. In *FoSSaCS (Lecture Notes in Computer Science, Vol. 2030)*. Springer, 1–24.

Gordon D. Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Logical Methods in Computer Science* 9, 4 (2013).

Matija Pretnar. 2015. An Introduction to Algebraic Effects and Handlers. *Electr. Notes Theor. Comput. Sci.* 319 (2015), 19–35. Invited tutorial paper.

Dana Scott and Christopher Strachey. 1971. *Proceedings of the Symposium on Computers and Automata* 21 (1971).

Alex K. Simpson. 1998. Lazy Functional Algorithms for Exact Real Functionals. In *MFCS (Lecture Notes in Computer Science, Vol. 1450)*. Springer, 456–464.

SML/NJ. 2020. SML/NJ website. http://www.smlnj.org

Michael Sperber, Kent R. Dybvig, Matthew Flatt, Anton van Stratten, Robby Bruce Findler, and Jacob Matthews. 2009. Revised^6 Report on the Algorithmic Language Scheme. *J. Funct. Program.* 19, S1 (2009), 1–301.