

Asymptotic Speedup via Effect Handlers

DANIEL HILLERSTRÖM

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

SAM LINDLEY

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

JOHN LONGLEY

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

Abstract

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

To our knowledge, these results are the first of their kind for control operators.

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}' ?
2. *Complexity*: Are there operations programmable in \mathcal{L} with some asymptotic runtime bound (e.g. $\mathcal{O}(n^2)$) 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 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: that is, 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 essentially because a program may interact with a given function only in ways prescribed by the language (for instance, by applying it to an argument), and typically has no access to the concrete representation of the function at the machine level.

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 $\mathbb{N}_\perp \times \mathbb{N}_\perp \rightarrow \mathbb{N}_\perp$); 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 recursion but not by iteration (Longley, 2019). Much of this territory, including the mathematical theory of

some of the natural definitions of computability in a higher-order setting, 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 $\mathcal{O}(n)$ 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 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.

Our purpose in this 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 $P q = \text{true}$. We shall consider boolean vectors of any length to be represented by the type $\text{Nat} \rightarrow \text{Bool}$; thus for each n , we are asking for an implementation of a certain third-order operation

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

Naturally, we do not expect such a generic operation to compete in efficiency with a bespoke counting operation for some specific predicate, but it is nonetheless interesting to ask how efficient it is possible to be with this more modular approach.

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 inspired by Berger (1990) achieves the effect of ‘pruned search’ where the predicate allows it (serving as a warning that counterintuitive phenomena can arise in this territory). This implementation is of interest in its own right and will be discussed in Section 7. Nonetheless, under a certain natural condition on P (namely that it must inspect all n components of the given vector before returning), both the above approaches will have $\Omega(n2^n)$ runtime.

What we will show is that in a typical call-by-value functional language without advanced control features, one cannot improve on this: *any* implementation of count_n must necessarily take time $\Omega(n2^n)$ on predicates P of a certain kind. Furthermore, we will show that the same lower bound also applies to a richer language supporting *affine effect handlers*, which suffices for the encoding of exceptions, local state, coroutines, and single-shot continuations. On the other hand, if we move to language with general effect handlers, it becomes possible to bring the runtime down to $\mathcal{O}(2^n)$: an asymptotic gain of a factor of n . We also show that our results transfer to the more familiar *generic search* problem: that of returning the list of all vectors q such that $P q = \text{true}$.

139 The idea behind the speedup is easily explained and will already be familiar, at least
 140 informally, to programmers who have worked with multi-shot continuations. Suppose for
 141 example $n = 3$, and suppose that the predicate P always inspects the components of its
 142 argument in the order 0, 1, 2. A naïve implementation of count_3 might start by applying the
 143 given P to $q_0 = (\text{true}, \text{true}, \text{true})$, and then to $q_1 = (\text{true}, \text{true}, \text{false})$. Clearly there is some
 144 duplication here: the computations of $P q_0$ and $P q_1$ will proceed identically up to the point
 145 where the value of the final component is requested. What we would like to do, then, is to
 146 record the state of the computation of $P q_0$ at just this point, so that we can later resume
 147 this computation with false supplied as the final component value in order to obtain the
 148 value of $P q_1$. (Similarly for all other internal nodes in the evident binary tree of boolean
 149 vectors.) Of course, such a ‘backup’ approach is easy to realize if one is implementing a
 150 bespoke search operation for some *particular* choice of P ; but to apply this idea of resuming
 151 previous subcomputations in the *generic* setting (that is, uniformly in P) requires some
 152 feature such as general effect handlers or multi-shot continuations.

153 One could also obviate the need for such a feature by choosing to present the predicate P
 154 in some other way, but from our present perspective this would be to move the goalposts:
 155 our intention is precisely to show that our languages differ in an essential way *as regards*
 156 *their power to manipulate data of type* $(\text{Nat} \rightarrow \text{Bool}) \rightarrow \text{Bool}$. Indeed, a key aspect of our
 157 approach, inherited from Longley and Normann (2015), is that by allowing ourselves to
 158 fix the way in which data is given to us, we are able to uncover a wealth of interesting
 159 expressivity differences between languages, despite the fact that they are in some sense inter-
 160 encodable. Such an approach also seems reasonable from the perspective of programming in
 161 the large: when implementing some program module one does not always have the freedom
 162 to choose the form or type of one’s inputs, and in such cases, the kinds of expressivity
 163 distinctions we are considering may potentially make a real practical difference.

164 This idea of using first-class control to achieve ‘backtracking’ has been exploited before
 165 and is fairly widely known (see e.g. (Kiselyov et al., 2005)), and there is a clear programming
 166 intuition that this yields a speedup unattainable in languages without such control features.
 167 Our main contribution in this paper is to provide, for the first time, a precise mathematical
 168 theorem that pins down this fundamental efficiency difference, thus giving formal substance
 169 to this intuition. Since our goal is to give a realistic analysis of the asymptotic runtimes
 170 achievable in various settings, but without getting bogged down in inessential implementa-
 171 tion details, we shall work concretely and operationally with a CEK-style abstract machine
 172 semantics as our basic model of execution time. The details of this model are only explicitly
 173 used for showing that our efficient implementation of generic count with effect handlers
 174 has the claimed $\mathcal{O}(2^n)$ runtime; but it also plays a background role as our reference model
 175 of runtime for the $\Omega(n2^n)$ lower bound results, even though we here work mostly with a
 176 simpler kind of operational semantics.

177 In the first instance, we formulate our results as a comparison between a purely functional
 178 base language λ_b (a version of call-by-value PCF) and an extension λ_h with general effect
 179 handlers. This allows us to present the key idea in a simple setting, but we then show how
 180 our runtime lower bound is also applicable to a more sophisticated language λ_a with affine
 181 effect handlers, intermediate in power between λ_b and λ_h and corresponding broadly to
 182 ‘single-shot’ uses of delimited control. Our proof involves some general machinery for
 183 reasoning about program evaluation in λ_a which may be of independent interest.
 184

In summary, our purpose is to exhibit an efficiency difference between single-shot and multi-shot versions of delimited control which, in our view, manifests a fundamental feature of the programming language landscape. Since many widely-used languages do not support multi-shot control features, this challenges a common assumption that all real-world programming languages are essentially ‘equivalent’ from an asymptotic point of view. We also situate our results within a broader context by informally discussing the attainable efficiency for generic count within a spectrum of weaker languages. 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 pure PCF-like language λ_b and an extension λ_h with general effect handlers.
- Section 4 defines abstract machines for λ_b and λ_h , yielding a runtime cost model.
- Section 5 introduces the generic count problem and some associated machinery, and presents an implementation in λ_h with essential runtime $\mathcal{O}(2^n)$.
- Section 6 discusses some extensions and variations of the foregoing result, adapting it to deal with a wider class of predicates and bridging the gap between generic count and generic search. We also briefly outline how one can use sufficient effect polymorphism to adapt the result to a setting with a type-and-effect system.
- Section 7 surveys a range of approaches to generic counting in languages weaker than λ_h , including the one suggested by Berger (1990), emphasizing how the attainable efficiency varies according to the language, but observing that none of these approaches match the $\mathcal{O}(2^n)$ runtime bound of our effectful implementation.
- Section 8 establishes that *any* generic count implementation within λ_b must have runtime $\Omega(n2^n)$ on predicates of a certain kind.
- Section 9 refines our definition of λ_h to yield a language λ_a for affine effect handlers, clarifying its relationship to λ_b and λ_h .
- Section 10 develops some machinery for reasoning about program evaluation in λ_a , and applies this to establish the $\Omega(n2^n)$ bound for generic count programs in this language.
- Section 11 reports on experiments showing that the theoretical efficiency difference we describe is manifested in practice, using implementations in OCaml of various search procedures.
- Section 12 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. Some of the less interesting proof details are relegated to the appendices.

Relation to prior work This article is an extended version of the following previously published paper and Chapter 7 of the first author’s PhD dissertation:

- Hillerström, D., Lindley, S. & Longley, J. (2020) Effects for efficiency: Asymptotic speedup with first-class control. *Proc. ACM Program. Lang.* **4**(ICFP), 100:1–100:29
- Hillerström, D. (2021) *Foundations for Programming and Implementing Effect Handlers*. Ph.D. thesis. The University of Edinburgh, Scotland, UK

The main new contribution in the present version is that we introduce a language λ_a for arbitrary affine effect handlers and develop the theory needed to extend our lower bound result to this language (Section 9), whereas in the previous version, only an extension with local state was considered. We have also included an account of the Berger search procedure (Section 7.3), and have simplified our original proof of the $\Omega(n2^n)$ bound for λ_b (Section 8). The benchmarks have been ported to OCaml 5.0 in such a way that the effectful procedures make use of effect handlers internally (Section 11).

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; Kammar et al., 2013; Kiselyov et al., 2013; Sivaramakrishnan et al., 2021; Leijen, 2017; Hillerström et al., 2020). 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: $\Sigma \stackrel{\text{def}}{=} \{\text{Branch} : \text{Unit} \rightarrow \text{Bool}\}$. The operation takes a single parameter of type `unit` and ultimately produces a boolean value. The pragmatic programming 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` $\stackrel{\text{def}}{=} \text{Heads} \mid \text{Tails}$, then we may implement a coin toss as follows.

```
toss : Unit → Toss
toss ⟨⟩ = if do Branch ⟨⟩ then Heads else Tails
```

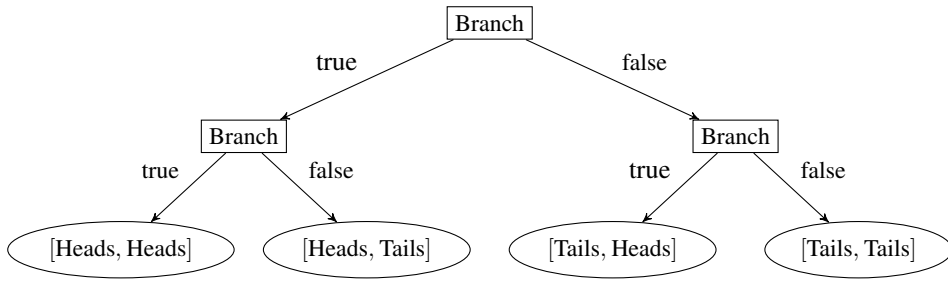
From the type signature it is clear that the computation returns a value of type `Toss`. It is not clear from the signature of `toss` whether it performs an effect. However, from the definition, it evidently performs the operation `Branch` with argument `⟨⟩` using the **do**-invocation form. The result of the operation determines whether the computation returns either `Heads` or `Tails`.

Systems such as Frank (Lindley et al., 2017; Convent et al., 2020), Helium (Biernacki et al., 2019, 2020), Koka (Leijen, 2017), and Links (Hillerström and Lindley, 2016; Hillerström et al., 2020) 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.

An effectful computation may be used as a subcomputation of another computation, e.g. we can use `toss` to implement a computation that performs two coin tosses.

```
tossTwice : Unit → List Toss
tossTwice ⟨⟩ = [toss ⟨⟩, toss ⟨⟩]
```

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 `tossTwice` is as follows.



It models the 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 two coin tosses.

```
handle tossTwice ⟨⟩ with
  val x      ↦ [x]
  Branch ⟨⟩ r ↦ r true ++ r false
```

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 `tossTwice`, 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 `tossTwice` computation from the invocation of `Branch` inside the first instance of `toss` up to its nearest enclosing handler.

Applying `r` to `true` resumes evaluation of `tossTwice` via the `true` branch, which causes another invocation of `Branch` to occur, resulting in yet another resumption. Applying this resumption yields a possible return value of `[Heads, Heads]`, which gets lifted into the singleton list `[[Heads, Heads]]`. Afterwards, the latter resumption is applied `false`, thus producing the value `[[Heads, Tails]]`. Before returning to the

first invocation of the initial resumption, the two lists get concatenated to obtain the intermediary result $[[\text{Heads}, \text{Heads}], [\text{Heads}, \text{Tails}]]$. Thereafter, the initial resumption is applied to false , which symmetrically returns the list $[[\text{Tails}, \text{Heads}], [\text{Tails}, \text{Tails}]]$. Finally, the two intermediary lists get concatenated to produce the final result $[[\text{Heads}, \text{Heads}], [\text{Heads}, \text{Tails}], [\text{Tails}, \text{Heads}], [\text{Tails}, \text{Tails}]]$.

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.

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

The ground types are Nat and Unit which classify natural number values and the unit value, respectively. The function type $A \rightarrow B$ 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).

Values

$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}$	$\frac{}{\Gamma \vdash \langle \rangle : \text{Unit}}$	$\frac{k \in \mathbb{N}}{\Gamma \vdash k : \text{Nat}}$	$\frac{c : A \rightarrow B}{\Gamma \vdash c : A \rightarrow B}$
$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x^A. M : A \rightarrow B}$	$\frac{\Gamma, f : A \rightarrow B, x : A \vdash M : B}{\Gamma \vdash \mathbf{rec}^{f^A \rightarrow B} x. M : A \rightarrow B}$	$\frac{\Gamma \vdash V : A \quad \Gamma \vdash W : B}{\Gamma \vdash \langle V, W \rangle : A \times B}$	$\frac{\Gamma \vdash V : A}{\Gamma \vdash (\mathbf{inl} V)^B : A + B}$
$\frac{\Gamma \vdash W : B}{\Gamma \vdash (\mathbf{inr} W)^A : A + B}$			

Computations

$\frac{\Gamma \vdash V : A \rightarrow B \quad \Gamma \vdash W : A}{\Gamma \vdash V W : B}$	$\frac{\Gamma \vdash V : A \times B \quad \Gamma, x : A, y : B \vdash N : C}{\Gamma \vdash \mathbf{let} \langle x, y \rangle = V \mathbf{in} N : C}$
$\frac{\Gamma \vdash V : A + B \quad \Gamma, x : A \vdash M : C \quad \Gamma, y : B \vdash N : C}{\Gamma \vdash \mathbf{case} V \{ \mathbf{inl} x \mapsto M; \mathbf{inr} y \mapsto N \} : C}$	
$\frac{\Gamma \vdash V : A}{\Gamma \vdash \mathbf{return} V : A}$	$\frac{\Gamma \vdash M : A \quad \Gamma, x : A \vdash N : C}{\Gamma \vdash \mathbf{let} x \leftarrow M \mathbf{in} N : C}$

Fig. 1: Typing Rules for λ_b

S-APP	$(\lambda x^A. M)V \rightsquigarrow M[V/x]$
S-APP-REC	$(\mathbf{rec}^{f^A} x. M)V \rightsquigarrow M[(\mathbf{rec}^{f^A} x. M)/f, V/x]$
S-CONST	$c V \rightsquigarrow \mathbf{return} (\ulcorner c \urcorner(V))$
S-SPLIT	$\mathbf{let} \langle x, y \rangle = \langle V, W \rangle \mathbf{in} N \rightsquigarrow N[V/x, W/y]$
S-CASE-INL	$\mathbf{case} (\mathbf{inl} V)^B \{ \mathbf{inl} x \mapsto M; \mathbf{inr} y \mapsto N \} \rightsquigarrow M[V/x]$
S-CASE-INR	$\mathbf{case} (\mathbf{inr} V)^A \{ \mathbf{inl} x \mapsto M; \mathbf{inr} y \mapsto N \} \rightsquigarrow N[V/y]$
S-LET	$\mathbf{let} x \leftarrow \mathbf{return} V \mathbf{in} N \rightsquigarrow N[V/x]$
S-LIFT	$\mathcal{E}[M] \rightsquigarrow \mathcal{E}[N], \quad \text{if } M \rightsquigarrow N$
Evaluation contexts $\mathcal{E} ::= [] \mid \mathbf{let} x \leftarrow \mathcal{E} \mathbf{in}$	

Fig. 2: Contextual Small-Step Operational Semantics

The constants have the following types.

$$\{(+), (-)\} : \text{Nat} \times \text{Nat} \rightarrow \text{Nat} \quad (=) : \text{Nat} \times \text{Nat} \rightarrow \text{Unit} + \text{Unit}$$

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 (Moggi, 1991; Flanagan et al., 1993). 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.

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

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

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

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

$$\lambda_{-}^A.M \stackrel{\text{def}}{=} \lambda x^A.M, \quad \text{where } x \notin FV(M)$$

We use the standard encoding of booleans as a sum:

$$\text{Bool} \stackrel{\text{def}}{=} \text{Unit} + \text{Unit} \quad \text{true} \stackrel{\text{def}}{=} \mathbf{inl}\langle \rangle \quad \text{false} \stackrel{\text{def}}{=} \mathbf{inr}\langle \rangle$$

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

3.2 Handler calculus

We now define λ_h as an extension of λ_b .

Operation symbols	$\ell \in \mathcal{L}$
Signatures	$\Sigma ::= \cdot \mid \{\ell : A \rightarrow B\} \cup \Sigma$
Handler types	$F ::= C \Rightarrow D$
Computations	$M, N ::= \dots \mid \mathbf{do}\ \ell\ V \mid \mathbf{handle}\ M\ \mathbf{with}\ H$
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 $A \rightarrow B$ classifies operations that take an argument of type A and return a result of type B . We write $\text{dom}(\Sigma) \subseteq \mathcal{L}$ 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

Computations

$$\begin{array}{c}
461 \\
462 \\
463 \\
464
\end{array}
\frac{\text{T-DO} \quad (\ell : A \rightarrow B) \in \Sigma \quad \Gamma \vdash V : A}{\Gamma \vdash \mathbf{do} \ell V : B} \qquad \frac{\text{T-HANDLE} \quad \Gamma \vdash M : C \quad \Gamma \vdash H : C \Rightarrow D}{\Gamma \vdash \mathbf{handle} M \mathbf{with} H : D}$$

Handlers

$$\begin{array}{c}
465 \\
466 \\
467 \\
468 \\
469 \\
470 \\
471
\end{array}
\frac{\text{T-HANDLER} \quad H^{\text{val}} = \{\mathbf{val} x \mapsto M\} \quad [H^\ell = \{\ell p r \mapsto N_\ell\}]_{\ell \in \text{dom}(\Sigma)} \quad \Gamma, x : C \vdash M : D \quad [\Gamma, p : A_\ell, r : B_\ell \rightarrow D \vdash N_\ell : D]_{(\ell : A_\ell \rightarrow B_\ell) \in \Sigma}}{\Gamma \vdash H : C \Rightarrow D}$$

Fig. 3: Additional Typing Rules for λ_h

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 ($B_\ell \rightarrow D$) 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.

$$\begin{array}{c}
486 \\
487 \\
488
\end{array}
\begin{array}{l}
H^{\text{val}} \stackrel{\text{def}}{=} \{\mathbf{val} x \mapsto M\}, \quad \text{where } \{\mathbf{val} x \mapsto M\} \in H \\
H^\ell \stackrel{\text{def}}{=} \{\ell p r \mapsto M\}, \quad \text{where } \{\ell p r \mapsto M\} \in H
\end{array}$$

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.

$$\begin{array}{c}
491 \\
492 \\
493 \\
494
\end{array}
\begin{array}{l}
\text{S-RET} \quad \mathbf{handle} (\mathbf{return} V) \mathbf{with} H \rightsquigarrow N[V/x], \quad \text{where } H^{\text{val}} = \{\mathbf{val} x \mapsto N\} \\
\text{S-OP} \quad \mathbf{handle} \mathcal{E}[\mathbf{do} \ell V] \mathbf{with} H \rightsquigarrow N[V/p, (\lambda y. \mathbf{handle} \mathcal{E}[\mathbf{return} y] \mathbf{with} H)/r], \\
\quad \text{where } H^\ell = \{\ell p r \mapsto N\}
\end{array}$$

The first rule invokes the success clause. The second rule handles an operation via the corresponding operation clause. If we were naively 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.

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

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

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

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

Theorem 1 (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.*

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. (2020). 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 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.

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

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 ($\llbracket - \rrbracket$) 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

Transition relation

553	M-APP	$\langle V W \mid \gamma \mid \sigma \rangle \longrightarrow \langle M \mid \gamma' [x \mapsto \llbracket W \rrbracket \gamma] \mid \sigma \rangle,$ if $\llbracket V \rrbracket \gamma = (\gamma', \lambda x^A. M)$
554		
555	M-REC	$\langle V W \mid \gamma \mid \sigma \rangle \longrightarrow \langle M \mid \gamma' [f \mapsto (\gamma', \mathbf{rec}^{f^{A \rightarrow B}} x.M),$ $x \mapsto \llbracket W \rrbracket \gamma] \mid \sigma \rangle,$ if $\llbracket V \rrbracket \gamma = (\gamma', \mathbf{rec}^{f^{A \rightarrow B}} x.M)$
556		
557		
558	M-CONST	$\langle V W \mid \gamma \mid \sigma \rangle \longrightarrow \langle \mathbf{return} (\ulcorner c \urcorner (\llbracket W \rrbracket \gamma)) \mid \gamma \mid \sigma \rangle,$ if $\llbracket V \rrbracket \gamma = c$
559		
560	M-SPLIT	$\langle \mathbf{let} \langle x, y \rangle = V \mathbf{in} N \mid \gamma \mid \sigma \rangle \longrightarrow \langle N \mid \gamma [x \mapsto v, y \mapsto w] \mid \sigma \rangle,$ if $\llbracket V \rrbracket \gamma = \langle v; w \rangle$
561		
562		
563	M-CASEL	$\langle \mathbf{case} V \{ \mathbf{inl} x \mapsto M;$ $\mathbf{inr} y \mapsto N \} \mid \gamma \mid \sigma \rangle \longrightarrow \langle M \mid \gamma [x \mapsto v] \mid \sigma \rangle,$ if $\llbracket V \rrbracket \gamma = \mathbf{inl} v$
564		
565		
566	M-CASER	$\langle \mathbf{case} V \{ \mathbf{inl} x \mapsto M;$ $\mathbf{inr} y \mapsto N \} \mid \gamma \mid \sigma \rangle \longrightarrow \langle N \mid \gamma [y \mapsto v] \mid \sigma \rangle,$ if $\llbracket V \rrbracket \gamma = \mathbf{inr} v$
567		
568	M-LET	$\langle \mathbf{let} x \leftarrow M \mathbf{in} N \mid \gamma \mid \sigma \rangle \longrightarrow \langle M \mid \gamma \mid (\gamma, x, N) :: \sigma \rangle$
569	M-RETCONT	$\langle \mathbf{return} V \mid \gamma \mid (\gamma', x, N) :: \sigma \rangle \longrightarrow \langle N \mid \gamma' [x \mapsto \llbracket V \rrbracket \gamma] \mid \sigma \rangle$
570		

Value interpretation

572	$\llbracket x \rrbracket \gamma = \gamma(x)$	$\llbracket n \rrbracket \gamma = n$	$\llbracket \lambda x^A. M \rrbracket \gamma = (\gamma, \lambda x^A. M)$
573	$\llbracket \langle \rangle \rrbracket \gamma = \langle \rangle$	$\llbracket c \rrbracket \gamma = c$	$\llbracket \mathbf{rec}^{f^{A \rightarrow B}} x.M \rrbracket \gamma = (\gamma, \mathbf{rec}^{f^{A \rightarrow B}} x.M)$
574			
575	$\llbracket \langle V, W \rangle \rrbracket \gamma = \langle \llbracket V \rrbracket \gamma, \llbracket W \rrbracket \gamma \rangle$	$\llbracket (\mathbf{inl} V)^B \rrbracket \gamma = (\mathbf{inl} \llbracket V \rrbracket \gamma)^B$	$\llbracket (\mathbf{inr} V)^A \rrbracket \gamma = (\mathbf{inr} \llbracket V \rrbracket \gamma)^A$
576			

Fig. 4: Abstract Machine Semantics for λ_b

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 $\langle \mathbf{return} V \mid \gamma \mid \square \rangle$ in which case the final return value is given by the denotation $\llbracket V \rrbracket \gamma$ of V under environment γ .

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 $\langle - \rangle$ from configurations to terms (Hillerström et al., 2020).

Transition relation

599	M-RESUME	$\langle V W \mid \gamma \mid \kappa \rangle \longrightarrow \langle \mathbf{return} W \mid \gamma \mid (\sigma, \chi) :: \kappa \rangle,$
		if $\llbracket V \rrbracket \gamma = (\sigma, \chi)$
601	M-LET	$\langle \mathbf{let} x \leftarrow M \mathbf{in} N \mid \gamma \mid (\sigma, \chi) :: \kappa \rangle \longrightarrow \langle M \mid \gamma \mid ((\gamma, x, N) :: \sigma, \chi) :: \kappa \rangle$
602	M-RETCONT	$\langle \mathbf{return} V \mid \gamma \mid ((\gamma', x, N) :: \sigma, \chi) :: \kappa \rangle \longrightarrow \langle N \mid \gamma' [x \mapsto \llbracket V \rrbracket \gamma] \mid (\sigma, \chi) :: \kappa \rangle$
603	M-HANDLE	$\langle \mathbf{handle} M \mathbf{with} H \mid \gamma \mid \kappa \rangle \longrightarrow \langle M \mid \gamma \mid ([], (\gamma, H)) :: \kappa \rangle$
604	M-RETHANDLER	$\langle \mathbf{return} V \mid \gamma \mid ([], (\gamma', H)) :: \kappa \rangle \longrightarrow \langle M \mid \gamma' [x \mapsto \llbracket V \rrbracket \gamma] \mid \kappa \rangle,$
605		if $H^{\mathbf{val}} = \{\mathbf{val} x \mapsto M\}$
606	M-HANDLE-OP	$\langle \mathbf{do} \ell V \mid \gamma \mid (\sigma, (\gamma', H)) :: \kappa \rangle \longrightarrow \langle M \mid \gamma' [p \mapsto \llbracket V \rrbracket \gamma,$
607		$r \mapsto (\sigma, (\gamma', H))] \mid \kappa \rangle,$
608		if $\ell : A \rightarrow B \in \Sigma$
609		and $H^\ell = \{\ell p r \mapsto M\}$
610		

Fig. 5: Abstract Machine Semantics for λ_h **4.2 Handler machine**

We now enrich the λ_b machine to a λ_h machine. We extend the syntax as follows.

612	Configurations	$\mathcal{C} \in \mathbf{Conf} ::= \langle M \mid \gamma \mid \kappa \rangle$
613	Resumptions	$\rho \in \mathbf{Res} ::= (\sigma, \chi)$
614	Continuations	$\kappa \in \mathbf{Cont} ::= [] \mid \rho :: \kappa$
615	Handler closures	$\chi \in \mathbf{HClo} ::= (\gamma, H)$
616	Machine values	$v, w \in \mathbf{MVal} ::= \dots \mid \rho$

The notion of configurations changes slightly in that the continuation component is replaced by a generalised continuation $\kappa \in \mathbf{Cont}$ (Hillerström et al., 2020); 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:

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

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 $\mathcal{O}(1)$ for the extension operation $(_ :: _)$. The topmost pure continuation on the handler machine may also be extended in time $\mathcal{O}(1)$, as extending it only requires reaching under the topmost handler closure. Environments, γ , can be realised using a map, with a time complexity of $\mathcal{O}(\log |\gamma|)$ 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 $\mathcal{O}(\log |\gamma|)$. The value interpretation function $\llbracket - \rrbracket \gamma$ is defined structurally on values. Its worst-time complexity is exhibited by a nesting of pairs of variables $\llbracket \langle x_1, \langle x_2, \dots, \langle x_{n-1}, x_n \rangle \dots \rangle \rangle \rrbracket \gamma$ which has complexity $\mathcal{O}(n \log |\gamma|)$.

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 $\mathcal{O}(|\sigma|)$ time operation.

Primitive operations on naturals Our model assumes that arithmetic operations on arbitrary natural numbers take $\mathcal{O}(1)$ 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 8). 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 (Simpson, 1998; Daniels, 2016).

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 $\mathbb{N}_n \rightarrow \mathbb{B}$, where $\mathbb{N}_n \stackrel{\text{def}}{=} \{0, \dots, n-1\}$. In both λ_b and λ_h we may represent such functions by terms of type $\text{Nat} \rightarrow \text{Bool}$. We will often informally write Nat_n in place of Nat to indicate that only the values $0, \dots, n-1$ 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

$$\begin{array}{ll} \text{Point} \stackrel{\text{def}}{=} \text{Nat} \rightarrow \text{Bool} & \text{Point}_n \stackrel{\text{def}}{=} \text{Nat}_n \rightarrow \text{Bool} \\ \text{Predicate} \stackrel{\text{def}}{=} \text{Point} \rightarrow \text{Bool} & \text{Predicate}_n \stackrel{\text{def}}{=} \text{Point}_n \rightarrow \text{Bool} \end{array}$$

and will be looking for programs

$$\text{count}_n : \text{Predicate}_n \rightarrow \text{Nat}$$

such that for suitable terms P representing semantic predicates $\Pi : \mathbb{B}^n \rightarrow \mathbb{B}$, $\text{count}_n P$ 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 :

$$q_0 \stackrel{\text{def}}{=} \lambda _ . \text{true} \quad q_1 \stackrel{\text{def}}{=} \lambda i . i = 0 \quad q_2 \stackrel{\text{def}}{=} \lambda i . \text{if } i = 0 \text{ then true else if } i = 1 \text{ then false else } \perp$$

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

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

$$T_0 \stackrel{\text{def}}{=} \lambda q . \text{true} \quad T_1 \stackrel{\text{def}}{=} \lambda q . (q 1 ; q 0 ; \text{true}) \quad T_2 \stackrel{\text{def}}{=} \lambda q . (q 0 ; q 0 ; \text{true})$$

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 $\mathbb{B}^1 \rightarrow \mathbb{B}$ (here $\&\&$ is shortcut ‘and’):

$$I_0 \stackrel{\text{def}}{=} \lambda q . q 0 \quad I_1 \stackrel{\text{def}}{=} \lambda q . \text{if } q 0 \text{ then true else false} \quad I_2 \stackrel{\text{def}}{=} \lambda q . (q 0) \&\& (q 0)$$

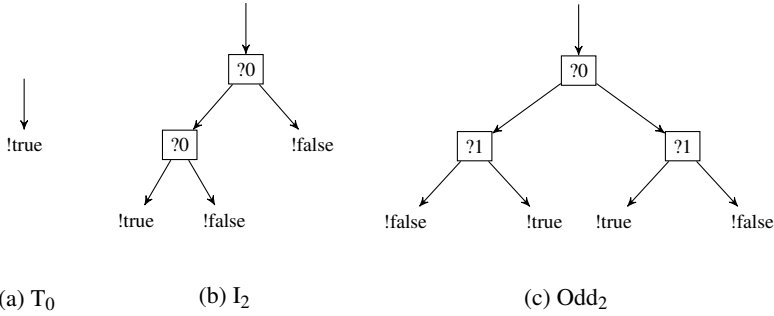


Fig. 6: Examples of Decision Trees

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

$$\text{Odd}_n \stackrel{\text{def}}{=} \lambda q. \text{fold } \otimes \text{ false } (\text{map } q [0, \dots, n-1])$$

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 $Q : \text{Point}_n$. 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 PQ 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 $\mathbb{N}_n \stackrel{\text{def}}{=} \{0, \dots, n - 1\}$, 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 2 (*n-points*). A closed value $Q : \text{Point}$ is said to be a syntactic *n-point* if:

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

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

$$\forall k \in \mathbb{N}_n, b \in \mathbb{B}. \llbracket Q \rrbracket(k) = b \Leftrightarrow Q\ k \rightsquigarrow^* \text{return } b$$

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

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 $\Pi : \mathbb{B}^n \rightarrow \mathbb{B}$. One slick way to define syntactic *n-predicates* would be as closed terms $P : \text{Predicate}$ 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 3 (untimed decision tree). (i) The address set **Addr** is simply the set \mathbb{B}^* of finite lists of booleans. If $bs, bs' \in \text{Addr}$, we write $bs \sqsubseteq bs'$ (resp. $bs \sqsubset bs'$) to mean that bs is a prefix (resp. proper prefix) of bs' .

(ii) The label set \mathbf{Lab} consists of queries parameterised by a natural number and answers parameterised by a boolean:

$$\mathbf{Lab} \stackrel{\text{def}}{=} \{?k \mid k \in \mathbb{N}\} \cup \{!b \mid b \in \mathbb{B}\}$$

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

- The domain of τ (written $\text{dom}(\tau)$) 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 4 (timed decision tree). *A timed decision tree is a partial function $\tau : \mathbf{Addr} \rightarrow \mathbf{Lab} \times \mathbb{N}$ such that its first projection $bs \mapsto \tau(bs).1$ is a decision tree. We write $\text{labs}(\tau)$ for the first projection ($bs \mapsto \tau(bs).1$) and $\text{steps}(\tau)$ for the second projection ($bs \mapsto \tau(bs).2$) of a timed decision tree.*

Here we think of $\text{steps}(\tau)(bs)$ 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 $q : \text{Point}$, 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. (i) Define $\mathcal{T} : \text{Conf}_q \rightarrow \mathbf{Addr} \rightarrow (\mathbf{Lab} \times \mathbb{N})$ to be the minimal family of partial functions satisfying the following equations:

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

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

(ii) The timed decision tree of a computation term is obtained by placing it in the initial configuration: $\mathcal{T}(M) \stackrel{\text{def}}{=} \mathcal{T}(\langle M, \emptyset[q \mapsto q], \kappa_0 \rangle)$.

(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: $\mathcal{U}(P) = \text{labs}(\mathcal{T}(P))$.

If the execution of a configuration \mathcal{C} runs forever or gets stuck at an unhandled operation, then $\mathcal{T}(\mathcal{C})(bs)$ 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 6. 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:

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

- 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 $b_0 b_1 \dots$ through τ , given inductively by:

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

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

Proposition 1. If P is an n -predicate and Q is an n -point, then $P Q \rightsquigarrow^* \mathbf{return} b$ where $b = \mathcal{U}(P) \bullet \llbracket Q \rrbracket$.

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 $\llbracket P \rrbracket$ given by $\llbracket P \rrbracket(\pi) = \mathcal{U}(P) \bullet \pi$.

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

Definition 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 \mathbf{Addr}_n , the set of bit vectors of length $\leq n$.
- There are no repeated queries along any path in τ :

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

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

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

925 It is also clear how for any n -standard tree τ we may construct a predicate P that denotes
 926 it, simply by mirroring the structure of τ with nested **if** expressions:

927
 928 **Definition 8** (canonical n -standard predicates). *Given an n -standard tree τ , we may asso-*
 929 *ciate to each address $bs \in \text{dom}(\tau)$ a λ_b term $T(\tau, bs)$ (with free variable $q : \text{Point}$) by reverse*
 930 *induction on the length of bs :*

$$932 \quad T(\tau, bs) = \text{return } b \quad \text{if } \tau(bs) = !b$$

$$933 \quad T(\tau, bs) = \text{if } q(k) \text{ then } T(\tau, bs ++ [true]) \text{ else } T(\tau, bs ++ [false]) \quad \text{if } \tau(bs) = ?k$$

934 We then define

$$935 \quad P(\tau) = \lambda q. T(\tau, [])$$

936 (so that clearly $\mathcal{U}(P(\tau)) = \tau$), and call $P(\tau)$ the canonical n -standard predicate for τ .
 937
 938

939 Note that the use of lists here is entirely at the meta-level, and none of the terms $T(\tau, bs)$
 940 themselves involve list data. Because of their simple, standardized form, canonical n -
 941 standard predicates will play a useful role in our lower bound analysis in Section 8
 942

943 5.3 Specification of counting programs

944 We can now specify what it means for a program $K : \text{Predicate} \rightarrow \text{Nat}$ to implement counting.
 945

946
 947 **Definition 9.** (i) *The count of a semantic n -predicate Π , written $\sharp\Pi$, is simply the number*
 948 *of semantic n -points $\pi \in \mathbb{B}^n$ for which $\Pi(\pi) = \text{true}$.*

949 (ii) *If P is any n -predicate, we say that K correctly counts P if $K P \rightsquigarrow^* \text{return } m$, where*
 950 *$m = \sharp[[P]]$.*
 951

952 This definition gives us the flexibility to talk about counting programs that operate on
 953 various classes of predicates, allowing us to state our results in their strongest natural form.
 954 On the positive side, we shall shortly see that there is a single ‘efficient’ program in λ_h
 955 that correctly counts all n -standard λ_h predicates for every n ; in Section 6.1 we improve
 956 this to one that correctly counts *all* n -predicates of λ_h . On the negative side, we shall show
 957 that an n -indexed family of counting programs written in λ_b , even if only required to work
 958 correctly on canonical n -standard λ_b predicates, can never compete with our λ_h program for
 959 asymptotic efficiency even in the most favourable cases.
 960

961 5.4 Efficient generic count with effects

962 We now present the simplest version of our effectful implementation of counting: one that
 963 works on n -standard predicates.
 964
 965
 966

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, meaning that prior computation performed by the predicate 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 $\text{Branch} : \text{Unit} \rightarrow \text{Bool} \in \Sigma$ 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.

```

effcount : ((Nat → Bool) → Bool) → Nat
effcount pred  $\stackrel{\text{def}}{=} \mathbf{handle\ pred\ (\lambda\_do\ Branch\ \langle \rangle)\ with}$ 
    val x       $\mapsto \mathbf{if\ } x \mathbf{\ then\ return\ 1\ else\ return\ 0}$ 
    Branch  $\langle \rangle\ r$   $\mapsto \mathbf{let\ } x_{\text{true}} \leftarrow r \mathbf{\ true\ in}$ 
     $\mathbf{let\ } x_{\text{false}} \leftarrow r \mathbf{\ false\ in\ } x_{\text{true}} + x_{\text{false}}$ 

```

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 $(\lambda_do\ Branch\ \langle \rangle)$ informally serves as a ‘superposition’ of all possible points.

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

Theorem 2. *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*

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

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

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

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 $\text{Hyp}(bs)$ by a laborious but entirely routine downwards induction

1013 on the length of bs . The proof combines counting of explicit machine steps with ‘oracular’
 1014 appeals to the assumed behaviour of P as modelled by $\mathcal{T}(P)$. Once $\text{Hyp}(\square)$ is established,
 1015 both halves of the theorem follow easily. Full details are given in Appendix C of Hillerström
 1016 et al. (2020). ■

1017 The above formula can clearly be simplified for certain reasonable classes of predicates.
 1018 For instance, suppose we fix some constant $c \in \mathbb{N}$, and let $\mathcal{P}_{n,c}$ be the class of all n -standard
 1019 predicates P for which all the edge times $\text{steps}(\mathcal{T}(P))(bs)$ are bounded by c . (Clearly,
 1020 many reasonable predicates will belong to $\mathcal{P}_{n,c}$ for some modest value of c .) Since the
 1021 number of sequences bs in question is less than 2^{n+1} , we may read off from the above
 1022 formula that for predicates in $\mathcal{P}_{n,c}$, the runtime of `effcount` is $\mathcal{O}(c2^n)$.

1023 Alternatively, should we wish to use the finer-grained cost model that assigns an $\mathcal{O}(\log |\gamma|)$
 1024 runtime to each abstract machine step (see Section 4.3), we may note that any environment
 1025 γ arising in the computation contains at most n entries introduced by the `let`-bindings in
 1026 `effcount`, and (if $P \in \mathcal{P}_{n,c}$) at most $\mathcal{O}(cn)$ entries introduced by P . Thus, the time for each
 1027 step in the computation remains $\mathcal{O}(\log c + \log n)$, and the total runtime for `effcount` is
 1028 $\mathcal{O}(c2^n(\log c + \log n))$.

1029 One might also ask about the execution time for an implementation of λ_h that performs
 1030 genuine copying of continuations, as in systems such as MLton (2020). As MLton copies
 1031 the entire continuation (stack), whose size is $\mathcal{O}(n)$, at each of the 2^n branches, continuation
 1032 copying alone takes time $\mathcal{O}(n2^n)$ and the effectful implementation offers no performance
 1033 benefit. More refined implementations (Farvardin and Reppy, 2020; Flatt and Dybvig, 2020)
 1034 that are able to take advantage of delimited control operators or sharing in copies of the
 1035 stack can bring the complexity of continuation copying back down to $\mathcal{O}(2^n)$.

1036 Finally, one might consider another dimension of cost, namely the space used by `effcount`.
 1037 Consider a class $\mathcal{Q}_{n,c,d}$ of n -standard predicates P for which the edge times in $\mathcal{T}(P)$
 1038 never exceed c and the sizes of pure continuations never exceed d . If we consider any
 1039 $P \in \mathcal{Q}_{n,c,d}$ then the total number of environment entries is bounded by cn , taking up space
 1040 $\mathcal{O}(cn(\log cn))$. We must also account for the pure continuations. There are n of these, each
 1041 taking at most d space. Thus the total space is $\mathcal{O}(n(d + c(\log c + \log n)))$.

1042 6 Extensions and variations

1043 Our complexity result is robust under several variations. We outline here how the idea
 1044 generalises beyond n -standard predicates, and adapts from generic count to generic search.
 1045 We also indicate how one may obtain the speedup in question in the presence of a type-and-
 1046 effect system.

1047 6.1 Beyond n -standard predicates

1048 The n -standard restriction on predicates serves to make the efficiency phenomenon stand
 1049 out as clearly as possible. However, we can relax the restriction by tweaking `effcount` to
 1050 handle repeated queries and missing queries. The trade-off is that the analysis of `effcount`
 1051 becomes more involved. The key to relaxing the n -standard restriction is the use of state to
 1052 keep track of which queries have been computed. We can give stateful implementations of
 1053
 1054
 1055
 1056
 1057
 1058

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 $r : B \rightarrow S \rightarrow D$, 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.

$$\text{Branch} : \text{Nat} \rightarrow \text{Bool}$$

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

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

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 `addn` and `lookupn` is $\mathcal{O}(\log n)$ (Okasaki, 1999). We can then use parameter-passing to support repeated queries as follows.

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

The state parameter s memoises query results, thus avoiding double-counting and enabling `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.

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

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 $\text{effcount}_n''$ to operate correctly on predicates that inspect n points at most once. We leave it as an exercise for the reader to combine $\text{effcount}'_n$ and $\text{effcount}''_n$ to handle both repeated queries and missing queries.

6.2 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

$$\text{search}_n : ((\text{Nat}_n \rightarrow \text{Bool}) \rightarrow \text{Bool}) \rightarrow \text{List}_{\text{Nat}_n \rightarrow \text{Bool}}$$

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 $x_{s_{\text{true}}}$ and $x_{s_{\text{false}}}$ as follows.

$$\begin{aligned} \text{effsearch}_n : ((\text{Nat}_n \rightarrow \text{Bool}) \rightarrow \text{Bool}) \rightarrow \text{List}_{\text{Nat}_n \rightarrow \text{Bool}} \\ \text{effsearch}_n \text{ pred} \stackrel{\text{def}}{=} & \mathbf{let} f \leftarrow \mathbf{handle} \text{ pred } (\lambda i. \mathbf{do} \text{ Branch } i) \mathbf{with} \\ & \mathbf{val} x \quad \mapsto \lambda q. \mathbf{if} x \mathbf{then} \text{ singleton } q \mathbf{else} \text{ nil} \\ & \text{Branch } i \ r \mapsto \lambda q. \mathbf{let} x_{s_{\text{true}}} \leftarrow r \ \text{true} \ (\lambda j. \mathbf{if} i = j \mathbf{then} \text{ true} \mathbf{else} q \ j) \ \mathbf{in} \\ & \quad \mathbf{let} x_{s_{\text{false}}} \leftarrow r \ \text{false} \ (\lambda j. \mathbf{if} i = j \mathbf{then} \text{ false} \mathbf{else} q \ j) \ \mathbf{in} \\ & \quad \text{concat } \langle x_{s_{\text{true}}}, x_{s_{\text{false}}} \rangle \\ & \mathbf{in} \text{ toConsList } (f \ (\lambda j. \perp)) \end{aligned}$$

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 if it is in the predicate. A little care is required to ensure that effsearch_n has runtime $\mathcal{O}(2^n)$; naïve use of cons-list concatenation would result in $\mathcal{O}(n2^n)$ 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: $\text{HList}_A \stackrel{\text{def}}{=} \text{List}_A \rightarrow \text{List}_A$. The empty Hughes list $\text{nil} : \text{HList}_A$ is defined as the identity function: $\text{nil} \stackrel{\text{def}}{=} \lambda xs. xs$.

$$\begin{aligned} \text{singleton}_A : A \rightarrow \text{HList}_A \\ \text{singleton}_A x \stackrel{\text{def}}{=} \lambda xs. x :: xs \\ \\ \text{concat}_A : \text{HList}_A \times \text{HList}_A \rightarrow \text{HList}_A \\ \text{concat}_A f g \stackrel{\text{def}}{=} \lambda xs. g (f xs) \\ \\ \text{toConsList}_A : \text{HList} \rightarrow \text{List}_A \\ \text{toConsList}_A f \stackrel{\text{def}}{=} f [] \end{aligned}$$

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

6.3 Type-and-effect system

Many practical implementations of effect handlers come equipped with rich type systems that track which effectful operations any function may perform (Bauer and Pretnar, 2014; Hillerström and Lindley, 2016; Leijen, 2017; Biernacki et al., 2019; Brachthäuser et al., 2020). One may wonder whether our result transfers to such a system as we make crucial use of the ability to *inject* an effectful operation into a computation, which a first glance might seem to require a change of (effect) types.

However, as we shall briefly outline, with sufficient polymorphism we need not change the effect types. Our generic count program does not perform any externally visible effects. Therefore, if we equip our simple type system with some form of rank-2 effect polymorphism, then we do not morally require a change of types even in the presence of the richer types provided by effect tracking.

Suppose we track the effects on function types, e.g. $A \rightarrow B!\varepsilon$ denotes a function that accepts a value of type A as input and produces some value of type B as output using effects ε . Here ε is intended to be an effect variable which may be instantiated to name concrete effectful operations that the function may perform such as $\text{Branch} : \langle \rangle \rightarrow \text{Bool}$. We shall not concern ourselves with a particular effect type formalism here, but rather just note that there are many approaches to realising such an effect system, e.g. using row types (Hillerström and Lindley, 2016; Leijen, 2017), subtyping (Bauer and Pretnar, 2014), intersection types (Brachthäuser et al., 2020), etc.

We can give a fully effect-parametric signature to generic count using rank-2 effect polymorphism.

$$\text{Count} : (\forall \varepsilon. (\text{Nat} \rightarrow \text{Bool}!\varepsilon) \rightarrow \text{Bool}!\varepsilon) \rightarrow \text{Nat}!\emptyset$$

Here \emptyset denotes that an application of `Count` does not perform any externally visible effects. The parameter type of `Count` is a rank-2 effect type. It effectively hides the implementation detail of the provided point from the predicate. Thus, the implementation of `Count` is allowed to supply a point that performs any effectful operation granted that the implementation guarantees to handle any such operation. This idea of using rank-2 polymorphism is an old idea which dates back to at least McCracken (1984); and it has been used in practice in Haskell as the primary means for state encapsulation since Launchbury and Jones (1994).

7 Generic count in weaker languages

We have shown that there is an implementation of generic count in λ_h with a runtime bound of $\mathcal{O}(2^n)$ for certain well-behaved predicates. Our eventual goal is to prove that such a runtime bound is unattainable in λ_b (Section 8), or indeed in the stronger language λ_a (Section 10). In this section, we provide some context for these results by surveying a range of possible approaches to generic counting in languages weaker than λ_h , emphasizing how the attainable efficiency varies according to the expressivity of the language. Since the purpose here is simply to situate our main results within a broader landscape which may itself call for further investigation, our discussion in this section will be informal and intuitive rather than mathematically rigorous.

7.1 Naïve count

The naïve approach, of course, is 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. Of course, this approach could be readily implemented in λ_b ; but it is also clear how it could be effected in an even weaker language, in which the *recursion* construct of λ_b is replaced by a weaker *iteration* construct. For instance, the following operator (definable in λ_b) allows one to achieve the effect of while-loops manipulating data of type A :

$$\begin{aligned} \text{while}_A &: (A \rightarrow \text{Bool}) \rightarrow A \rightarrow (A \rightarrow A) \rightarrow A \\ \text{while}_A \text{ test } x f &\stackrel{\text{def}}{=} \mathbf{if} \text{ test } x \mathbf{then} \text{ while}_A \text{ test } (f x) f \mathbf{else} x \end{aligned}$$

Let us write λ_i for the sublanguage of λ_b allowing while_A for each type A , but disallowing all uses of **rec** elsewhere. Then it is a straightforward coding exercise to write a λ_i program

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

that implements generic counting using the naïve strategy.

The evaluation of an n -standard predicate on an individual n -point must clearly take time $\Omega(n)$. It is therefore clear that in whatever way the naïve count strategy is implemented, the runtime on any n -standard predicate P must be $\Omega(n2^n)$. If P is not n -standard, the $\Omega(n)$ bound on each point application need not apply, but we may still say that a naïve count for 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 . In the next section we illustrate the germ of this idea, and in Section 7.3 we show how it gives rise to a practically superior counting program within λ_b .

7.2 The nesting trick

The germ of the idea may be illustrated even within λ_i . Suppose that we first construct some program

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

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, 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 $\text{bestshot}_n P$ takes only constant time, all the real work being deferred until the argument of type Nat_n is supplied.

Now consider the following program:

$$\begin{aligned} \text{lazycount}_n &\stackrel{\text{def}}{=} \lambda \text{pred.} \mathbf{if} \text{ pred } (\text{bestshot}_n \text{ pred}) \\ &\quad \mathbf{then} \text{ naivecount}_n \text{ pred} \\ &\quad \mathbf{else} \text{ return } 0 \end{aligned}$$

```

1243 bestshotn : Predicaten → Pointn
1244 bestshotn pred def≡ bestshot'n pred []
1245
1246 bestshot'n : Predicaten → ListBool → Pointn
1247 bestshot'n pred start def≡ let f ← memoise (λ ⟨.⟩.bestshot''n pred start) in
1248     return (λ i. if i < |start| then start.i else (f ⟨.⟩).i)
1249
1250 bestshot''n : Predicaten → ListBool → ListBool
1251 bestshot''n pred start def≡ if |start| = n then return start
1252     else let f ← bestshot'n pred (append start [true]) in
1253     if pred f then return [f 0, . . . , f (n - 1)]
1254     else bestshot''n pred (append start [false])

```

Fig. 7: An implementation of bestshot in λ_b with memoisation

Here the term $pred$ ($bestshot_n pred$) 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 $lazycount_n$ is a correct implementation of generic count, and also that if $pred$ is the predicate $\lambda q.false$ then $lazycount_n pred$ returns 0 within $O(1)$ time, thus violating the $\Omega(2^n)$ lower bound suggested above.

This might seem like a footling point, as $lazycount_n$ 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 in λ_b 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. In contrast to the naïve strategy, however, this approach relies crucially on the use of recursion, and cannot be implemented in a language such as λ_i with mere iteration.

We shall refer to this λ_b 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). We give an implementation of Bergercount in the next section.

7.3 Berger count

Berger’s original program (Berger, 1990) introduced a remarkable search operator for predicates on *infinite* streams of booleans, and has played an important role in higher-order computability theory (Longley and Normann, 2015). What we wish to highlight here is that if one applies the algorithm to predicates on *finite* boolean vectors, the resulting program, though no longer interesting from a computability perspective, still holds some interest from a complexity standpoint: indeed, it yields what seems to be the best known implementation of generic count within a PCF-style ‘functional’ language (provided one accepts the use of a primitive for call-by-need evaluation).

We give the gist of an adaptation of Berger’s search algorithm on finite spaces. The key ingredient of Berger’s search algorithm is the $bestshot_n$ function, which given any n -standard predicate P returns a point satisfying P if one exists, or dummy point $\lambda i.false$ if not. Figure 7 depicts the implementation of this function. It is implemented by via two

```

1289 Bergercountn : Predicaten → Nat
1290 Bergercountn pred def = count'n pred [] 0
1291
1292 count'n : Predicaten → ListBool → Nat → Nat
1293 count'n pred start acc def = if |start| = n then acc + (if pred (λi.start.i) then return 1
1294                               else return 0)
1295                               else let f ← bestshot'n pred start in
1296                               if pred f then count''n start [f 0, . . . , f (n - 1)] acc
1297                               else return acc
1298
1299 count''n : Predicaten → ListBool → ListBool → Nat → Nat
1300 count''n pred start leftmost acc def = if |start| = n then acc + 1
1301                               else let b ← leftmost.start in
1302                               let acc' ← count''n pred (append start [b])
1303                               leftmost acc in
1304                               if b then count'n pred (append start [false]) acc'
1305                               else return acc'

```

Fig. 8: An implementation of Berger count in λ_b

mutually recursive auxiliary functions whose workings are admittedly hard to elucidate in a few words. The function $\text{bestshot}'_n$ is a generalisation of bestshot_n that makes a best shot at finding a point π satisfying given predicate and matching some specified list start in some initial segment of its components $[\pi(0), \dots, \pi(i-1)]$. It works ‘lazily’, drawing its values from start wherever possible, and performing an actual search only when required. This actual search is undertaken by $\text{bestshot}''_n$, which proceeds by first searching for a solution that extends the specified list with true; but if no such solution is forthcoming, it settles for false as the next component of the point being constructed. The whole procedure relies on a subtle combination of laziness, recursion and implicit nesting of calls to the provided predicate which means that the search is self-pruning in regions of the binary tree where the predicate only demands some initial segment $q\ 0, \dots, q\ (i-1)$ of its argument q .

The above program makes use of an operation

$$\text{memoise} : (\text{Unit} \rightarrow \text{List Bool}) \rightarrow (\text{Unit} \rightarrow \text{List Bool})$$

which transforms a given thunk into an equivalent ‘memoised’ version, i.e. one that caches its value after its first invocation and immediately returns this value on all subsequent invocations. Such an operation may readily be implemented with the help of local state, or alternatively may simply be added as a primitive in its own right. The latter has the advantage that it preserves the purely ‘functional’ character of the language, in the sense that every program is observationally equivalent to a λ_b program, namely the one obtained by replacing memoise by the identity.

Figure 8 depicts an implementation that exploits the above idea to yield a generic count program (this development appears to be new). Again, Bergercount_n is implemented by means of two mutually recursive auxiliary functions. The function count'_n counts the solutions to the provided predicate pred that start with the specified list of booleans, adding

1335 their number to a previously accumulated total given by acc . The function $count''_n$ does the
 1336 same thing, but exploiting the knowledge that a best shot at the ‘leftmost’ solution to P
 1337 within this subtree has already been computed. (We are visualising n -points as forming
 1338 a binary tree with true to the left of false at each fork.) Thus, $count''_n$ will not re-examine
 1339 the portion of the subtree to the left of this candidate solution, but rather will start at this
 1340 solution and work rightward.

1341 This gives rise to an n -count program that can work efficiently on predicates that tend
 1342 to ‘fail fast’: more specifically, predicates P that inspect the components of their argument
 1343 q in order $q\ 0, q\ 1, q\ 2, \dots$, and which are frequently able to return false after inspecting
 1344 just a small number of these components. Generalising our program from binary to k -ary
 1345 branching trees, we see that the n -queens problem provides a typical example: most points
 1346 in the space can be seen not to be solutions by inspecting just the first few components.
 1347 Our experimental results in Section 11 attest to the viability of this approach and its
 1348 overwhelming superiority over the naïve functional method.

1349 By contrast, the above program is *not* able to exploit parts of the tree where our predicate
 1350 ‘succeeds fast’, i.e. returns true after seeing just a few components. Unlike the effectful
 1351 count program of Section 5.4, which may sometimes add 2^{n-d} to the count in a single
 1352 step, the Berger approach can only count solutions one at a time. Thus, supposing P is an
 1353 n -standard predicate the evaluation of $count_n P$ that returns a natural number c must take
 1354 time $\Omega(c)$.

1355 7.4 Pruned count

1357 To do better than Bergercount, it seems that we must ascend to a more powerful language.
 1358 We now briefly outline another approach, using ideas from Longley (1999), which yields a
 1359 more efficient form of pruned search in an extension of λ_b with *local state* of ground type.
 1360 Since local state can certainly be encoded using affine effect handlers with no essential
 1361 loss of efficiency, this approach falls within the ambit of what can be achieved within the
 1362 language λ_a to be introduced in Section 9.

1363 The key idea is that each time we apply a predicate to a point, we may use local state to
 1364 detect which components of the point are actually inspected by the predicate. We do this
 1365 using a third-order function *Modulus*, which encloses the point in a wrapper that logs all
 1366 calls to the point, then passes this wrapped point to the predicate:

```

1368 Modulus : Predicate → Point → (Bool × ListNat)
1369 Modulus pred point  $\stackrel{\text{def}}{=} \mathbf{let\ } log \leftarrow \text{ref}([\ ] : \text{ListNat}) \mathbf{in}$ 
1370    $\mathbf{let\ } wrap \leftarrow \lambda i. (log := i :: !log; \mathbf{return\ } point\ i) \mathbf{in}$ 
1371    $\mathbf{let\ } b \leftarrow \text{pred\ } wrap \mathbf{in}$ 
1372    $\mathbf{return\ } \langle b, !log \rangle$ 

```

1373 This is somewhat different from the modulus functional considered in Longley (1999), which
 1374 returns a *sorted* list of the arguments to which the point is applied. This has the theoretically
 1375 pleasant consequence that the modulus is an example of a *sequentially realizable* functional
 1376 — its externally observable behaviour is purely functional (i.e. extensional) although the
 1377 function it implements cannot be realized in pure λ_b . However, this property is purchased
 1378

at the cost of the extra work needed to return a sorted list, and is of little relevance to our present concerns.

The essential point is that if `Modulus pred point` returns $\langle b, ilist \rangle$, we know immediately that `pred point'` would also return the value b for every `point'` that agrees with `point` at the components listed in `ilist`. With some further coding, this property can be used as the basis of a program

$$\text{prunedcount}_n : ((\text{Nat}_n \rightarrow \text{Bool}) \rightarrow \text{Bool}) \rightarrow \text{Nat}$$

that takes care, at each stage, to apply the predicate to some ‘new’ point at which the value is not already known on the basis of previous calls, and which then increments the accumulator by either 0 (if the predicate returns false) or the appropriate 2^{n-d} (if it returns true). In contrast to `Bergercount`, this has the effect of pruning the search space both where the predicate fails fast and where it succeeds fast.

Of course, the ability to prune in the ‘true’ case makes no difference for search problems such as n -queens, where the predicate never returns true without inspecting all components. Even for searches of this kind, however, `prunedcount` performs significantly better in practice than `Bergercount`, which achieves its pruning of ‘false’ subtrees by much more convoluted means. (The difference is clearly manifested by the experimental results reported in Section 11.) Indeed, in the absence of advanced control features, we are not aware of any approach to generic counting that essentially does better than `prunedcount`.

It is clear, however, that in the case of n -standard predicates, which always inspect all n components of their points, no pruning at all is possible, and neither `Bergercount` nor `prunedcount` improves on the $\Omega(n2^n)$ runtime of `naivecount`.

8 A lower bound for λ_b

The above discussion strongly suggests that the $\mathcal{O}(2^n)$ runtime of our λ_h generic count implementation is unattainable in λ_b , but also points out the existence of phenomena in this area that defy intuition (Escardó (2007) gives some striking further examples). In this section, we prove rigorously that *any* implementation of generic counting in λ_b must have runtime $\Omega(n2^n)$ on certain n -standard predicates. In the following two sections, we shall apply a similar analysis to the richer language λ_a . This mathematically robust 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.

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

1427 correspondence between these two execution models such that for the evaluation of any
 1428 closed term, the number of abstract machine steps is always at least the number of small-step
 1429 reductions. Thus, if we are able to show that the number of small-step reductions for any
 1430 generic count program in λ_b on the predicates of interest is $\Omega(n2^n)$, this will establish the
 1431 desired lower bound on the runtime.

1432 To establish a formal contrast with λ_b , it will in fact suffice to show a lower bound of
 1433 $\Omega(n2^n)$ on the *worst-case* runtime for generic count programs in λ_b . For this purpose, it is
 1434 convenient to focus on a specialized class of predicate terms that will be easy to work with.
 1435 We therefore declare that our intention is initially to analyse the runtime of any generic
 1436 count program in λ_b on any *canonical n -standard* predicate as in Definition 8. However, we
 1437 shall subsequently remark that in fact the same lower bound applies to arbitrary n -standard
 1438 predicates.

1439 Let us suppose, then, that K is a program of λ_b that correctly counts all canonical
 1440 n -standard predicates of λ_b for some specific n . We now establish a key lemma, which
 1441 vindicates the naïve intuition that if P is a canonical n -standard predicate, the only way for
 1442 K to discover the correct value for $\sharp[[P]]$ is to perform 2^n separate applications $P Q$ (allowing
 1443 for the possibility that these applications need not be performed ‘in turn’ but might be
 1444 nested in some complex way).

1445 **Lemma 1** (No shortcuts). *Suppose K correctly counts all n -standard predicates of λ_b . If P
 1446 is a canonical n -standard predicate, then K applies P to at least 2^n distinct n -points. More
 1447 formally, for any of the 2^n possible semantic n -points $\pi : \mathbb{N}_n \rightarrow \mathbb{B}$, there is a term $\mathcal{E}[P Q]$
 1448 appearing in the small-step reduction of $K P$ such that Q is an n -point and $[[Q]] = \pi$.*

1451 **Proof** Suppose π is some semantic n -point. Since P is canonical, we have $P = P(\tau)$ for
 1452 some τ . Let l be the maximal path through τ associated with π : that is, the one we construct
 1453 by responding to each query $?k$ with $\pi(k)$. Then l is a leaf node such that $\tau(l) = !(\tau \bullet \pi)$.
 1454 Let τ' be obtained from τ by simply negating this answer value at l , and take $P' = P(\tau')$.

1455 Since the numbers of true-leaves in τ and τ' differ by 1, it is clear that if K indeed correctly
 1456 counts all canonical n -standard predicates, then the values returned by $K P$ and $K P'$ will
 1457 have an absolute difference of 1. On the other hand, we shall argue that if the computation
 1458 of $K P$ never actually ‘visits’ the leaf l in question, then K will be unable to detect any
 1459 difference between P and P' . The situation is reminiscent of Milner’s *context lemma* (Milner,
 1460 1977), which loosely says that essentially the only way to observe a difference between two
 1461 programs is to apply them to some argument on which they differ.

1462 Without loss of generality we shall assume that $\tau(l) = \text{true}$ and $\tau'(l) = \text{false}$. This means
 1463 that for some term context $C[-]$ with a single occurrence of a hole of type Bool , we have
 1464 $P \equiv \lambda q. C[\text{true}]$ and $P' \equiv \lambda q. C[\text{false}]$.

1465 Now consider the reduction sequence starting from $K (\lambda q. C[-])$ (treating the hole ‘-’ as
 1466 an additional variable). This cannot be infinite, for then the reduction of $K P$ would also be
 1467 infinite, so K would not correctly count all canonical n -standard predicates. Neither can this
 1468 reduction terminate in a numeral c , for then both $K P$ and $K P'$ would evaluate to c , whereas
 1469 the correct results should differ by 1. We conclude that the reduction of $K (\lambda q. C[-])$ gets
 1470 stuck at some term with the hole in evaluation position. We write this term as $D[-]$, where
 1471

the $D[\]$ abstracts only this head occurrence of the hole (there may be other occurrences of the hole within D).

We now trace back through the reduction $K (\lambda q. C[-]) \rightsquigarrow^* L[-]$ looking at the ancestors of this head occurrence of ‘-’, and identifying the last point in the reduction at which this ancestor occurs under a binder λq . It is clear that at this point we have a term $\mathcal{E}[(\lambda q. C[-])Q]$ with \mathcal{E} an evaluation context, $C[\]$ a context abstracting only this ancestor occurrence of ‘-’, and Q a closed term of type Point. This reduces in the next step to $\mathcal{E}[E[-]]$ where $E[-] \equiv C[-][Q/q]$.

We now claim that Q is an n -point and $\llbracket Q \rrbracket = \pi$ as required. For this, we consider the reduction sequence starting from $E[-]$. Again, this cannot be infinite, as that would yield an infinite reduction for $\mathcal{E}[E[-]]$ and hence for $K P$. Nor can $E[-]$ evaluate to a numeral m , for then $\mathcal{E}[E[-]] \rightsquigarrow^* \mathcal{E}[m]$ where the latter contains no residuals of the ‘-’ in $E[-]$, so the designated hole in $\mathcal{E}[(\lambda q. C[-])Q]$ is not the ancestor of that in $D[-]$ after all. We conclude that $E[-] \rightsquigarrow^* F[-]$, where ‘-’ occurs in evaluation position in $F[-]$ and $\mathcal{E}[F[-]] \equiv D[-]$.

We now appeal to the fact that $P \equiv \lambda q. C[\text{true}]$ is canonical, so that $C[-]$ is simply a complex of nested **if**-expressions as in Definition 8, with a hole replacing the leaf literal at the position indicated by the path l . It is now clear that the only way for the hole in $E[-]$ to become exposed in evaluation position is for each of the branch conditions $Q(k)$ to evaluate to $\pi(k)$ so that the evaluation indeed follows the path l . But because τ is n -standard, each of $Q(0), \dots, Q(n-1)$ occurs exactly once on this path, so the above is exactly the condition for Q to be an n -point with value π . ■

Corollary 1. *Suppose K and P are as in Lemma 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 $\llbracket Q \rrbracket = \pi$.*

Proof Suppose $\pi \in \mathbb{B}^n$. By Lemma 1, the computation of $K P$ contains some $\mathcal{E}[P Q]$ where $\llbracket Q \rrbracket = \pi$, 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 $\mathcal{F}[-] \stackrel{\text{def}}{=} \mathcal{E}[\mathcal{E}'[-]]$. ■

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 $\llbracket Q \rrbracket$. 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 3. *If K is a λ_b program that correctly counts all canonical n -standard λ_b predicates, and P is any canonical n -standard λ_b predicate, then the evaluation of $K P$ must take time $\Omega(n2^n)$.* □

In Hillerström et al. (2020) a more complex proof was given, modelled on traditional proofs of Milner’s context lemma. This established the slightly stronger conclusion that the evaluation of $K P$ takes time $\Omega(n2^n)$ for *all* n -standard predicates P , not just the canonical ones (under the strengthened hypothesis that K correctly counts all n -standard λ_b predicates).

It is worth noting where our 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’ λ_do Branch $\langle \rangle$. Thus, it need no longer be the case that the reduction of each term Qk yields a value: it may get stuck at some invocation of ℓ , so that control will then pass to the effect handler.

9 Affine effect handlers

Having established our $\Omega(n2^n)$ runtime bound for implementations of generic count in the relatively simple setting of λ_b , we now wish to show that the same bound applies for a much richer language λ_a supporting *affine* effect handlers: intuitively those in which each resumption r may be invoked at most once. This will show that the multiple invocation of r within our effcount program is essential to its efficiency, and will formally locate the fundamental efficiency gap as occurring between λ_a and λ_h . Since affine effect handlers suffice for encoding many language features such as exceptions (Pretnar, 2015), local state (Plotkin and Pretnar, 2009), coroutines (Kawahara and Kameyama, 2020), and single-shot continuations (Sivaramakrishnan et al., 2021), this will come close to showing that the speedup we have discussed is unattainable in real languages such as Standard ML, Java, and Python (for some appropriate class of predicate terms).

In this section, we present the definition of our language λ_a , outlining its relationship to λ_h and λ_b and showing that it indeed suffices for expressing the features mentioned above. In the following section, we will prove some key properties of evaluation in this language, and use these to establish a version of Theorem 3 for λ_a .

Our language λ_a will be essentially a sublanguage of λ_h in which the relevant restriction on the use of resumption variables is enforced by means of an *affine type system* in the tradition of linear logic. Many approaches are possible here, for instance: Girard’s intuitionistic linear logic ILL (Girard, 1987), Barber’s dual intuitionistic linear logic DILL (Barber, 1996), and Benton’s adjoint calculus (Benton, 1994). We choose to work with a variant of fine-grain call-by-value based on DILL; an advantage over vanilla ILL is that it readily admits a local encoding of our intuitionistic base calculus.

9.1 λ_a as a dual intuitionistic-affine calculus

We present the type system of λ_a in terms of dual-context judgements of the form $\Delta; \Gamma \vdash \square : A$, stating that a term \square (which may be a value term V or a computation term M) has type A under *intuitionistic type environment* Δ and *affine type environment*. Informally, variables in the intuitionistic environment may be used zero, one or many times within \square , while those in the affine environment may be used at most once. The effect of this is that the intuitionistic environment admits both Weakening and Contraction, and the affine environment admits Weakening but not Contraction.

The syntax of λ_a is as follows:

1565	Types	$A, B, C, D ::= \text{Nat} \mid \text{Unit} \mid A \multimap B \mid A \otimes B \mid A \oplus B \mid !A$
1566	Type Environments	$\Gamma ::= \cdot \mid \Gamma, x : A$
1567	Handler types	$F ::= C \Rightarrow D$
1568	Values	$V, W ::= x \mid k \mid c \mid \lambda x^A. M \mid \mathbf{rec}^{f^{A \rightarrow B}} x.M$ $\mid \langle \rangle \mid \langle V, W \rangle \mid (\mathbf{inl} V)^B \mid (\mathbf{inr} W)^A \mid !W$
1569	Computations	$M, N ::= V W \mid \mathbf{let} \langle x, y \rangle = V \mathbf{in} N$ $\mid \mathbf{case} V \{ \mathbf{inl} x \mapsto M; \mathbf{inr} y \mapsto N \}$ $\mid \mathbf{return} V \mid \mathbf{let} x \leftarrow M \mathbf{in} N \mid \mathbf{let} !x = V \mathbf{in} N$ $\mid \mathbf{do} \ell V \mid \mathbf{handle} M \text{ With } H$
1570	Handlers	$H ::= \{ \mathbf{val} x \mapsto M \} \mid \{ \ell p r \mapsto N \} \uplus H$

The type constructors \multimap and $!$ are borrowed from linear logic. Here we informally understand $A \multimap B$ as a type of operations that may use their argument of type A at most once, and $!A$ as a type of values of type A that may be used as many times as desired.

The typing rules are given in Figure 9. To understand the workings of this type system, it is helpful to think of the linear arrow \multimap and the affine environment Γ as playing the primary role: for instance, lambda abstraction is supported for affine variables but not intuitionistic ones. The sole purpose of the intuitionistic environment is to allow for multiple uses of values of $!$ -type: the rule TL-LETBANG allows such a value to be bound to an intuitionistic variable. Note too that values of $!$ -type are formed via the TL-BANG rule, which allows a value $W : A$ to be ‘promoted’ to a reusable value $!W : !A$ if all free variables that went into the making of W are themselves reusable (we here write $!\Gamma$ to mean that every type in Γ is of the form $!A$ for some A).

The crucial restriction in the rule for handlers is that the resumption variable r is now affine. Notice that the return clause may involve affine variables as it will be invoked at most once (this idea will be substantiated in Section 10 below). By contrast, the operation clauses cannot involve affine variables as they may be invoked multiple times.

There is also a small subtlety with **rec**. The function argument f is bound in the intuitionistic type environment, allowing f to be used many times within M . The operational intention is that f can be unfolded to $\lambda x.M$ as often as necessary; for this reason, it is required that M involves no affine variables other than x .

All of the above syntactic forms are shared with λ_h , with the exception of $!W$ and **let** $!x = V$ **in** N . To give a small-step operational semantics for λ_a , we may therefore take all the operational rules for λ_h as given in Section 3 (along with the machinery of evaluation contexts and handler contexts), together with the new rule:

$$\text{S-BANG} \quad \mathbf{let} \ !x = !W \mathbf{in} \ N \rightsquigarrow N[W/x]$$

This completes our definition of λ_a .

9.2 Relationship to λ_h and λ_b

We now outline how we intend to view λ_a as a sublanguage of λ_h , and λ_b as a sublanguage of λ_a . A brief sketch will suffice here, as these translations will only play a minor role in what follows and are mentioned here mainly for the sake of orientation.

Values

$\frac{}{\Delta, x : A; \Gamma \vdash x : A}$	$\frac{}{\Delta; \Gamma, x : A \vdash x : A}$	$\frac{}{\Delta; \cdot \vdash \langle \rangle : \text{Unit}}$	$\frac{\text{TL-NAT}}{k \in \mathbb{N}} \frac{}{\Delta; \Gamma \vdash k : \text{Nat}}$
$\frac{\text{TL-CONST}}{c : A \multimap B} \frac{}{\Delta; \Gamma \vdash c : A \multimap B}$	$\frac{\text{TL-LAM}}{\Delta; \Gamma, x : A \vdash M : B} \frac{}{\Delta; \Gamma \vdash \lambda x^A. M : A \multimap B}$	$\frac{\text{TL-REC}}{\Delta, f : A \multimap B; x : A \vdash M : B} \frac{}{\Delta; \cdot \vdash \mathbf{rec}^{f^{A \multimap B}} x. M : A \multimap B}$	
$\frac{\text{TL-PROD}}{\Delta; \Gamma_1 \vdash V : A \quad \Delta; \Gamma_2 \vdash W : B} \frac{}{\Delta; \Gamma_1, \Gamma_2 \vdash \langle V, W \rangle : A \otimes B}$	$\frac{\text{TL-INL}}{\Delta; \Gamma \vdash V : A} \frac{}{\Delta; \Gamma \vdash (\mathbf{inl} V)^B : A \oplus B}$	$\frac{\text{TL-INR}}{\Delta; \Gamma \vdash W : B} \frac{}{\Delta; \Gamma \vdash (\mathbf{inr} W)^A : A \oplus B}$	
$\frac{\text{TL-BANG}}{\Delta; \Gamma \vdash W : A \quad !\Gamma} \frac{}{\Delta; \Gamma \vdash !W : !A}$			

Computations

$\frac{\text{TL-APP}}{\Delta; \Gamma_1 \vdash V : A \multimap B \quad \Delta; \Gamma_2 \vdash W : A} \frac{}{\Delta; \Gamma_1, \Gamma_2 \vdash V W : B}$			
$\frac{\text{TL-SPLIT}}{\Delta; \Gamma_1 \vdash V : A \otimes B \quad \Delta; \Gamma_2, x : A, y : B \vdash N : C} \frac{}{\Delta; \Gamma_1, \Gamma_2 \vdash \mathbf{let} \langle x, y \rangle = V \mathbf{in} N : C}$			
$\frac{\text{TL-CASE}}{\Delta; \Gamma_1 \vdash V : A \oplus B \quad \Delta; \Gamma_2, x : A \vdash M : C \quad \Delta; \Gamma_2, y : B \vdash N : C} \frac{}{\Delta; \Gamma_1, \Gamma_2 \vdash \mathbf{case} V \{ \mathbf{inl} x \mapsto M; \mathbf{inr} y \mapsto N \} : C}$			
$\frac{\text{TL-RETURN}}{\Delta; \Gamma \vdash V : A} \frac{}{\Delta; \Gamma \vdash \mathbf{return} V : A}$	$\frac{\text{TL-LET}}{\Delta; \Gamma_1 \vdash M : A \quad \Delta; \Gamma_2, x : A \vdash N : C} \frac{}{\Delta; \Gamma_1, \Gamma_2 \vdash \mathbf{let} x \leftarrow M \mathbf{in} N : C}$		
$\frac{\text{TL-LETBANG}}{\Delta; \Gamma_1 \vdash V : !A \quad \Delta, x : A; \Gamma_2 \vdash N : C} \frac{}{\Delta; \Gamma_1, \Gamma_2 \vdash \mathbf{let} !x = V \mathbf{in} N : C}$			
$\frac{\text{T-DO}}{(\ell : A \rightarrow B) \in \Sigma} \frac{}{\Delta; \Gamma \vdash \mathbf{do} \ell V : B}$	$\frac{\text{T-HANDLE}}{\Delta; \Gamma_1 \vdash M : C \quad \Delta; \Gamma_2 \vdash H : C \Rightarrow D} \frac{}{\Delta; \Gamma_1 \vdash \mathbf{handle} M \mathbf{with} H : D}$		

Handlers

$\frac{\text{T-HANDLER}}{\Delta; \Gamma, x : C \vdash M : D \quad [H^\ell = \{\ell p r \mapsto N_\ell\}]_{\ell \in \text{dom}(\Sigma)} \quad [\Delta; p : A_\ell, r : B_\ell \multimap D \vdash N_\ell : D]_{(\ell : A_\ell \rightarrow B_\ell) \in \Sigma}} \frac{}{\Delta; \Gamma \vdash H : C \Rightarrow D}$	
--	--

Fig. 9: Typing Rules for λ_a

For the inclusion $\lambda_a \hookrightarrow \lambda_h$, the broad idea is simply that any well-typed term of λ_a will certainly remain well-typed if all linearity restrictions on variables are waived. Formally, we may define a translation $(-)^{\dagger}$ that erases the intuitionistic/affine distinction completely. The translation on types may be defined by

$$\begin{aligned} \text{Nat}^{\dagger} &= \text{Nat} \\ \text{Unit}^{\dagger} &= \text{Unit} \\ (A \multimap B)^{\dagger} &= A^{\dagger} \rightarrow B^{\dagger} \\ (A \otimes B)^{\dagger} &= A^{\dagger} \times B^{\dagger} \\ (A \oplus B)^{\dagger} &= A^{\dagger} + B^{\dagger} \end{aligned}$$

The translation on terms is given in the obvious way for the syntactic forms common to λ_a and λ_h , the two new forms being treated by

$$\begin{aligned} (!W)^{\dagger} &= W^{\dagger} \\ (\mathbf{let} \ !x = V \ \mathbf{in} \ N)^{\dagger} &= \mathbf{let} \ x \leftarrow \mathbf{return} \ V^{\dagger} \ \mathbf{in} \ N^{\dagger} \end{aligned}$$

A typing judgement $\Delta; \Gamma \vdash \square : A$ of λ_a then becomes a judgement $(\Delta, \Gamma)^{\dagger} \vdash \square^{\dagger} : A^{\dagger}$, where Δ, Γ is the result of rolling Δ and Γ into a single environment, and $(\Delta, \Gamma)^{\dagger}$ is the result of applying $(-)^{\dagger}$ to the types of all its variables.

Under this translation, it is easy to check that every derivable typing judgement in λ_a yields one in λ_h , and also that if $M \rightsquigarrow M'$ in λ_a then $M^{\dagger} \rightsquigarrow M'^{\dagger}$ in λ_h .

For the inclusion $\lambda_b \hookrightarrow \lambda_a$, we give a translation $(-)^*$ based on the familiar Girard translation from intuitionistic types to linear ones, wrapping each subformula of a type by a '!' except for the return types of functions. The translation on types is as follows.

$$\begin{aligned} \text{Nat}^* &= \text{Nat} \\ \text{Unit}^* &= \text{Unit} \\ (A \rightarrow B)^* &= !(A^*) \multimap B^* \\ (A \times B)^* &= !(A^*) \otimes !(B^*) \\ (A + B)^* &= !(A^*) \oplus !(B^*) \\ (C \Rightarrow D)^* &= C^* \Rightarrow D^* \end{aligned}$$

A type environment Γ of λ_b translated to the environment Γ^* ; \cdot of λ_a : that is, all variables are treated as intuitionistic. The translation of value and computation terms therefore needs to eliminate '!' types at bindings in favour of intuitionistic variables. We give here a selection of the clauses for the translation on terms; for all syntactic forms not covered here, the translation is defined homomorphically on term structure in the obvious way.

$$\begin{aligned} (\lambda x^A . M)^* &= \lambda z^{!(A^*)} . \mathbf{let} \ !x = z \ \mathbf{in} \ M^* \\ (\mathbf{rec} \ f^{A \rightarrow B} \ x . M) &= \mathbf{rec} \ f^{!(A^*) \multimap B^*} \ z . \mathbf{let} \ !x = z \ \mathbf{in} \ M^* \\ (V \ W)^* &= V^* (! (W^*)) \\ (\mathbf{let} \ \langle x, y \rangle = V \ \mathbf{in} \ N)^* &= \mathbf{let} \ \langle x', y' \rangle = V^* \ \mathbf{in} \ \mathbf{let} \ !x = x' \ \mathbf{in} \ \mathbf{let} \ !y = y' \ \mathbf{in} \ N^* \\ (\mathbf{case} \ V \ \{\mathbf{inl} \ x \mapsto M; \mathbf{inr} \ y \mapsto N\}) &= \mathbf{case} \ V^* \ \{\mathbf{inl} \ x \mapsto \mathbf{let} \ !x = x' \ \mathbf{in} \ M^*; \\ &\quad \mathbf{inr} \ y \mapsto \mathbf{let} \ !y = y' \ \mathbf{in} \ N^*\} \\ (\mathbf{let} \ x \leftarrow M \ \mathbf{in} \ N)^* &= \mathbf{let} \ x' \leftarrow M^* \ \mathbf{in} \ \mathbf{let} \ !x = x' \ \mathbf{in} \ N^* \end{aligned}$$

The converse to the above proposition — that if $M \rightsquigarrow^m M''$ then $\langle M \mid \emptyset \rangle \rightsquigarrow^m \langle M' \mid \Xi \rangle$ for some M', Ξ — is not quite clear at this point, because of the worry that an application of S-RES might be blocked because the relevant \hat{r} is not present in $\text{dom } \Xi$, having been deleted by an earlier application of S-RES. We shall see shortly, however, that such blocking never happens, so that our two semantics do indeed work perfectly in lockstep.

Let us say a configuration $\langle M' \mid \Xi \rangle$ is *naturally arising* if it appears in the course of reduction of $\langle M \mid \emptyset \rangle$ for some closed M .

In the typing rules for λ_a , the critical typing restriction is that in the handler clauses $\ell p r \mapsto N_\ell$, the variable r is used *linearly* within N_ℓ . This does not mean that r can occur at most once within N_ℓ (in view of the TL-CASE rule); and even if it does, the variable r may subsequently be copied in the course of a β -reduction (again because of TL-CASE). However, the linearity restriction does buy us the following crucial property:

Lemma 2 (Single-shot resumptions). *For any naturally arising $\langle M \mid \Xi \rangle$ and any $\hat{r} \in \text{dom}(\Xi)$, the reduction sequence starting from $\langle M \mid \Xi \rangle$ contains at most one instance of S-RES for the variable \hat{r} .*

Proof Since $\langle M \mid \Xi \rangle$ is naturally arising, we have $\langle M_0 \mid \emptyset \rangle \rightsquigarrow^* \langle M \mid \Xi \rangle$ for some M_0 , and we may as well assume that $\langle M \mid \Xi \rangle$ appears as early as possible in this reduction, i.e. at the point where \hat{r} is introduced, so that M has the form $N[V/p, \hat{r}/r]$ as in the S-OP' rule.

We first claim that M, Ξ satisfy the following two conditions, writing r_0, \dots, r_{k-1} for the elements of $\text{dom}(\Xi)$.

1. \hat{r} appears in at most one of the $k + 1$ terms $M, \Xi(r_0), \dots, \Xi(r_{k-1})$.
2. If N is one of these terms and \hat{r} appears in N , then no two occurrences of \hat{r} within N share the same set of enclosing **case** clauses. (A **case** clause is a subphrase **inl** $x \mapsto P$ or **inr** $y \mapsto Q$ within a **case** expression.)

Condition 1 holds because \hat{r} is fresh and so does not appear in any of the $\Xi(r_i)$. Condition 2 is a general property of occurrences of linear variables within terms: An inspection of the typing rules shows that the TL-CASE rule is the only possible source of multiple occurrences of \hat{r} , and it is clear that if we know the set of enclosing **case** clauses then the occurrence is uniquely determined.

Next, we claim that Conditions 1 and 2 above are maintained as invariants by all the reduction rules of our new semantics. Since Condition 2 is a general property of linear variables, and our reduction rules are easily seen to respect the type system, the preservation of this condition is automatic, so it will suffice to show that Condition 1 is preserved. We reason by cases on the possible forms for a reduction $\langle M \mid \Xi \rangle \rightsquigarrow \langle M' \mid \Xi' \rangle$.

- For S-OP' (applied within some handler context \mathcal{H} and introducing a fresh \hat{r}'): Suppose \hat{r} appears within the relevant subterm **handle** $\mathcal{E}[\text{do } \ell V]$ **with** H (the situation for occurrences of \hat{r} elsewhere is straightforward). Since this subterm is in evaluation position, it is not within a **case** clause, so by Conditions 1 and 2 for $\langle M \mid \Xi \rangle$, there are no other occurrences of \hat{r} elsewhere, and all occurrences are within just one of $\mathcal{E}[\], V, H$. If they are within V , then because of the linearity of p within N , then \hat{r} may appear within the resulting term $N[V/p, \hat{r}'/r]$, but will not appear in the new

$\Xi'(\hat{r}')$ or elsewhere. If within $\mathcal{E}[\]$ or H , the occurrences of \hat{r} will all be moved to $\Xi'(\hat{r}')$, and there will be none elsewhere.

- For S-RES (applied to some subterm $\hat{r}'W$ within some \mathcal{H}): If \hat{r} occurs within W , it may appear within the resulting $R[W/y]$, but not elsewhere. If \hat{r} occurs within $\Xi'(\hat{r}')$ (i.e. within R), then it does not appear elsewhere in $\langle M \mid \Xi \rangle$. So after the application of S-RES and the deletion of \hat{r}' from Ξ , \hat{r} may appear within $R[W/y]$ but nowhere else.
- For the rules carried over from the original semantics (applied within some \mathcal{H}), the preservation of Condition 1 is immediate, since Ξ is unchanged.

To complete the proof, suppose that within the reduction sequence from some naturally arising $\langle M \mid \Xi \rangle$ we have an application of S-RES for a given variable \hat{r} : that is, we have some configuration $\langle \mathcal{H}[\hat{r}W] \mid \Xi' \rangle$. Since this satisfies Conditions 1 and 2, we see that the highlighted occurrence of \hat{r} is its only appearance with $\mathcal{H}[\hat{r}W]$ or the range of Ξ' , and it follows that \hat{r} does not appear at all within the resulting configuration $\langle \mathcal{H}[R[W/y]] \mid \Xi' \setminus \hat{r} \rangle$. There is therefore no danger of a later instance of S-RES for \hat{r} . ■

We can now lay to rest the worry mentioned earlier:

Proposition 3. (i) For any naturally arising $\langle M \mid \Xi \rangle$, all free variables appearing within M or any $\Xi(\hat{r})$ are contained in $\text{dom } \Xi$.

(ii) If $M \rightsquigarrow^m M''$ under the original rules, then $\langle M \mid \emptyset \rangle \rightsquigarrow^m \langle M' \mid \Xi \rangle$ for some M', Ξ .

Proof (i) The property in question clearly holds for initial configurations $\langle M \mid \emptyset \rangle$ with M closed, and it is easy to see that it is preserved by all reduction steps, given that S-RES completely expunges the variable \hat{r} as established above.

(ii) From (i) we know that an application of S-RES will never be blocked because the lookup $\Xi(\hat{r})$ fails. A reduction $M \rightsquigarrow^m M''$ can therefore be lifted to one $\langle M \mid \emptyset \rangle \rightsquigarrow^m \langle M' \mid \Xi \rangle$ by an easy induction on m . ■

Lemma 2 is the crucial property of λ_a on which our whole argument hinges. This property is flagrantly violated by λ_h , as illustrated by effcount with its essential use of multi-shot resumptions. Our next task is to show how, in view of Lemma 2, the evaluation of a given subterm may be tracked in a sequential way through a reduction sequence.

10.2 Tracking of active subterms

We shall say a subterm S of M is *active* if it occurs in an evaluation position, i.e. $M = \mathcal{H}[S]$ for some handler context S . We introduce the following concepts for tracking the evaluation of S through the reduction of M with respect to some resumption context Ξ .

Clearly, if $\langle M \mid \Xi \rangle$ is naturally arising and $M = \mathcal{H}[S]$, any reduction sequence $\langle S \mid \Xi \rangle \rightsquigarrow^* \langle S' \mid \Xi' \rangle$ will yield a reduction sequence

$$\langle M \mid \Xi \rangle \equiv \langle \mathcal{H}[S] \mid \Xi \rangle \rightsquigarrow^* \langle \mathcal{H}[S'] \mid \Xi' \rangle$$

We then say the occurrence of S' highlighted by $\mathcal{H}[S']$ is an *active reduct* of the original occurrence of S highlighted by $\mathcal{H}[S]$.

In this situation, there are three possibilities:

1. The reduction of $\langle S \mid \Xi \rangle$ may continue forever.
2. The reduction of $\langle S \mid \Xi \rangle$ may terminate in some $\langle V \mid \Xi' \rangle$ where V is a closed value.
3. The reduction may get stuck at some configuration $\langle \mathcal{E}[\mathbf{do} \ell V] \mid \Xi' \rangle$, where the $\mathbf{do} \ell V$ is not handled within $\mathcal{E}[\mathbf{do} \ell V]$ itself.

In case 3, $\mathcal{H}[-]$ will have the form $\mathcal{H}'[\mathbf{handle} \mathcal{F} - \mathbf{with} H]$ with \mathcal{F} an evaluation context, and the S-OP' rule will then apply to $\langle \mathbf{handle} \mathcal{F}[\mathcal{E}[\mathbf{do} \ell V]] \mathbf{with} H \mid \Xi' \rangle$. This will result in a new resumption environment entry

$$\hat{r} \mapsto \lambda y. \mathbf{handle} \mathcal{F}[\mathcal{E}[\mathbf{return} y]] \mathbf{with} H$$

and we may call the subterm $\mathcal{E}[\mathbf{return} y]$ here a *dormant reduct* of the original S .

As the reduction of the original $\langle M \mid \Xi \rangle$ continues, this environment entry will remain unaffected until, if ever, \hat{r} is activated by S-RES (and by Lemma 2, this will happen at most once). This activation step will have the form

$$\langle \mathcal{H}'_1[\hat{r}W] \mid \Xi_1 \rangle \rightsquigarrow \langle \mathcal{H}'_1[\mathbf{handle} \mathcal{F}[\mathcal{E}[\mathbf{return} W]] \mathbf{with} H] \mid \Xi_1 \rangle$$

where $\mathcal{H}'_1[\mathbf{handle} \mathcal{F}[-] \mathbf{with} H]$ is itself a handler context, which we shall write as $\mathcal{H}_1[-]$ for compatibility with our earlier convention. So writing S_1 for $\mathcal{E}[\mathbf{return} W]$, we have arrived at

$$\langle M \mid \Xi \rangle \rightsquigarrow^* \langle \mathcal{H}_1[S_1] \mid \Xi_1 \rangle,$$

and we shall again designate this occurrence of S_1 as an *active reduct* of the original S .

For the purpose of tracking the fate of the original subterm S , it will also be convenient also to say that $\langle S \mid \Xi \rangle$ gives rise in this context to a *pseudo-reduction sequence*

$$\langle S \mid \Xi \rangle \rightsquigarrow^* \langle \mathcal{E}[\mathbf{do} \ell V] \mid \Xi' \rangle \rightsquigarrow^! \langle S_1 \mid \Xi_1 \rangle$$

in which all steps but the last are genuine reductions, but the step flagged by $\rightsquigarrow^!$ is considered as a ‘pseudo-reduction’ (note that this step has a seemingly ‘non-deterministic’ character in that it depends crucially on information from outside $\langle \mathcal{E}[\mathbf{do} \ell V] \mid \Xi' \rangle$). The point is simply to have a way of saying how the evaluation of $\mathcal{E}[\mathbf{do} \ell V]$ continues after being temporarily suspended by a switch to another thread of control.

We may now repeat exactly the same procedure starting from $\langle \mathcal{H}_1[S_1] \mid \Xi_1 \rangle$, potentially yielding further (active and dormant) reducts of the original S :

$$\langle S \mid \Xi \rangle \rightsquigarrow^* \langle \mathcal{E}[\mathbf{do} \ell V] \mid \Xi' \rangle \rightsquigarrow^! \langle S_1 \mid \Xi_1 \rangle \rightsquigarrow^* \langle \mathcal{E}_1[\mathbf{do} \ell_1 V_1] \mid \Xi'_1 \rangle \rightsquigarrow^! \langle S_2 \mid \Xi_2 \rangle \rightsquigarrow^* \dots$$

In this way, we obtain an extended pseudo-reduction sequence for S , consisting of ordinary reduction sequences interspersed with pseudo-reductions of the above kind, jumping straight from some $\langle \mathcal{E}_i[\mathbf{do} \ell_i V_i] \mid \Xi_i \rangle$ to the corresponding $\langle \mathcal{E}_i[\mathbf{return} W_i] \mid \Xi_{i+1} \rangle$.

This pseudo-reduction sequence may continue forever, or it may end because an instance of \mathbf{do} gives rise to a resumption that is never subsequently invoked, or it may terminate in some $\langle V \mid \Xi'_i \rangle$ where V is a closed value. In the last case, we say the evaluation of the original S *completes* (in the context of $\langle \mathcal{H}[S] \mid \Xi \rangle$).

It is thanks to Lemma 2 that the evaluation behaviour of S may be represented in this way by a single linear reduction sequence rather than by a branching tree. The notion of

pseudo-reduction sequence thus allows us to reason about subterm evaluations much as in the familiar setting, rendering the thread-switching machinery largely transparent, its only trace being in the ‘non-deterministic’ character of the pseudo-reduction steps.

10.3 Application to generic count

We now apply the above notions to the analysis of generic counting in λ_a , obtaining a lower bound analogous to that of Theorem 3 for λ_b . Proceeding as in Section 8, we fix $n \in \mathbb{N}$, and suppose that K is some program of λ_a that correctly counts all canonical n -standard predicates P (noting that all such predicates are actually terms from our base language λ_b). Once again, focussing on this restricted class of predicates will greatly simplify our task, while still giving all we need for a worst-case lower bound.

We recall here that we are thinking of λ_b as included in λ_a via the *intuitionistic encoding* \star defined in Section 9. Since our intention is that our lower bound for λ_a should generalize the one for λ_b , this means that the types Point and Predicate appear within λ_a as

$$\text{Point} \stackrel{\text{def}}{=} !\text{Nat} \multimap \text{Bool}, \quad \text{Predicate} \stackrel{\text{def}}{=} !\text{Point} \multimap \text{Bool}$$

Formally, then, we will be considering the reduction behaviour of $\langle K (!P) \mid \emptyset \rangle$, where $P : \text{Predicate}$ is the \star -translation of a canonical n -standard predicate, K is a generic count program of λ_a assumed to count all such predicates correctly. By hypothesis, this reduction will terminate in some $\langle \text{return } c \mid \Xi_{\text{end}} \rangle$ where c is a numeral.

We take it to be clear what is meant by a *residual* of the predicate P occurring anywhere in this reduction — noting that such residuals may appear both in the term and in the resumption environment, and may proliferate without restriction. We suppress the tedious formal definition.

By an *application of P* , we shall mean an occurrence of a term $P Q$ in evaluation position in some reduct of $\langle K (!P) \mid \emptyset \rangle$ (so that $\langle K (!P) \mid \emptyset \rangle \rightsquigarrow^* \langle \mathcal{H}[P Q] \mid \Xi_0 \rangle$ for some handler context \mathcal{H}), where we require that the P in $P Q$ is a residual of the original P . As a significant consequence of our typing of P , the argument Q here will be of type $!\text{Point}$, meaning that no resumption variable \hat{r} may appear in Q (recall that resumption variables are not of $!$ -type). However, such terms Q may well exhibit effectful behaviour in other ways: they may contain **do** invocations to be handled elsewhere within K , and they may contain their own **handle** expressions.

For any such application of P , we may consider the pseudo-reduction sequence for $P Q$ arising from the reduction of $\langle \mathcal{H}[P Q] \mid \Xi_0 \rangle$. In view of the special form of canonical predicates, it is clear that this pseudo-reduction sequence will have the following form, or some initial segment thereof:

$$\begin{aligned} \langle P Q \mid \Xi_0 \rangle &\rightsquigarrow^* \langle \mathcal{E}_0[Q k_0] \mid \Xi_0 \rangle && \rightsquigarrow^{*,!} \langle \mathcal{E}_0[b_0] \mid \Xi_1 \rangle \\ &\rightsquigarrow^* \langle \mathcal{E}_1[Q k_1] \mid \Xi_1 \rangle && \rightsquigarrow^{*,!} \langle \mathcal{E}_1[b_1] \mid \Xi_2 \rangle \\ &\dots && \\ &\rightsquigarrow^* \langle \mathcal{E}_{n-1}[Q k_{n-1}] \mid \Xi_{n-1} \rangle && \rightsquigarrow^{*,!} \langle \mathcal{E}_{n-1}[b_{n-1}] \mid \Xi_n \rangle \\ &\rightsquigarrow^* \langle \text{return } b \mid \Xi_n \rangle \end{aligned}$$

We think of this as a sequence of ‘ P -phases’ alternating with ‘ Q -phases’. The former are the portions designated by \rightsquigarrow^* : these are very short sequences of genuine reductions

concerned with the evaluation of the canonical n -standard predicate P in this instance. The latter are the portions designated by $\rightsquigarrow^{*,!}$, which may mix genuine reduction steps with pseudo ones. Clearly, no Q -phase can run forever, since the whole computation $\langle K (!P) \mid \emptyset \rangle \rightsquigarrow^* \langle \text{return } c \mid \Xi_{\text{end}} \rangle$ is finite. Nevertheless, it is possible that a computation for $P Q$ may hang because one of the Q -phases is suspended by a **do** operation and never thereafter resumed. We say an application of P is *successful* if it evaluates all the way to some boolean b as indicated above.

In the case of a successful application, we see that the P -phases consist of precisely the reductions that feature in the definition of the tree $\mathcal{U}(P)$: in the notation of the above reduction scheme, the computation is tracing out the path $b_0 \dots b_{n-1}$ through this tree. Bearing in mind that P is assured to be n -standard, we may conclude that $\{k_0, \dots, k_{n-1}\} = \{0, \dots, n-1\}$ and that $\mathcal{U}(P)(b_0 \dots b_{n-1}) = !b$.

We may now, ‘with hindsight’, identify the semantic n -point to which P was in effect applied: namely, the point π given by $\pi(k_i) = b_i$ for each i . Notice that in the setting of λ_b , this semantic point could be read off at the point of application simply as $\llbracket Q \rrbracket$; however, this is not possible here, since the behaviour of **do** operations within Q need not be determined by Q itself, and indeed may vary according to the context in which Q appears. Nonetheless, in the above situation, it is convenient to refer to our $P Q$ as an *application of P to π* .

The crucial claim is now the following:

Lemma 3. *For each of the 2^n semantic n -points π , the reduction of $\langle K (!P) \mid \emptyset \rangle$ contains an application of P to π .*

Proof Essentially the same argument as for Lemma 1. Given any semantic point π , we may identify the corresponding leaf literal within the body of P , and write $C[-]$ for the context that abstracts on this leaf literal. Assuming without loss of generality that this literal is true, we then have $P \equiv \lambda q. C[\text{true}]$, and we also set $P' \equiv \lambda q. C[\text{false}]$.

We now consider the reduction sequence from $\langle K (\lambda q. C[-]) \mid \emptyset \rangle$, carrying the hole ‘-’ through the computation. Just as in Lemma 1, we argue that this hole must at some stage reach evaluation position, and that this must happen during the evaluation of a certain subterm $(\lambda q. C[-])Q$. (The same argument applies here because we are tracking subterms through genuine reductions, not pseudo-reductions.)

In the present setting, we cannot conclude that $\llbracket Q \rrbracket = \pi$, but even so, the same argument shows that (replacing ‘-’ by true) we have a successful application of P to π . ■

Theorem 4. *If K is a λ_a program that correctly counts all canonical n -standard λ_b predicates, and P is an canonical n -standard λ_b predicate, then the evaluation of $K P$ takes time $\Omega(n2^n)$.*

Proof It is clear from the above analysis that each successful application of P involves, at least, the n reductions of the subterms $\mathcal{E}_i[b_i]$. To complete the proof, we just need to check that none of these reductions can be shared by two applications of P associated with different semantic points.

Suppose, then that we have such a reduction within a successful application of P , appearing within the overall computation of $K (!P)$. We note that this reduction step contracts some

subterm **if** b_i **then** R **else** R' to R or R' as appropriate (or rather, we have a short sequence of reductions that performs the desugared version of this), and this serves to identify the subterm $\mathcal{E}_i[b_i]$ of relevance. Since we know that this subterm appears as part of a successful application of P , we may track the fate of this subterm by a pseudo-reduction sequence, and will be assured that this terminates in some **return** b as indicated above. Furthermore, this occurrence of b will be a residual of some leaf literal within P , and the location of this leaf will determine the point π to which P was applied. There is therefore no possibility that one of these steps can belong to applications of P to distinct points π, π' . ■

As with Theorem 3, one may also obtain a variant of this theorem with both occurrences of ‘canonical’ omitted, at the cost of significant extra complication in the proof.

Taken together, Theorems 4 and 2 highlight the fundamental efficiency gap between λ_a and λ_h .

11 Experiments

The theoretical efficiency gap between realisations of λ_b and λ_h manifests in practice. We report here on runtime experiments undertaken in OCaml involving two search-like problems: the familiar n -queens problem, and the problem of computing definite integrals of mathematical functions in the setting of exact real-number computation. Our work here builds on earlier experiments described by Daniels (2016).

Tables 1 and 2 show the speedup from using an effectful implementation of generic search over various other implementations. We discuss the benchmarks and results in further detail below.

Methodology For both our search problems, we evaluated an effectful implementation of generic search against three other implementations:

- Naïve: a pure functional implementation of the simple procedure described in Section 7.1,
- Berger: a lazy pure functional generic search procedure as outlined in Section 7.3.
- Pruned: the pruned search procedure of Section 7.4, using a Modulus operator implemented using local state. This is essentially the best currently known approach to generic search that does not involve advanced control features.

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 (3 minutes for single solution, 8 minutes for enumerations), are reported as $-$. The experiments were conducted using, at the time of writing, latest stock OCaml 5.0.0-beta1 with factory settings on an AMD Ryzen 9 5900X 12-core CPU powered workstation running Ubuntu 20.04.

OCaml supports only single-shot continuations out-of-the-box, but the effectful procedure requires multi-shot continuations to function correctly. Therefore, we use the package `multicont 1.0.0-rc.3`, which provides facilities for programming with multi-shot continuations in OCaml. Under the hood the library opaquely performs a copy-on-invoke of a regular single-shot continuation such that an additional copy of the continuation is available

Parameter	First solution			All solutions		
	20	24	28	8	10	12
Naïve	–	–	–	365.76	6633.47	–
Berger	13.89	21.72	31.83	3.91	3.51	3.18
Pruned	3.75	4.90	5.86	1.75	1.99	1.97
Bespoke	0.24	0.28	0.30	0.24	0.21	0.22

Table 1: Runtime of the n -Queens procedures relative to the effectful implementation

later. In contrast to OCaml’s continuations, the multi-shot continuations provided by this package are garbage collected. The package can be installed directly via OPAM or built from source. The interested reader may see the following hyperlink more information:

<https://github.com/dhil/ocaml-multicont>

The complete source code and data for the benchmarks are available at:

<https://github.com/dhil/asymptotic-speedup-via-effect-handlers-code-jfp>

Queens The classic n -queens problem can be directly cast as a search problem of the kind considered in Section 5, mildly generalising so as to allow searches over the space $\mathbb{N}_n \rightarrow \mathbb{N}_n$ rather than just $\mathbb{N}_n \rightarrow \mathbb{B}$: a potential solution or ‘point’ corresponds to a vector (q_0, \dots, q_{n-1}) where q_i is the row of the unique queen in the i th column. We evaluated four implementations of generic search, and as a control we also included a bespoke implementation hand-optimised for the problem. We performed two experiments: finding the first solution for $n \in \{20, 24, 28\}$ and enumerating all solutions for $n \in \{8, 10, 12\}$.

As expected, the naïve implementation performs very poorly indeed. The Berger procedure is in some cases more competitive, and the pruned procedure even more so, but still slower than the baseline effectful version. Unsurprisingly, the baseline is significantly slower than the bespoke implementation.

Exact real integration Our second problem involves a more elaborate application of the same ideas, this time to a ‘search’ over all paths through a tree of infinite depth (though still finitely branching). This example was one of our main inspirations, and it was by simplifying and distilling the phenomena arising here that we were led to the formulation of generic search in the finite setting and to the main results of this paper.

Glossing over several points, the idea is as follows. Let us represent real numbers in the interval $[0, 1]$ by streams of binary digits, where we think of all possible such streams as paths through the full infinite binary tree, and let us represent a continuous mathematical function $f : [0, 1] \rightarrow [0, 1]$ by a function on such streams. Our task is to evaluate the definite integral $\int_0^1 f$ to within any specified error bound $\varepsilon > 0$. Informally, we can achieve this if for every $x \in [0, 1]$ we know the value of $f(x)$ to within ε . We may therefore proceed as follows. First we evaluate f to the required precision at the real 0, represented by the stream $0, 0, 0, \dots$. We will obtain a result v_0 having consumed finitely many digits of the input stream, say k of them. Since we have not looked at any of the subsequent digits, this actually tells us not only that $|f(0) - v| < \varepsilon$, but that $|f(x) - v| < \varepsilon$ for any $x \in [0, 2^{-k}]$. This

Parameter	Id	Squaring			Logistic				
	20	14	17	20	1	2	3	4	5
Naïve	6.58	18.22	22.38	27.28	23.44	63.75	36.67	–	–
Berger	3.62	7.67	7.83	8.34	8.76	11.98	11.67	12.02	12.62
Pruned	1.25	1.67	1.54	1.60	1.70	2.51	2.20	3.52	3.84

Table 2: Runtime of exact real integration procedures relative to the effectful implementation gives us a contribution of $v \cdot 2^{-k}$ towards our integral. We therefore continue by evaluating f to the required precision at the right endpoint of this interval, represented by the stream $0^{k-1}, 1, 0, 0, 0 \dots$. This will return a result after consuming say k' digits, giving us the desired information about $f(x)$ for every $x \in [2^{-k}, 2^{-k} + 2^{-k'}]$. We continue creeping along the unit interval in this way until we reach the endpoint 1. (Since the computation of f to any required precision is assumed to succeed for any input stream, computable or otherwise, it follows by König's Lemma (König, 1927) that we will indeed reach 1 after a finite number of steps.) At this point, we have enough information to return the value of the integral to within ϵ .

The relation to the concerns of this paper should now be apparent. We are wishing to implement a general integration operator that performs the above parametrically in any given f (or more precisely in any given representation of such an f by a function on streams). In a language without advanced control features, we essentially have no option but to start the evaluation of f afresh on each new stream: there is no way to take advantage of the fact that the evaluation on $0, 0, 0, \dots$ and on $0^{k-1}, 1, 0, 0, 0 \dots$ will proceed identically up to the point where the k th input digit is examined. With general effect handlers, however, such an optimization is indeed possible, much as we have explained in this paper — the main difference being that we are now traversing a tree of infinite depth, and we have no bound in advance on the depth to which we will be required to explore.

For the sake of simplicity, we have here sketched the idea as though reals were represented by ordinary binary sequences. It is well known, however, that ordinary binary representations are inadequate for the purpose of exact real computation, and a common alternative (see e.g. Wiedmer (1980)) is to work instead with streams of *signed binary* digits $-1, 0, 1$, meaning that every real number has multiple representations. This somewhat complicates the details of how integrals are computed, but does not affect the essential idea.

Our 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 $3.62 \times$. For the squaring function the speedups over the Berger procedure are between 7 and $8 \times$, whereas the pruned procedure remains more competitive as the effectful procedure provides only a modest speedup of roughly $1.50 \times$. More significant speedups are achieved when we integrate the logistic map $x \mapsto 1 - 2x^2$ at a fixed precision of 15. The achieved speedup generally gets better as we make the function harder to compute by iterating it up to 5 times — with the exception of naïve implementation at iteration 3, which surprisingly performs significantly better than at iteration 2. Between the Berger and effectful procedures, and the pruned and effectful procedures the speedup ratios steadily increase as the function becomes harder to compute.

12 Conclusions and future work

We presented a PCF-inspired language λ_b , an extension λ_h with general effect handlers, and a milder extension λ_a with affine effect handlers. We proved that λ_h supports an asymptotically more efficient implementation of generic search than any possible implementation in λ_b or even λ_a . We observed this effect in practice on several benchmarks. Since λ_a is powerful enough to encode features such as exceptions, local state and coroutines, our results strongly suggest that the speedup we have discussed is unattainable in languages such as Standard ML, Java and Python.

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). We have also indicated in Section 6.3 how the same speedup may also be obtained in the presence of a type-and-effect system.

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.

Finally, we have suggested that our gap between λ_a and λ_h can be seen as an instance of a much more general phenomenon, whereby the attainable efficiency for performing some task may vary according to the expressivity of the programming language. Thus, in the case of generic counting for n -predicates:

- In λ_i (a language with iteration but not recursion), one cannot systematically (and uniformly in n) achieve either ‘false’ pruning or ‘true’ pruning.
- In λ_b , we may systematically and uniformly achieve ‘false’ pruning but not ‘true’ pruning (Bergercount).
- In λ_a , we may systematically achieve both ‘false’ and ‘true’ pruning (prunedcount), but no sharing of computations is possible.
- In λ_h , pruning and sharing of computations are systematically possible (effcount).

However, in this paper we have established only to the last of these gaps with mathematical rigour. The others having been discussed only informally in Section 7. We believe that using methods similar to those of this paper, it should be possible to formulate and prove precise statements pinning down the other differences, thus giving mathematical substance to the above claims. This wider programme of examining the language expressivity spectrum through the lens of algorithmic complexity seems to us worthy of significant further attention.

Acknowledgements 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. Specifically, we are grateful to one of the reviewers for the encouragement to establish a lower bound result for arbitrary affine effects.

Daniel Hillerström and Sam Lindley were supported by the UKRI Future Leaders Fellowship “Effect Handler Oriented Programming” (reference number MR/T043830/1).

References

- Barber, A. (1996) Dual Intuitionistic Linear Logic. Technical report ECS-LFCS-96-347, University of Edinburgh.
- Bauer, A. (2018) What is algebraic about algebraic effects and handlers? *CoRR*. **abs/1807.05923**.
- Bauer, A. & Pretnar, M. (2014) An effect system for algebraic effects and handlers. *Log. Methods Comput. Sci.* **10**(4).
- Bauer, A. & Pretnar, M. (2015) Programming with algebraic effects and handlers. *J. Log. Algebr. Meth. Program.* **84**(1), 108–123.
- Bell, J. & Stevens, B. (2009) A survey of known results and research areas for n-queens. *Discret. Math.* **309**(1), 1–31.
- Benton, N. & Kennedy, A. (2001) Exceptional syntax journal of functional programming. *J. Funct. Program.* **11**(4), 395–410.
- Benton, P. (1994) A mixed linear and non-linear logic: Proofs, terms and models. CSL. Springer. pp. 121–135.
- Berger, U. (1990) *Totale Objekte und Mengen in der Bereichstheorie*. Ph.D. thesis. Ludwig Maximillians-Universität. Munich.
- Biernacki, D., Piróg, M., Polesiuk, P. & Sieczkowski, F. (2019) Abstracting algebraic effects. *PACMPL*. **3**(POPL), 6:1–6:28.
- Biernacki, D., Piróg, M., Polesiuk, P. & Sieczkowski, F. (2020) Binders by day, labels by night: effect instances via lexically scoped handlers. *PACMPL*. **4**(POPL), 48:1–48:29.
- Bird, R., Jones, G. & de Moor, O. (1997) More haste less speed: lazy versus eager evaluation. *J. Funct. Program.* **7**(5), 541–547.
- Bird, R. S. (2006) Functional pearl: A program to solve sudoku. *J. Funct. Program.* **16**(6), 671–679.
- Brachthäuser, J. I., Schuster, P. & Ostermann, K. (2020) Effekt: Capability-passing style for type- and effect-safe, extensible effect handlers in scala. *J. Funct. Program.* **30**, e8.
- Cartwright, R. & Felleisen, M. (1992) Observable sequentiality and full abstraction. POPL. ACM Press. pp. 328–342.
- Convent, L., Lindley, S., McBride, C. & McLaughlin, C. (2020) Doo bee doo bee doo. *J. Funct. Program.* **30**. To appear.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C. (2009) *Introduction to Algorithms, Third Edition*. MIT Press. third edition.
- Daniels, R. (2016) *Efficient Generic Searches and Programming Language Expressivity*. Master’s thesis. School of Informatics, the University of Edinburgh. Scotland.
- Danvy, O. & Filinski, A. (1990) Abstracting control. LISP and Functional Programming. ACM. pp. 151–160.
- Dolan, S., White, L., Sivaramakrishnan, K., Yallop, J. & Madhavapeddy, A. (2015) Effective concurrency through algebraic effects. OCaml Workshop.
- Escardó, M. H. (2007) Infinite sets that admit fast exhaustive search. LICS. IEEE Computer Society. pp. 443–452.
- Farvardin, K. & Reppy, J. H. (2020) From folklore to fact: comparing implementations of stacks and continuations. PLDI. ACM. pp. 75–90.
- Felleisen, M. (1987) *The Calculi of Lambda-nu-cs Conversion: A Syntactic Theory of Control and State in Imperative Higher-order Programming Languages*. Ph.D. thesis. Department of Computer Science. Indianapolis, IN, USA. AAI8727494.
- Felleisen, M. (1988) The theory and practice of first-class prompts. POPL. ACM Press. pp. 180–190.

- 2209 Felleisen, M. (1991) On the expressive power of programming languages. *Sci. Comput. Prog.* **17**(1–3),
2210 35–75.
- 2211 Felleisen, M. & Friedman, D. P. (1987) Control operators, the SECD-machine, and the λ -calculus.
2212 The Proceedings of the Conference on Formal Description of Programming Concepts III, Ebberup,
2213 Denmark. Elsevier. pp. 193–217.
- 2214 Flanagan, C., Sabry, A., Duba, B. F. & Felleisen, M. (1993) The essence of compiling with
2215 continuations. PLDI. ACM. pp. 237–247.
- 2216 Flatt, M. & Dybvig, R. K. (2020) Compiler and runtime support for continuation marks. PLDI. ACM.
2217 pp. 45–58.
- 2218 Forster, Y., Kammar, O., Lindley, S. & Pretnar, M. (2019) On the expressive power of user-defined
2219 effects: Effect handlers, monadic reflection, delimited control. *J. Funct. Program.* **29**, e15.
- 2220 Girard, J.-Y. (1987) Linear logic. *Theor. Comp. Sci.* **50**, 1–101.
- 2221 Hillerström, D. & Lindley, S. (2016) Liberating effects with rows and handlers. TyDe@ICFP. ACM.
2222 pp. 15–27.
- 2223 Hillerström, D. & Lindley, S. (2018) Shallow effect handlers. APLAS. Springer. pp. 415–435.
- 2224 Hillerström, D., Lindley, S. & Atkey, R. (2020) Effect handlers via generalised continuations. *J. Funct.*
2225 *Program.* **30**, e5.
- 2226 Hillerström, D., Lindley, S., Atkey, R. & Sivaramakrishnan, K. C. (2017) Continuation passing style
2227 for effect handlers. FSCD. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. pp. 18:1–18:19.
- 2228 Hillerström, D., Lindley, S. & Longley, J. (2020) Effects for efficiency: Asymptotic speedup with
2229 first-class control. *Proc. ACM Program. Lang.* **4**(ICFP), 100:1–100:29.
- 2230 Hillerström, D., Lindley, S. & Longley, J. (2020) Effects for efficiency: Asymptotic speedup with
2231 first-class control. *CoRR*. **abs/2007.00605**.
- 2232 Hillerström, D. (2021) *Foundations for Programming and Implementing Effect Handlers*. Ph.D. thesis.
2233 The University of Edinburgh, Scotland, UK.
- 2234 Hughes, J. (1986) A novel representation of lists and its application to the function "reverse". *Inf.*
2235 *Process. Lett.* **22**(3), 141–144.
- 2236 Jones, N. (2001) The expressive power of higher-order types, or, life without CONS. *J. Funct. Program.*
2237 **11**, 5–94.
- 2238 Kammar, O., Lindley, S. & Oury, N. (2013) Handlers in action. ICFP. ACM. pp. 145–158.
- 2239 Kawahara, S. & Kameyama, Y. (2020) One-shot algebraic effects as coroutines. TFP. Springer. pp.
2240 159–179.
- 2241 Kiselyov, O., Sabry, A. & Swords, C. (2013) Extensible effects: an alternative to monad transformers.
2242 Haskell. ACM. pp. 59–70.
- 2243 Kiselyov, O., Shan, C., Friedman, D. P. & Sabry, A. (2005) Backtracking, interleaving, and terminating
2244 monad transformers: (functional pearl). ICFP. ACM. pp. 192–203.
- 2245 Knuth, D. (1997) *The Art of Computer Programming, Volume 1: Fundamental Algorithms (third*
2246 *edition)*. Addison-Wesley.
- 2247 König, D. (1927) Über eine Schlussweise aus dem Endlichen ins Unendliche. *Acta Sci. Math. (Szeged)*.
2248 **3**(2–3), 121–130.
- 2249 Launchbury, J. & Jones, S. L. P. (1994) Lazy functional state threads. PLDI. ACM. pp. 24–35.
- 2250 Leijen, D. (2017) Type directed compilation of row-typed algebraic effects. POPL. ACM. pp. 486–499.
- 2251 Levy, P. B., Power, J. & Thielecke, H. (2003) Modelling environments in call-by-value programming
2252 languages. *Inf. Comput.* **185**(2), 182–210.
- 2253 Lindley, S., McBride, C. & McLaughlin, C. (2017) Do be do be do. POPL. ACM. pp. 500–514.
- 2254 Longley, J. (1999) When is a functional program not a functional program? ICFP. ACM. pp. 1–7.
- 2255 Longley, J. (2018) The recursion hierarchy for PCF is strict. *Logical Methods in Comput. Sci.* **14**(3:8),
2256 1–51.
- 2257 Longley, J. (2019) Bar recursion is not computable via iteration. *Computability*. **8**(2), 119–153.
- 2258 Longley, J. & Normann, D. (2015) *Higher-Order Computability*. Theory and Applications of
2259 Computability. Springer.
- 2260 McCracken, N. (1984) The typechecking of programs with implicit type structure. Semantics of Data
2261 Types. Berlin, Heidelberg. Springer Berlin Heidelberg. pp. 301–315.
- 2262
- 2263
- 2264

- Milner, R. (1977) Fully abstract models of typed λ -calculi. *Theor. Comput. Sci.* **4**(1), 1–22.
- MLton. (2020) MLton website.
- Moggi, E. (1991) Notions of computation and monads. *Inf. Comput.* **93**(1), 55–92.
- Okasaki, C. (1999) *Purely functional data structures*. Cambridge University Press.
- Pippenger, N. (1996) Pure versus impure lisp. *POPL*. ACM. pp. 104–109.
- Piróg, M., Polesiuk, P. & Sieczkowski, F. (2019) Typed equivalence of effect handlers and delimited control. *FSCD*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. pp. 30:1–30:16.
- Plotkin, G. (1977) LCF considered as a programming language. *Theor. Comput. Sci.* **5**(3), 223–255.
- Plotkin, G. D. & Power, J. (2001) Adequacy for algebraic effects. *FoSSaCS*. Springer. pp. 1–24.
- Plotkin, G. D. & Pretnar, M. (2009) Handlers of algebraic effects. *ESOP*. Springer. pp. 80–94.
- Plotkin, G. D. & Pretnar, M. (2013) Handling algebraic effects. *Logical Methods in Computer Science.* **9**(4).
- Pretnar, M. (2015) An introduction to algebraic effects and handlers. *Electr. Notes Theor. Comput. Sci.* **319**, 19–35. Invited tutorial paper.
- Simpson, A. K. (1998) Lazy functional algorithms for exact real functionals. *MFCS*. Springer. pp. 456–464.
- Sivaramakrishnan, K. C., Dolan, S., White, L., Kelly, T., Jaffer, S. & Madhavapeddy, A. (2021) Retrofitting effect handlers onto ocaml. *PLDI*. ACM. pp. 206–221.
- Sperber, M., Dybvig, K. R., Flatt, M., van Stratten, A., Findler, R. B. & Matthews, J. (2009) Revised⁶ report on the algorithmic language Scheme. *J. Funct. Program.* **19**(S1), 1–301.
- Wiedmer, E. (1980) Computing with infinite objects. *Theor. Comp. Sci.* **10**, 133–155.

A Correctness of the base machine

We now show that the base abstract machine is correct with respect to the operational semantics, that is, the abstract machine faithfully simulates the operational semantics. Initial states provide a canonical way to map a computation term onto the abstract machine. A more interesting question is how to map an arbitrary configuration to a computation term. Figure 10 describes such a mapping $\llbracket - \rrbracket$ from configurations to terms via a collection of mutually recursive functions defined on configurations, continuations, computation terms, value terms, and machine values. The mapping makes use of two operations on environments, γ , which we define now.

Definition 10. We write $\text{dom}(\gamma)$ for the domain of γ , and $\gamma \setminus \{x_1, \dots, x_n\}$ for the restriction of environment γ to $\text{dom}(\gamma) \setminus \{x_1, \dots, x_n\}$.

The $\llbracket - \rrbracket$ function enables us to classify the abstract machine reduction rules according to how they relate to the operational semantics. The rule (M-LET) is administrative in the sense that $\llbracket - \rrbracket$ is invariant under this rule. This leaves the β -rules (M-APP), (M-SPLIT), (M-CASE), and (M-RETCNT). Each of these corresponds directly with performing a reduction in the operational semantics.

Definition 11 (Auxiliary reduction relations). We write \longrightarrow_a for administrative steps (M-LET) and \simeq_a for the symmetric closure of \longrightarrow_a^* . We write \longrightarrow_β for β -steps (all other rules) and \Longrightarrow for a sequence of steps of the form $\longrightarrow_a^* \longrightarrow_\beta$.

Configurations

Pure continuations

$$\langle\langle M \mid \gamma \mid \sigma \rangle\rangle = \langle\langle \sigma \rangle\rangle (\langle\langle M \rangle\rangle \gamma) \quad \langle\langle \square \rangle\rangle M = M$$

$$\langle\langle \gamma, x, N \rangle\rangle :: \langle\langle \sigma \rangle\rangle M = \langle\langle \sigma \rangle\rangle (\mathbf{let} \ x \leftarrow M \ \mathbf{in} \ \langle\langle N \rangle\rangle (\gamma \setminus \{x\}))$$

Computation terms

$$\langle\langle V \ W \rangle\rangle \gamma = \langle\langle V \rangle\rangle \gamma \ \langle\langle W \rangle\rangle \gamma$$

$$\langle\langle \mathbf{let} \ \langle x; y \rangle = V \ \mathbf{in} \ N \rangle\rangle \gamma = \mathbf{let} \ \langle x; y \rangle = \langle\langle V \rangle\rangle \gamma \ \mathbf{in} \ \langle\langle N \rangle\rangle (\gamma \setminus \{x, y\})$$

$$\langle\langle \mathbf{case} \ V \ \{\mathbf{inl} \ x \mapsto M; \mathbf{inr} \ y \mapsto N\} \rangle\rangle \gamma = \mathbf{case} \ \langle\langle V \rangle\rangle \gamma \ \{\mathbf{inl} \ x \mapsto \langle\langle M \rangle\rangle (\gamma \setminus \{x\});$$

$$\mathbf{inr} \ y \mapsto \langle\langle N \rangle\rangle (\gamma \setminus \{y\})\}$$

$$\langle\langle \mathbf{return} \ V \rangle\rangle \gamma = \mathbf{return} \ \langle\langle V \rangle\rangle \gamma$$

$$\langle\langle \mathbf{let} \ x \leftarrow M \ \mathbf{in} \ N \rangle\rangle \gamma = \mathbf{let} \ x \leftarrow \langle\langle M \rangle\rangle \gamma \ \mathbf{in} \ \langle\langle N \rangle\rangle (\gamma \setminus \{x\})$$

Value terms and values

$$\langle\langle x \rangle\rangle \gamma = \langle\langle v \rangle\rangle, \quad \text{if } \gamma(x) = v$$

$$\langle\langle x \rangle\rangle \gamma = x, \quad \text{if } x \notin \text{dom}(\gamma)$$

$$\langle\langle n \rangle\rangle \gamma = n$$

$$\langle\langle \lambda x^A. M \rangle\rangle \gamma = \lambda x^A. \langle\langle M \rangle\rangle (\gamma \setminus \{x\})$$

$$\langle\langle \mathbf{rec} \ f \ x^A. M \rangle\rangle \gamma = \mathbf{rec} \ f \ x^A. \langle\langle M \rangle\rangle (\gamma \setminus \{f, x\})$$

$$\langle\langle \langle \rangle \rangle \rangle \gamma = \langle \rangle$$

$$\langle\langle \langle V, W \rangle \rangle \rangle \gamma = \langle\langle V \rangle\rangle \gamma, \ \langle\langle W \rangle\rangle \gamma$$

$$\langle\langle \langle \mathbf{inl} \ V \rangle^B \rangle \rangle \gamma = \langle\langle \mathbf{inl} \ \langle\langle V \rangle\rangle \gamma^B \rangle \rangle$$

$$\langle\langle \langle \mathbf{inr} \ W \rangle^A \rangle \rangle \gamma = \langle\langle \mathbf{inr} \ \langle\langle W \rangle\rangle \gamma^A \rangle \rangle$$

$$\langle\langle n \rangle\rangle = n$$

$$\langle\langle \langle \gamma, \lambda x^A. M \rangle \rangle \rangle = \lambda x^A. \langle\langle M \rangle\rangle (\gamma \setminus \{x\})$$

$$\langle\langle \langle \gamma, \mathbf{rec} \ f \ x^A. M \rangle \rangle \rangle = \mathbf{rec} \ f \ x^A. \langle\langle M \rangle\rangle (\gamma \setminus \{f, x\})$$

$$\langle\langle \langle \rangle \rangle \rangle = \langle \rangle$$

$$\langle\langle \langle v; w \rangle \rangle \rangle = \langle\langle v \rangle\rangle; \ \langle\langle w \rangle\rangle$$

$$\langle\langle \langle \mathbf{inl} \ v \rangle^B \rangle \rangle = \langle\langle \mathbf{inl} \ \langle\langle v \rangle\rangle^B \rangle \rangle$$

$$\langle\langle \langle \mathbf{inr} \ w \rangle^A \rangle \rangle = \langle\langle \mathbf{inr} \ \langle\langle w \rangle\rangle^A \rangle \rangle$$

$$\langle\langle \langle \sigma^A \rangle \rangle \rangle = \lambda x^A. \langle\langle \langle \sigma \rangle \rangle (\mathbf{return} \ x) \rangle \rangle$$

Fig. 10: Mapping from Base Machine Configurations to Terms

The following lemma describes how we can simulate each reduction in the operational semantics by a sequence of administrative steps followed by one β -step in the abstract machine.

Lemma 4. *Suppose M is a computation and \mathcal{C} is configuration such that $\langle\langle \mathcal{C} \rangle\rangle = M$, then if $M \rightsquigarrow N$ there exists \mathcal{C}' such that $\mathcal{C} \Longrightarrow \mathcal{C}'$ and $\langle\langle \mathcal{C}' \rangle\rangle = N$, or if $M \not\rightsquigarrow$ then $\mathcal{C} \not\Longrightarrow$.*

Proof By induction on the derivation of $M \rightsquigarrow N$. ■

The correspondence here is rather strong: there is a one-to-one mapping between \rightsquigarrow and $\Longrightarrow / \simeq_a$ (where we write R/S for the quotient of relation R by relation S). The inverse of the lemma is straightforward as the semantics is deterministic. Notice that Lemma 4 does not require that M be well-typed. We have chosen here not to perform type-erasure, but the results can be adapted to semantics in which all type annotations are erased.

Theorem 5 (Base simulation). *If $\vdash M : A$ and $M \rightsquigarrow^+ N$ such that N is normal, then $\langle M \mid \emptyset \mid \square \rangle \longrightarrow^+ \mathcal{C}$ such that $\langle\langle \mathcal{C} \rangle\rangle = N$, or $M \not\rightsquigarrow$ then $\langle M \mid \emptyset \mid \square \rangle \not\rightarrow$.*

Proof By repeated application of Lemma 4. ■

Configurations**Continuations**

$$\langle M \mid \gamma \mid \kappa \rangle = \langle \kappa \rangle (\langle M \rangle \gamma)$$

$$\langle \langle \rangle \rangle M = M$$

$$\langle \langle \sigma, \chi \rangle :: \kappa \rangle M = \langle \kappa \rangle (\langle \chi \rangle (\langle \sigma \rangle (M)))$$

Handler Closures and Definitions

$$\langle \langle \gamma, H \rangle \rangle M = \mathbf{handle} \ M \ \mathbf{with} \ \langle H \rangle \gamma \quad \langle \{ \mathbf{val} \ x \mapsto M \} \rangle \gamma = \{ \mathbf{val} \ x \mapsto \langle M \rangle (\gamma \setminus \{x\}) \}$$

$$\langle \{ \ell \ x \ r \mapsto M \} \uplus H \rangle \gamma = \{ \ell \ x \ r \mapsto \langle M \rangle (\gamma \setminus \{x, r\}) \} \uplus \langle H \rangle \gamma$$

Computation Terms and Machine Values

$$\langle \mathbf{handle} \ M \ \mathbf{with} \ H \rangle \gamma = \mathbf{handle} \ \langle M \rangle \gamma \ \mathbf{with} \ \langle H \rangle \gamma \quad \langle \langle \gamma, H \rangle^D \rangle \gamma = \lambda x^D. \langle \langle \gamma, H \rangle \rangle (\mathbf{return} \ x)$$

$$\langle \mathbf{do} \ \ell \ V \rangle \gamma = \mathbf{do} \ \ell \ \langle V \rangle \gamma$$

Fig. 11: Mapping from Handler Machine Configurations to Terms

B Correctness of the handler machine

The correctness result for the base machine can mostly be repurposed for the handler machine as we need only recheck the cases for (M-LET) and (M-RETCONT) and check the cases for handlers. Figure 11 shows the necessary changes to the $\langle _ \rangle$ function.

Lemma 5. *Suppose M is a computation and \mathcal{C} is configuration such that $\langle \mathcal{C} \rangle = M$, then if $M \rightsquigarrow N$ there exists \mathcal{C}' such that $\mathcal{C} \Longrightarrow \mathcal{C}'$ and $\langle \mathcal{C}' \rangle = N$, or if $M \not\rightsquigarrow$ then $\mathcal{C} \not\Longrightarrow$.*

Proof By induction on the derivation of $M \rightsquigarrow N$. ■

Theorem 6 (Handler simulation). *If $\vdash M : A$ and $M \rightsquigarrow^+ N$ such that N is normal, then $\langle M \mid \emptyset \mid \kappa_0 \rangle \longrightarrow^+ \mathcal{C}$ such that $\langle \mathcal{C} \rangle = N$, or $M \not\rightsquigarrow$ then $\langle M \mid \emptyset \mid \kappa_0 \rangle \not\longrightarrow$.*

Proof By repeated application of Lemma 5. ■