*JFP* **34**, e5, 54 pages, 2024. © The Author(s), 2024. Published by Cambridge University Press. This is an Open Access article, distributed under the terms of the Creative Commons Attribution licence (https://creativecommons.org/licenses/by/4.0/), which permits unrestricted re-use, distribution and reproduction, provided the original article is properly cited.
doi:10.1017/S0956796824000030

# *Asymptotic speedup via effect handlers*

DANIEL HILLERSTRÖM
*Laboratory for Foundations of Computer Science,*
*The University of Edinburgh, Edinburgh EH8 9YL, UK*
(*e-mail:* daniel.hillerstrom@ed.ac.uk)

SAM LINDLEY
*Laboratory for Foundations of Computer Science,*
*The University of Edinburgh, Edinburgh EH8 9YL, UK*
(*e-mail:* sam.lindley@ed.ac.uk)

JOHN LONGLEY
*Laboratory for Foundations of Computer Science,*
*The University of Edinburgh, Edinburgh EH8 9YL, 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 functional base language $\lambda_b$ and an extension $\lambda_h$ with general *effect handlers*. We prove that $\lambda_h$ admits an asymptotically more efficient implementation of generic count than any implementation in $\lambda_b$. We also show that this gap remains even when $\lambda_b$ is extended to a language $\lambda_a$ 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 & Felleisen, 1992). Such operations are 'non-functional' in the sense that their output is not determined solely by the extension of their input (seen as a mathematical function $\mathbb{N}_\perp \times \mathbb{N}_\perp \to \mathbb{N}_\perp$); however, there are also programs with 'functional' behaviour that can be implemented with control or local state but not without them (Longley, 1999). More recent results have exhibited differences lower down in the language expressivity spectrum: for instance, in a purely functional setting *à la* Haskell, the expressive power of *recursion* increases strictly with its type level (Longley, 2018), and there are natural operations computable by 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 & 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 $P\,q = \mathrm{true}$. We shall consider boolean vectors of any length to be represented by the type $\mathrm{Nat} \to \mathrm{Bool}$; thus for each $n$, we are asking for an implementation of a certain third-order operation

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

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 $\mathrm{count}_n$ 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 a 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 implementation method transfers to the more familiar *generic search* problem: that of returning the list of all vectors $q$ such that $P\,q = \mathrm{true}$.

The idea behind the speedup is easily explained and will already be familiar, at least informally, to programmers who have worked with multi-shot continuations. Suppose for example $n = 3$, and suppose that the predicate $P$ always inspects the components of its argument in the order $0, 1, 2$. A naïve implementation of $\mathrm{count}_3$ might start by applying the given $P$ to $q_0 = (\mathrm{true}, \mathrm{true}, \mathrm{true})$, and then to $q_1 = (\mathrm{true}, \mathrm{true}, \mathrm{false})$. Clearly, there is some duplication here: the computations of $P\,q_0$ and $P\,q_1$ will proceed identically up to the point where the value of the final component is requested. What we would like to do, then, is to record the state of the computation of $P\,q_0$ at just this point, so that we can later resume this computation with false supplied as the final component value in order to obtain the value of $P\,q_1$. (Similarly for all other internal nodes in the evident binary tree of boolean vectors.) Of course, such a 'backup' approach is easy to realise if one is implementing a bespoke search operation for some *particular* choice of $P$; but to apply this idea of resuming previous subcomputations in the *generic* setting (that is, uniformly in $P$) requires some feature such as general effect handlers or multi-shot continuations.

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

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

In the first instance, we formulate our results as a comparison between a purely functional base language $\lambda_{\mathrm{b}}$ (a version of call-by-value PCF) and an extension $\lambda_{\mathrm{h}}$ with general effect handlers. This allows us to present the key idea in a simple setting, but we then show how our runtime lower bound is also applicable to a more sophisticated language $\lambda_{\mathrm{a}}$ with affine effect handlers, intermediate in power between $\lambda_{\mathrm{b}}$ and $\lambda_{\mathrm{h}}$ and corresponding broadly to 'single-shot' uses of delimited control. Our proof involves some general machinery for reasoning about program evaluation in $\lambda_{\mathrm{a}}$ which may be of independent interest.

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 (Plotkin & Pretnar, 2013) 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 & 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 $\lambda_b$ and an extension $\lambda_h$ with general effect handlers.
- Section 4 defines abstract machines for $\lambda_b$ and $\lambda_h$, yielding a runtime cost model.
- Section 5 introduces the generic count problem and some associated machinery and presents an implementation in $\lambda_h$ with runtime $\mathcal{O}(2^n)$ (perhaps with small additional logarithmic factors according to the precise details of the cost model).
- 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 $\lambda_h$, including the one suggested by Berger (1990), emphasising 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 $\lambda_b$ must have runtime $\Omega(n2^n)$ on predicates of a certain kind.
- Section 9 refines our definition of $\lambda_h$ to yield a language $\lambda_a$ for affine effect handlers, clarifying its relationship to $\lambda_b$ and $\lambda_h$.
- Section 10 develops some machinery for reasoning about program evaluation in $\lambda_a$, 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 $\lambda_b$ and $\lambda_h$ 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 $\lambda_a$ 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 $\lambda_b$ (Section 8). The benchmarks have been ported to OCaml 5.1.1 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 & Power, 2001; Plotkin & Pretnar, 2013). Subsequently, they have emerged as a practical programming abstraction for modular effectful programming (Convent et al., 2020; Kammar et al., 2013; Kiselyov et al., 2013; Bauer & Pretnar, 2015; Leijen, 2017; Hillerström et al., 2020; Sivaramakrishnan et al., 2021). 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 & 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 $\Sigma$ of typed *operation symbols* along with an equational theory that describes the properties of the operations (Plotkin & Power, 2001). An example of an algebraic effect is *nondeterminism*, whose signature consists of a single nondeterministic choice operation: $\Sigma \overset{\text{def}}{=} \{\mathrm{Branch} : \mathrm{Unit} \to \mathrm{Bool}\}$. The operation takes a single parameter of type unit and ultimately produces a boolean value. The pragmatic programmatic view of algebraic effects differs from the original development as no implementation accounts for equations over operations yet.

As a simple example, let us use the operation Branch to model a coin toss. Suppose we have a data type $\mathrm{Toss} \overset{\text{def}}{=} \mathrm{Heads} \mid \mathrm{Tails}$, then we may implement a coin toss as follows.

$$\begin{array}{l} \mathrm{toss} : \mathrm{Unit} \to \mathrm{Toss} \\ \mathrm{toss}\ \langle\rangle = \mathbf{if}\ \mathbf{do}\ \mathrm{Branch}\ \langle\rangle\ \mathbf{then}\ \mathrm{Heads}\ \mathbf{else}\ \mathrm{Tails} \end{array}$$

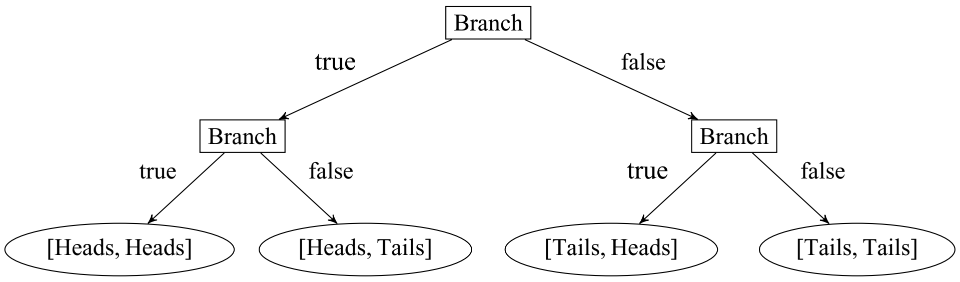
From the type signature, it is clear that the computation returns a value of type Toss. It is not clear from the signature of toss whether it performs an effect. However, from the definition, it evidently performs the operation Branch with argument $\langle\rangle$ using the **do**-invocation form. The result of the operation determines whether the computation returns either Heads or Tails. Systems such as Effekt (Brachthäuser et al., 2020), Frank

(Lindley et al., 2017; Convent et al., 2020), Helium (Biernacki et al., 2019, 2020), Koka (Leijen, 2017), and Links (Hillerström & Lindley, 2016; Hillerström et al., 2020) include type-and-effect systems (or in the case of Effekt a capability type system) which track the use of effectful operations, whilst systems such as Eff (Bauer & Pretnar, 2015) and Multicore OCaml (Dolan et al., 2015)/OCaml 5 (Sivaramakrishnan et al., 2021) choose 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.

$$\begin{aligned}&\mathsf{tossTwice} : \mathsf{Unit} \to \mathsf{List\ Toss}\\ &\mathsf{tossTwice}\ \langle\rangle = [\mathsf{toss}\ \langle\rangle, \mathsf{toss}\ \langle\rangle]\end{aligned}$$

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.

$$\begin{aligned}&\mathbf{handle}\ \mathsf{tossTwice}\ \langle\rangle\ \mathbf{with}\\ &\quad \mathbf{val}\ x \qquad\quad \mapsto [x]\\ &\quad \mathsf{Branch}\ \langle\rangle\ r \mapsto r\ \mathsf{true} +\!\!+\ r\ \mathsf{false}\end{aligned}$$

The **handle**-construct generalises the exceptional syntax of Benton & Kennedy (2001). This handler has a *success* clause and an *operation* clause. 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 to 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 [[Heads, Heads], [Heads, Tails]]. Thereafter, the initial resumption is applied to false, which symmetrically returns the list [[Tails, Heads], [Tails, Tails]]. Finally, the two intermediary lists get concatenated to produce the final result [[Heads, Heads], [Heads, Tails], [Tails, Heads], [Tails, Tails]].

## 3 Calculi

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

### 3.1 Base calculus

The base calculus $\lambda_{\mathrm{b}}$ 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 $\lambda_{\mathrm{b}}$ is as follows.

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

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

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. We also use $f$, $g$, $h$, $q$ for variables of function type and $i, j$ for variables of type Nat. The value terms are standard.

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 those given in Figure 1, along with the familiar Exchange, Weakening and Contraction rules for environments. (Note that thanks to Weakening we are able to type terms such as $(\lambda x^A.(\lambda x^B.x))$, even though environments are not permitted to contain duplicate variables.) 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

**Values**

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

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

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

**Computations**

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

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

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

Fig. 1. Typing rules for $\lambda_b$.

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

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

Fig. 2. Contextual small-step operational semantics.

type environment $\Gamma$, where $\Box$ is either a value term ($V$) or a computation term ($M$). The constants have the following types.

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

We give a small-step operational semantics for $\lambda_b$ with *evaluation contexts* in the style of Felleisen (1987). The reduction relation $\rightsquigarrow$ is defined on computation terms via the rules given in Figure 2. 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 $R$.

Most often, we are interested in $\rightsquigarrow$ as a relation on *closed* terms. However, we will sometimes consider it as a relation on terms involving free variables, with the stipulation that none of these free variables also occur as *bound* variables within the terms. Since we never perform reductions under a binder, this means that the notation $M[V/x]$ in our rules may be taken simply to mean $M$ with $V$ textually substituted for free occurrences of $x$ (no variable capture is possible). We also take $\ulcorner c \urcorner$ to mean the usual interpretation of constant $c$ as a meta-level function on closed values.

The type soundness of our system is easily verified. This is subsumed by the property we shall formally state for the richer language $\lambda_\text{h}$ in Theorem 1 below.

When dealing with reductions $N \leadsto N'$, we shall often make use of the idea that certain subterm occurrences within $N'$ arise from corresponding identical subterms of $N$. For instance, in the case of a reduction $(\lambda x^A.M)V \leadsto M[V/x]$, we shall say that any subterm occurrence $P$ within any of the substituted copies of $V$ on the right-hand side is a *descendant* of the corresponding subterm occurrence within the $V$ on the left-hand side. (Descendants are called *residuals* e.g. in Barendregt, 1984.) Similarly, any subterm occurrence $Q$ of $M[V/x]$ not overlapping with any of these substituted copies of $V$ is a *descendant* of the corresponding occurrence of an identical subterm within the $M$ on the left. This notion extends to the other reduction rules in the evident way; we suppress the formal details. If $P'$ is a descendant of $P$, we also say that $P$ is an *ancestor* of $P'$. By transitivity we extend these notions to the relations $\leadsto^+$ and $\leadsto^*$. Note that if $N \leadsto^* N'$ then a subterm occurrence in $N'$ may have at most one ancestor in $N$ but a subterm occurrence in $N$ may have many descendants in $N'$.

**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:

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

We define sequencing of computations in the standard way.

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

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

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

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

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

We use the standard encoding of booleans as a sum:

$$\text{Bool} \overset{\text{def}}{=} \text{Unit} + \text{Unit} \qquad \text{true} \overset{\text{def}}{=} \textbf{inl}\;\langle\rangle \qquad \text{false} \overset{\text{def}}{=} \textbf{inr}\;\langle\rangle$$

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

### 3.2 Handler calculus

We now define $\lambda_\text{h}$ as an extension of $\lambda_\text{b}$.

$$\begin{array}{lrl}
\text{Signatures} & \Sigma & ::= \cdot \mid \{\ell : A \to B\} \cup \Sigma \\
\text{Handler types} & F & ::= C \Rightarrow D \\
\text{Computations} & M, N & ::= \cdots \mid \textbf{do}\;\ell\;V \mid \textbf{handle}\;M\;\textbf{with}\;H \\
\text{Handlers} & H & ::= \{\textbf{val}\;x \mapsto M\} \mid \{\ell\;p\;r \mapsto N\} \uplus H
\end{array}$$

**Computations**

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

**Handlers**

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

Fig. 3. Additional typing rules for $\lambda_{\mathrm{h}}$.

We assume a fixed *effect signature* $\Sigma$ that associates types $\Sigma(\ell)$ to finitely many operation symbols $\ell$. An operation type $A \to B$ classifies operations that take an argument of type $A$ and return a result of type $B$. 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 a single success clause ($\{\mathbf{val}\, x \mapsto M\}$) and an operation clause ($\{\ell\, p\, r \mapsto N\}$) for each operation $\ell$ in $\Sigma$; here the $x, p, r$ are considered as bound variables. Following Plotkin & Pretnar (2013), we adopt the convention that a handler with missing operation clauses (with respect to $\Sigma$) is syntactic sugar for one in which all missing clauses perform explicit forwarding:

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

The typing rules for $\lambda_{\mathrm{h}}$ are those of $\lambda_{\mathrm{b}}$ (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 $\ell$ appears in the effect signature $\Sigma$ 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 $p$ ($A_\ell$) and resumption $r$ ($B_\ell \to D$) in operation clause $H^\ell$ is determined by the type of $\ell$; the return type of $r$ is $D$, as the body of the resumption will itself be handled by $H$. We write $H^\ell$ and $H^{\mathrm{val}}$ for projecting success and operation clauses.

$$\begin{aligned} H^{\mathrm{val}} &\overset{\mathrm{def}}{=} \{\mathbf{val}\, x \mapsto M\}, &&\text{where } \{\mathbf{val}\, x \mapsto M\} \in H \\ H^{\ell} &\overset{\mathrm{def}}{=} \{\ell\, p\, r \mapsto M\}, &&\text{where } \{\ell\, p\, r \mapsto M\} \in H \end{aligned}$$

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

$$\begin{array}{lll} \textsc{S-Ret} & \mathbf{handle}\, (\mathbf{return}\, V)\, \mathbf{with}\, H \rightsquigarrow N[V/x], & \text{where } H^{\mathrm{val}} = \{\mathbf{val}\, x \mapsto N\} \\ \textsc{S-Op} & \mathbf{handle}\, \mathcal{E}[\mathbf{do}\, \ell\, V]\, \mathbf{with}\, H \rightsquigarrow N[V/p,\ (\lambda y.\mathbf{handle}\, \mathcal{E}[\mathbf{return}\, y]\, \mathbf{with}\, H)/r], & \\ & & \text{where } H^{\ell} = \{\ell\, p\, r \mapsto N\} \text{ and } y \text{ is fresh} \end{array}$$

The first rule invokes the success clause. The second rule handles an operation via the corresponding operation clause.

To allow for the evaluation of subterms within **handle** expressions, we extend our earlier grammar for evaluation contexts to one for *handler contexts*:

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

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

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

However, it is critical that the rule S-Op is restricted to pure evaluation contexts $\mathcal{E}$ rather than handler contexts. This ensures that the **do** invocation is handled by the innermost handler (recalling our convention that all handlers handle all operations). If arbitrary handler contexts $\mathcal{H}$ were permitted in this rule, the semantics would become non-deterministic, as any handler in scope could be selected.

The ancestor-descendant relation for subterm occurrences extends to $\lambda_h$ in the obvious way.

We now characterise normal forms and state the standard type soundness property of $\lambda_h$.

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

**Theorem 1** (Type Soundness for $\lambda_h$). *If $\vdash M : C$, then either there exists $\vdash N : C$ such that $M \rightsquigarrow^* N$ and $N$ is normal with respect to $\Sigma$, or $M$ diverges.*

It is worth observing that our language does not prohibit 'operation extrusion': even if we begin with a term in which all **do** invocations fall within the scope of a handler, this property need not be preserved by reductions, since a **do** invocation may pass another **do** to the outermost handler. Such behaviour may be readily ruled out using a type-and-effect system, but this additional machinery is not necessary for our present purposes.

## 4 Abstract machine semantics

Thus far we have introduced the base calculus $\lambda_b$ and its extension with effect handlers $\lambda_h$. 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 & Friedman, 1987) for $\lambda_b$ 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 $\gamma$ which binds free variables. The third component contains the continuation which instructs the machine

**Transition relation**

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

**Value interpretation**

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

Fig. 4. Abstract machine semantics for $\lambda_b$.

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

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

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

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

A final state is given by a configuration of the form $\langle \mathbf{return}\, V \mid \gamma \mid [] \rangle$ in which case the final return value is given by the denotation $[\![V]\!]\gamma$ of $V$ under environment $\gamma$.

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

## *4.2 Handler machine*

We now enrich the $\lambda_b$ machine to a $\lambda_h$ machine. We extend the syntax as follows.

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

The notion of configurations changes slightly in that the continuation component is replaced by a generalised continuation $\kappa \in \mathsf{Cont}$ (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 ($\chi$). A handler closure consists of an environment and a handler definition, where the former binds the free variables that occur in the latter. The machine is initialised by placing a term in a configuration alongside the empty environment ($\emptyset$) and the identity continuation ($\kappa_0$). The latter is a singleton list containing the identity resumption, which consists of the identity pure continuation paired with the identity handler closure:

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

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 $\lambda_h$. Extending the result for the base machine, we formally state and prove the correspondence in Appendix B.

**Transition relation**

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

Fig. 5. Abstract machine semantics for $\lambda_{\mathrm{h}}$.

## *4.3 Realisability and asymptotic complexity*

As witnessed by the work of Hillerström & Lindley (2016), 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, $\gamma$, can be realised using a map, with a time complexity of $\mathcal{O}(\log|\gamma|)$ for extension and lookup (Okasaki, 1999). We can use the same technique to realise label lookup, $H^{\ell}$, with time complexity $\mathcal{O}(\log|\Sigma|)$. However, in Section 5.4, we shall work only with a single effect operation, so $|\Sigma| = 1$, meaning that in our analysis we can practically treat label lookup as being a constant time operation.

The worst-case time complexity of a single machine transition is exhibited by rules which involves the value translation function, $[\![-]\!]\gamma$, as it is defined structurally on values. Its worst-time complexity is exhibited by a nesting of pairs of variables $[\![\langle x_1, \langle x_2, \cdots, \langle x_{n-1}, x_n\rangle \cdots\rangle\rangle]\!]\gamma$ 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 $((\sigma,\chi) :: \_)$ would be a $\mathcal{O}(|\sigma| + |\chi|)$ time operation, where $|\chi|$ is the size of the handler closure $\chi$.

**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 $\lambda_h$ supports implementations of certain operations with an asymptotic runtime bound that cannot be achieved in $\lambda_b$ (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 $\lambda_h$, then show that no equivalent program in $\lambda_b$ 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 & Stevens, 2009) and graph colouring can be cast as instances of generic search, and similar ideas have been explored in connection with 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 $\mathbb{N}_n \to \mathbb{B}$, where $\mathbb{N}_n \overset{\text{def}}{=} \{0, \ldots, n-1\}$. In both $\lambda_b$ and $\lambda_h$ we may represent such functions by terms of type $\mathrm{Nat} \to \mathrm{Bool}$. We will often informally write $\mathrm{Nat}_n$ in place of Nat to indicate that only the values $0, \ldots, n-1$ are relevant, but this convention has no formal status since our setup does not support dependent types.

To summarise, in both $\lambda_b$ and $\lambda_h$ we will be working with the types

$$\begin{array}{rlrl} \mathrm{Point} \overset{\text{def}}{=} \mathrm{Nat} \to \mathrm{Bool} & & \mathrm{Point}_n \overset{\text{def}}{=} \mathrm{Nat}_n \to \mathrm{Bool} \\ \mathrm{Predicate} \overset{\text{def}}{=} \mathrm{Point} \to \mathrm{Bool} & & \mathrm{Predicate}_n \overset{\text{def}}{=} \mathrm{Point}_n \to \mathrm{Bool} \end{array}$$

and will be looking for programs

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

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

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:

$$\mathrm{q}_0 \overset{\text{def}}{=} \lambda\_.\mathrm{true} \qquad \mathrm{q}_1 \overset{\text{def}}{=} \lambda i.i = 0 \qquad \mathrm{q}_2 \overset{\text{def}}{=} \lambda i.\ \mathbf{if}\ i = 0\ \mathbf{then}\ \mathrm{true}\ \mathbf{else}\ \mathbf{if}\ i = 1\ \mathbf{then}\ \mathrm{false}\ \mathbf{else}\ \bot$$

(Here $\bot$ is the diverging term $(\mathbf{rec}\, f\; i.f\; i)\,\langle\rangle$.) Then $\mathrm{q}_0$ represents $\langle \mathrm{true}, \ldots, \mathrm{true}\rangle \in \mathbb{B}^n$ for any $n$; $\mathrm{q}_1$ represents $\langle \mathrm{true}, \mathrm{false}, \ldots, \mathrm{false}\rangle \in \mathbb{B}^n$ for any $n \geq 1$; and $\mathrm{q}_2$ represents $\langle \mathrm{true}, \mathrm{false}\rangle \in \mathbb{B}^2$.

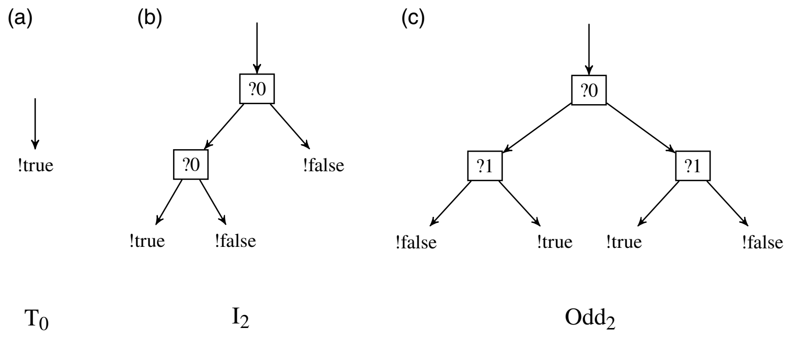


Fig. 6. Examples of decision trees.

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

$$\mathrm{T}_0 \stackrel{\text{def}}{=} \lambda q.\mathrm{true} \qquad \mathrm{T}_1 \stackrel{\text{def}}{=} \lambda q.(q\ 1; q\ 0; \mathrm{true}) \qquad \mathrm{T}_2 \stackrel{\text{def}}{=} \lambda q.(q\ 0; q\ 0; \mathrm{true})$$

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

Likewise, the following all represent the 'identity' predicate $\mathbb{B}^1 \to \mathbb{B}$, in a sense to be made precise below (here && is shortcut 'and'):

$$\mathrm{I}_0 \stackrel{\text{def}}{=} \lambda q.q\ 0 \qquad \mathrm{I}_1 \stackrel{\text{def}}{=} \lambda q.\ \mathbf{if}\ q\ 0\ \mathbf{then}\ \mathrm{true}\ \mathbf{else}\ \mathrm{false} \qquad \mathrm{I}_2 \stackrel{\text{def}}{=} \lambda q.(q\ 0)\ \&\&\ (q\ 0)$$

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

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

Here fold and map are the standard combinators on lists, and $\otimes$ is exclusive-or. Applying $\mathrm{Odd}_2$ to $\mathrm{q}_0$ yields false; applying it to $\mathrm{q}_1$ or $\mathrm{q}_2$ yields true.

We can think of a predicate term $P$ as participating in a 'dialogue' with a given point $Q : \mathrm{Point}_n$. 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 $\mathrm{I}_2$, one of the !false leaves will be 'unreachable' if we are working in $\lambda_\mathrm{b}$ (but reachable in a language supporting mutable state).

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

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

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

Of the examples we have given, the tree for $\mathrm{T}_0$ is 0-standard; those for $\mathrm{I}_0$ and $\mathrm{I}_1$ are 1-standard; that for $\mathrm{Odd}_n$ 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 $\lambda_{\mathrm{h}}$, but everything can clearly be relativised to $\lambda_{\mathrm{b}}$ with no change to the meaning in the case of $\lambda_{\mathrm{b}}$ terms. For the purpose of this subsection, we fix $n \in \mathbb{N}$, set $\mathbb{N}_n \overset{\mathrm{def}}{=} \{0, \dots, n-1\}$, 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 $\lambda_{\mathrm{h}}$, 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* $Q$ : Point *is said to be a* syntactic $n$-point *if:*

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

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

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

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

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

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

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

**Definition 3** (untimed decision tree). *(i) The address set* $\mathsf{Addr}$ *is simply the set* $\mathbb{B}^*$ *of finite lists of booleans. If* $bs, bs' \in \mathsf{Addr}$*, we write* $bs \sqsubseteq bs'$ *(resp.* $bs \sqsubset bs'$*) to mean that* $bs$ *is a prefix (resp. proper prefix) of* $bs'$*.*

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

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

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

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

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

**Definition 4** (timed decision tree). *A timed decision tree is a partial function* $\tau : \mathsf{Addr} \rightharpoonup \mathsf{Lab} \times \mathbb{N}$ *such that its first projection* $bs \mapsto \tau(bs).1$ *is a decision tree. We write* $\mathsf{labs}(\tau)$ *for the first projection (*$bs \mapsto \tau(bs).1$*) and* $\mathsf{steps}(\tau)$ *for the second projection (*$bs \mapsto \tau(bs).2$*) of a timed decision tree.*

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

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

Our definition captures this idea using abstract machine configurations. We write $\text{Conf}_q$ for the set of $\lambda_h$ 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* $\mathcal{T} : \mathrm{Conf}_q \to \mathsf{Addr} \rightharpoonup (\mathsf{Lab} \times \mathbb{N})$ *to be the minimal family of partial functions satisfying the following equations:*

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

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

*(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* $P : \mathrm{Predicate}$ *is* $\mathcal{T}(P\, q)$*. Since* $q$ *plays the role of a dummy argument, we will usually omit it and write* $\mathcal{T}(P)$ *for* $\mathcal{T}(P\, q)$*.*

*(iv) The untimed decision tree* $\mathcal{U}(P)$ *is obtained from* $\mathcal{T}(P)$ *via first projection:* $\mathcal{U}(P) = \mathsf{labs}(\mathcal{T}(P))$*.*

If the execution of a configuration $\mathcal{C}$ 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* $\tau$ *is an* $n$-predicate tree *if it satisfies the following:*

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

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

- *All paths in* $\tau$ *are finite (so every maximal path terminates in an answer node).*

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

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

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

This path will terminate at some answer node $b_0 b_1 \ldots b_{r-1}$ of $\tau$, 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* $P\, Q \leadsto^* \mathbf{return}\, b$ *where* $b = \mathcal{U}(P) \bullet [\![Q]\!]$*.*

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

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

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

**Definition 7** ($n$-standard trees and predicates)**.** *An n-predicate tree $\tau$ is said to be* $n$-standard *if the following hold:*

- *The domain of $\tau$ is precisely $\mathsf{Addr}_n$, the set of bit vectors of length $\leq n$.*
- *There are no repeated queries along any path in $\tau$:*

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

*A timed decision tree $\tau$ is n-standard if its underlying untimed decision tree $\mathsf{labs}(\tau)$ is too. An n-predicate $P$ is n-standard if $\mathcal{U}(P)$ is n-standard.*

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

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

**Definition 8** (canonical $n$-standard predicates)**.** *Given an n-standard tree $\tau$, we may associate to each address $bs \in dom(\tau)$ a $\lambda_b$ term $T_q(\tau, bs)$ (with free variable $q : \mathrm{Point}$) by reverse induction on the length of bs:*

$$\begin{array}{lcll} T_q(\tau, bs) & = & \textbf{return}\, b & \textit{if}\ \tau(bs) = !b \\ T_q(\tau, bs) & = & \textbf{if}\, q(k)\, \textbf{then}\, T_q(\tau, bs +\!\!+ [\mathrm{true}])\, \textbf{else}\, T_q(\tau, bs +\!\!+ [\mathrm{false}]) & \textit{if}\ \tau(bs) = ?k \end{array}$$

*We then define*

$$P(\tau) \ = \ \lambda q.\, T_q(\tau, [])$$

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

*In practice, we will omit the subscript q from uses of T.*

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

## *5.3 Specification of counting programs*

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

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

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

This definition gives us the flexibility to talk about counting programs that operate on various classes of predicates, allowing us to state our results in their strongest natural form.

On the positive side, we shall shortly see that there is a single 'efficient' program in $\lambda_{\mathrm{h}}$ that correctly counts all $n$-standard $\lambda_h$ predicates for every $n$; in Section 6.1 we improve this to one that correctly counts *all* $n$-predicates of $\lambda_h$. On the negative side, we shall show that an $n$-indexed family of counting programs written in $\lambda_{\mathrm{b}}$, even if only required to work correctly on canonical $n$-standard $\lambda_b$ predicates, can never compete with our $\lambda_{\mathrm{h}}$ program for asymptotic efficiency even in the most favourable cases.

### 5.4 *Efficient generic count with effects*

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

Our program uses a variation of the handler for nondeterministic computation that we gave in Section 2. The main idea is to implement points as 'nondeterministic computations' using the Branch operation such that the handler may respond to every query twice, by invoking the provided resumption with true and subsequently false. The key insight is that the resumption restarts computation at the invocation site of Branch, 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 Branch : Unit $\to$ Bool $\in \Sigma$ is a distinguished operation that may not be handled in the definition of any input predicate (it has to be forwarded according to the default convention). The algorithm is then as follows.

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

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

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 $\lambda_{\mathrm{h}}$:*

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

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

**Proof** [Outline.] Suppose $bs \in \mathsf{Addr}_n$, with length $j$. 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 Hyp($bs$) for the claim that $\mathcal{C}_{bs}$ correctly counts this subtree and does so within the following number of steps:

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

The $9 * (2^{n-j} - 1)$ expression is the number of machine steps contributed by the Branch-case inside the handler, whilst the $2 * 2^{n-j}$ expression is the number of machine steps contributed by the **val**-case. We prove Hyp($bs$) by a laborious but entirely routine downwards induction on the length of $bs$. The proof combines counting of explicit machine steps with 'oracular' appeals to the assumed behaviour of $P$ as modelled by $\mathcal{T}(P)$. Once Hyp([]) is established, both halves of the theorem follow easily. Full details are given in Appendix C of Hillerström et al. (2020). ■

The above formula can clearly be simplified for certain reasonable classes of predicates. For instance, suppose we fix some constant $c \in \mathbb{N}$, and let $\mathcal{P}_{n,c}$ be the class of all $n$-standard predicates $P$ for which all the edge times $\mathsf{steps}(\mathcal{T}(P))(bs)$ are bounded by $c$. (Many reasonable predicates will belong to $\mathcal{P}_{n,c}$ for some modest value of $c$: for instance, the membership test for some regular language $\mathcal{L} \subseteq \{0, 1\}^*$, or even for many languages defined by deterministic pushdown automata if cons-lists be added to our language.) Since the number of sequences $bs$ in question is less than $2^{n+1}$, we may read off from the above formula that for predicates in $\mathcal{P}_{n,c}$, the runtime of effcount is $\mathcal{O}(c2^n)$.

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

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

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

## 6 Extensions and variations

Our efficient implementation method is robust under several variations. We outline here how the idea generalises beyond $n$-standard predicates and adapts from generic count to generic search. We also indicate how one may obtain the speedup in question in the presence of a type-and-effect system.

### 6.1 Beyond $n$-standard predicates

The $n$-standardness restriction on predicates serves to make the efficiency phenomenon stand out as clearly as possible. However, we can relax the restriction by tweaking effcount to handle repeated queries and missing queries. The trade-off is that the analysis of effcount becomes more involved. The key to relaxing the $n$-standardness restriction is the use of state to keep track of which queries have been computed. We can give stateful implementations of effcount without changing its type signature by using *parameter-passing* (Kammar et al., 2013; Pretnar, 2015) to internalise state within a handler. Parameter-passing abstracts every handler clause such that the current state is supplied before the evaluation of a clause continues and the state is threaded through resumptions: a resumption becomes a two-argument curried function $r : B \to S \to D$, where the first argument of type $B$ is the return type of the operation and the second argument is the updated state of type $S$.

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

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

For fixed $n$, we assume a type of natural number to boolean maps, $\mathrm{Map}_n$, with the following interface.

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

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 $\lambda_b$ such that the time complexity of $\mathrm{add}_n$ and $\mathrm{lookup}_n$ is $\mathcal{O}(\log n)$ (Okasaki, 1999). We can then use parameter-passing to support repeated queries as follows.

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

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

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

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

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 $\mathrm{effcount}''_n$ to operate correctly on predicates that inspect $n$ points at most once. We leave it as an exercise for the reader to combine $\mathrm{effcount}'_n$ and $\mathrm{effcount}''_n$ 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 key difference is that a generic search procedure must materialise a list of solutions, thus its type is

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

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

```
$\mathrm{effsearch}_n : ((\mathrm{Nat}_n \to \mathrm{Bool}) \to \mathrm{Bool}) \to \mathrm{List}_{\mathrm{Nat}_n \to \mathrm{Bool}}$
$\mathrm{effsearch}_n\, pred \overset{\text{def}}{=} \mathbf{let}\, f \leftarrow \mathbf{handle}\, pred\, (\lambda i.\mathbf{do}\ \mathrm{Branch}\ i)\ \mathbf{with}$
                         $\mathbf{val}\ x \mapsto \lambda q.\mathbf{if}\, x\, \mathbf{then}\ \mathrm{singleton}\ q\ \mathbf{else}\ \mathrm{nil}$
                         $\mathrm{Branch}\ i\ r \mapsto \lambda q.\mathbf{let}\, xs_{\mathrm{true}} \leftarrow r\ \mathrm{true}\ (\lambda j.\mathbf{if}\ i = j\ \mathbf{then}\ \mathrm{true}\ \mathbf{else}\ q\ j)\ \mathbf{in}$
                                    $\mathbf{let}\, xs_{\mathrm{false}} \leftarrow r\ \mathrm{false}\ (\lambda j.\mathbf{if}\ i = j\ \mathbf{then}\ \mathrm{false}\ \mathbf{else}\ q\ j)\ \mathbf{in}$
                                    $\mathrm{append}\ \langle xs_{\mathrm{true}}, xs_{\mathrm{false}}\rangle$
                $\mathbf{in}\ \mathrm{toConsList}\ (f\ (\lambda j.\bot))$
```

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 $\mathrm{effsearch}_n$ 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: $\mathrm{HList}_A \overset{\text{def}}{=} \mathrm{List}_A \to \mathrm{List}_A$. The empty Hughes list $\mathrm{nil} : \mathrm{HList}_A$ is defined as the identity function: $\mathrm{nil} \overset{\text{def}}{=} \lambda xs.xs$.

$$\begin{aligned}
&\mathrm{singleton}_A : A \to \mathrm{HList}_A\\
&\mathrm{singleton}_A\; x \overset{\text{def}}{=} \lambda xs.x :: xs\\[1ex]
&\mathrm{append}_A : \mathrm{HList}_A \to \mathrm{HList}_A \to \mathrm{HList}_A\\
&\mathrm{append}_A\; f\; g \overset{\text{def}}{=} \lambda xs.g\;(f\;xs)\\[1ex]
&\mathrm{toConsList}_A : \mathrm{HList} \to \mathrm{List}_A\\
&\mathrm{toConsList}_A\; f \overset{\text{def}}{=} f\;[]
\end{aligned}$$

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 & Pretnar, 2014; Hillerström & 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. $A \to B!\varepsilon$ denotes a function that accepts a value of type $A$ as input and produces some value of type $B$ as output using effects $\varepsilon$. Here $\varepsilon$ is intended to be an effect variable which may be instantiated to name concrete effectful operations that the function may perform such as $\mathrm{Branch} : \mathrm{Unit} \to \mathrm{Bool}$. 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 & Lindley, 2016; Leijen, 2017), subtyping (Bauer & 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.

$$\mathrm{Count} : (\forall \varepsilon.(\mathrm{Nat} \to \mathrm{Bool}!\varepsilon) \to \mathrm{Bool}!\varepsilon) \to \mathrm{Nat}!\emptyset$$

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 at least to McCracken (1984); it has been used in practice in Haskell as the primary means for state encapsulation since Launchbury & Jones (1994).

## 7 Generic count in weaker languages

We have shown that there is an implementation of generic count in $\lambda_h$ 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 $\lambda_b$ (Section 8), or indeed in the stronger language $\lambda_a$ (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 $\lambda_h$, emphasising 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 $\lambda_b$; but it is also clear how it could be effected in an even weaker language, in which the *recursion* construct of $\lambda_b$ is replaced by a weaker *iteration* construct. (For a comparison of the power of iteration and recursion, see Longley, 2019.) For instance, the following operator (definable in $\lambda_b$) allows one to achieve the effect of while-loops manipulating data of type $A$:

$$\begin{aligned}
&\mathsf{while}_A \ : (A \to \mathrm{Bool}) \to A \to (A \to A) \to A \\
&\mathsf{while}_A\ \mathit{test}\, x\, f \overset{\text{def}}{=} \mathbf{if}\ \mathit{test}\ x\ \mathbf{then}\ \mathsf{while}_A\ \mathit{test}\ (f\ x)\, f\ \mathbf{else}\ x
\end{aligned}$$

Let us write $\lambda_i$ for the sublanguage of $\lambda_b$ allowing $\mathsf{while}_A$ for each type $A$, but disallowing all uses of **rec** elsewhere. Then it is a straightforward coding exercise to write a $\lambda_i$ program

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

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 $\lambda_b$, 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 $\lambda_b$.

### *7.2 The nesting trick*

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

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

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, $\mathrm{bestshot}_n$ embodies Hilbert's choice operator $\varepsilon$ 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 $\mathrm{bestshot}_n\ P$ takes only constant time, all the real work being deferred until the argument of type $\mathrm{Nat}_n$ is supplied.

Now consider the following program:

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

Here the term *pred* ($\mathrm{bestshot}_n$ *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 $\mathrm{lazycount}_n$ is a correct implementation of generic count, and also that if *pred* is the predicate $\lambda q.\mathrm{false}$ then $\mathrm{lazycount}_n\ \mathit{pred}$ returns 0 within $\mathcal{O}(1)$ time, thus violating the $\Omega(2^n)$ lower bound suggested above.

This might seem like a footling point, as $\mathrm{lazycount}_n$ 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 $\lambda_b$ 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 $\lambda_i$ with mere iteration.

We shall refer to this $\lambda_b$ program as Bergercount, as it is modelled largely on Berger's PCF implementation of the so-called *fan functional* (Berger, 1990; Longley & 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 & 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 $\mathrm{bestshot}_n$ function, which given any $n$-standard predicate $P$ returns a point satisfying $P$ if one exists, or the dummy point $\lambda i.\mathrm{false}$ if not. Figure 7 depicts the implementation of this function. It is implemented by via two mutually recursive auxiliary functions whose workings are admittedly hard to elucidate in a few words. The function $\mathrm{bestshot}'_n$ is a generalisation of $\mathrm{bestshot}_n$ that makes a best

```
$\mathrm{bestshot}_n : \mathrm{Predicate}_n \to \mathrm{Point}_n$
$\mathrm{bestshot}_n\ pred \overset{\mathrm{def}}{=} \mathrm{bestshot}'_n\ pred\ []$

$\mathrm{bestshot}'_n : \mathrm{Predicate}_n \to \mathrm{List}_{\mathrm{Bool}} \to \mathrm{Point}_n$
$\mathrm{bestshot}'_n\ pred\ start \overset{\mathrm{def}}{=} \mathbf{let}\ g \leftarrow \mathrm{memoise}\ (\lambda\langle\rangle.\mathrm{bestshot}''_n\ pred\ start)\ \mathbf{in}$
                              $\mathbf{let}\ f \leftarrow \mathbf{return}\ (\lambda i.\mathbf{let}\ xs \leftarrow g\ \langle\rangle\ \mathbf{in}\ \mathrm{nth}\ xs\ i)\ \mathbf{in}$
                              $\mathbf{return}\ (\lambda i.\mathbf{if}\ i < |start|\ \mathbf{then}\ \mathrm{nth}\ start\ i\ \mathbf{else}\ f\ i)$

$\mathrm{bestshot}''_n : \mathrm{Predicate}_n \to \mathrm{List}_{\mathrm{Bool}} \to \mathrm{List}_{\mathrm{Bool}}$
$\mathrm{bestshot}''_n\ pred\ start \overset{\mathrm{def}}{=} \mathbf{if}\ |start| = n\ \mathbf{then\ return}\ start$
                              $\mathbf{else\ let}\ f \leftarrow \mathrm{bestshot}'_n\ pred\ (\mathrm{append}\ start\ [\mathrm{true}])\ \mathbf{in}$
                                  $\mathbf{if}\ pred\ f\ \mathbf{then\ return}\ [f\ 0, \ldots, f\ (n-1)]$
                                  $\mathbf{else}\ \mathrm{bestshot}''_n\ pred\ (\mathrm{append}\ start\ [\mathrm{false}])$
```

Fig. 7. An implementation of bestshot in $\lambda_b$ with memoisation.

shot at finding a point $\pi$ satisfying given predicate and matching some specified list *start* in some initial segment of its components $[\pi(0), \ldots, \pi(i-1)]$. Here we use two customary operations on lists: we write $|-|$ for the list length function and nth for the function which projects the $n$th element of a list. The $\mathrm{bestshot}'_n$ function works 'lazily', drawing its values from *start* wherever possible, and performing an actual search only when required. This actual search is undertaken by $\mathrm{bestshot}''_n$, which proceeds by first searching for a solution that extends the specified list with true (using the standard list append function); 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 $q\ 0,\ldots,q\ (i-1)$ of its argument $q$.

The above program makes use of an operation

$$\mathrm{memoise} : (\mathrm{Unit} \to \mathrm{List\ Bool}) \to (\mathrm{Unit} \to \mathrm{List\ Bool})$$

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 $\lambda_b$ 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, $\mathrm{Bergercount}_n$ is implemented by means of two mutually recursive auxiliary functions. The function $\mathrm{count}'_n$ counts the solutions to the provided predicate *pred* that start with the specified list of booleans, adding their number to a previously accumulated total given by *acc*. The function $\mathrm{count}''_n$ does the same thing, but exploiting the knowledge that a best shot at the 'leftmost' solution to $P$ within this subtree has already been computed. (We are visualising $n$-points as forming a binary tree with true to the left of false at each fork.) Thus, $\mathrm{count}''_n$ will not re-examine the portion of the subtree to the left of this candidate solution, but rather will start at this solution and work rightward.

This gives rise to an $n$-count program that can work efficiently on predicates that tend to 'fail fast': more specifically, predicates $P$ that inspect the components of their argument

```
$\text{Bergercount}_n : \text{Predicate}_n \to \text{Nat}$
$\text{Bergercount}_n\ pred \overset{\text{def}}{=} \text{count}'_n\ pred\ []\ 0$

$\text{count}'_n : \text{Predicate}_n \to \text{List}_{\text{Bool}} \to \text{Nat} \to \text{Nat}$
$\text{count}'_n\ pred\ start\ acc \overset{\text{def}}{=}$ if $|start| = n$ then $acc +$ (if $pred\ (\lambda i.\text{nth}\ start\ i)$ then return 1
                                                                  else return 0)
                                   else let $f \leftarrow \text{bestshot}'_n\ pred\ start$ in
                                        if $pred\ f$ then $\text{count}''_n\ pred\ start\ [f\ 0, \ldots, f\ (n-1)]\ acc$
                                        else return $acc$

$\text{count}''_n : \text{Predicate}_n \to \text{List}_{\text{Bool}} \to \text{List}_{\text{Bool}} \to \text{Nat} \to \text{Nat}$
$\text{count}''_n\ pred\ start\ leftmost\ acc \overset{\text{def}}{=}$ if $|start| = n$ then $acc + 1$
                                              else let $b \leftarrow \text{nth}\ leftmost\ |start|$ in
                                                   let $acc' \leftarrow \text{count}''_n\ pred\ (\text{append}\ start\ [b])$
                                                                              $leftmost\ acc$ in
                                                   if $b$ then $\text{count}'_n\ pred\ (\text{append}\ start\ [\text{false}])\ acc'$
                                                   else return $acc'$
```

Fig. 8. An implementation of Berger count in $\lambda_b$.

$q$ in order $q$ 0, $q$ 1, $q$ 2, ..., and which are frequently able to return false after inspecting just a small number of these components. Generalising our program from binary to $k$-ary branching trees, we see that the $n$-queens problem provides a typical example: most points in the space can be seen not to be solutions by inspecting just the first few components. Our experimental results in Section 11 attest to the viability of this approach and its overwhelming superiority over the naïve functional method.

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

### *7.4 Pruned count*

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

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

```
$\text{Modulus} : \text{Predicate} \to \text{Point} \to (\text{Bool} \times \text{List}_{\text{Nat}})$
$\text{Modulus}\ pred\ point \overset{\text{def}}{=}$ let $log \leftarrow \text{ref}([\,] : \text{List}_{\text{Nat}})$ in
                             let $wrap \leftarrow \lambda i.(log := i :: !log;$ return $point\ i)$ in
                             let $b \leftarrow pred\ wrap$ in
                             return $\langle b, !log\rangle$
```

This is somewhat different from the modulus functional considered in Longley (1999), which returns a *sorted* list of the arguments to which the point is applied. The latter has the theoretically pleasant consequence that the modulus is an example of a *sequentially realisable* functional – its externally observable behaviour is purely functional (i.e. extensional) although the function it implements cannot be realised in pure $\lambda_b$. However, this property is purchased 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 $\langle b, ilist\rangle$, then 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*. This property can be used as the basis of a program

$$\mathrm{prunedcount}_n\ :((\mathrm{Nat}_n \to \mathrm{Bool}) \to \mathrm{Bool}) \to \mathrm{Nat}$$

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 $\lambda_\mathrm{b}$

The above discussion strongly suggests that the $\mathcal{O}(2^n)$ runtime of our $\lambda_\mathrm{h}$ generic count implementation is unattainable in $\lambda_\mathrm{b}$, 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 $\lambda_\mathrm{b}$ 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 $\lambda_\mathrm{a}$. 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; Hillerström et al., 2020). 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 $\lambda_b$ rather than the more elaborate abstract machine model employed in Section 4.1. This is because, as observed in Section 4.1, there is a tight correspondence between these two execution models such that for the evaluation of any closed term, the number of abstract machine steps is always at least the number of small-step reductions. Thus, if we are able to show that the number of small-step reductions for any generic count program in $\lambda_b$ on the predicates of interest is $\Omega(n2^n)$, this will establish the desired lower bound on the runtime.

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

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

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

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

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

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

Now consider the reduction sequence starting from $K\ (\lambda q.\ C[-])$ (treating the hole '$-$' as an additional variable). This cannot be infinite, for then the reduction of $K\ P$ would also be infinite, since valid reduction steps are closed under substituting true for '$-$'; thus $K$ would not correctly count all canonical $n$-standard predicates. Neither can this reduction terminate in a numeral $k$, for then both $K\ P$ and $K\ P'$ would evaluate to $k$ for a similar reason, whereas the correct results should differ by 1. Nor can it terminate in just the term '$-$', as this does not have the correct type. We conclude that the reduction of $K\ (\lambda q.\, C[-])$ gets stuck at some term with the hole in head position: more precisely, since '$-$' formally has type $\langle\rangle + \langle\rangle$, we see by inspection of the reduction rules that it must get stuck at some term $\mathcal{D}[\textbf{case}\ -\ \{\cdots\}]$, where $\mathcal{D}$ is an evaluation context. We write this term as $D[-]$, where the $D[\ ]$ abstracts only this head occurrence of the hole (there may well be other occurrences of the hole within $D$). From the form of evaluation contexts, we know that this hole occurrence does not appear under a $\lambda$ binder.

We now trace back through the reduction $K\ (\lambda q.\, C[-]) \rightsquigarrow^* D[-]$ looking at the ancestors of this occurrence of '$-$', and identifying the last point in the reduction at which this ancestor occurs within a descendant of the original $\lambda q.\, C[-]$. Since $C[-]$ has no free variables other than the hole occurrence, and the only rule for eliminating a $\lambda$ is S-App, 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 $[\![Q]\!] = \pi$ as required. For this, we 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 follows that $E[-]$ itself is a complex of nested **if**-expressions with branch conditions $Q(k)$ and with the hole at one of the leaves. It is now clear that the only way for this hole to become later exposed (as it is in $D[-]$) is for each of the branch conditions $Q(k)$ to evaluate to $\pi(k)$, so that the evaluation indeed follows the path $l$ and we have $E[-] \rightsquigarrow^* -$. But because $\tau$ is $n$-standard, each of $Q(0), \ldots, Q(n-1)$ occurs exactly once on this path, so the above is exactly the condition for $Q$ to be an $n$-point with value $\pi$. ■

**Corollary 1.** *Suppose $K$ and $P$ are as in Lemma 1. For any semantic $n$-point $\pi$ and any natural number $k < n$, the reduction sequence for $K\ P$ contains a term $\mathcal{F}[Q\ k]$, where $\mathcal{F}$ is an evaluation context and $[\![Q]\!] = \pi$.*

**Proof** Suppose $\pi \in \mathbb{B}^n$. By Lemma 1, the computation of $K\ P$ contains some $\mathcal{E}[P\ Q]$ where $[\![Q]\!] = \pi$, and the above analysis of the computation of $P\ Q$ shows that it contains a term $\mathcal{E}'[Q\ k]$ for each $k < n$. The corollary follows, taking $\mathcal{F}[-] \overset{\text{def}}{=} \mathcal{E}[\mathcal{E}'[-]]$. ■

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 $\pi$ from $\mathcal{F}[Q\ k]$ as $[\![Q]\!]$. There are thus at least $n2^n$ distinct terms in the reduction sequence for $K\ P$, so the reduction has length $\geq n2^n$. We have thus proved:

**Theorem 3.** *If $K$ is a $\lambda_b$ program that correctly counts all canonical $n$-standard $\lambda_b$ predicates, and $P$ is any canonical $n$-standard $\lambda_b$ 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 $\lambda_{\mathrm{b}}$ predicates).

It is worth noting where our argument breaks down if applied to $\lambda_{\mathrm{h}}$. In $\lambda_{\mathrm{b}}$, 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 $\pi$. This is intuitively why we may only learn about one point at a time. In $\lambda_{\mathrm{h}}$, this is not the case, because of the presence of operation symbols. For instance, our effcount program from Section 5.4 will apply $P$ to the 'generic point' $\lambda\_.\,\mathbf{do}\,\mathrm{Branch}\,\langle\rangle$. Thus, it need no longer be the case that the reduction of each term $Q\,k$ yields a value: it may get stuck at some invocation of $\ell$, 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 $\lambda_{\mathrm{b}}$, we now wish to show that the same bound applies for a much richer language $\lambda_{\mathrm{a}}$ 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 $\lambda_{\mathrm{a}}$ and $\lambda_{\mathrm{h}}$. Since affine effect handlers suffice for encoding many language features such as exceptions (Pretnar, 2015), local state (Plotkin & Pretnar, 2009), coroutines (Kawahara & 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 $\lambda_{\mathrm{a}}$, outlining its relationship to $\lambda_{\mathrm{h}}$ and $\lambda_{\mathrm{b}}$. 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 $\lambda_{\mathrm{a}}$.

Our language $\lambda_{\mathrm{a}}$ will be essentially a sublanguage of $\lambda_{\mathrm{h}}$ 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 $\lambda_{\mathrm{a}}$ as a dual intuitionistic-affine calculus*

We present the type system of $\lambda_{\mathrm{a}}$ in terms of dual-context judgements $\Delta; \Gamma \vdash \Box : A$, stating that a term $\Box$ (which may be a value term $V$ or a computation term $M$) has type $A$ under *intuitionistic type environment* $\Delta$ and *affine type environment* $\Gamma$. Informally, variables in the intuitionistic environment may be used zero, one or many times within $\Box$, while those in the affine environment may be used at most once.

As before, environments are lists assigning types to variables. For hygiene, we suppose we have disjoint lexical categories of intuitionistic and affine variables (each ranged over by metavariables $x, y$), and the variables within each of the environments $\Delta, \Gamma$ are required to be distinct.

The syntax of $\lambda_a$ is as follows:

| | |
|---|---|
| Types | $A, B, C, D ::= \mathrm{Nat} \mid \mathrm{Unit} \mid A \multimap B \mid A \otimes B \mid A \oplus B \mid !A$ |
| Type Environments | $\Delta ::= \cdot \mid \Delta, x : A$ |
| | $\Gamma ::= \cdot \mid \Gamma, x : A$ |
| Handler types | $F ::= C \Rightarrow D$ |
| Values | $V, W ::= x \mid k \mid c \mid \lambda x^A.\, M \mid \mathbf{rec}\, f^{A \to B}\, x.M$ |
| | $\mid\ \langle\rangle \mid \langle V, W\rangle \mid \mathbf{inl}^B\, V \mid \mathbf{inr}^A\, W \mid !W$ |
| Computations | $M, N ::= V\, W \mid \mathbf{let}\, \langle x, y\rangle = V\, \mathbf{in}\, N$ |
| | $\mid\ \mathbf{case}\, V\, \{\mathbf{inl}\, x \mapsto M; \mathbf{inr}\, y \mapsto N\}$ |
| | $\mid\ \mathbf{return}\, V \mid \mathbf{let}\, x \leftarrow M\, \mathbf{in}\, N \mid !M \mid \mathbf{let!}\, x = V\, \mathbf{in}\, N$ |
| | $\mid\ \mathbf{do}\, \ell\, V \mid \mathbf{handle}\, M\, \mathbf{with}\, H$ |
| Handlers | $H ::= \{\mathbf{val}\, x \mapsto M\} \mid \{\ell\, p\, r \mapsto N\} \uplus H$ |

The type constructors $\multimap$, $\otimes$, $\oplus$, and $!$ are borrowed from linear logic. Here we informally understand Nat, Unit, $A \multimap B$, $A \times B$, and $A \oplus B$ as types of values that may be used at most once, and $!A$ as a type of values of type $A$ that may be used as many times as desired. (The way DILL manifests the latter is to allow a value of $!A$ to be used at most once by binding it to an intuitionistic variable of type $A$ which can subsequently be used as many times as desired. Thus, technically $!A$ is affine just like all other types, but it provides access to an unlimited source of identical affine values of type $A$.)

The typing rules those shown in Figure 9, along with Exchange, Weakening and Contraction for the intuitionistic environment, and Exchange and Weakening (but not Contraction) for the affine one. To understand the workings of this type system, it is helpful to think of the affine arrow $\multimap$ and the affine environment $\Gamma$ 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 $\Gamma$ is of the form $!A$ for some $A$). Similarly, TL-BangComp allows promotion of computations. This latter form introduces some minor complications and is not strictly speaking necessary, but we include it nonetheless in order to admit a straightforward translation of let-binding in the inclusion $\lambda_b \hookrightarrow \lambda_a$ outlined at the end of this section.

The crucial restrictions in the rule for handlers are that the operation argument $p$ and the resumption variable $r$ are now affine. Notice that the success 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

**Values**

$$\overset{\textsc{TL-IVar}}{\dfrac{x : A \in \Delta}{\Delta; \Gamma \vdash x : A}} \qquad \overset{\textsc{TL-AVar}}{\dfrac{x : A \in \Gamma}{\Delta; \Gamma \vdash x : A}} \qquad \overset{\textsc{TL-Unit}}{\dfrac{}{\Delta; \Gamma \vdash \langle\rangle : \mathrm{Unit}}} \qquad \overset{\textsc{TL-Nat}}{\dfrac{k \in \mathbb{N}}{\Delta; \Gamma \vdash k : \mathrm{Nat}}} \qquad \overset{\textsc{TL-Const}}{\dfrac{c : A \multimap B}{\Delta; \Gamma \vdash c : A \multimap B}}$$

$$\overset{\textsc{TL-Lam}}{\dfrac{\Delta; \Gamma, x : A \vdash M : B}{\Delta; \Gamma \vdash \lambda x^{A}.\, M : A \multimap B}} \qquad \overset{\textsc{TL-Rec}}{\dfrac{\Delta, f : A \multimap B; x : A \vdash M : B}{\Delta; \cdot \vdash \mathbf{rec}\, f^{A \multimap B}\, x.\, M : A \multimap B}}$$

$$\overset{\textsc{TL-Prod}}{\dfrac{\Delta; \Gamma_1 \vdash V : A \qquad \Delta; \Gamma_2 \vdash W : B}{\Delta; \Gamma_1, \Gamma_2 \vdash \langle V, W\rangle : A \otimes B}} \qquad \overset{\textsc{TL-Inl}}{\dfrac{\Delta; \Gamma \vdash V : A}{\Delta; \Gamma \vdash \mathbf{inl}^{B}\, V : A \oplus B}} \qquad \overset{\textsc{TL-Inr}}{\dfrac{\Delta; \Gamma \vdash W : B}{\Delta; \Gamma \vdash \mathbf{inr}^{A}\, W : A \oplus B}}$$

$$\overset{\textsc{TL-Bang}}{\dfrac{\Delta; \Gamma \vdash W : A \qquad !\Gamma}{\Delta; \Gamma \vdash !W : !A}}$$

**Computations**

$$\overset{\textsc{TL-App}}{\dfrac{\Delta; \Gamma_1 \vdash V : A \multimap B \qquad \Delta; \Gamma_2 \vdash W : A}{\Delta; \Gamma_1, \Gamma_2 \vdash V\, W : B}} \qquad \overset{\textsc{TL-Split}}{\dfrac{\Delta; \Gamma_1 \vdash V : A \otimes B \qquad \Delta; \Gamma_2, x : A, y : B \vdash N : C}{\Delta; \Gamma_1, \Gamma_2 \vdash \mathbf{let}\, \langle x, y\rangle = V\, \mathbf{in}\, N : C}}$$

$$\overset{\textsc{TL-Case}}{\dfrac{\Delta; \Gamma_1 \vdash V : A \oplus B \qquad \Delta; \Gamma_2, x : A \vdash M : C \qquad \Delta; \Gamma_2, y : B \vdash N : C}{\Delta; \Gamma_1, \Gamma_2 \vdash \mathbf{case}\, V\, \{\mathbf{inl}\, x \mapsto M; \mathbf{inr}\, y \mapsto N\} : C}}$$

$$\overset{\textsc{TL-Return}}{\dfrac{\Delta; \Gamma \vdash V : A}{\Delta; \Gamma \vdash \mathbf{return}\, V : A}} \qquad \overset{\textsc{TL-Let}}{\dfrac{\Delta; \Gamma_1 \vdash M : A \qquad \Delta; \Gamma_2, x : A \vdash N : C}{\Delta; \Gamma_1, \Gamma_2 \vdash \mathbf{let}\, x \leftarrow M\, \mathbf{in}\, N : C}}$$

$$\overset{\textsc{TL-BangComp}}{\dfrac{\Delta; \Gamma \vdash M : C \qquad !\Gamma}{\Delta; \Gamma \vdash !M : !C}} \qquad \overset{\textsc{TL-LetBang}}{\dfrac{\Delta; \Gamma_1 \vdash V : !A \qquad \Delta, x : A; \Gamma_2 \vdash N : C}{\Delta; \Gamma_1, \Gamma_2 \vdash \mathbf{let}!\, x = V\, \mathbf{in}\, N : C}}$$

$$\overset{\textsc{TL-Do}}{\dfrac{(\ell : A \to B) \in \Sigma \qquad \Delta; \Gamma \vdash V : A}{\Delta; \Gamma \vdash \mathbf{do}\, \ell\, V : B}} \qquad \overset{\textsc{TL-Handle}}{\dfrac{\Delta; \Gamma_1 \vdash M : C \qquad \Delta; \Gamma_2 \vdash H : C \Rightarrow D}{\Delta; \Gamma_1, \Gamma_2 \vdash \mathbf{handle}\, M\, \mathbf{with}\, H : D}}$$

**Handlers**

$$\overset{\textsc{TL-Handler}}{\dfrac{\begin{array}{cc} H^{\mathrm{val}} = \{\mathbf{val}\, x \mapsto M\} & [H^{\ell} = \{\ell\, p\, r \mapsto N_{\ell}\}]_{\ell \in dom(\Sigma)} \\ \Delta; \Gamma, x : C \vdash M : D & [\Delta; p : A_{\ell}, r : B_{\ell} \multimap D \vdash N_{\ell} : D]_{(\ell : A_{\ell} \to B_{\ell}) \in \Sigma} \end{array}}{\Delta; \Gamma \vdash H : C \Rightarrow D}}$$

Fig. 9. Typing Rules for $\lambda_{\mathrm{a}}$

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 $\lambda_{\mathrm{h}}$, with the exception of $!W$, $!M$, and $\mathbf{let}!\, x = V\, \mathbf{in}\, N$.

To give a small-step operational semantics for $\lambda_{\mathrm{a}}$, we may therefore take all the operational rules for $\lambda_{\mathrm{h}}$ as given in Section 3 (along with the machinery of evaluation contexts and handler contexts), together with the new rules

$$\begin{array}{ll} \text{S-B}\textsc{ang} & \mathbf{let}\,!\, x = !W \,\mathbf{in}\, N \rightsquigarrow N[W/x] \\ \text{S-B}\textsc{ang}\text{C}\textsc{omp} & !(\mathbf{return}\ V) \rightsquigarrow \mathbf{return}\ !V \end{array}$$

and additional evaluation and handler contexts for promoted computations:

$$\begin{array}{ll} \text{Evaluation contexts} & \mathcal{E} ::= \cdots \mid !\mathcal{E} \\ \text{Handler contexts} & \mathcal{H} ::= \cdots \mid !\mathcal{H} \end{array}$$

As usual, we take $\rightsquigarrow^*$ to be the transitive closure of $\rightsquigarrow$, and define the notions of ancestor and descendant in the evident way. This completes our definition of $\lambda_a$.

The notion of normal form may now be defined just as in Definition 1. Once again, the following is straightforward to verify:

**Theorem 4** (Type Soundness for $\lambda_a$). *If* $\vdash M : C$*, then either there exists* $\vdash N : C$ *such that* $M \rightsquigarrow^* N$ *and* $N$ *is normal with respect to* $\Sigma$*, or* $M$ *diverges.*

### *9.2 Relationship to* $\lambda_h$ *and* $\lambda_b$

We now outline how we intend to view $\lambda_a$ as a sublanguage of $\lambda_h$, and $\lambda_b$ as a sublanguage of $\lambda_a$. 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.

For the inclusion $\lambda_a \hookrightarrow \lambda_h$, the broad idea is simply that any well-typed term of $\lambda_a$ will certainly remain well-typed if all affineness 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

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

The translation on terms is given in the obvious way for the syntactic forms common to $\lambda_a$ and $\lambda_h$, the three new forms being treated by

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

A typing judgement $\Delta; \Gamma \vdash \Box : A$ of $\lambda_a$ then becomes a judgement $(\Delta, \Gamma)^\dagger \vdash \Box^\dagger : A^\dagger$, where $\Delta, \Gamma$ is the result of rolling $\Delta$ and $\Gamma$ 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 $\lambda_a$ yields one in $\lambda_h$, and also that if $M \rightsquigarrow M'$ in $\lambda_a$ then $M^\dagger \rightsquigarrow^* M'^\dagger$ in $\lambda_h$.

For the inclusion $\lambda_b \hookrightarrow \lambda_a$, we give a translation $(-)^\star$ inspired by 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.

$$\begin{aligned} \mathrm{Nat}^\star &= \mathrm{Nat} \\ \mathrm{Unit}^\star &= \mathrm{Unit} \end{aligned}$$

$$\begin{aligned}(A \to B)^\star &= !(A^\star) \multimap B^\star\\ (A \times B)^\star &= !(A^\star) \otimes !(B^\star)\\ (A + B)^\star &= !(A^\star) \oplus !(B^\star)\end{aligned}$$

A type environment $\Gamma$ of $\lambda_b$ translated to the environment $\Gamma^\star; \cdot$ of $\lambda_a$: 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.

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

Once again, it is routine to check that every derivable typing judgement in $\lambda_b$ yields one in $\lambda_a$, and that if $M \leadsto M'$ in $\lambda_b$ then $M^\star \leadsto^* M'^\star$ in $\lambda_a$.

## 10 Affine effect computations and generic count

We begin with some general machinery for managing resumptions and for tracking the evaluation of subterms through reductions, allowing for the 'thread-switching' behaviour that $\lambda_a$ supports. We expect that this machinery will be quite widely applicable to any kind of reasoning about the behaviour of effectful programs in $\lambda_a$. In Section 10.3, we apply this machinery to the specific scenario of the generic count problem.

Throughout this section, 'subterm' will always mean 'subterm occurrence'.

### *10.1 Tracking of resumptions*

To make the role of resumptions more explicit, it will be convenient to recast the small-step operational semantics for $\lambda_a$ slightly, presenting it as a reduction system for pairs $\langle M \mid \Xi\rangle$, where $M$ is a term and $\Xi$ is a *resumption environment*, mapping finitely many *resumption variables* $\hat{r}$ to terms of the form $\lambda y.\,\mathbf{handle}\,\mathcal{E}[\mathbf{return}\,y]\,\mathbf{with}\,H$. Note that these terms may themselves involve other resumption variables.

The only reduction rules in which $\Xi$ plays an active role are the following. We write $\Xi\backslash\hat{r}$ for $\Xi$ with the entry for $\hat{r}$ deleted.

$$\begin{array}{lll}\text{S-OP}' & \langle \mathbf{handle}\,\mathcal{E}[\mathbf{do}\,\ell\;V]\,\mathbf{with}\,H \mid \Xi\rangle \leadsto \langle N[V/p,\,\hat{r}/r] & \\ & \qquad \mid \Xi, \hat{r} \mapsto \lambda y.\,\mathbf{handle}\,\mathcal{E}[\mathbf{return}\,y]\,\mathbf{with}\,H\rangle & \\ & \qquad \text{where } H^\ell = \{\ell\,p\,r \mapsto N\},\ \hat{r}\text{ fresh} & \\ \text{S-RES} & \langle \hat{r}W \mid \Xi\rangle \leadsto \langle R[W/y] \mid \Xi\backslash\hat{r}\rangle & \text{where } \Xi(\hat{r}) = \lambda y.\,R\\ \text{S-LIFT}' & \langle \mathcal{H}[M] \mid \Xi\rangle \leadsto \langle \mathcal{H}[M'] \mid \Xi'\rangle, & \text{if } \langle M \mid \Xi\rangle \leadsto \langle M' \mid \Xi'\rangle\end{array}$$

All other reduction rules are carried over from the original semantics in the obvious way: for each reduction rule $M \rightsquigarrow M'$ except for S-Op, we now have a rule $\langle M \mid \Xi \rangle \rightsquigarrow \langle M' \mid \Xi \rangle$. To initiate a reduction sequence for a closed term $M$, we start from the configuration $\langle M \mid \emptyset \rangle$.

The main purpose of this semantics is to make explicit the points at which resumptions are invoked (as the points at which S-Res is applied). In the original semantics, such steps appear simply as $\beta$-reductions, which may not be distinguishable, on the face of it, from other $\beta$-reductions that occur.

It is intuitively clear that the reduction of $\langle M \mid \emptyset \rangle$ under the new semantics proceeds in lockstep with the reduction of $M$ under the original semantics. One half of this is formalised by the following proposition (we write $\rightsquigarrow^m$ for reduction in exactly $m$ steps).

**Proposition 2.** *For any $m$, if $\langle M \mid \emptyset \rangle \rightsquigarrow^m \langle M' \mid \Xi \rangle$ then $M \rightsquigarrow^m M''$, where $M''$ is obtained from $M'$ by repeatedly expanding all resumption variables as specified by $\Xi$ until no longer possible.*

**Proof** Easy induction on $m$. Note that in the case of S-Op$'$, the number of rounds of expansion needed may increase by 1. ■

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 $M'$, $\Xi$ — 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 *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 $\lambda_a$, the critical typing restriction is that in the handler clauses $\ell\, p\, r \mapsto N_\ell$, the variable $r$ is used *affinely* within $N_\ell$. This does not mean that $r$ can occur at most once within $N_\ell$ (in view of the TL-Case rule); and even if it does, the variable $r$ may subsequently be copied in the course of a $\beta$-reduction (again because of TL-Case). However, the affineness restriction does buy us the following crucial property:

**Lemma 2** (Single-shot resumptions)**.** *For any naturally arising $\langle M \mid \Xi \rangle$ and any $\hat{r} \in dom(\Xi)$, 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 in this situation, $M$, $\Xi$ satisfy the following two conditions, writing $\hat{r}_0, \ldots, \hat{r}_{k-1}$ for the elements of $dom(\Xi)$.

1. $\hat{r}$ appears in at most one of the $k+1$ terms $M, \Xi(\hat{r}_0), \ldots, \Xi(\hat{r}_{k-1})$.
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 $\Xi(\hat{r}_i)$. Condition 2 is a general property of occurrences of affine 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 affine 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** $\mathcal{E}$[**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 affineness of $p$ within $N$, $\hat{r}$ 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 $\Xi$, $\hat{r}$ may appear in $R[W/y]$ 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 $\Xi$ 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 within $\mathcal{H}[\hat{r}W]$ or the range of $\Xi'$, and it follows that $\hat{r}$ does not appear at all within the resulting configuration $\langle \mathcal{H}[R[W/y]] \mid \Xi' \backslash \hat{r} \rangle$. 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 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 $M'$, $\Xi$.*

**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 within the proof of Lemma 2.

(ii) From (i) we know that an application of S-RES will never be blocked by the failure of a lookup $\Xi(\hat{r})$ fails. A reduction $M \rightsquigarrow^m M''$ can therefore be lifted to one $\langle M \mid \emptyset \rangle \rightsquigarrow^m \langle M' \mid \Xi \rangle$ where $M', \Xi, M''$ are related as in Proposition 2, by induction on $m$ and an easy comparison between the two reduction systems. ■

Lemma 2 is the crucial property of $\lambda_a$ on which our whole argument hinges. This property is flagrantly violated by $\lambda_h$, 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 $\mathcal{H}$. We introduce the following concepts for tracking the evaluation of $S$ through the reduction of $M$ with respect to some resumption context $\Xi$.

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

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

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 four possibilities:

1. The reduction of $\langle S \mid \Xi \rangle$ may continue forever.
2. The reduction may terminate in some $\langle \mathbf{return}\ V \mid \Xi' \rangle$ where $V$ is a value.
3. The reduction of $\langle S \mid \Xi \rangle$ 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 anywhere within $\mathcal{H}[\mathcal{E}[\mathbf{do}\ \ell\ V]]$ — in this situation, we say the entire computation is *absolutely blocked*.
4. The reduction may get stuck at some $\langle \mathcal{E}[\mathbf{do}\ \ell\ V] \mid \Xi' \rangle$, where the $\mathbf{do}\ \ell\ V$ is not handled within $\mathcal{E}[\mathbf{do}\ \ell\ V]$ itself, but is handled further out within $\mathcal{H}[-]$.

In case 4, $\mathcal{H}[-]$ will have the form $\mathcal{H}'[\mathbf{handle}\ \mathcal{F}[-]\ \mathbf{with}\ H]$ where $\mathcal{F}$ is 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

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

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

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

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 to say that $\langle S \mid \Xi \rangle$ gives rise in this context to a *pseudo-reduction sequence*

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

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 $S$:

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

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 be absolutely blocked, or it may end with a dormant reduct in a resumption environment entry that is never subsequently activated, or it may terminate in some $\langle \mathbf{return}\,V \mid \Xi_i' \rangle$ where $V$ is a 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.

It is also clear that the notions of ancestor and descendant make sense for subterms appearing within pseudo-reduction sequences, providing one considers configurations $\langle S \mid \Xi \rangle$ as a whole: a subterm within the main term may have descendants within the resumption environment, and *vice versa*. For a pseudo-reduction step $\langle S \mid \Xi \rangle \rightsquigarrow^! \langle S' \mid \Xi' \rangle$, we say a subterm of the right-hand side is a descendant of one on the left iff it is a descendant with respect to the genuine reduction sequence that witnesses this pseudo-step.

### *10.3 Application to generic count*

We now apply the above notions to the analysis of generic counting in $\lambda_a$, obtaining a lower bound analogous to that of Theorem 3 for $\lambda_b$. Proceeding as in Section 8, we fix $n \in \mathbb{N}$, and suppose that $K$ is some program of $\lambda_a$ that correctly counts all canonical $n$-standard predicates $P$ (noting that all such predicates are actually terms from our base language $\lambda_b$). Once again, focusing 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 $\lambda_b$ as included in $\lambda_a$ via the *intuitionistic encoding* $(-)^\star$ defined in Section 9. Since our intention is that our lower bound for $\lambda_a$ should generalise the one for $\lambda_b$, this means that the types Point and Predicate appear within $\lambda_a$ as

$$\mathrm{Point} \stackrel{\mathrm{def}}{=} !\mathrm{Nat} \multimap \mathrm{Bool}, \qquad \mathrm{Predicate} \stackrel{\mathrm{def}}{=} !\mathrm{Point} \multimap \mathrm{Bool}$$

Formally, then, we will be considering the reduction behaviour of $\langle K\,(!P) \mid \emptyset \rangle$, where $P : \mathrm{Predicate}$ is the $\star$-translation of a canonical $n$-standard predicate, $K$ is a generic count program of $\lambda_a$ assumed to count all such predicates correctly. (We may assume without loss of generality that $K$ is a closed term.) By hypothesis, this reduction will terminate in some $\langle \textbf{return}\, k \mid \Xi_{end} \rangle$ where $k$ is a numeral.

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 descendant of the original $P$. As a significant consequence of our typing of $P$, the argument $Q$ here will be of type !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 the canonical predicate $P$ (see Definition 8), it is clear that this pseudo-reduction sequence will have the following form, or some initial portion thereof:

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

Here each $\mathcal{E}_i[-]$ has its unique hole occurrence at the head of an **if**-expression corresponding to one of the nested **if**-expressions within $P$ itself.

We think of the above pseudo-reduction as a sequence of '$P$-phases' alternating with '$Q$-phases'. The former are the portions designated by $\rightsquigarrow^*$: these are 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 $\langle K\,(!P) \mid \emptyset \rangle \rightsquigarrow^* \langle \textbf{return}\, k \mid \Xi_{end} \rangle$ is finite. Likewise, the pseudo-reduction for $P\,Q$ can never block absolutely, as this too would prevent the whole computation from completing. Nonetheless, it is quite 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 $b_0 \ldots b_{n-1}$ through this tree. Bearing in mind that $P$ is assured to be $n$-standard, we may conclude that $\{k_0, \ldots, k_{n-1}\} = \{0, \ldots, n-1\}$ and that $\mathcal{U}(P)(b_0 \ldots b_{n-1}) = !b$.

We may now, 'with hindsight', identify the semantic $n$-point to which $P$ was in effect applied: namely, the point $\pi$ given by $\pi(k_i) = b_i$ for each $i$. Notice that in the setting of $\lambda_b$, this semantic point could be read off at the point of application simply as $[\![Q]\!]$; 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* $\pi$. Since, as we have seen, the point $\pi$ may be read off from the pseudo-reduction sequence for $P\,Q$, we have shown:

**Lemma 3.** *No application of $P$ can be an application of $P$ to more than one point $\pi$.*

The crucial claim is now the following:

**Lemma 4.** *For each of the $2^n$ semantic n-points $\pi$, the reduction of $\langle K\,(!P) \mid \emptyset\rangle$ contains an application of $P$ to $\pi$.*

**Proof** Essentially the same argument as for Lemma 1. We may suppose $P = P(\tau)$. Given any semantic point $\pi$, we may identify the associated path $b_0 \ldots b_{n-1}$ through $\tau$ and the corresponding leaf literal within the body of $P$; we write $C[-]$ for the context that abstracts on this leaf literal. Assuming without loss of generality that this literal is true, we then have $P \equiv \lambda q.\, C[\mathrm{true}]$, and we also set $P' \equiv \lambda q.\, C[\mathrm{false}]$.

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 point reach the head of a **case** expression in evaluation position, and by looking at the last ancestor of this hole occurrence that does not appear within a descendant of the original $\lambda q.\, C[-]$, we see that at this point we have a configuration $\langle \mathcal{H}[(\lambda q.\, C[-])Q] \mid \Xi\rangle$ with $Q$ a closed term of type Point. This reduces in the next step to $\langle \mathcal{H}[E[-]] \mid \Xi\rangle$, where $E[-] \equiv C[-][Q/q]$. (Note that we are here considering the entire top-level computation and are tracking subterms through genuine reductions, not pseudo-reductions.)

In the present setting, we cannot conclude that $[\![Q]\!] = \pi$ — indeed, $[\![Q]\!]$ as defined in Definition 2 has no clear meaning if $Q$ invokes operations that it cannot handle. Nevertheless, it is again the case that $E[-]$ is a complex of nested **if** expressions with various branch conditions $Q\,k$, and with the unique hole at the leaf at position $l$. The idea now is that the only way for the hole to become later exposed at the head of an active **case** expression is for these enclosing **if** expressions to be successively stripped away until the hole is exposed, and this can only happen if each of the relevant conditions $Q\,k$ evaluates (by pseudo-reduction) to the corresponding $\pi(k)$.

More formally, we have that $E[-]$ has the form **if** $Q\,k_0 \;\cdots$, which desugars to **let** $z \leftarrow Q\,k_0$ **in case** $z\,\{\cdots\}$. Now consider the pseudo-reduction sequence for $Q\,k_0$. This cannot continue for ever or be absolutely blocked, for then the same would happen with true substituted for '$-$' and the computation of $P(!Q)$ would not complete. We also claim that it cannot end with a dormant reduct that is never reactivated. To see this, we first note that the pseudo-reduction sequence for **let** $z \leftarrow Q\,k_0$ **in** $\cdots$ exactly tracks the one for $Q\,k_0$ for as far as this latter pseudo-reduction extends (this is easy to check, since no handler intervenes between these terms). So if the sequence for $Q\,k_0$ ended in a dormant reduct, the same would be true for **let** $z \leftarrow Q\,k_0$ **in** $\cdots$, whose pseudo-reduction sequence carries with it the critical hole occurrence. Thus, this hole occurrence would remain forever dormant in the resumption environment and so would never become exposed.

We conclude that the pseudo-reduction of $Q\,k_0$ must complete with some boolean value $b$, so that the enclosing **let** expression reduces to **case** $b\,\{\cdots\}$. Furthermore, this value $b$ must be $b_0 = \pi(k_0)$ if the hole is not to be eliminated at the next step. After applying one of the S-Case rules, we are now left with one of the **if** expressions from the second level of the original $E[-]$, say with branch condition $Q\,k_1$, and the same argument now shows that this must pseudo-reduce to the boolean value $b_1 = \pi(k_1)$. Continuing in this way, we arrive at a pseudo-reduction of $\langle(\lambda q.\, C[-])Q \mid \Xi\rangle$ to **return** $-$, and under specialisation of '$-$' to true, we obtain precisely a successful pseudo-reduction of $\langle P\, Q \mid \Xi\rangle$ to true. Moreover, since our computation has traced the path $b_0 \dots b_{n-1}$ through the term structure of $P$, we have here an application of $P$ to $\pi$ in the sense introduced above. ■

**Theorem 5.** *If $K$ is a $\lambda_a$ program that correctly counts all canonical $n$-standard $\lambda_b$ predicates, and $P \equiv \lambda q.\, C[\mathrm{true}]$ is a canonical $n$-standard $\lambda_b$ predicate, then the evaluation of $K\,(!P)$ takes time $\Omega(n2^n)$.*

**Proof** It is clear from Lemmas 3 and 4 that in the evaluation of $K\,(!P)$ there are at least $2^n$ successful applications of $P$, and that each of these involves, at the very least, the $n$ S-CASE reductions of the subterms $\mathcal{E}_i[b_i] \equiv \textbf{if}\, b_i\ \cdots \equiv \textbf{case}\, b_i\, \{\cdots\}$ (each of which costs one abstract machine step). To complete the proof, we just need to check that none of these reduction steps can be shared by two successful applications of $P$ associated with different semantic points $\pi, \pi'$. For this, we show that given such an S-CASE reduction step, we may uniquely locate the application $P\, Q$ that gave rise to it.

Suppose, then, that within the evaluation of $K\,(!P)$ we are given such a reduction step, reducing a subterm **case** $b_i\, \{\textbf{inl}\, \langle\rangle \mapsto R; \textbf{inr}\, \langle\rangle \mapsto R'\}$ to $R$ or $R'$ as appropriate. Note that this subterm does not appear within a $\lambda$ expression, since we never reduce under a $\lambda$. Moreover, if this reduction step indeed features in the pseudo-reduction for some $P\, Q$ as displayed above, then $R$ and $R'$ are themselves descendants of subterms within $C[\mathrm{true}][Q/q]$, and indeed are either boolean literals or expressions $\textbf{if}\, Q\, k'\ \cdots$.

Let us now trace the ancestors of $R$ (say) back as far as possible through the entire computation of $K\,(!P)$. There are two cases. If $R$ is a boolean **return** $b$, this will have an ancestor within $P$ itself in the application $P\, Q$, and it is clear from the displayed form of the pseudo-reduction that this will be the latest ancestor that occurs under a $\lambda$ (within the main term as opposed to the resumption environment); and this property serves to pinpoint the application $P\, Q$. If $R \equiv \textbf{if}\, Q\, k'\ \cdots$, then its earliest ancestor will be within the term $C[\mathrm{true}][Q/q]$ to which $P\, Q$ contracts, when $R$ came into existence as the result of the substitution $[Q/q]$. Once again, this information suffices to pinpoint $P\, Q$. Thus, it is uniquely determined with which successful application the given S-CASE step is associated. ■

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 5 and 2 highlight the efficiency gap between $\lambda_a$ and $\lambda_h$.

## 11 Experiments

The theoretical efficiency gap between realisations of $\lambda_b$ and $\lambda_h$ 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.

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

| | First solution | | | All solutions | | |
|---|---|---|---|---|---|---|
| *Parameter* | 20 | 24 | 28 | 8 | 10 | 12 |
| Naïve | – | – | – | 353.67 | 5903.77 | – |
| Berger | 11.36 | 17.93 | 25.85 | 1.71 | 1.67 | 1.60 |
| Pruned | 3.45 | 4.46 | 5.01 | 1.60 | 1.67 | 1.71 |
| Bespoke | 0.21 | 0.24 | 0.28 | 0.17 | 0.18 | 0.19 |

Table 2. Runtime of exact real integration procedures relative to the effectful implementation

| | Id | Squaring | | | Logistic | | | | |
|---|---|---|---|---|---|---|---|---|---|
| *Parameter* | 20 | 14 | 17 | 20 | 1 | 2 | 3 | 4 | 5 |
| Naïve | 4.48 | 26.58 | 30.65 | 28.99 | 21.89 | 61.18 | 45.29 | – | – |
| Berger | 1.21 | 11.23 | 15.95 | 8.89 | 8.18 | 19.35 | 12.52 | 11.29 | 11.53 |
| Pruned | 1.07 | 2.11 | 1.56 | 1.71 | 1.59 | 4.04 | 4.02 | 3.63 | 3.50 |

**Methodology.** For both our search problems, we consider general predicates rather than only $n$-standard predicates. We evaluate 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 (lower is better). 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 5.1.1` with factory settings on an AMD Ryzen 9 5900X 12-core CPU powered workstation running Ubuntu 22.04.

OCaml supports only single-shot continuations out-of-the-box, but the effectful procedure requires multi-shot continuations to function correctly. Therefore, we used the package `multicont 1.0.1`, 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 later. In contrast to OCaml's continuations, the multi-shot continuations provided by this package are garbage collected.

**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 $\mathbb{N}_n \to \mathbb{N}_n$ rather than just $\mathbb{N}_n \to \mathbb{B}$: a potential solution or 'point' corresponds to a vector $(q_0, \ldots, q_{n-1})$ 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 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 $f : [0, 1] \to [0, 1]$ by a function on such streams. Our task is to evaluate the definite integral $\int_0^1 f$ to within any specified error bound $\epsilon > 0$. Informally, we can achieve this if for every $x \in [0, 1]$ we know the value of $f(x)$ to within $\epsilon$. We may therefore proceed as follows. First, we evaluate $f$ to the required precision at the real 0, represented by the stream $0, 0, 0, \ldots$. We will obtain a result $v$ 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 $|f(0) - v| < \epsilon$, but that $|f(x) - v| < \epsilon$ for any $x \in [0, 2^{-k}]$. This gives us a contribution of $v.2^{-k}$ towards our integral. We therefore continue by evaluating $f$ to the required precision at the right endpoint $2^k$ of this interval, represented by the stream $0^{k-1}, 1, 0, 0, 0 \ldots$ (where $0^{k-1}$ is a sequence of $k-1$ zeros). 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. It cannot happen, for example, that our sampled $x$ values accumulate at some $\Sigma_{i=0}^{\infty} 2^{-k_i}$ where $k_0 < k_1 < \cdots$, since the evaluation of $f$ to the required precision on the input stream $0^{k_0-1}, 1, 0^{k_1-k_0-1}, 1, 0^{k_2-k_1-1}, 1, \ldots$ is itself guaranteed to consume only finitely many digits, so once these digits are present, we will be able to advance further than this supposed limit point.) At this point, we have enough information to return the value of the integral to within $\epsilon$.

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 $0, 0, 0, \ldots$ and on $0^{k-1}, 1, 0, 0, 0 \ldots$ will proceed identically up to the point where the $k$th input digit is examined. With general effect handlers, however, such an optimisation 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 (that is, computing the integral to within $2^{-20}$), the pruned procedure is competitive with the effectful procedure. Though the effectful procedure beats the Berger procedure, providing a relative speedup of $1.21\times$. For the squaring function, the speedups over the Berger procedure are between 9 and $16\times$, whereas the pruned procedure remains more competitive as the effectful procedure provides only a modest speedup of between $1.5 - 2.1\times$. 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. The relative speedup over the Berger procedure is $8 - 19\times$, whereas the speedup over the pruned procedure is $1.5 - 4\times$.

## 12 Conclusions and future work

We presented a PCF-inspired language $\lambda_{\mathrm{b}}$, an extension $\lambda_{\mathrm{h}}$ with general effect handlers, and a milder extension $\lambda_{\mathrm{a}}$ with affine effect handlers. We proved that $\lambda_{\mathrm{h}}$ supports an asymptotically more efficient implementation of generic search than any possible implementation in $\lambda_{\mathrm{b}}$ or even $\lambda_{\mathrm{a}}$. We observed this effect in practice on several benchmarks. Since $\lambda_{\mathrm{a}}$ 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 $\lambda_{\mathrm{h}}$ 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 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 $\lambda_a$ and $\lambda_h$ 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 $\lambda_i$ (a language with iteration but not recursion), one cannot systematically (and uniformly in $n$) achieve either 'false' pruning or 'true' pruning.
- In $\lambda_b$, we may systematically and uniformly achieve 'false' pruning but not 'true' pruning (Bergercount).
- In $\lambda_a$, we may systematically achieve both 'false' and 'true' pruning (prunedcount), but no sharing of computations is possible.
- In $\lambda_h$, pruning and sharing of computations are systematically possible (effcount).

In this paper, we have established 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.

## Data availability statement

The data and code that support the experiments in Section 11 are openly available via `https://doi.org/10.5281/zenodo.10671782`.

## 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. We are grateful to one of the reviewers of our ICFP paper for the encouragement to establish a lower bound result for arbitrary affine effects.

## Funding

This work was supported by the UKRI Future Leaders Fellowship "Effect Handler Oriented Programming" (S.L., reference number MR/T043830/1).

## Conflicts of interest

We (the authors) certify that we have no competing interests to declare that are relevant to the content of this article.

## Author ORCID

D. Hillerström, https://orcid.org/0000-0003-4730-9315; S. Lindley, https://orcid.org/0000-0002-1360-4714.

## References

Barber, A. (1996) *Dual Intuitionistic Linear Logic*. Technical report ECS-LFCS-96-347, University of Edinburgh.

Barendregt, H. P. (1984) *The Lambda Calculus: Its Syntax and Semantics*, revised ed. North-Holland.

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. *J. Funct. Program.* **11**(4), 395–410.

Benton, P. (1994) A mixed linear and non-linear logic: Proofs, terms and models. In CSL. Springer, pp. 121–135.

Berger, U. (1990) *Totale Objekte und Mengen in der Bereichstheorie*. Ph.D. thesis. Munich: Ludwig Maximillians-Universtität.

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) Effects as capabilities: Effect handlers and lightweight effect polymorphism. *Proc. ACM Program. Lang.* **4**(OOPSLA), 126:1–126:30.

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. In POPL. ACM Press, pp. 328–342.

Convent, L., Lindley, S., McBride, C. & McLaughlin, C. (2020) Doo bee doo bee doo. *J. Funct. Program.* **30**, e9.

Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C. (2009) *Introduction to Algorithms*, 3rd ed. MIT Press.

Daniels, R. (2016) *Efficient Generic Searches and Programming Language Expressivity*. Master's thesis. UK: School of Informatics, the University of Edinburgh.

Danvy, O. & Filinski, A. (1990) Abstracting control. In LISP and Functional Programming. ACM, pp. 151–160.

Dolan, S., White, L., Sivaramakrishnan, K., Yallop, J. & Madhavapeddy, A. (2015) Effective concurrency through algebraic effects. In OCaml Workshop.

Escardó, M. H. (2007) Infinite sets that admit fast exhaustive search. In LICS. IEEE Computer Society, pp. 443–452.

Farvardin, K. & Reppy, J. H. (2020) From folklore to fact: Comparing implementations of stacks and continuations. In 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. Indianapolis, IN, USA: Department of Computer Science. AAI8727494.

Felleisen, M. (1988) The theory and practice of first-class prompts. In POPL. ACM Press, pp. 180–190.

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

Felleisen, M. & Friedman, D. P. (1987) Control operators, the SECD-machine, and the λ-calculus. In The Proceedings of the Conference on Formal Description of Programming Concepts III, Ebberup, Denmark. Elsevier, pp. 193–217.

Flanagan, C., Sabry, A., Duba, B. F. & Felleisen, M. (1993) The essence of compiling with continuations. In PLDI. ACM, pp. 237–247.

Flatt, M. & Dybvig, R. K. (2020) Compiler and runtime support for continuation marks. In PLDI. ACM, pp. 45–58.

Forster, Y., Kammar, O., Lindley, S. & Pretnar, M. (2019) On the expressive power of user-defined effects: Effect handlers, monadic reflection, delimited control. *J. Funct. Program.* **29**, e15.

Girard, J.-Y. (1987) Linear logic. *Theor. Comp. Sci.* **50**, 1–101.

Hillerström, D. (2021) *Foundations for Programming and Implementing Effect Handlers*. Ph.D. thesis. UK: The University of Edinburgh.

Hillerström, D. & Lindley, S. (2016) Liberating effects with rows and handlers. In TyDe@ICFP. ACM, pp. 15–27.

Hillerström, D., Lindley, S. & Atkey, R. (2020) Effect handlers via generalised continuations. *J. Funct. Program.* **30**, e5.

Hillerström, D., Lindley, S., Atkey, R. & Sivaramakrishnan, K. C. (2017) Continuation passing style for effect handlers. In FSCD. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 18:1–18:19.

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., Lindley, S. & Longley, J. (2020) Effects for efficiency: Asymptotic speedup with first-class control. CoRR. abs/2007.00605.

Hughes, J. (1986) A novel representation of lists and its application to the function "reverse". *Inf. Process. Lett.* **22**(3), 141–144.

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

Kammar, O., Lindley, S. & Oury, N. (2013) Handlers in action. In ICFP. ACM, pp. 145–158.

Kawahara, S. & Kameyama, Y. (2020) One-shot algebraic effects as coroutines. In TFP. Springer, pp. 159–179.

Kiselyov, O., Sabry, A. & Swords, C. (2013) Extensible effects: An alternative to monad transformers. In Haskell. ACM. pp. 59–70.

Kiselyov, O., Shan, C., Friedman, D. P. & Sabry, A. (2005) Backtracking, interleaving, and terminating monad transformers: (functional pearl). In ICFP. ACM, pp. 192–203.

Knuth, D. (1997) *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, 3rd ed. Addison-Wesley.

König, D. (1927) Über eine Schlussweise aus dem Endlichen ins Unendliche. *Acta Sci. Math. (Szeged)* **3**(2–3), 121–130.

Launchbury, J. & Jones, S. L. P. (1994) Lazy functional state threads. In PLDI. ACM, pp. 24–35.

Leijen, D. (2017) Type directed compilation of row-typed algebraic effects. In POPL. ACM, pp. 486–499.

Levy, P. B., Power, J. & Thielecke, H. (2003) Modelling environments in call-by-value programming languages. *Inf. Comput.* **185**(2), 182–210.

Lindley, S., McBride, C. & McLaughlin, C. (2017) Do be do be do. In POPL. ACM, pp. 500–514.

Longley, J. (1999) When is a functional program not a functional program? In ICFP. ACM, pp. 1–7.

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

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

Longley, J. & Normann, D. (2015) *Higher-Order Computability*. Theory and Applications of Computability. Springer.

McCracken, N. (1984) The typechecking of programs with implicit type structure. In Semantics of Data Types. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 301–315.
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. In POPL. ACM, pp. 104–109.
Piróg, M., Polesiuk, P. & Sieczkowski, F. (2019) Typed equivalence of effect handlers and delimited control. In FSCD. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 30:1–30:16.
Plotkin, G. D. (1977) LCF considered as a programming language. *Theor. Comput. Sci.* **5**(3), 223–255.
Plotkin, G. D. & Power, J. (2001) Adequacy for algebraic effects. In FoSSaCS. Springer, pp. 1–24.
Plotkin, G. D. & Pretnar, M. (2009) Handlers of algebraic effects. In ESOP. Springer, pp. 80–94.
Plotkin, G. D. & Pretnar, M. (2013) Handling algebraic effects. *Logical Methods Comput. Sci.* **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. In MFCS. Springer, pp. 456–464.
Sivaramakrishnan, K. C., Dolan, S., White, L., Kelly, T., Jaffer, S. & Madhavapeddy, A. (2021) Retrofitting effect handlers onto OCaml. In PLDI. ACM, pp. 206–221.
Sperber, M., Dybvig, K. R., Flatt, M., van Stratten, A., Findler, R. B. & Matthews, J. (2009) Revised$^6$ 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 A1 describes such a mapping $(\!|-|\!)$ 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, $\gamma$, which we define now.

**Definition 10.** *We write dom$(\gamma)$ for the domain of $\gamma$, and $\gamma\backslash\{x_1,\ldots,x_n\}$ for the restriction of environment $\gamma$ to dom$(\gamma)\backslash\{x_1,\ldots,x_n\}$.*

The $(\!|-|\!)$ 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 $(\!|-|\!)$ is invariant under this rule. This leaves the $\beta$-rules (M-App), (M-Split), (M-Case), and (M-RetCont). 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 $\longrightarrow_\beta$ for $\beta$-steps (all other rules) and $\Longrightarrow$ for a sequence of steps of the form $\longrightarrow_a^*\longrightarrow_\beta$.*

**Configurations**

$$(\!|\langle M \mid \gamma \mid \sigma\rangle|\!) = (\!|\sigma|\!)((\!|M|\!)\gamma)$$

**Pure continuations**

$$\begin{aligned}(\!|[\,]|\!)M &= M\\ (\!|(\gamma, x, N) :: \sigma|\!)M &= (\!|\sigma|\!)(\mathbf{let}\, x \leftarrow M \,\mathbf{in}\, (\!|N|\!)(\gamma\backslash\{x\}))\end{aligned}$$

**Computation terms**

$$\begin{aligned}(\!|V\ W|\!)\gamma &= (\!|V|\!)\gamma\ (\!|W|\!)\gamma\\ (\!|\mathbf{let}\,\langle x; y\rangle = V \,\mathbf{in}\, N|\!)\gamma &= \mathbf{let}\,\langle x; y\rangle = (\!|V|\!)\gamma \,\mathbf{in}\, (\!|N|\!)(\gamma\backslash\{x, y\})\\ (\!|\mathbf{case}\, V\,\{\mathbf{inl}\, x \mapsto M; \mathbf{inr}\, y \mapsto N\}|\!)\gamma &= \mathbf{case}\,(\!|V|\!)\gamma\,\{\mathbf{inl}\, x \mapsto (\!|M|\!)(\gamma\backslash\{x\});\\ &\qquad\quad \mathbf{inr}\, y \mapsto (\!|N|\!)(\gamma\backslash\{y\})\}\\ (\!|\mathbf{return}\, V|\!)\gamma &= \mathbf{return}\,(\!|V|\!)\gamma\\ (\!|\mathbf{let}\, x \leftarrow M \,\mathbf{in}\, N|\!)\gamma &= \mathbf{let}\, x \leftarrow (\!|M|\!)\gamma \,\mathbf{in}\, (\!|N|\!)(\gamma\backslash\{x\})\end{aligned}$$

**Value terms and values**

$$\begin{aligned}(\!|x|\!)\gamma &= (\!|v|\!), \quad \text{if } \gamma(x) = v\\ (\!|x|\!)\gamma &= x, \quad \text{if } x \notin dom(\gamma)\\ (\!|n|\!)\gamma &= n\\ (\!|\lambda x^A.M|\!)\gamma &= \lambda x^A.(\!|M|\!)(\gamma\backslash\{x\})\\ (\!|\mathbf{rec}\, f\, x^A.M|\!)\gamma &= \mathbf{rec}\, f\, x^A.(\!|M|\!)(\gamma\backslash\{f, x\})\\ (\!|\langle\rangle|\!)\gamma &= \langle\rangle\\ (\!|\langle V, W\rangle|\!)\gamma &= \langle(\!|V|\!)\gamma, (\!|W|\!)\gamma\rangle\\ (\!|\mathbf{inl}^B\, V|\!)\gamma &= (\mathbf{inl}\,(\!|V|\!)\gamma)^B\\ (\!|\mathbf{inr}^A\, W|\!)\gamma &= (\mathbf{inr}\,(\!|W|\!)\gamma)^A\end{aligned}$$

$$\begin{aligned}(\!|n|\!) &= n\\ (\!|(\gamma, \lambda x^A.M)|\!) &= \lambda x^A.(\!|M|\!)(\gamma\backslash\{x\})\\ (\!|(\gamma, \mathbf{rec}\, f\, x^A.M)|\!) &= \mathbf{rec}\, f\, x^A.(\!|M|\!)(\gamma\backslash\{f, x\})\\ (\!|\langle\rangle|\!) &= \langle\rangle\\ (\!|\langle v; w\rangle|\!) &= \langle(\!|v|\!); (\!|w|\!)\rangle\\ (\!|\mathbf{inl}^B\, v|\!) &= \mathbf{inl}^B\,(\!|v|\!)\\ (\!|\mathbf{inr}^A\, w|\!) &= \mathbf{inr}^A\,(\!|w|\!)\\ (\!|\sigma^A|\!) &= \lambda x^A.(\!|\sigma|\!)(\mathbf{return}\, x)\end{aligned}$$

Fig. A1. 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 $\beta$-step in the abstract machine.

**Lemma 5.** *Suppose $M$ is a computation and $\mathcal{C}$ is configuration such that $(\!|\mathcal{C}|\!) = M$, then if $M \rightsquigarrow N$ there exists $\mathcal{C}'$ such that $\mathcal{C} \Longrightarrow \mathcal{C}'$ and $(\!|\mathcal{C}'|\!) = N$, 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 $\Longrightarrow / \simeq_{\mathrm{a}}$ (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 5 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 6** (Base simulation). *If $\vdash M : A$ and $M \rightsquigarrow^+ N$ where $N$ is normal, then $\langle M \mid \emptyset \mid [\,]\rangle \longrightarrow^+ \mathcal{C}$ such that $(\!|\mathcal{C}|\!) = N$, or if $M \not\rightsquigarrow$ then $\langle M \mid \emptyset \mid [\,]\rangle \not\longrightarrow$.*

**Proof** By repeated application of Lemma 5. ■

## 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 B1 shows the necessary changes to the $(\!|-|\!)$ function.

**Configurations**

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

**Continuations**

$$\llparenthesis [] \rrparenthesis M = M$$
$$\llparenthesis (\sigma, \chi) :: \kappa \rrparenthesis M = \llparenthesis \kappa \rrparenthesis (\llparenthesis \chi \rrparenthesis (\llparenthesis \sigma \rrparenthesis (M)))$$

**Handler Closures and Definitions**

$$\llparenthesis (\gamma, H) \rrparenthesis M = \mathbf{handle}\ M\ \mathbf{with}\ \llparenthesis H \rrparenthesis \gamma \qquad \llparenthesis \{\mathbf{val}\ x \mapsto M\} \rrparenthesis \gamma = \{\mathbf{val}\ x \mapsto \llparenthesis M \rrparenthesis (\gamma \backslash \{x\})\}$$
$$\llparenthesis \{\ell\ p\ r \mapsto M\} \uplus H \rrparenthesis \gamma = \{\ell\ p\ r \mapsto \llparenthesis M \rrparenthesis (\gamma \backslash \{p, r\})\} \uplus \llparenthesis H \rrparenthesis \gamma$$

**Computation Terms and Machine Values**

$$\llparenthesis \mathbf{handle}\ M\ \mathbf{with}\ H \rrparenthesis \gamma = \mathbf{handle}\ \llparenthesis M \rrparenthesis \gamma\ \mathbf{with}\ \llparenthesis H \rrparenthesis \gamma \qquad \llparenthesis (\gamma, H)^D \rrparenthesis = \lambda x^D . \llparenthesis (\gamma, H) \rrparenthesis (\mathbf{return}\ x)$$
$$\llparenthesis \mathbf{do}\ \ell\ V \rrparenthesis \gamma = \mathbf{do}\ \ell\ \llparenthesis V \rrparenthesis \gamma$$

Fig. B1. Mapping from handler machine configurations to terms.

**Lemma 6.** *Suppose $M$ is a computation and $\mathcal{C}$ is a configuration such that $\llparenthesis \mathcal{C} \rrparenthesis = M$, then if $M \leadsto N$ there exists $\mathcal{C}'$ such that $\mathcal{C} \Longrightarrow \mathcal{C}'$ and $\llparenthesis \mathcal{C}' \rrparenthesis = N$, or if $M \not\leadsto$ then $\mathcal{C} \not\Longrightarrow$.*

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

**Theorem 7** (Handler simulation)**.** *If $\vdash M : A$ and $M \leadsto^+ N$ such that $N$ is normal, then $\langle M \mid \emptyset \mid \kappa_0 \rangle \longrightarrow^+ \mathcal{C}$ such that $\llparenthesis \mathcal{C} \rrparenthesis = N$, or $M \not\leadsto$ then $\langle M \mid \emptyset \mid \kappa_0 \rangle \not\longrightarrow$.*

**Proof** By repeated application of Lemma 6. ■