Effects for Efficiency

Asymptotic Speedup with First-Class Control

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

As Filinski showed in the 1990s, delimited control operators can express all monadic effects. Plotkin and Pretnar's effect handlers offer a modular form of delimited control providing a uniform mechanism for concisely implementing features ranging from async/await to probabilistic programming.

We study the fundamental efficiency of delimited control. Specifically, we show that effect handlers enable an asymptotic improvement in runtime complexity for a certain class of programs. We consider the *generic search* problem and define a pure PCF-like base language λ_{b} and its extension with effect handlers λ_{h}. We show that λ_{h} admits an asymptotically more efficient implementation of generic search than any λ_{b} implementation of generic search. We also show that this efficiency gap remains when λ_{b} is extended with mutable state.

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

1 INTRODUCTION

In today's programming languages we find a wealth of powerful constructs and features — exceptions, higher-order store, dynamic method dispatch, coroutining, 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 type A 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. '$O(n^2)$') 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

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

applicable whatever programming language we are working with, and moreover that all respectable languages can represent all algorithms of interest; thus, one does not expect the best achievable asymptotic run-time for a typical algorithm (say in number theory or graph theory) to be sensitive to the choice of programming language, except perhaps in marginal cases. (It should be admitted, however, that proving general theorems to this effect may be harder than one might suppose: see for example Section 1 of [Pippenger 1996].)

The situation changes radically, however, if we consider *higher-order* operations: programmable operations whose inputs may themselves be programmable operations. (At this point, we suppose that the languages we wish to compare all support higher-order data in some way: in particular, that their type systems are rich enough to admit encodings of all simple types generated from the familiar ground types via '$\rightarrow$'.) Here it turns out that both what is computable and the efficiency with which it can be computed can be highly sensitive to the selection of language features present. This is in fact true more widely for *abstract data types*, of which higher-order types can be seen as a special case: a higher-order value will of course be represented within the machine as ground data, but a program within the language typically has no access to this internal representation, and can interact with the value only by applying it to an argument.

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

Relatively few results of this character have so far been established on the complexity side. Pippenger [1996] gives an example of an 'online' operation on infinite sequences of atomic symbols (essentially a function from streams to streams) such that the first n output symbols can be produced within time $O(n)$ if one is working in an 'impure' version of Lisp (in which mutation of 'cons' pairs is admitted), but with a worst-case runtime no better than $\Omega(n \log n)$ for any implementation in pure Lisp (without such mutation). This example was reconsidered by Bird et al. [1997] who showed that the same speedup can be achieved in a pure language by using lazy evaluation. Jones [2001] explores the approach of manifesting expressivity and efficiency differences between certain languages (which differ according to both the forms of iteration or recursion they admit and also the use of higher types that they allow) by artificially restricting attention to 'cons-free' programs; in this setting, the classes of representable first-order functions for the various languages are found to coincide with some well-known complexity classes.

The purpose of the present paper is to give a clear example of such an inherent complexity difference higher up in the expressivity spectrum. Specifically, we consider the following *generic search* 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 p for which $P(p) = \mathrm{true}$. We shall consider boolean vectors of any length to be represented by the type $\mathsf{Nat} \rightarrow \mathsf{Bool}$; thus, for each n, we are asking for an implementation of a certain third-order operation

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

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

The 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 naive implementation of count_3 might start by applying the given P to $p_0 = (\mathsf{true}, \mathsf{true}, \mathsf{true})$, and then to $p_1 = (\mathsf{true}, \mathsf{true}, \mathsf{false})$. Clearly there is some duplication here: the computations of $P\,p_0$ and $P\,p_1$ 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 $P\,p_0$ 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 $P\,p_1$. (Similarly for all other internal nodes in the evident binary tree of boolean vectors.) Of course, this 'backup' approach would be standardly applied if one were implementing a bespoke search operation for some *particular* choice of P (corresponding, say, to the n-queens problem); but to apply this idea of resuming previous subcomputations in the generic setting (that is, uniformly in P) requires some special language feature such as effect handlers or multi-shot continuations. One could also obviate the need for such a feature by choosing to present the predicate P in some other way, but from our present perspective this would be to move the goalposts: our intention is precisely to show that our languages differ in an essential way *as regards their power to manipulate data of type* $(\mathsf{Nat} \to \mathsf{Bool}) \to \mathsf{Bool}$.

This idea of using first-class control to achieve 'backtracking' has been exploited before and is fairly widely known (see e.g. [Kiselyov et al. 2005]), and there is a clear programming intuition that this yields a speedup unattainable in languages without such control features. Our main contribution in this paper is to provide, for the first time, a precise mathematical theorem that pins down this fundamental efficiency difference, thus giving formal substance to the above-mentioned intuition. Since our goal is to give a realistic analysis of the efficiency achievable in various settings without getting bogged down in inessential implementation details, we shall work concretely and operationally with the languages in question, using a CEK-style abstract machine semantics as our basic model of execution time, and with some specific programs in these languages. In the first instance, we formulate our results as a comparison between a purely functional base language (a version of call-by-value PCF) and an extension with first-class control; we then indicate how these results can be extended to base languages with other features such as mutable state.

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

The rest of the paper is structured as follows.

- Section 2 provides an introduction to effect handlers as a programming abstraction.
- Section 3 presents a PCF-like language λ_b and its extension λ_h with effect handlers.
- Section 4 defines abstract machines for λ_b and λ_h, yielding a runtime cost model.
- Section 5 proves formal complexity results for generic search in λ_b ($\Omega(n2^n)$) and λ_h ($O(2^n)$).
- Section 6 shows that our results scale to richer settings including support for a wider class of predicates, an extension of the base language with state, and a non-trivial algorithm for generic search that exploits memoisation to perform pruned search.
- Section 7 evaluates implementations of generic search based on λ_b and λ_h in Standard ML.
- Section 8 concludes.

The languages λ_b and λ_h are rather minimal versions of previously studied systems — we only include the machinery needed for illustrating the generic search efficiency phenomenon. Full proofs of our main complexity results are available in the appendices of the anonymised supplementary material.

2 EFFECT HANDLERS PRIMER

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

Viewed through the lens of universal algebra, an algebraic effect is given by a signature Σ of finitary *operation symbols* defined over some nonempty carrier set A, along with an equational theory that describes the properties of the operations [Plotkin and Power 2001]. An example of an algebraic effect is *nondeterminism*, whose signature consists of a single nondeterministic choice operation: $\Sigma := \{\mathsf{Branch} : 1 \to \mathsf{Bool}\}$. The operation takes a single parameter of type unit and ultimately produces a boolean value. The pragmatic programmatic view of algebraic effects differs from the original development as no implementation accounts for equations over operations yet.

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

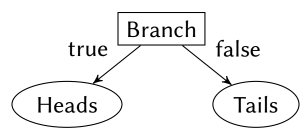
$$\begin{array}{l}\mathsf{toss} : 1 \to \mathsf{Toss}\\ \mathsf{toss}\,\langle\rangle = \textbf{if do}\ \mathsf{Branch}\,\langle\rangle\ \textbf{then}\ \mathsf{Heads}\ \textbf{else}\ \mathsf{Tails}\end{array}$$

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

We may, in the style of Lindley [2014], view an effectful computation as a tree, where the interior nodes correspond to operation invocations and the leaves correspond to return values. The computation tree for toss is as follows.

$$
\begin{array}{lrl}
\text{Types} & A, B, C, D ::= & \mathsf{Nat} \mid 1 \mid A \to B \mid A \times B \mid A + B \\
\text{Type Environments} & \Gamma ::= & \cdot \mid \Gamma, x : A \\
\text{Values} & V, W \in \mathsf{Val} ::= & x \mid n \mid c \mid \lambda x^A.\, M \mid \mathbf{rec}\, f^A\, x.M \\
& \mid & \langle\rangle \mid \langle V, W \rangle \mid (\mathbf{inl}\; V)^B \mid (\mathbf{inr}\; W)^A \\
\text{Computations} & M, N \in \mathsf{Comp} ::= & V\, W \mid \mathbf{let}\, \langle x, y \rangle = V\; \mathbf{in}\; N \\
& \mid & \mathbf{case}\; V\, \{\mathbf{inl}\; x \mapsto M; \mathbf{inr}\; y \mapsto N\} \\
& \mid & \mathbf{return}\; V \mid \mathbf{let}\; x \leftarrow M\; \mathbf{in}\; N
\end{array}
$$

Fig. 1. Syntax of λ_b



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

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

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

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

3 CALCULI

In this section, we present our base language λ_b and its extension with effect handlers λ_h.

3.1 Base Calculus

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

The syntax of λ_b is given in Figure 1. The ground types are Nat and 1 which classify natural number values and the unit value, respectively. We write ground A to assert that type A is a ground type. The function type $A \to B$ represents functions that map values of type A to values of type B. The binary product type $A \times B$ represents a pair of values whose first and second components have types A and B respectively. The sum type $A \times B$ represents tagged values of either type A or B. Type environments Γ map term variables to their types.

Values

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ \textsc{T-Var} \qquad \frac{}{\Gamma \vdash \langle\rangle : 1}\ \textsc{T-Unit} \qquad \frac{n \in \mathbb{N}}{\Gamma \vdash n : \mathsf{Nat}}\ \textsc{T-Nat} \qquad \frac{c : A \to B}{\Gamma \vdash c : A \to B}\ \textsc{T-Const}$$

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

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

Computations

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

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

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

Fig. 2. Typing Rules for λ_b

We let n range over natural numbers and c range over primitive operations on natural numbers $(+, -, =)$. We generally use lowercase letters x, y, z and more to denote term variables. By convention we use f, g, and h for variables of function type, i and j for variables of type Nat, and r and k to denote resumptions and continuations, with the exception that we will use uppercase P to denote predicates. Value terms comprise variables (x), the unit value ($\langle\rangle$), natural number literals (n), primitive constants (c), lambda abstraction ($\lambda x^A.\, M$), recursion ($\mathbf{rec}\ f^A\ x.M$), pairs ($\langle V, W\rangle$), left ($(\mathbf{inl}\ V)^B$) and right ($(\mathbf{inr}\ W)^A$) injections. We will occasionally blur the distinction between object and meta language by writing A for the meta level type of closed value terms of type A. All elimination forms are computation terms. Abstraction is eliminated using application ($V\ W$). The product eliminator ($\mathbf{let}\ \langle x, y\rangle = V\ \mathbf{in}\ N$) splits a pair V into its constituents and binds them to x and y, respectively. Sums are eliminated by a case split ($\mathbf{case}\ V\ \{\mathbf{inl}\ x \mapsto M; \mathbf{inr}\ y \mapsto N\}$). A trivial computation ($\mathbf{return}\ V$) returns value V. The sequencing expression ($\mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N$) evaluates M and binds the result value to x in N.

The typing rules are given in Figure 2. We require two typing judgements: one for values and the other for computations. The judgement $\Gamma \vdash \square : A$ states that a $\square$-term has type A under type environment Γ, where $\square$ is either a value term (V) or a computation term (M). The constants have the following types.

$$\{(+), (-)\} : \langle \mathsf{Nat}, \mathsf{Nat}\rangle \to \mathsf{Nat} \qquad\qquad (=) : \langle \mathsf{Nat}, \mathsf{Nat}\rangle \to \mathsf{Bool}$$

We give a small-step operational semantics for λ_b with *evaluation contexts* in the style of Felleisen [1987]. The reduction rules are given in Figure 3. We write $M[V/x]$ for M with V substituted for x

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

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

Fig. 3. Contextual Small-Step Operational Semantics

and $\ulcorner c \urcorner$ for the usual interpretation of constant c as a meta-level function on closed values. The reduction relation $\leadsto$ is defined on computation terms. The statement $M \leadsto N$ 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. We write R/S for the quotient of relation R by relation S.

Syntactic sugar. For convenience we often write code in direct-style assuming the standard left-to-right call-by-value elaboration into fine-grain call-by-value [Flanagan et al. 1993]. For example, the expression $f\ (h\ w) + g\ \langle\rangle$ is syntactic sugar for:

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

We use the standard encoding of booleans as sums:

$$\mathsf{Bool} := 1 + 1 \qquad \mathsf{true} := \mathbf{inl}\ \langle\rangle \qquad \mathsf{false} := \mathbf{inr}\ \langle\rangle$$

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

We also define sequencing of computations in the standard way.

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

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

$$\lambda\langle\rangle.M := \lambda x^1.M, \quad \text{where } x \notin FV(M)$$

and more generally if the binder has a type other than 1, then we write

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

We elide type annotations when clear from context.

3.2 Handler Calculus

We now define λ_{h} as an extension of λ_{b}. First we define notation for operation symbols, signatures, and handler types.

$$\begin{array}{ll}
\text{Operation symbols} & \ell \in \mathcal{L} \\
\text{Signatures} & \Sigma ::= \cdot \mid \{\ell : A \to B\} \cup \Sigma \\
\text{Handler types} & F ::= C \Rightarrow D
\end{array}$$

We assume a countably infinite set of operation symbols $\mathcal{L}$. An effect signature Σ is a map from operation symbols to their types, thus we assume that each operation symbol in a signature is distinct. An operation type $A \to B$ denotes an operation that takes an argument of type A and returns a result of type B. We write $dom(\Sigma) \subseteq \mathcal{L}$ for the set of operation symbols in a signature Σ. An effect 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

T-Do
$$\frac{(\ell : A \to B) \in \Sigma \qquad \Gamma \vdash V : A}{\Gamma \vdash \mathbf{do}\ \ell\ V : B}$$

T-Handle
$$\frac{\Gamma \vdash M : C \qquad \Gamma \vdash H : C \Rightarrow D}{\Gamma \vdash \mathbf{handle}\ M\ \mathbf{with}\ H : D}$$

Handlers

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

Fig. 4. Additional Typing Rules for λ_{h}

The typing rules for λ_{h} are those of λ_{b} (Figure 2) plus three additional rules for operations, handling, and handlers given in Figure 4. The T-Do rule ensures that an operation invocation is only well-typed if the operation ℓ appears in the effect signature Σ and the argument type A matches the type of the provided argument V. The result type B determines the type of the invocation. The T-Handle rule is straightforward. 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 value clause must match the input type C. The type of the parameter p (A_ℓ) and resumption r ($B_\ell \to D$) in operation clause H^{ℓ} is determined by the signature for ℓ; the return type of r is D, as the body of the resumption will itself be handled by H. We write H^{ℓ} and H^{val} for projecting success and operation clauses.

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

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

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

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

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

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

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

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

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

Definition 3.1 (Computation normal forms). We say that a computation term N is normal with respect to $\ell \in \Sigma$, if N is either of the form $\mathbf{return}\ V$, or $\mathcal{E}[\mathbf{do}\ \ell\ W]$.

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

4 ABSTRACT MACHINE SEMANTICS

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

4.1 Base Machine

We choose a *CEK*-style abstract machine semantics [Felleisen and Friedman 1987] for $\lambda_b$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 γ which binds free variables. The third component contains the continuation which instructs the machine how to proceed once evaluation of the current computation is complete. The syntax of abstract machine states is as follows.

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

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

The abstract machine semantics is given in Figure 5. The transition relation ($\longrightarrow$) makes use of the value interpretation ($[\![-]\!]$) on value terms and machine values. The machine is initialised by placing a term in a configuration alongside the empty environment ($\emptyset$) and identity pure continuation ($[]$). The rules (M-App), (M-Rec), (M-Const), (M-Split), (M-CaseL), and (M-CaseR) eliminate values. The (M-Let) rule extends the current pure continuation with let bindings. The (M-RetCont) rule extends the environment in the top frame of the pure continuation with a returned value. Given an input of a well-typed closed computation term $\vdash M : A$, the machine will either diverge or return a value of type A. A final state is given by a configuration of the form $\langle \mathbf{return}\; V \mid \gamma \mid [] \rangle$ in which case the final return value is given by the denotation $[\![V]\!]\gamma$ of V under environment γ.

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

4.2 Handler Machine

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

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

The notion of configurations changes slightly in that the continuation component is replaced by a generalised continuation $\kappa \in \mathsf{Cont}$ [Hillerström et al. 2020]; a continuation is now a list of pairs containing a pure continuation (as in the base machine) and a handler closure (χ). A handler closure

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

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

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

Transition relation

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

Fig. 6. Abstract Machine Semantics for λ_{h}

consists of an environment and a handler definition, where the former binds the free variables that occur in the latter. The identity continuation is an empty pure continuation paired with the identity handler closure:

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

Machine values are augmented to include handler closures, as an operation invocation causes the topmost handler closure 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 6 depicts the new and modified rules. The (M-Handle) rule pushes a handler closure along with an empty pure continuation onto the continuation stack. The (M-RetHandler) rule transfers control to the success clause of the current handler once the pure continuation is empty. The (M-Handle-Op) rule transfers control to the matching operation clause on the topmost handler, and during the process it reifies the handler closure. Finally, the (M-Resume) rule applies a reified handler closure, by pushing it onto the continuation stack. The handler machine has two possible final states: either it yields a value or it gets stuck on an unhandled operation.

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

4.3 Realisability and Asymptotic Complexity

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

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

Continuation copying. On the handler machine the topmost continuation frame can be copied in constant time due to the persistent runtime and the layout of machine continuations. An alternative design would be to make the runtime non-persistent, as in MLton [2020], in which case copying a continuation frame $((\sigma, (\gamma, \_)) :: \_)$ would be a $\mathcal{O}(|\sigma| + |\gamma|)$ time operation.

Primitive operations on naturals. Our model assumes that arithmetic operations on arbitrary natural numbers take $\mathcal{O}(1)$ time. This is common practice in the study of algorithms when the main interest lies elsewhere (see [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 have essentially 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 EFFICIENT GENERIC SEARCH

We now come to the crux of the paper. In this section we prove that λ_{h} accommodates some programmable operations with an asymptotic runtime bound that cannot be achieved in λ_{b}. Whilst the positive half of this claim essentially consolidates a known piece of folklore, the negative half appears to be a genuinely new result. To obtain our results, it suffices to find just one efficient program in λ_{h} and show that *no* equivalent program in λ_{b} can achieve the same asymptotic complexity. We take *generic search* as our example.

Generic search is a modular search procedure that finds solutions to a given search problem 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] and the n-queens problem [Bell and Stevens 2009] can be cast as instances of generic search. Other instantiations include problems from game theory such as computing Nash equilibria, problems from graph theory such as graph colouring, and problems from real analysis such as real number integration [Daniels 2016; Simpson 1998].

To simplify the presentation, we compute the number of solutions (generic count), rather than materialising all solutions (generic search). With little extra effort one can tweak the development to compute exact solutions. Informally, a generic count program takes as input a predicate and returns the number of times the predicate yields true. A predicate returns a boolean value which signifies whether its input satisfies the predicate. As input a predicate takes a bit vector of length $n > 0$, which we represent as a first-order function $\mathsf{Nat} \to \mathsf{Bool}$. Ultimately we ask for implementations of a program, count, whose type is

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

where Nat_n admits elements of the set $\mathbb{N}_n := \{0, \dots, n-1\}$. We often omit the n index when clear from context; in particular it does not appear explicitly in the types of our programs as our formalism does not support dependent types.

Before giving the necessary formal machinery to state and prove the result, we first introduce the concepts informally.

5.1 Predicates and Points

Higher-order functions are the key to our modular formulation of generic search. We define a predicate of size n as a closed value of the following type

$$\mathsf{Predicate}_n := \mathsf{Point}_n \to \mathsf{Bool}$$

where n is a natural number, and a point is also a closed value of the following type

$$\mathsf{Point}_n := \mathsf{Nat}_n \to \mathsf{Bool}$$

Intuitively, a point implements a vector of boolean values where the natural number argument serves as an index into the vector. A point need not be a total function; indeed points we concern ourselves with are typically partial.

Examples. Let us consider some simple examples of predicates and points. As a first example consider the constant point, $\mathsf{p}_{\mathsf{true}} := \lambda\_.\mathsf{true}$. A slightly more interesting point is

$$\mathsf{p}_2 := \lambda i.\mathbf{if}\ i = 0\ \mathbf{then}\ \mathsf{true}\ \mathbf{else}\ \mathbf{if}\ i = 1\ \mathbf{then}\ \mathsf{false}\ \mathbf{else}\ \bot\ i$$

where $\bot := \mathbf{rec}\ f\ i.f\ i$ is the always-diverging point.

Now let us move onto some example predicates. We can give a whole family of constant true predicates. For example tt_0 returns true irrespective of its point.

$$\mathsf{tt}_0 := \lambda p.\mathsf{true}$$

We can define a variation, tt_2, which inspects two components of its point, but still returns true.

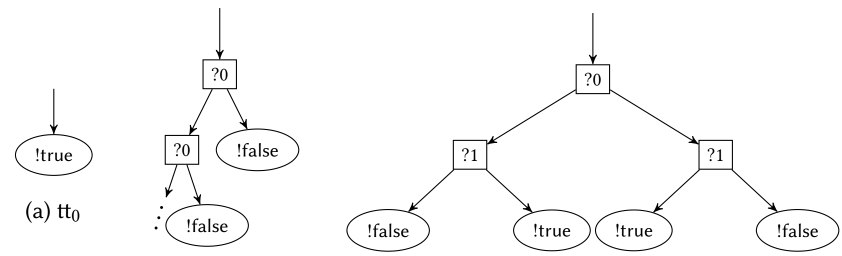
$$\mathsf{tt}_2 := \lambda p.p\,1; p\,0; \mathsf{true}$$

This predicate is slightly more interesting than tt_0 as it is defined only for points defined on Nat_n for $n \geq 2$. A predicate may inspect the same component of its point more than once

$$\mathsf{red}_1 := \lambda p.p\,0; p\,0$$

thus performing redundant work. Another class of predicates are divergent predicates such as

$$\mathsf{div}_1 := \mathbf{rec}\ div\ p.\mathbf{if}\ p\,0\ \mathbf{then}\ div\,p\ \mathbf{else}\ \mathsf{false}$$



(a) tt_0 (b) div_1 (c) odd_2

Fig. 7. Example Decision Tree Models

which diverges whenever the 0-th index of the point yields true. Thus both $\mathrm{div}_1\ \mathrm{p}_{\mathrm{true}}$ and $\mathrm{div}_1\ \mathrm{p}_2$ never terminate. Finally, let us consider a predicate which determines whether a point contains an odd number of true components

$$\mathrm{odd}_n := \lambda p.\mathrm{fold}\ \otimes\ \mathrm{false}\,(\mathrm{map}\ p\ [0, \ldots, n-1])$$

where fold and map are the standard combinators on lists and $\otimes$ is exclusive-or. This predicate is only well-defined for $n > 0$. Applying odd_2 to p_2 yields true; applying it to $\mathrm{p}_{\mathrm{true}}$ yields false.

Predicate Models. In essence a predicate is a decision procedure, which participates in a 'dialogue' with a supplied point $p : \mathrm{Point}_n$. The predicate may *query* (i.e. invoke) the components of p, and p then *responds* (i.e. returns). Ultimately this dialogue may *answer* whether the point satisfies the predicate. We can model the behaviour of a predicate as an unrooted binary decision tree characterising the predicate's interaction with p, where each interior node is labelled with a query $?i$ (for $i \in \mathbb{N}_n$) whose left subtree corresponds to $p\ i$ being true and whose right subtree corresponds to $p\ i$ being false, and each leaf is labelled with an answer !true or !false according to whether p satisfies the predicate. The trees are unrooted to account for the computation that occurs in between the application of a predicate to p and the first query or answer.

Figure 7 depicts models of some of the example predicates given above. The model of tt_0 is simply an unrooted leaf (Figure 7a). The model of div_1 is an infinite left-branching tree (Figure 7b). The model of odd_2 is a complete binary tree (Figure 7c). A further example is the model of the unconditionally divergent predicate $\mathrm{div} := \textbf{rec}\ \mathit{div}\ p.\mathit{div}\ p$, which is empty.

Restrictions. In order to obtain a meaningful complexity result we must constrain the predicates of interest. At one extreme, counting the size of a divergent predicate like div is meaningless. At the other extreme, a constant predicate like tt_0 exhibits no interesting computational characteristics; other constant predicates like tt_2 inspect their provided point. Predicates like red_1 perform redundant work. Such redundancy can be eliminated via insertion of a local let binding.

Thus we restrict attention to predicates that for $n > 0$:

(1) terminate when applied to any point p; and
(2) inspect each bit $0 < i < n$ of p exactly once.

Of the examples so far, the ones satisfying these conditions are tt_2 and odd_n. Predicates satisfying 1 and 2 are exactly those whose models form complete binary trees (as in Figure 7c), which we call *n-standard*. We provide a rigorous definition of n-standard predicates in Section 5.3. To satisfy 1, we also require that points terminate on their defined domain Nat_n. We call a point that is defined on $0 < i < n$ an n-point.

5.2 Effectful Generic Counting

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

We fix the effect signature $\Sigma := \{\mathsf{Branch} : 1 \rightarrow \mathsf{Bool}\}$. The algorithm is then as follows.

$$\begin{array}{l}
\mathsf{effcount} : ((\mathsf{Nat} \rightarrow \mathsf{Bool}) \rightarrow \mathsf{Bool}) \rightarrow \mathsf{Nat} \\
\mathsf{effcount}\, P := \mathbf{handle}\; P\,(\lambda_.\mathbf{do}\;\mathsf{Branch}\;\langle\rangle)\;\mathbf{with} \\
\qquad\qquad \begin{array}{lll}
\mathbf{val}\; b & \mapsto & \mathbf{if}\; b\;\mathbf{then}\;\mathbf{return}\;1\;\mathbf{else}\;\mathbf{return}\;0 \\
\mathsf{Branch}\;\langle\rangle\; r & \mapsto & \mathbf{let}\; x_{\mathsf{true}} \leftarrow r\,\mathsf{true}\;\mathbf{in} \\
 & & \mathbf{let}\; x_{\mathsf{false}} \leftarrow r\,\mathsf{false}\;\mathbf{in}\; x_{\mathsf{true}} + x_{\mathsf{false}}
\end{array}
\end{array}$$

The handler applies predicate P to a single 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. A curious detail about effcount is that it works for all n-standard predicates without having to know the exact value of n. This is because the point $(\lambda\_.\mathbf{do}\;\mathsf{Branch}\;\langle\rangle)$ represents the superposition of all possible points. The sharing enabled by the use of the resumption is exactly the 'magic' we need to make it possible to implement generic counting more efficiently in λ_{h} than in λ_{b}.

5.3 Predicates, Points, and their Models, Formally

We now formalise the notions of n-standard predicates, points, and their models. For simplicity, we formalise these concepts using the operational semantics and abstract machine for the base language λ_{b}; this means that the above concepts will be defined only for predicates expressible in λ_{b}. There is in principle no strong need for this restriction — with a little extra effort, corresponding concepts can be defined for λ_{h} predicates, and our efficiency result for effcount will be applicable to these too — but we choose to avoid this inessential complication.

We begin by formalising the decision tree model of predicates. We first introduce the label set, Lab, consisting of queries and answers.

Notation. We write $bs \sqsubset bs'$ to mean that list bs is a prefix of list bs'.

Definition 5.1 (label set). The label set Lab consists of queries parameterised by a natural number and answers parameterised by a boolean.

$$\mathsf{Lab} := \{?n \mid n \in \mathbb{N}\} \cup \{!\mathsf{true}, !\mathsf{false}\}$$

We model a decision tree as a partial function from lists of booleans to labels; each boolean list specifies a cursor into the tree as a path from the root of the tree.

Definition 5.2 ((untimed) decision tree). A decision tree is a partial function $t : \mathbb{B}^* \rightharpoonup \mathsf{Lab}$ from lists of booleans to node labels with the following properties:

- The domain of t, $dom(t)$, is prefix closed.
- If $t(bs) = !b$ then $t(bs')$ is undefined for all $bs' \sqsupset bs$. In other words answer nodes are always leaves.

Timed decision trees are decorated with timing data that records the number of machine steps.

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

We now relate predicates to decision trees by way of an interpretation of configurations as decision trees.

Notation. We write $a \simeq b$ for Kleene equality: either both a and b are undefined or both are defined and $a = b$.

Definition 5.4. The timed decision tree of a configuration is defined by the following equations

$$\begin{array}{ll}
\mathcal{T}(\langle \mathbf{return}\ \mathsf{true} \mid \gamma \mid [] \rangle)\,[] = (!\mathsf{true}, 0) & \mathcal{T}(\langle p\ V \mid \gamma \mid \sigma \rangle)\,(b :: bs) \simeq \mathcal{T}(\langle \mathbf{return}\ b \mid \gamma \mid \sigma \rangle)\,bs \\
\mathcal{T}(\langle \mathbf{return}\ \mathsf{false} \mid \gamma \mid [] \rangle)\,[] = (!\mathsf{false}, 0) & \mathcal{T}(\langle M \mid \gamma \mid \sigma \rangle)\,bs \simeq \mathcal{I}(\mathcal{T}(\langle M' \mid \gamma' \mid \sigma' \rangle)\,bs), \\
\mathcal{T}(\langle p\ V \mid \gamma \mid \sigma \rangle)\,[] = (?[\![V]\!]\gamma, 0) & \quad \text{if } \langle M \mid \gamma \mid \sigma \rangle \longrightarrow \langle M' \mid \gamma' \mid \sigma' \rangle
\end{array}$$

where $\mathcal{I}(\ell, s) = (\ell, s + 1)$ and p is a distinguished free variable such that in all of the above equations $\gamma(p) = \gamma'(p) = p$. The decision tree of a computation term is obtained by placing it in the initial configuration: $\mathcal{T}(M) := \mathcal{T}(\langle M, \emptyset[p \mapsto p], \kappa_0 \rangle)$. The decision tree of a predicate P is $\mathcal{T}(P\,p)$. Since p is a distinguished variable, we often omit it and write $\mathcal{T}(P)$ for $\mathcal{T}(P\,p)$.

We can define a construction procedure, $\mathcal{U}$, for untimed decision trees using $\mathcal{T}$ as follows: $\mathcal{U}(P) := bs \mapsto \mathcal{T}(P)(bs).1$.

Definition 5.5 (n-standard trees and n-standard predicates). For any $n > 0$ a decision tree t is said to be n-standard if:

- the domain of t consists of all the lists whose length is at most n, i.e., $dom(t) = \{bs : \mathbb{B}^* \mid |bs| \le n\}$;
- every leaf node in t is an answer node, i.e., for all $bs \in dom(t)$ if $|bs| = n$ then $t(bs) = !b$, for some $b \in \mathbb{B}$; and
- there are no repeated queries along any path in t: for all $bs, bs' \in dom(t), j \in \mathbb{N}$, if $bs \sqsubseteq bs'$ and $t(bs) = t(bs') = ?j$ then $bs = bs'$.

A timed decision tree t is n-standard if its underlying untimed decision tree $(bs \mapsto t(bs).1)$ is. A predicate P is said to be n-standard if its decision tree $\mathcal{T}(P)$ is an n-standard tree.

As alluded to in Section 5.1 n-standard decision tree models are exactly those that form a complete binary tree such that each path contains no repeated queries. The third condition in the definition requires only that there are no repeated queries along any path in the model; it does not impose a particular ordering on those queries.

We now move onto formalising points. Our model of points is only used for extensional reasoning about programs in the λ_{b}-language as we can reason intensionally about the single point used by effcount in the λ_{h}-language. As remarked in Section 5.1, points may in general be partial, however, the points that we shall consider all have the property that they terminate whenever applied to an element of their defined domain (Nat_n for some $n > 0$).

Definition 5.6 (n-points). For any $n > 0$ a closed value $p : \mathsf{Point}_n$ is said to be an n-point if

$$\forall i \in \mathbb{N}_n. p\ i \rightsquigarrow^* \mathbf{return}\ \mathsf{true} \lor p\ i \rightsquigarrow^* \mathbf{return}\ \mathsf{false}.$$

A semantic n-point π is the denotation of an n-point p, i.e. a mathematical function $\mathbb{N}_n \to \mathsf{Bool}$. For any n-point p its corresponding semantic n-point is given by $\pi = \mathbb{P}[\![p]\!]$, where $\mathbb{P}[\![-]\!]$ is the realisation of the operational behaviour of p

$$\begin{array}{l}
\mathbb{P}[\![-]\!] : \mathsf{Point}_n \to (\mathbb{N}_n \to \mathbb{B}) \\
\mathbb{P}[\![p]\!] := i \in \mathbb{N}_n \mapsto p\ i
\end{array}$$

Moreover, any two n-points p_0 and p_1 are said to be *distinct* if their corresponding semantic n-points differ, i.e.:

$$\exists i \in \mathbb{N}_n . \mathbb{P}[\![P_0]\!]\ i \neq \mathbb{P}[\![P_1]\!]\ i$$

5.4 Specification of Generic Counting

We now formally define generic counting.

Definition 5.7. A counting function is a partial function of type $\mathbb{B}^* \rightharpoonup \mathbb{N}$.

As with the decision tree functions, the list argument to a counting function serves as a cursor into the model of the predicate. However, in this case, the function computes the sum of the true answers in the subtree pointed to by the cursor. Thus in order to compute the sum of all true answers we apply the counting function to the empty list. The following definition provides a procedure for constructing a counting function for any predicate.

Definition 5.8. The counting function for a configuration is defined by the following equations.

$$\begin{aligned}
C(\langle \mathbf{return}\ \mathsf{true} \mid \gamma \mid [] \rangle)\,[] &= 1 \\
C(\langle \mathbf{return}\ \mathsf{false} \mid \gamma \mid [] \rangle)\,[] &= 0 \\
C(\langle p\, V \mid \gamma \mid \sigma \rangle)\,[] &= C(\langle \mathbf{return}\ \mathsf{true} \mid \gamma \mid \sigma \rangle)\,[] + C(\langle \mathbf{return}\ \mathsf{false} \mid \gamma \mid \sigma \rangle)\,[] \\
C(\langle p\, V \mid \gamma \mid \sigma \rangle)\,(b :: bs) &\simeq C(\langle \mathbf{return}\ b \mid \gamma \mid \sigma \rangle)\,bs \\
C(\langle M \mid \gamma \mid \sigma \rangle)\,bs &\simeq C(\langle M' \mid \gamma' \mid \sigma' \rangle)\,bs, \qquad \text{if } \langle M \mid \gamma \mid \sigma \rangle \longrightarrow \langle M' \mid \gamma' \mid \sigma' \rangle
\end{aligned}$$

where p is a distinguished free variable such that in all of the above equations $\gamma(p) = \gamma'(p) = p$. As with $\mathcal{T}$, we write $C(P)$ for $C(P\ p)$.

Definition 5.9 (generic count program). A program $C : ((\mathsf{Nat} \to \mathsf{Bool}) \to \mathsf{Bool}) \to \mathsf{Nat}$ is said to be an n-count program if for every n-standard predicate P

$$C\ P \rightsquigarrow^+ \mathbf{return}\ C(P)([])$$

The restriction to n-standard predicates might at first seem rather tiresome and unnatural, but in the context of our work it has two motivations. First, it allows us to present the essence of our effectful generic counting algorithm in its simplest, cleanest form (compare the effcount program given above with the more widely applicable versions in Section 6.1 below). Second, it will enable us in Section 5.6 to present our main negative result in a particularly sharp form: in the base language λ_{b}, no n-count program can compete with effcount *even on n-standard predicates.*

5.5 Complexity of Effectful Generic Counting

In this section we formulate correctness and asymptotic bounds for running the effectful generic counting program effcount on a predicate P. Full proofs are in Appendix C.

A key feature of the proof is that we must alternate between intensional and extensional modes of reasoning. As effcount is a fixed program, we can reason intensionally about its behaviour and thereby directly observe machine transitions. But we must also consider the transitions of P. Since the code for P is unknown we cannot employ the same reasoning technique. Instead, we reason extensionally by making use of the fact that the timed decision tree model of P contains the exact number of transitions that P performs in each branch of computation.

THEOREM 5.10. *For all $n > 0$ and any n-standard predicate P it holds that*

(1) *The program* effcount *is a generic counting program.*
(2) *The runtime complexity of* effcount P *is given by:*

$$\sum_{bs \in \mathbb{B}^*}^{|bs| \leq n} \mathsf{steps}(\mathcal{T}(P))(bs) + O(2^n)$$

Proof. Both items can be proved by downwards induction on the length of bs and alternating, as needed, between intensional reasoning about reduction steps within effcount and extensional reasoning about reduction steps for P. We give the full details in Appendix C. □

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

As a related aside, one might also ask about the execution time for an implementation of λ_{h} that performs genuine copying of continuations, as in systems such as MLton [2020]. We will not present the details of such an implementation, but it is informally clear that our $O(2^n)$ bound would still apply as long as the continuations associated with internal nodes of $\mathcal{T}(P)$ never becomes too large. Specifically, we might consider a class $\mathcal{Q}_{n,c,k}$ of n-standard predicates P for which the edge times in $\mathcal{T}(P)$ never exceed c and the sizes of the continuations never exceed k. (Once again, for reasonable c and k this gives us a respectable class of predicates.) Then it is intuitively clear that for such predicates, the total continuation-copying time will be $O(2^n)$, so that the overall runtime will still be $O(2^n)$.

5.6 Pure Generic Counting

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

One might ask at this point whether the claimed lower bound could not be obviated by means of some known continuation passing style (CPS) or monadic transform of effect handlers [Hillerström et al. 2017; Leijen 2017]. This can indeed be done, but only by dint of changing the type of our predicates P — which, as noted in the introduction, would defeat the purpose of our present enquiry. Our intention is precisely to investigate the relative power of various languages for manipulating predicates that are presented to us in a certain way which we do not have the luxury of choosing.

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

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

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

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

Suppose that we first construct some program

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

which, given a predicate P, returns some n-point p such that $P\ p$ evaluates to true whenever this is possible (i.e. whenever some such point exists). If P returns false on every n-point, we require simply that $\mathsf{bestshot}_n\ P$ returns some arbitrary n-point. (In other words, $\mathsf{bestshot}_n$ embodies Hilbert's choice operator ε on predicates.) It is once again routine to construct such a program by naive means; and we may moreover assume that for any P, the evaluation of $\mathsf{bestshot}_n\ P$ takes only constant time, all the real work being deferred until the argument of type Nat_n is supplied.

Now consider the following program:

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

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

This might seem a rather footling point, as $\mathsf{lazycount}_n$ offers this efficiency gain *only* on (some implementations of) the everywhere false predicate. However, by means of a recursive application of such a nesting trick, we may arrive at a generic count program that spectacularly defies the $\Omega(2^n)$ lower bound for an interesting class of (non-n-standard) predicates, and indeed proves quite viable for counting solutions to 'n-queens' and similar problems. We shall refer to this program BergerCount, since it is modelled largely on Berger's PCF implementation of the so-called *fan functional* ([Berger 1990]; see also [Longley and Normann 2015]). This program is of some interest in its own right, and will be briefly presented in Section 6.3. As we shall see, BergerCount actually requires a mild extension of λ_{b} with a 'memoisation' primitive to achieve the effect of call-by-need evaluation; but such a language can still be seen as purely 'functional' in the same sense as Haskell.

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

We now proceed to the proof itself. In the interests of clarity, we first present a proof in the basic setting of λ_{b}; later we will see how the approach scales to languages with state (Section 6.2).

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

We now establish a key lemma, which vindicates the naive intuition that in the n-standard case, the only way to discover the correct value for $\mathsf{count}\ P$ is to perform 2^n separate applications $P\ p$ (albeit allowing for the possibility that these applications need not be performed 'in turn' but might be nested in some complex way). We outline the proof here; full details are in Appendix D.

Lemma 5.11 (No shortcuts). *If C is an n-count program and P is an n-standard predicate, then C 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 $\mathcal{E}[P\ p]$ appearing in the small-step reduction of $C\ P$ such that p is a closed value (hence an n-point) and $\mathbb{P}[\![p]\!] = \pi$.*

Proof. Suppose C and P are as above, and suppose for contradiction that π is some semantic n-point such that no corresponding application $P\ p$ ever arises in the course of computing $C\ P$. Let t be the untimed decision tree for P. Now consider the leaf node in t corresponding to the point π, and let t' be the tree obtained from t' by simply negating the boolean value at this leaf node. It is then a fairly simple matter to construct a predicate P' whose decision tree is t'.

Since the numbers of true-leaves in t and t' differ by 1, it is clear that if C is indeed a correct n-count program, then the values returned by $C\ P$ and $C\ P'$ will have an absolute difference of 1. On the other hand, we will argue that if the computation of $C\ P$ never actually 'visits' the leaf node in question, then C is unable to detect any difference between P and P'.

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

> Suppose $C\ P \rightsquigarrow^* \mathcal{E}[P\ p[P]]$ where $\mathcal{E}$ is an evaluation context, and the context $p[-]$ abstracts all occurrences of P that are residuals of the key occurrence in $C\ P$. If $P\ p[P] \rightsquigarrow^m \mathbf{return}\ V$, then also $P'\ p[P'] \rightsquigarrow^* \mathbf{return}\ V$.

To show this, we note that the tree t provides an analysis of the reduction behaviour of $P\ p[P]$, and this behaviour can be seen to be mimicked by $P'\ p[P']$ using the induction hypothesis together with the fact that P' has tree t' and $p[P]$ does not denote the point π.

From the above claim one may now read off that if $C\ P \rightsquigarrow^* \mathbf{return}\ c$ then also $C\ P' \rightsquigarrow^* \mathbf{return}\ c$. This gives the desired contradiction, as we have already noted that these values must be different. □

Corollary 5.12. *Suppose C and P are as in the preceding Lemma. For any semantic n-point π, the reduction sequence for $C\ P$ contains at least n occurrences of terms $\mathcal{F}[p\ i]$, where $\mathcal{F}[-]$ is an evaluation context, p is an n-point denoting π, and i is a natural number value.*

Proof. Let π be any semantic n-point. By the previous lemma, the reduction sequence for $C\ P$ contains some term $\mathcal{E}[P\ p]$ where p is an n-point denoting π; and the n-standardness of P tells us that the reduction sequence for $P\ p$ contains n occurrences of terms $\mathcal{G}[p\ i]$ where i is a natural number value and $\mathcal{G}$ is an evaluation context. Hence the reduction sequence for $C\ P$ contains n occurrences of terms $\mathcal{F}[p\ i] \equiv \mathcal{E}[\mathcal{G}[p\ i]]$. □

The desired lower bound now follows. Since our n-points p are assumed to be values, it is clearly impossible for the same term to be of the form $\mathcal{E}[p\ i]$ and $\mathcal{E}'[p'\ i']$ for two distinct n-points p, p' and evaluation contexts $\mathcal{E}, \mathcal{E}'$. It is therefore immediate from our corollary that the reduction sequence for $C\ P$ consists of at least $n2^n$ distinct terms, i.e. the reduction has length $\geq n2^n$.

Theorem 5.13. *If C is an n-count program and P is any n-standard predicate, then the evaluation of $C\ P$ must take time $\Omega(n2^n)$.* □

As we shall see, the above argument goes through with just minor adjustments for an extension of λ_b with exceptions, and also for a language containing the memoisation primitive required for BergerCount. For a stateful language, however, some further ingredients are required: we will return to this in Section 6.

6 EXTENSIONS AND VARIATIONS

Our complexity result is robust in that continues to hold in more general settings. We outline here how it generalises beyond n-standard predicates and to richer base languages.

6.1 Beyond n-Standard Predicates

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

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

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

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

$$\begin{aligned} \mathsf{empty}_n &: \mathsf{Map}_n \\ \mathsf{add}_n &: \langle \mathsf{Nat}_n, \mathsf{Bool} \rangle \to \mathsf{Map}_n \to \mathsf{Map}_n \\ \mathsf{lookup}_n &: \mathsf{Nat}_n \to \mathsf{Map}_n \to 1 + \mathsf{Bool} \end{aligned}$$

Invoking the lookup function lookup i *map* returns $\mathbf{inl}\ \langle\rangle$ if i is not present in *map*, and $\mathbf{inr}\ ans$ if i is present, where $ans : \mathsf{Bool}$ is the value associated with i. We can realise suitable maps in λ_b such that the time complexity of add_n and lookup_n is $O(\log n)$ [Okasaki 1999].

We can now use parameter-passing to support repeated queries as follows.

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

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

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

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

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

6.2 Extending λ_{b} with State

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

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

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

The simplest way to circumvent this difficulty is to define a predicate to be a closed value $P : \mathsf{Predicate}_n$ *within the sublanguage* λ_{b}. For such predicates, the notions of decision tree, counting function and n-standardness are unproblematic. Our result will establish a runtime lower bound of $\Omega(n2^n)$ for count programs $C \in \lambda_{\mathrm{s}}$ applied only to predicates P of this kind.

On the other hand, since C itself may be stateful, we cannot exclude the possibility that $C\ P$ will apply P to terms p that are themselves stateful. Such a term p will no longer unambiguously denote some semantic point π, and this means the proof of Section 5.6 will not go through as it stands.

To adapt our proof to the setting of λ_{s}, a little more machinery will be helpful. If C is an n-count program and P an n-standard predicate, we expect that the evaluation of $C\ P$ will at various points feature terms $\mathcal{E}[P\ p]$ which are then reduced in subsequent steps to some $\mathcal{E}[\mathbf{return}\ W]$, via a reduction sequence which, modulo $\mathcal{E}[-]$, has the following form:

$$P\ p \leadsto^* \mathcal{E}_0[p\ i_0] \leadsto^* \mathcal{E}_0[\mathbf{return}\ b_0] \leadsto^* \cdots \leadsto^* \mathcal{E}_{n-1}[p\ i_{n-1}] \leadsto^* \mathcal{E}_{n-1}[\mathbf{return}\ b_{n-1}] \leadsto^* \mathbf{return}\ W$$

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

Since P is n-standard, we know that each thread will consist of $n+1$ P-sections separated by n p-sections. Indeed, it is clear that this computation traces the path $b_0 \ldots b_{n-1}$ through the decision tree for P, with $i_0, \ldots, i_{n-1}$ the corresponding internal node labels. We may now construe $b_0 \ldots b_{n-1}$

as a semantic point $\pi : \mathbb{N}_n \to \mathbb{B}$, and call it the semantic point associated with (the thread arising from) the application occurrence $P\ p$.

The following lemma now serves as a surrogate for Lemma 5.11:

Lemma 6.1. *Let P be an n-standard predicate. For any semantic point $\pi : \mathbb{N}_n \to \mathbb{B}$, the evaluation of $C\ P$ involves an application occurrence $P\ p$ associated with π.*

The proof of this lemma is not too different from that of Lemma 5.11: if π were a point with no associated thread, there would be an unvisited leaf in the decision tree, and we could manufacture an n-standard predicate P' whose tree differed from that of P only at this leaf. We can then show, by induction on length of reductions, that any portion of the evaluation of $C\ P$ can be suitably mimicked with P replaced by P'. Naturally, this idea now needs to be formulated at the level of configurations rather than plain terms: in the course of reducing $(C\ P, 0, [])$, we may encounter configurations (M, l, s) in which residual occurrences of P have found their way into s as well as M, so in order to replace P by P' we must abstract on all these occurrences via an evident notion of *configuration context*. With this adjustment, however, the argument of Lemma 5.11 goes through.

Since each thread involves at least the n terms $\mathcal{E}_j[p\ i_j]$, our proof of the $\Omega(n2^n)$ bound is complete provided we can show that no two threads overlap: more precisely, none of the above terms $\mathcal{E}_j[p\ i_j]$ can belong to the P-section of more than one thread. The difficulty here is that because syntactic points no longer have unambiguous denotations, the relevant π can no longer be simply read off from p: indeed, it is entirely possible that our computation may involve two instances of the same application $P\ p$ giving rise to entirely different threads owing to the presence of state. Fortunately, however, we may reason as follows.

Let us suppose that $P\ p$ and $P\ p'$ are any two application occurrences arising in the evaluation of $C\ P$, with $P\ p$ appearing before $P\ p'$, and suppose these respectively give rise to threads T, T'. We wish to show that the P-sections of T do not overlap with those of T'. There are three cases:

- If T' does not start until after T has finished, then of course T, T' are disjoint.
- If T' starts within some p-section $\mathcal{E}_j[p\ i_j] \leadsto^* \mathcal{E}_j[\mathbf{return}\ b_j]$ of T, then it is not hard to see that T' must also end within this same p-section, as the evaluation of $P\ p'$ forms part of the evaluation of $p\ i_j$.
- It is not possible for T' to start within a P-section of T. This follows from the fact that a 'residual occurrence' of P (that is, one arising as a residual of the P in $C\ P$) cannot itself contain other residual occurrences of P; thus, for any term arising from the reduction of $P\ p$ (discounting $P\ p$ itself), every residual occurrence of P occurs within some p.

Arguing along such lines, one can show that any two threads are indeed 'disjoint', in such a way that there must be at least $n2^n$ steps in the overall reduction sequence.

6.3 Berger Count

We now briefly outline the BergerCount program alluded to in Section 5.6, in order to fill out our overall picture of the relationship between language expressivity and potential program efficiency.

Berger's original program [Berger 1990] introduced a remarkable search operator for predicates on *infinite* streams of booleans, and has played an important role in higher-order computability theory [Longley and Normann 2015]. What we wish to highlight here is that if one applies the algorithm to predicates on *finite* boolean vectors, the resulting program, though no longer interesting from a computability perspective, still holds some interest from a complexity standpoint: indeed, it yields what seems to be the best available implementation of generic counting 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.

$$\begin{array}{l}
\mathsf{bestshot}_n : \mathsf{Predicate}_n \to \mathsf{Point}_n \\
\mathsf{bestshot}_n\ P := \mathsf{bestshot}'_n\ P\ []
\end{array}$$

$$\begin{array}{l}
\mathsf{bestshot}'_n : \mathsf{Predicate}_n \to \mathsf{List\ Bool} \to \mathsf{Point}_n \\
\mathsf{bestshot}'_n\ P\ \mathit{start} := \mathbf{let}\ f \leftarrow \mathsf{memoise}\ (\lambda\langle\rangle.\mathsf{bestshot}''_n\ P\ \mathit{start})\ \mathbf{in} \\
\qquad\qquad\qquad\qquad \mathbf{return}\ (\lambda i.\mathbf{if}\ i < |\mathit{start}|\ \mathbf{then}\ \mathit{start}.i\ \mathbf{else}\ (f\ \langle\rangle).i)
\end{array}$$

$$\begin{array}{l}
\mathsf{bestshot}''_n : \mathsf{Predicate}_n \to \mathsf{List\ Bool} \to \mathsf{List\ Bool} \\
\mathsf{bestshot}''_n\ P\ \mathit{start} := \mathbf{if}\ |\mathit{start}| = n\ \mathbf{then\ return}\ \mathit{start} \\
\qquad\qquad\qquad\qquad \mathbf{else\ let}\ f \leftarrow \mathsf{bestshot}'_n\ P\ (\mathsf{append}\ \mathit{start}\ [\mathsf{true}])\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\quad \mathbf{if}\ P\ f\ \mathbf{then\ return}\ [f\ 0, \ldots, f\ (n-1)] \\
\qquad\qquad\qquad\qquad\quad \mathbf{else}\ \mathsf{bestshot}''_n\ P\ (\mathsf{append}\ \mathit{start}\ [\mathsf{false}])
\end{array}$$

The function $\mathsf{bestshot}_n$ will return a point satisfying the given predicate P if there is one, or the dummy point $\lambda i.\mathsf{false}$ if there is none. This is implemented by means of two mutually recursive auxiliary functions whose workings are admittedly hard to elucidate in a few words. The function $\mathsf{bestshot}'_n$ is a generalisation of $\mathsf{bestshot}_n$ that makes a best shot at finding a point p satisfying P and matching some specified list *start* in some initial segment of its components $[p\ 0, \ldots, p\ (i-1)]$. This works 'lazily', drawing its values from *start* wherever possible, and performing an actual search only when required. This actual search is undertaken by $\mathsf{bestshot}''_n$, which proceeds by first searching for a solution that extends the specified list with true; but if no such solution is forthcoming, it settles for false as the next component of the point being constructed. The whole procedure relies on a subtle combination of laziness, recursion and implicit nesting of calls to P which means that the search is self-pruning in regions of the binary tree where P only demands some initial segment $p\ 0,\ldots,p\ (i-1)$ of its argument p.

The above program makes use of an operation

$$\mathsf{memoise} : (1 \to \mathsf{List\ Bool}) \to (1 \to \mathsf{List\ Bool})$$

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 in λ_s, or alternatively may simply be added as a primitive in its own right (we omit the details). The latter has the advantage that it preserves the purely 'functional' character of the language, in the sense that every program is observationally equivalent to a λ_b program, namely the one obtained by replacing memoise by the identity.

We now show how the above idea may be exploited to yield a generic count program (this part of our work appears to be new).

$$\begin{array}{l}
\mathsf{BergerCount}_n : \mathsf{Predicate}_n \to \mathsf{Nat} \\
\mathsf{BergerCount}_n\ P := \mathsf{count}'_n\ P\ []\ 0
\end{array}$$

$$\begin{array}{l}
\mathsf{count}'_n : \mathsf{Predicate}_n \to \mathsf{List\ Bool} \to \mathsf{Nat} \to \mathsf{Nat} \\
\mathsf{count}'_n\ P\ \mathit{start}\ \mathit{acc} := \mathbf{if}\ |\mathit{start}| = n\ \mathbf{then}\ \mathit{acc} + (\mathbf{if}\ P(\lambda i.\mathit{start}.i)\ \mathbf{then\ return}\ 1\ \mathbf{else\ return}\ 0) \\
\qquad\qquad\qquad\qquad \mathbf{else\ let}\ f \leftarrow \mathsf{bestshot}'_n\ P\ \mathit{start}\ |\mathit{start}|\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\quad \mathbf{if}\ P\ f\ \mathbf{then}\ \mathsf{count}''_n\ \mathit{start}\ [f\ 0, \ldots, f\ (n-1)]\ \mathit{acc}\ \mathbf{else\ return}\ \mathit{acc}
\end{array}$$

$$\begin{array}{l}
\mathsf{count}''_n : \mathsf{Predicate}_n \to \mathsf{List\ Bool} \to \mathsf{List\ Bool} \to \mathsf{Nat} \to \mathsf{Nat} \\
\mathsf{count}''_n\ P\ \mathit{start}\ \mathit{leftmost}\ \mathit{acc} := \mathbf{if}\ |\mathit{start}| = n\ \mathbf{then}\ \mathit{acc} + 1 \\
\qquad\qquad\qquad\qquad \mathbf{else\ let}\ b \leftarrow \mathit{leftmost}.|\mathit{start}|\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\quad \mathbf{let}\ \mathit{acc}' \leftarrow \mathsf{count}''_n\ (\mathsf{append}\ \mathit{start}\ [b])\ \mathit{leftmost}\ \mathit{acc}\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\quad \mathbf{if}\ b\ \mathbf{then}\ \mathsf{count}'_n\ (\mathsf{append}\ \mathit{start}\ [\mathsf{false}])\ \mathit{acc}'\ \mathbf{else\ return}\ \mathit{acc}'
\end{array}$$

Again, $\mathsf{BergerCount}_n$ is implemented by means of two mutually recursive auxiliary functions. The function count'_n counts the solutions to P that start with the specified list of booleans, adding their number to a previously accumulated total given by *acc*. The function count''_n does the same thing,

	Queens						Integration								
	First solution			All solutions			Id	Squaring			Logistic				
Parameter	20	24	28	8	10	12	20	14	17	20	1	2	3	4	5
Naïve	∞	∞	∞	274.18	∞	∞	17.17	50.61	65.8	80.58	∞	∞	∞	∞	∞
Berger	9.29	12.69	∞	2.11	2.81	3.41	5.59	23.30	25.65	27.50	26.10	33.27	34.02	32.76	31.00
Pruned	2.03	2.37	2.66	1.29	1.42	1.52	2.27	4.39	5.00	5.08	4.80	6.25	7.18	8.09	8.80
Bespoke	0.13	0.12	0.12	0.15	0.05	0.04									

Table 1. Runtimes Relative to the Effectful Implementation

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, 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 p in order p 0, p 1, p 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 7 attest to the viability of this approach and its overwhelming superiority over the naive functional method.

By contrast, the above program is *not* able to take advantage of parts of the tree where our predicate 'succeeds fast', i.e. returns true after seeing only a few components. Unlike the effectful count program of Section 5.2, 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, any evaluation of $\mathrm{count}_n\ P$ that returns a natural number c must take time $\Omega(c)$. These observations informally indicate the likely extent of the efficiency gap between effectful and purely functional computation when it comes to non-n-standard predicates.

7 EXPERIMENTS

The theoretical efficiency gap between realisations of λ_{b} and λ_{h} manifests in practice. We have observed it empirically on instantiations of n-queens and exact real number integration, which can be cast as generic search. Table 1 shows the speedup of using an effectful implementation of generic search over various pure implementations. We discuss the benchmarks and results in further detail below.

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

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

Each benchmark was run 11 times. The reported figure is the median runtime ratio between the particular implementation and the baseline effectful implementation. Benchmarks that failed to terminate within a threshold (1 minute for single solution, 8 minutes for enumerations), are reported as ∞. The experiments were conducted in SML/NJ v110.78 with factory settings on an Intel Xeon CPU E5-1620 v2 @ 3.70GHz powered workstation running Ubuntu 16.04. The effectful implementation uses an encoding of delimited control akin to effect handlers based on top of SML/NJ's call/cc.

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

Exact Real Integration. The integration benchmarks are adapted from Simpson [1998]. We integrate three different functions with varying precision in the interval $[0, 1]$. For the identity function (Id) at precision 20 the speedup relative to Berger is 5.59×. For the squaring function the speedups are larger at higher precisions: at precision 14 the speedup is 4.39× over the pruned integrator, whilst it is 5.08× at precision 20. The speedups are more extreme against the naïve and Berger integrators. We also integrate the logistic map $x \mapsto 1 - 2x^2$ at a fixed precision of 15. We make the function harder to compute by iterating it up to 5 times. Between the pruned and effectful integrator the speedup ratio increases as the function becomes harder to compute.

8 CONCLUSIONS AND FUTURE WORK

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

We have also verified that the same lower bound applies to a language λ_e which extends λ_b with (Benton-Kennedy style [Benton and Kennedy 2001]) *exceptions* and handlers — and even for the combined language λ_se with both state and exceptions. As was the case for λ_s, it is helpful to insist here that our predicates themselves are terms of λ_b. However, the adaptations of our proof method required for λ_e are less interesting and far-reaching than those for λ_s so we do not present them here. We also remark informally that λ_se seems to bring us close to the expressive power of real languages such as Standard ML, Java, and Python, strongly suggesting that the speedup we have discussed is unattainable in these language.

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

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

ACKNOWLEDGMENTS

We would like to thank James McKinna for insightful discussions about this work. Daniel Hillerström was supported by EPSRC grant EP/L01503X/1 (EPSRC Centre for Doctoral Training in Pervasive Parallelism). Sam Lindley was supported by EPSRC grant EP/K034413/1 (From Data Types to Session Types—A Basis for Concurrency and Distribution).

REFERENCES

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Andrzej Filinski. 1994. Representing Monads. In *POPL*. ACM Press, 446–457.

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

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

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

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

Daniel Hillerström, Sam Lindley, and Robert Atkey. 2020. Effect Handlers via Generalised Continuations. *J. Funct. Program.* 30 (2020). To appear.

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

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

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

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

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

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

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

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

Sam Lindley. 2014. Algebraic effects and effect handlers for idioms and arrows. In *WGP@ICFP*. ACM, 47–58.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Configurations

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

Pure continuations

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

Computation terms

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

Value terms and values

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

Fig. 8. Mapping from Base Machine Configurations to Terms

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 8 describes such a mapping $\llparenthesis - \rrparenthesis$ from configurations to terms via a collection of mutually recursive functions defined on configurations, continuations, computation terms, value terms, and machine values. The mapping makes use of two operations on environments, γ, which we define now.

Definition A.1. We write $dom(\gamma)$ for the domain of γ, and $\gamma \backslash \{x_1, \ldots, x_n\}$ for the restriction of environment γ to $dom(\gamma) \backslash \{x_1, \ldots, x_n\}$.

The $\llparenthesis - \rrparenthesis$ 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 $\llparenthesis - \rrparenthesis$ is invariant under this rule. This leaves the β-rules (M-App), (M-Split), (M-Case), and (M-RetCont). Each of these corresponds directly with performing a reduction in the operational semantics.

Definition A.2 (Auxiliary reduction relations). We write $\longrightarrow_{\mathrm{a}}$ for administrative steps (M-Let) and $\simeq_{\mathrm{a}}$ for the symmetric closure of $\longrightarrow_{\mathrm{a}}^*$. We write $\longrightarrow_\beta$ for β-steps (all other rules) and $\Longrightarrow$ for a sequence of steps of the form $\longrightarrow_{\mathrm{a}}^* \longrightarrow_\beta$.

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

Lemma A.3. *Suppose M is a computation and C is configuration such that $\llparenthesis C \rrparenthesis = M$, then if $M \leadsto N$ there exists C' such that $C \Longrightarrow C'$ and $\llparenthesis C' \rrparenthesis = N$, or if $M \not\leadsto$ then $C \not\Longrightarrow$.*

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

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

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

Computation Terms and Machine Values

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

Fig. 9. Mapping from Handler Machine Configurations to Terms

The correspondence here is rather strong: there is a one-to-one mapping between $\rightsquigarrow$ and $\Longrightarrow / \simeq_{\mathrm{a}}$. The inverse of the lemma is straightforward as the semantics is deterministic. Notice that Lemma A.3 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 A.4 (Base simulation). *If $\vdash M : A$ and $M \rightsquigarrow^+ N$ such that N is normal, then $\langle M \mid \emptyset \mid []\rangle \longrightarrow^+ C$ such that $\llparenthesis C\rrparenthesis = N$, or $M \not\rightsquigarrow$ then $\langle M \mid \emptyset \mid []\rangle \not\longrightarrow$.*

Proof. By repeated application of Lemma A.3. □

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 9 shows the necessary changes to the $\llparenthesis - \rrparenthesis$ function.

Lemma B.1. *Suppose M is a computation and C is configuration such that $\llparenthesis C\rrparenthesis = M$, then if $M \rightsquigarrow N$ there exists C' such that $C \Longrightarrow C'$ and $\llparenthesis C'\rrparenthesis = N$, or if $M \not\rightsquigarrow$ then $C \not\Longrightarrow$.*

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

Theorem B.2 (Handler simulation). *If $\vdash M : A$ and $M \rightsquigarrow^+ N$ such that N is normal, then $\langle M \mid \emptyset \mid \kappa_0\rangle \longrightarrow^+ C$ such that $\llparenthesis C\rrparenthesis = N$, or $M \not\rightsquigarrow$ then $\langle M \mid \emptyset \mid \kappa_0\rangle \not\longrightarrow$.*

Proof. By repeated application of Lemma B.1. □

C PROOF DETAILS FOR THE COMPLEXITY OF EFFECTFUL GENERIC COUNTING

In this appendix we give proof details and artefacts for Theorem 5.10. Throughout this section we let H_{count} denote the handler definition of count, that is

$$H_{\mathsf{count}} := \left\{ \begin{array}{ll} \textbf{val } x & \mapsto \textbf{if } x \textbf{ then return } 1 \textbf{ else return } 0 \\ \mathsf{Branch}\ \langle\rangle\ r & \mapsto \textbf{let } x \leftarrow r\ \mathsf{true} \textbf{ in} \\ & \quad\ \textbf{let } y \leftarrow r\ \mathsf{false} \textbf{ in} \\ & \quad\ x + y \end{array} \right\}$$

The timed decision tree model embeds timing information. For the proof we must also know the abstract machine environment and the pure continuation. Thus we decorate timed decision trees with this information.

Definition C.1 (decorated timed decision trees). A decorated timed decision tree is a partial function $t : \mathbb{B}^* \rightharpoonup (\mathsf{Lab} \times \mathsf{Nat}) \times (\mathsf{Env} \times \mathsf{PureCont})$ such that its first projection $bs \mapsto t(bs).1$ is a timed decision tree. As an abbreviation, we define $\mathsf{DT} := \mathbb{B}^* \rightharpoonup (\mathsf{Lab} \times \mathsf{Nat}) \times (\mathsf{Env} \times \mathsf{PureCont})$.

We extend the projections labs and steps in the obvious way to work over decorated timed decision trees. We define two further projections. The first $\mathsf{env}(t) := bs \mapsto t(bs).2.1$ projects the environment, whilst the second $\mathsf{pure}(t) := bs \mapsto t(bs).2.2$ projects the pure continuation.

The following definition gives a procedure for constructing a decorated timed decision tree. The construction is similar to that of Definition 5.4.

Definition C.2. The decorated timed decision tree of a configuration is defined by the following equations

$$\begin{aligned}
\mathcal{D} &: \mathsf{Conf} \rightharpoonup \mathsf{DT} \\
\mathcal{D}(\langle \mathbf{return}\ \mathsf{true} \mid \gamma \mid [] \rangle)\,[] &= ((!\mathsf{true}, 0), (\gamma, [])) \\
\mathcal{D}(\langle \mathbf{return}\ \mathsf{false} \mid \gamma \mid [] \rangle)\,[] &= ((!\mathsf{false}, 0), (\gamma, [])) \\
\mathcal{D}(\langle p\ V \mid \gamma \mid \sigma \rangle)\,[] &= ((?[\![V]\!]\gamma, 0), (\gamma, \sigma)) \\
\mathcal{D}(\langle p\ V \mid \gamma \mid \sigma \rangle)\,(b :: bs) &\simeq \mathcal{D}(\langle \mathbf{return}\ b \mid \gamma \mid \sigma \rangle)\ bs \\
\mathcal{D}(\langle M \mid \gamma \mid \sigma \rangle)\ bs &\simeq \mathcal{I}(\mathcal{D}(\langle M' \mid \gamma' \mid \sigma' \rangle)\ bs), \\
&\quad \text{if } \langle M \mid \gamma \mid \sigma \rangle \longrightarrow \langle M' \mid \gamma' \mid \sigma' \rangle
\end{aligned}$$

where $\mathcal{I}((\ell, s), (\gamma, \sigma)) := ((\ell, s+1), (\gamma, \sigma))$ and p is a distinguished free variable such that in all of the above equations $\gamma(p) = \gamma'(p) = p$.

We shall write $\mathcal{D}(P)$ to mean $\mathcal{D}(\langle P\ p \mid \emptyset[p \mapsto p] \mid [] \rangle)$.

We define some functions, that given a list of booleans and a n-standard predicate, compute configurations of the effectful abstract machine at particular points of interest during evaluation of the given predicate. Let $\chi_{\mathsf{count}}(V) := (\emptyset[pred \mapsto [\![V]\!]\emptyset], H_{\mathsf{count}})$ denote the handler closure of H_{count}.

Notation. For an n-standard predicate P we write $|P| = n$ for the size of the predicate. Furthermore, we define χ_{id} for the identity handler closure $(\emptyset, \{\mathbf{val}\ x \mapsto x\})$.

Definition C.3 (computing machine configurations). For any given n-standard predicate P and a list of booleans bs, such that $|bs| \leq n$, we can compute machine configurations at points of interest during evaluation of count P.

To make the notation slightly simpler we use the following conventions whenever n, t, and c appear free: $n = |P|$, $t = \mathcal{D}(P)$, and $c = \mathcal{C}(P)$.

- The function arrive either computes the configuration at a query node, if $|bs| < n$, or the configuration at an answer node.

$$\begin{aligned}
&\mathsf{arrive} : \mathbb{B}^* \times \mathsf{Val} \rightharpoonup \mathsf{Conf} \\
&\mathsf{arrive}(bs, P) := \langle V\ j \mid \gamma \mid (\sigma, \chi_{\mathsf{count}}(P)) :: \mathsf{residual}(bs, P) \rangle, \quad \text{if } |bs| < n \\
&\text{where } \gamma = \mathsf{env}(t)(bs), ?j = \mathsf{labs}(t)(bs), \text{ and } [\![V]\!]\gamma = (\mathsf{env}^{\perp}(P), \lambda_.\mathbf{do}\ \mathsf{Branch}\ \langle\rangle) \\
&\mathsf{arrive}(bs, P) := \langle \mathbf{return}\ b \mid \gamma \mid ([], \chi_{\mathsf{count}}(P)) :: \mathsf{residual}(bs, P) \rangle, \quad \text{if } |bs| = n \\
&\qquad\qquad \text{where } \gamma = \mathsf{env}(t)(bs) \text{ and } !b = \mathsf{labs}(t)(bs)
\end{aligned}$$

- Correspondingly, the depart function computes the configuration either after the completion of a query or handling of an answer.

$$\begin{aligned}
&\mathsf{depart} : \mathbb{B}^* \times \mathsf{Val} \rightharpoonup \mathsf{Conf} \\
&\mathsf{depart}(bs, P) := \langle \mathbf{return}\ m \mid \gamma \mid \mathsf{residual}(bs, P) \rangle, \quad \text{if } |bs| < n \\
&\text{where } \gamma = \mathsf{env}^{\uparrow}_{\mathsf{false}}(bs, P) \text{ and } m = c(\mathsf{true} :: bs) + c(\mathsf{false} :: bs) \\
&\mathsf{depart}(bs, P) := \langle \mathbf{return}\ m \mid \gamma \mid \mathsf{residual}(bs, P) \rangle, \quad \text{if } |bs| = n \\
&\qquad\qquad \text{where } \gamma = \mathsf{env}^{\perp}(P) \text{ and } m = c(bs)
\end{aligned}$$

The two clauses of depart yield slightly different configurations. The first clause computes a configuration inside the operation clause of H_{count}. The configuration is exactly tail-configuration after summing up the two respective values returned by the two invocations of resumption. Whilst the second clause computes the tail-configuration inside of the success clause of H_{count} after handling a return value of the predicate.

- The residual function computes the residual continuation structure which contains the bits of computations to perform after handling a complete path in a decision tree.

$$\begin{aligned} \mathsf{residual} &: \mathbb{B}^* \times \mathsf{Val} \rightharpoonup \mathsf{Cont} \\ \mathsf{residual}(bs, P) &:= [(\mathsf{purecont}(bs, P), \chi_{id})] \end{aligned}$$

- The function purecont computes the pure continuation.

$$\begin{aligned} \mathsf{purecont} &: \mathbb{B}^* \times \mathsf{Val} \rightharpoonup \mathsf{PureCont} \\ \mathsf{purecont}([], P) &:= [] \\ \mathsf{purecont}(\mathsf{true} :: bs, P) &:= (\gamma, x_{\mathsf{true}}, \mathbf{let}\ x_{\mathsf{false}} \leftarrow r\ \mathsf{false}\ \mathbf{in}\ x_{\mathsf{true}} + x_{\mathsf{false}}) :: \mathsf{purecont}(bs, P), \\ &\qquad \text{where } \gamma = \mathsf{env}^{\downarrow}_{\mathsf{true}}(\mathsf{true} :: bs, P) \\ \mathsf{purecont}(\mathsf{false} :: bs, P) &:= (\gamma, x_{\mathsf{false}}, x_{\mathsf{true}} + x_{\mathsf{false}}) :: \mathsf{purecont}(bs, P), \\ &\qquad \text{where } \gamma = \mathsf{env}^{\downarrow}_{\mathsf{false}}(\mathsf{false} :: bs, P) \end{aligned}$$

- The function $\mathsf{env}^{\perp}$ computes the initial environment of the handler. The family of functions $\mathsf{env}^{\downarrow}_{b \in \mathbb{B}}$ contains two functions, one for each instantiation of b, which describe how to compute the environment prior *descending* down a branch as the result of invoking a resumption with b. Analogously, the functions in the family $\mathsf{env}^{\uparrow}_{b \in \mathbb{B}}$ describe how to compute the environment after *ascending* from the resumptive exploration of a branch.

$$\begin{aligned} \mathsf{env}^{\perp} &: \mathsf{Val} \to \mathsf{Env} \\ \mathsf{env}^{\perp}(P) &:= \emptyset[pred \mapsto [\![P]\!]\emptyset] \end{aligned}$$

$$\begin{aligned} \mathsf{env}^{\downarrow}_{\mathsf{true}} &: \mathbb{B}^* \times \mathsf{Val} \rightharpoonup \mathsf{Env} \\ \mathsf{env}^{\downarrow}_{\mathsf{true}}(bs, P) &:= \mathsf{env}^{\perp}(V)[r \mapsto (\sigma, \chi_{\mathsf{count}}(P))], \\ &\qquad \text{where } \sigma = \mathsf{pure}(t)(bs) \end{aligned}$$

$$\begin{aligned} \mathsf{env}^{\downarrow}_{\mathsf{false}} &: \mathbb{B}^* \times \mathsf{Val} \rightharpoonup \mathsf{Env} \\ \mathsf{env}^{\downarrow}_{\mathsf{false}}(bs, P) &:= \gamma[x \mapsto i], \\ &\qquad \text{where } \gamma = \mathsf{env}^{\downarrow}_{\mathsf{true}}(bs, P) \text{ and } i = c(\mathsf{true} :: bs) \end{aligned}$$

$$\begin{aligned} \mathsf{env}^{\uparrow}_{\mathsf{false}} &: \mathbb{B}^* \times \mathsf{Val} \rightharpoonup \mathsf{Env} \\ \mathsf{env}^{\uparrow}_{\mathsf{false}}(bs, P) &:= \gamma[y \mapsto j], \\ &\qquad \text{where } \gamma = \mathsf{env}^{\downarrow}_{\mathsf{false}}(bs, P) \text{ and } j = c(\mathsf{false} :: bs) \end{aligned}$$

We require an auxiliary lemma, because we need to be able to reason about bits of predicate computation, specifically when the predicate is first applied, going from a departure configuration to an arrival configuration, and from a departure configuration to an answer configuration. The following lemma states that for an n-standard predicate, handler machine transitions in lock-step with the base machine.

For a given predicate P we write $\chi_{\mathsf{count}}(P)^{\mathbf{val}}$ to mean $\chi_{\mathsf{count}}(P)^{\mathbf{val}} = (\emptyset[pred \mapsto [\![P]\!]\emptyset], H_{\mathsf{count}})^{\mathbf{val}} = H^{\mathbf{val}}_{\mathsf{count}}$, that is the projection of the success clause of H_{count}.

Lemma C.4. *For any given n-standard predicate P and a list of booleans $bs \in \mathbb{B}^*$ such that $|bs| \leq n$ along with two value V : Bool and $b \in \mathbb{B}$, then the base machine and handler machine transition in lock-step in either way*

(1) *If $|bs| = []$, then*

$$\begin{aligned}&\langle P\ p \mid \gamma \mid []\rangle\\ \longrightarrow_{\mathsf{steps}(t)([])}\ &\langle p\ i \mid \gamma' \mid \sigma\rangle,\end{aligned}$$

where $?i = \mathsf{labs}(t)([])$, $\gamma = \emptyset[P \mapsto P]$, $\gamma' = \mathsf{env}(t)([])$, and $\sigma = \mathsf{pure}(t)([])$; implies the handler machine perform the same amount of transitions

$$\begin{aligned}&\langle P\ p \mid \gamma \mid ([], \chi_{\mathsf{count}}(P)) :: \mathsf{residual}(P, [])\rangle[(\lambda\_.\mathbf{do}\ \mathsf{Branch}\ \langle\rangle)/p]\\ \longrightarrow_{\mathsf{steps}(t)([])}\ &\langle p\ i \mid \gamma' \mid (\sigma, \chi_{\mathsf{count}}(P)) :: \mathsf{residual}(P, [])\rangle[(\lambda\_.\mathbf{do}\ \mathsf{Branch}\ \langle\rangle)/p]\end{aligned}$$

(2) *For $bs = b :: bs'$ we have the following two subcases*

- *If $|bs| < n$, then*

$$\begin{aligned}&\langle \mathbf{return}\ b \mid \gamma \mid \sigma\rangle\\ \longrightarrow_{\mathsf{steps}(t)(b::bs)}\ &\langle p\ i \mid \gamma' \mid \sigma\rangle,\end{aligned}$$

where $?i = \mathsf{labs}(t)(b :: bs)$, $\gamma = \mathsf{env}^{\downarrow}_{b}$, $\gamma' = \mathsf{env}(t)(b :: bs)$, and $\sigma = \mathsf{pure}(t)(bs)$; implies the handler machine perform the same amount of transitions

$$\begin{aligned}&\langle \mathbf{return}\ b \mid \gamma \mid (\sigma, \chi_{\mathsf{count}}(P)) :: \mathsf{residual}(P, b :: bs, n, t, c)\rangle[(\lambda\_.\mathbf{do}\ \mathsf{Branch}\ \langle\rangle)/p]\\ \longrightarrow_{\mathsf{steps}(t)(b::bs)}\ &\langle p\ i \mid \gamma' \mid (\sigma, \chi_{\mathsf{count}}(P)) :: \mathsf{residual}(P, b :: bs, n, t, c)\rangle[(\lambda\_.\mathbf{do}\ \mathsf{Branch}\ \langle\rangle)/p]\end{aligned}$$

- *If $|bs| = n$, then*

$$\begin{aligned}&\langle \mathbf{return}\ b \mid \gamma \mid \sigma\rangle\\ \longrightarrow_{\mathsf{steps}(t)(b::bs')}\ &\langle \mathbf{return}\ b' \mid \gamma' \mid []\rangle,\end{aligned}$$

where $!b' = \mathsf{labs}(t)(b :: bs)$, $\gamma = \mathsf{env}(t)(bs)$, $\gamma' = \mathsf{env}(t)(b :: bs)$, and $\sigma = \mathsf{pure}(t)(bs)$; implies the handler machine perform the same amount of transitions

$$\begin{aligned}&\langle \mathbf{return}\ b \mid \gamma \mid (\sigma, \chi_{\mathsf{count}}(P)) :: \mathsf{residual}(P, b :: bs, n, t, c)\rangle[(\lambda\_.\mathbf{do}\ \mathsf{Branch}\ \langle\rangle)/p]\\ \longrightarrow_{\mathsf{steps}(t)(b::bs')}\ &\langle \mathbf{return}\ b' \mid \gamma' \mid ([], \chi_{\mathsf{count}}(P)) :: \mathsf{residual}(P, b :: bs, n, t, c)\rangle[(\lambda\_.\mathbf{do}\ \mathsf{Branch}\ \langle\rangle)/p]\end{aligned}$$

Proof. Proof by induction on the transition relation $\longrightarrow$. □

Let control : Conf $\rightharpoonup$ Val denote a partial function that hoists a value out of a given machine configuration, that is

$$\mathsf{control}(\langle M \mid \gamma \mid \kappa\rangle) := \begin{cases}[\![V]\!]\gamma & \text{if } M = \mathbf{return}\ V\\ \bot & \text{otherwise}\end{cases}$$

The following lemma performs most of the heavy lifting for the proof of Theorem 5.10.

Lemma C.5. *Suppose P is an n-standard predicate, then for any list of booleans $bs \in \mathbb{B}^*$ such that $|bs| \leq n$*

$$\mathsf{arrive}(bs, P) \rightsquigarrow^{T(bs,n)} \mathsf{depart}(bs, P),$$

and control(depart(bs, P)) $\leq 2^{n-|bs|}$ *with the function T defined as*

$$T(bs, n) = \begin{cases} 9 * (2^{n-|bs|} - 1) + 2^{n-|bs|+1} + \sum_{bs' \in \mathbb{B}^*}^{1 \leq |bs'| \leq n-|bs|} \mathsf{steps}(t)(bs' +\!\!+\, bs) & \textit{if } |bs| < n \\ 2 & \textit{if } |bs| = n \end{cases}$$

Proof. By downward induction on bs.

Base step We have that $|bs| = n$. Since the predicate is n-standard we further have that $n \geq 1$. We proceed by direct calculation.

$$\begin{aligned}
& \mathsf{arrive}(bs, P) \\
= \quad & (\text{definition of arrive when } n = |bs|) \\
& \langle \mathbf{return}\ b \mid \gamma \mid ([], \chi_{\mathsf{count}}(P)) :: \mathsf{residual}(bs, P) \rangle \\
& \qquad\qquad \text{where } \gamma = \mathsf{env}(t)(bs) \text{ and } !b = \mathsf{labs}(t)(bs) \\
\longrightarrow \quad & (\text{M-RetHandler}, \chi_{\mathsf{count}}(P)^{\mathbf{val}} = \{\mathbf{val}\ x \mapsto \cdots\}) \\
& \langle \mathbf{if}\ x\ \mathbf{then}\ \mathbf{return}\ 1\ \mathbf{else}\ \mathbf{return}\ 0 \mid \gamma'[x \mapsto [\![b]\!]\gamma'] \mid \mathsf{residual}(bs, P) \rangle \\
& \qquad\qquad \text{where } \gamma' = \chi_{\mathsf{count}}(P).1
\end{aligned}$$

The value b can assume either of two values. We consider first the case $b = \mathsf{true}$.

$$\begin{aligned}
= \quad & (\text{assumption } b = \mathsf{true}, \text{definition of } [\![-]\!]\ (2 \text{ value steps})) \\
& \langle \mathbf{if}\ x\ \mathbf{then}\ \mathbf{return}\ 1\ \mathbf{else}\ \mathbf{return}\ 0 \mid \gamma'[x \mapsto \mathsf{true}] \mid \mathsf{residual}(bs, P) \rangle \\
\longrightarrow \quad & (\text{M-Case-inl (and log } |\gamma'[x \mapsto \mathsf{true}]| = 1 \text{ environment operations)}) \\
& \langle \mathbf{return}\ 1 \mid \gamma'[x \mapsto \mathsf{true}] \mid \mathsf{residual}(bs, n, P, t, c) \rangle \\
= \quad & (\text{definition of depart when } n = |bs|) \\
& \mathsf{depart}(bs, P)
\end{aligned}$$

We have that control(depart(bs, P)) $= 1 \leq 2^0 = 2^{n-|bs|}$. Next, we consider the case when $b = \mathsf{false}$.

$$\begin{aligned}
= \quad & (\text{assumption } b = \mathsf{false}, \text{definition of } [\![-]\!]\ (2 \text{ value steps})) \\
& \langle \mathbf{if}\ x\ \mathbf{then}\ \mathbf{return}\ 1\ \mathbf{else}\ \mathbf{return}\ 0 \mid \gamma'[x \mapsto \mathsf{false}] \mid \mathsf{residual}(bs, P) \rangle \\
\longrightarrow \quad & (\text{M-Case-Inl (and log } |\gamma'[x \mapsto \mathsf{false}]| = 1 \text{ environment operations)}) \\
& \langle \mathbf{return}\ 0 \mid \gamma'[x \mapsto \mathsf{false}] \mid \mathsf{residual}(bs, n, P, t, c) \rangle \\
= \quad & (\text{definition of depart when } n = |bs|) \\
& \mathsf{depart}(bs, P)
\end{aligned}$$

Again, we have that control(depart(bs, P)) $= 0 \leq 2^0 = 2^{n-|bs|}$.

Step analysis. In either case, the machine uses exactly 2 transitions. Thus we get that

$$2 = T(bs, n), \quad \text{when } |bs| = n$$

Inductive step The induction hypothesis states that for all $b \in \mathbb{B}$ and $|bs| < n$

$$\mathsf{arrive}(b :: bs, P) \rightsquigarrow^{T(b::bs, n)} \mathsf{depart}(b :: bs, P),$$

such that $\text{control}(\text{depart}(b :: bs, P)) \leq 2^{n-|b::bs|}$. We proceed by direct calculation.

$$
\begin{aligned}
& \text{arrive}(bs, P) \\
= \quad & (\text{definition of arrive when } n < |bs|) \\
& \langle V\ j \mid \gamma \mid (\sigma, \chi_{\mathsf{count}}(P)) :: \text{residual}(bs, P)\rangle \\
& \text{where } ?j = \text{labs}(t)(bs), \gamma = \text{env}(t)(bs), \sigma = \text{pure}(t)(bs), \text{ and } V = (\text{env}^{\perp}(P), \lambda\_.\mathbf{do}\ \mathsf{Branch}\ \langle\rangle) \\
\longrightarrow \quad & (\textsc{M-App}) \\
& \langle \mathbf{do}\ \mathsf{Branch}\ \langle\rangle \mid \gamma'[\_ \mapsto [\![j]\!]\gamma'] \mid (\sigma, \chi_{\mathsf{count}}(P)) :: \text{residual}(bs, P)\rangle \\
& \hfill \text{where } \gamma' = \text{env}^{\perp}(P) \\
\longrightarrow \quad & (\textsc{M-Handle-Op}, \chi_{\mathsf{count}}(P)^{\mathsf{Branch}} = \{\mathsf{Branch}\ \langle\rangle\ r \mapsto \cdots\}) \\
& \left\langle \begin{array}{l} \mathbf{let}\ x_{\mathsf{true}} \leftarrow r\ \mathsf{true}\ \mathbf{in} \\ \mathbf{let}\ x_{\mathsf{false}} \leftarrow r\ \mathsf{false}\ \mathbf{in} \\ x_{\mathsf{true}} + x_{\mathsf{false}} \end{array} \mid \gamma[r \mapsto [\![(\sigma, \chi_{\mathsf{count}}(P))]\!]\gamma] \mid \text{residual}(bs, P) \right\rangle \\
& \hfill \text{where } \gamma = \text{env}^{\perp}(P) \\
= \quad & (\text{definition of } [\![-]\!] \text{ (1 value step)}) \\
& \left\langle \begin{array}{l} \mathbf{let}\ x_{\mathsf{true}} \leftarrow r\ \mathsf{true}\ \mathbf{in} \\ \mathbf{let}\ x_{\mathsf{false}} \leftarrow r\ \mathsf{false}\ \mathbf{in} \\ x_{\mathsf{true}} + x_{\mathsf{false}} \end{array} \mid \gamma' \mid \text{residual}(bs, P) \right\rangle \\
& \hfill \text{where } \gamma' = \gamma[r \mapsto (\sigma, \chi_{\mathsf{count}}(P))] \\
\longrightarrow \quad & (\textsc{M-Let}, \text{definition of residual}) \\
& \langle r\ \mathsf{true} \mid \gamma' \mid \text{residual}(\mathsf{true} :: bs, P)\rangle \\
\longrightarrow \quad & (\textsc{M-Resume}, [\![r]\!]\gamma' = (\sigma, \chi_{\mathsf{count}}(P)) \text{ (log } |\gamma'| = 1 \text{ environment operations)}) \\
& \langle \mathbf{return}\ \mathsf{true} \mid \gamma' \mid (\sigma, \chi_{\mathsf{count}}(P)) :: \text{residual}(\mathsf{true} :: bs, P)\rangle
\end{aligned}
$$

We now use Lemma C.4 to reason about the progress of the predicate computation σ. There are two cases consider, either $1 + |bs| < n$ or $1 + |bs| = n$.

Case $1 + |bs| < n$. We obtain the following configuration.

$\longrightarrow^{\mathsf{steps}(t)(\mathsf{true}::bs)}$ (by Lemma C.4)

$\langle V\ j \mid \gamma'' \mid (\sigma', \chi_{\mathsf{count}}(P)) :: \mathsf{residual}(\mathsf{true} :: bs, P)\rangle$

where $?j = \mathsf{labs}(t)(\mathsf{true} :: bs), \gamma'' = \mathsf{env}(t)(\mathsf{true} :: bs), \sigma' = \mathsf{pure}(t)(\mathsf{true} :: bs)$

and $[\![V]\!]\gamma'' = (\mathsf{env}^{\perp}(P), \lambda\_.\mathbf{do}\ \mathsf{Branch}\ \langle\rangle)$

$=$ (definition of arrive when $1 + |bs| < n$)

$\mathsf{arrive}(\mathsf{true} :: bs, P)$

$\longrightarrow^{T(\mathsf{true}::bs,\, n)}$ (induction hypothesis)

$\mathsf{depart}(\mathsf{true} :: bs, P)$

$=$ (definition of depart when $1 + |bs| < n$)

$\langle \mathbf{return}\ i \mid \gamma \mid \mathsf{residual}(\mathsf{true} :: bs, P)\rangle$

where $i = c(\mathsf{true} :: \mathsf{true} :: bs) + c(\mathsf{false} :: \mathsf{true} :: bs)$ and $\gamma = \mathsf{env}^{\uparrow}_{\mathsf{false}}(\mathsf{true} :: bs, P)$

$=$ (definition of residual and purecont)

$\langle \mathbf{return}\ i \mid \gamma \mid [((\gamma', x_{\mathsf{true}}, \mathbf{let}\ x_{\mathsf{false}} \leftarrow r\ \mathsf{false}\ \mathbf{in}\ x_{\mathsf{true}} + x_{\mathsf{false}}) :: \mathsf{purecont}(bs, P), \chi_{id})]\rangle$

where $\gamma' = \mathsf{env}^{\downarrow}_{\mathsf{true}}(bs, P)$

$\longrightarrow$ (M-RetCont)

$\langle \mathbf{let}\ x_{\mathsf{false}} \leftarrow r\ \mathsf{false}\ \mathbf{in}\ x_{\mathsf{true}} + x_{\mathsf{false}} \mid \gamma'' \mid [(\mathsf{purecont}(bs, P), \chi_{id})]\rangle$

where $\gamma'' = \gamma'[x_{\mathsf{true}} \mapsto [\![i]\!]\gamma']$

$\longrightarrow$ (M-Let)

$\langle r\ \mathsf{false} \mid \gamma'' \mid [((\gamma'', x_{\mathsf{false}}, x_{\mathsf{true}} + x_{\mathsf{false}}) :: \mathsf{purecont}(bs, P), \chi_{id})]\rangle$

$=$ (definition of purecont and residual)

$\langle r\ \mathsf{false} \mid \gamma'' \mid \mathsf{residual}(\mathsf{false} :: bs, P)\rangle$

$\longrightarrow$ (M-Resume)

$\langle \mathbf{return}\ \mathsf{false} \mid \gamma'' \mid (\sigma, \chi_{\mathsf{count}}(P)) :: \mathsf{residual}(\mathsf{false} :: bs, P)\rangle$

where $\sigma = \mathsf{pure}(t)(bs)$

$\longrightarrow^{\mathsf{steps}(t)(\mathsf{false}::bs)}$ (by Lemma C.4 and assumption $|\mathsf{false} :: bs| < n$)

$\langle V\ j \mid \gamma \mid (\sigma, \chi_{\mathsf{count}}(P)) :: \mathsf{residual}(\mathsf{false} :: bs, P)\rangle$

where $?j = \mathsf{labs}(t)(\mathsf{false} :: bs), \sigma = \mathsf{pure}(t)(\mathsf{false} :: bs), \gamma = \mathsf{env}(t)(\mathsf{false} :: bs)$

and $[\![V]\!]\gamma = (\mathsf{env}^{\perp}(P), \lambda\_.\mathbf{do}\ \mathsf{Branch}\ \langle\rangle)$

$=$ (definition of arrive when $1 + |bs| < n$)

$\mathsf{arrive}(\mathsf{false} :: bs, P)$

$\longrightarrow^{T(\mathsf{false}::bs,\, n)}$ (induction hypothesis)

$\mathsf{depart}(\mathsf{false} :: bs, P)$

$=$ (definition of depart when $1 + |bs| < n$)

$\langle \mathbf{return}\ j \mid \gamma \mid \mathsf{residual}(\mathsf{false} :: bs, P)\rangle$

where $j = c(\mathsf{true} :: \mathsf{false} :: bs) + c(\mathsf{false} :: \mathsf{false} :: bs)$ and $\gamma = \mathsf{env}^{\uparrow}_{\mathsf{false}}(\mathsf{false} :: bs, P)$

$=$ (definition of residual and purecont)

$\langle \mathbf{return}\ j \mid \gamma \mid [((\gamma'', x_{\mathsf{false}}, x_{\mathsf{true}} + x_{\mathsf{false}}) :: \mathsf{purecont}(bs, P), \chi_{id})]\rangle$

$\longrightarrow$ (M-RetCont)

$\langle x_{\mathsf{true}} + x_{\mathsf{false}} \mid \gamma''[x_{\mathsf{false}} \mapsto [\![j]\!]\gamma''] \mid \mathsf{residual}(bs, P)\rangle$

$\longrightarrow$ (M-Plus)

$\langle \mathbf{return}\ m \mid \gamma''[x_{\mathsf{false}} \mapsto [\![j]\!]\gamma''] \mid \mathsf{residual}(bs, P)\rangle$

where

$$m = c(\mathsf{true} :: \mathsf{true} :: bs) + c(\mathsf{false} :: \mathsf{true} :: bs) + c(\mathsf{true} :: \mathsf{false} :: bs) + c(\mathsf{false} :: \mathsf{false} :: bs)$$
$$= c(\mathsf{true} :: bs) + c(\mathsf{false} :: bs) = c(bs) \le 2^{n-|bs|}$$

$=$ (definition of depart when $|bs| < n$)

$\mathsf{depart}(bs, P)$

Step analysis. The total number of machine transitions is given by

$$
\begin{aligned}
& 9 + \mathrm{steps}(t)(\mathrm{true} :: bs) + T(\mathrm{true} :: bs, n) + \mathrm{steps}(t)(\mathrm{false} :: bs) + T(\mathrm{false} :: bs, n) \\
= \quad & (\text{reorder}) \\
& 9 + T(\mathrm{true} :: bs, n) + \mathrm{steps}(t)(\mathrm{false} :: bs) + \mathrm{steps}(t)(\mathrm{true} :: bs) + \mathrm{steps}(t)(\mathrm{false} :: bs) \\
= \quad & (\text{definition of } T) \\
& 9 + 9 * (2^{n-|\mathrm{true}::bs|} - 1) + 9 * (2^{n-|\mathrm{false}::bs|} - 1) + 2^{n-|\mathrm{true}::bs|+1} + 2^{n-|\mathrm{false}::bs|+1} \\
& + \sum_{bs' \in \mathbb{B}^*}^{1 \le |bs'| \le n-|\mathrm{true}::bs|} \mathrm{steps}(t)(bs' +\!\!+ \mathrm{true} :: bs) + \sum_{bs' \in \mathbb{B}^*}^{1 \le |bs'| \le n-|\mathrm{false}::bs|} \mathrm{steps}(t)(bs' +\!\!+ \mathrm{false} :: bs) \\
& + \mathrm{steps}(t)(\mathrm{true} :: bs) + \mathrm{steps}(t)(\mathrm{false} :: bs) \\
= \quad & (\text{simplify}) \\
& 9 + 9 * (2^{n-|\mathrm{true}::bs|} - 1) + 9 * (2^{n-|\mathrm{false}::bs|} - 1) + 2^{n-|bs|+1} \\
& + \sum_{bs' \in \mathbb{B}^*}^{1 \le |bs'| \le n-|\mathrm{true}::bs|} \mathrm{steps}(t)(bs' +\!\!+ \mathrm{true} :: bs) + \sum_{bs' \in \mathbb{B}^*}^{1 \le |bs'| \le n-|\mathrm{false}::bs|} \mathrm{steps}(t)(bs' +\!\!+ \mathrm{false} :: bs) \\
& + \mathrm{steps}(t)(\mathrm{true} :: bs) + \mathrm{steps}(t)(\mathrm{false} :: bs) \\
= \quad & (\text{merge sums}) \\
& 9 + 9 * (2^{n-|\mathrm{true}::bs|} - 1) + 9 * (2^{n-|\mathrm{false}::bs|} - 1) + 2^{n-|bs|+1} \\
& + \left(\sum_{bs' \in \mathbb{B}^*}^{2 \le |bs'| \le n-|bs|} \mathrm{steps}(t)(bs' +\!\!+ bs) \right) + \mathrm{steps}(t)(\mathrm{true} :: bs) + \mathrm{steps}(t)(\mathrm{false} :: bs) \\
= \quad & (\text{rewrite binary sum}) \\
& 9 + 9 * (2^{n-|\mathrm{true}::bs|} - 1) + 9 * (2^{n-|\mathrm{false}::bs|} - 1) + 2^{n-|bs|+1} \\
& + \sum_{bs' \in \mathbb{B}^*}^{2 \le |bs'| \le n-|bs|} \mathrm{steps}(t)(bs' +\!\!+ bs) + \sum_{bs' \in \mathbb{B}^*}^{1 \le |bs'| \le 1} \mathrm{steps}(t)(bs' +\!\!+ bs) \\
= \quad & (\text{merge sums}) \\
& 9 + 9 * (2^{n-|\mathrm{true}::bs|} - 1) + 9 * (2^{n-|\mathrm{false}::bs|} - 1) + 2^{n-|bs|+1} + \sum_{bs' \in \mathbb{B}^*}^{1 \le |bs'| \le n-|bs|} \mathrm{steps}(t)(bs' +\!\!+ bs) \\
= \quad & (\text{factoring}) \\
& 9 + 2 * 9 * (2^{n-|bs|-1} - 1) + 2^{n-|bs|+1} + \sum_{bs' \in \mathbb{B}^*}^{1 \le |bs'| \le n-|bs|} \mathrm{steps}(t)(bs' +\!\!+ bs) \\
= \quad & (\text{distribute}) \\
& 9 + 9 * (2^{n-|bs|} - 2) + 2^{n-|bs|+1} + \sum_{bs' \in \mathbb{B}^*}^{1 \le |bs'| \le n-|bs|} \mathrm{steps}(t)(bs' +\!\!+ bs) \\
= \quad & (\text{distribute}) \\
& 9 + 9 * 2^{n-|bs|} - 18 + 2^{n-|bs|+1} + \sum_{bs' \in \mathbb{B}^*}^{1 \le |bs'| \le n-|bs|} \mathrm{steps}(t)(bs' +\!\!+ bs) \\
= \quad & (\text{simplify}) \\
& 9 * 2^{n-|bs|} - 9 + 2^{n-|bs|+1} + \sum_{bs' \in \mathbb{B}^*}^{1 \le |bs'| \le n-|bs|} \mathrm{steps}(t)(bs' +\!\!+ bs) \\
= \quad & (\text{factoring}) \\
& 9 * (2^{n-|bs|} - 1) + 2^{n-|bs|+1} + \sum_{bs' \in \mathbb{B}^*}^{1 \le |bs'| \le n-|bs|} \mathrm{steps}(t)(bs' +\!\!+ bs) \\
= \quad & (\text{definition of } T) \\
& T(bs, n)
\end{aligned}
$$

Case $1 + |bs| = n$. We obtain the following configuration.

$$
\begin{aligned}
&\longrightarrow^{\mathsf{steps}(t)(\mathsf{true}::bs)} \quad \text{(by Lemma C.4)}\\
&\quad \langle \mathbf{return}\ b \mid \gamma'' \mid ([], \chi_{\mathsf{count}}(P)) :: \mathsf{residual}(\mathsf{true} :: bs, P)\rangle\\
&\qquad\qquad \text{where } !b = \mathsf{labs}(t)(\mathsf{true} :: bs), \gamma'' = \mathsf{env}(t)(\mathsf{true} :: bs)\\
&= \quad \text{(definition of arrive when } 1 + |bs| = n)\\
&\quad \mathsf{arrive}(\mathsf{true} :: bs, P)\\
&\longrightarrow^{T(\mathsf{true}::bs,\, n)} \quad \text{(induction hypothesis)}\\
&\quad \mathsf{depart}(\mathsf{true} :: bs, P)\\
&= \quad \text{(definition of depart when } 1 + |bs| = n)\\
&\quad \langle \mathbf{return}\ i \mid \gamma \mid \mathsf{residual}(\mathsf{true} :: bs, P)\rangle\\
&\qquad\qquad \text{where } i = c(\mathsf{true} :: bs) \leq 2^{n-|\mathsf{true}::bs|} = 1 \text{ and } \gamma = \mathsf{env}^{\perp}(P)\\
&= \quad \text{(definition of residual and purecont)}\\
&\quad \langle \mathbf{return}\ i \mid \gamma \mid [((\gamma', x_{\mathsf{true}}, \mathbf{let}\ x_{\mathsf{false}} \leftarrow r\ \mathsf{false}\ \mathbf{in}\ x_{\mathsf{true}} + x_{\mathsf{false}}) :: \mathsf{purecont}(bs, P), \chi_{id})]\rangle\\
&\longrightarrow \quad \text{(M-RetCont)}\\
&\quad \langle \mathbf{let}\ x_{\mathsf{false}} \leftarrow r\ \mathsf{false}\ \mathbf{in}\ x_{\mathsf{true}} + x_{\mathsf{false}} \mid \gamma'[x_{\mathsf{true}} \mapsto [\![i]\!]\gamma'] \mid [(\mathsf{purecont}(bs, P), \chi_{id})]\rangle\\
&= \quad \text{(definition of } [\![-]\!] \text{ (1 value step))}\\
&\quad \langle \mathbf{let}\ x_{\mathsf{false}} \leftarrow r\ \mathsf{false}\ \mathbf{in}\ x_{\mathsf{true}} + x_{\mathsf{false}} \mid \gamma'' \mid [(\mathsf{purecont}(bs, P), \chi_{id})]\rangle\\
&\qquad\qquad \text{where } \gamma'' = \gamma'[x_{\mathsf{true}} \mapsto i]\\
&\longrightarrow \quad \text{(M-Let, definition of residual)}\\
&\quad \langle r\ \mathsf{false} \mid \gamma'' \mid \mathsf{residual}(\mathsf{false} :: bs, P)\rangle\\
&\longrightarrow \quad \text{(M-Resume)}\\
&\quad \langle \mathbf{return}\ \mathsf{false} \mid \gamma'' \mid (\sigma, \chi_{\mathsf{count}}(P)) :: \mathsf{residual}(\mathsf{false} :: bs, P)\rangle\\
&\qquad\qquad \text{where } \sigma = \mathsf{pure}(t)(bs)\\
&\longrightarrow^{\mathsf{steps}(t)(\mathsf{false}::bs)} \quad \text{(by Lemma C.4 and assumption } 1 + |bs| = n)\\
&\quad \langle \mathbf{return}\ b \mid \gamma \mid ([], \chi_{\mathsf{count}}(P)) :: \mathsf{residual}(\mathsf{false} :: bs, P)\rangle\\
&\qquad\qquad \text{where } !b = \mathsf{labs}(t)(\mathsf{false} :: bs), \gamma = \mathsf{env}(t)(\mathsf{false} :: bs)\\
&= \quad \text{(definition of arrive when } 1 + |bs| = n)\\
&\quad \mathsf{arrive}(\mathsf{false} :: bs, P)\\
&\longrightarrow^{T(\mathsf{false}::bs,\, n)} \quad \text{(induction hypothesis)}\\
&\quad \mathsf{depart}(\mathsf{false} :: bs, P)\\
&= \quad \text{(definition of depart when } 1 + |bs| = n)\\
&\quad \langle \mathbf{return}\ j \mid \gamma \mid \mathsf{residual}(\mathsf{false} :: bs, P)\rangle\\
&\qquad\qquad \text{where } j = c(\mathsf{false} :: bs) \leq 2^{n-|\mathsf{false}::bs|} = 1 \text{ and } \gamma = \mathsf{env}^{\perp}(P)\\
&= \quad \text{(definition of residual and purecont)}\\
&\quad \langle \mathbf{return}\ j \mid \gamma \mid [((\gamma', x_{\mathsf{false}}, x_{\mathsf{true}} + x_{\mathsf{false}}) :: \mathsf{purecont}(bs, P), \chi_{id})]\rangle\\
&\qquad\qquad \text{where } \gamma' = \mathsf{env}^{\downarrow}_{\mathsf{false}}(bs, P)\\
&\longrightarrow \quad \text{(M-RetCont)}\\
&\quad \langle x_{\mathsf{true}} + x_{\mathsf{false}} \mid \gamma'' \mid [(\mathsf{purecont}(bs, P), \chi_{id})]\rangle\\
&\qquad\qquad \text{where } \gamma'' = \gamma'[x_{\mathsf{false}} \mapsto [\![j]\!]\gamma'] = \gamma'[x_{\mathsf{false}} \mapsto j]\\
&\longrightarrow \quad \text{(M-Plus)}\\
&\quad \langle \mathbf{return}\ m \mid \gamma'' \mid [(\mathsf{purecont}(bs, P), \chi_{id})]\rangle\\
&\qquad\qquad \text{where } m = c(\mathsf{true} :: bs) + c(\mathsf{false} :: bs) \leq 2^{n-|bs|}\\
&= \quad \text{(definition of residual and depart when } |bs| < n)\\
&\quad \mathsf{depart}(bs, P)
\end{aligned}
$$

Step analysis. The total number of machine transitions is given by

$$
\begin{aligned}
& 9 + \mathsf{steps}(t)(\mathsf{true} :: bs) + T(\mathsf{true} :: bs, n) + \mathsf{steps}(t)(\mathsf{false} :: bs) + T(\mathsf{false} :: bs, n) \\
= \quad & \text{(reorder)} \\
& 9 + T(\mathsf{true} :: bs, n) + T(\mathsf{false} :: bs, n) + \mathsf{steps}(t)(\mathsf{true} :: bs) + \mathsf{steps}(t)(\mathsf{false} :: bs) \\
= \quad & \text{(definition of } T \text{ when } |bs| + 1 = n) \\
& 9 + 2 + 2 + \mathsf{steps}(t)(\mathsf{true} :: bs) + \mathsf{steps}(t)(\mathsf{false} :: bs) \\
= \quad & \text{(simplify)} \\
& 9 + 2^2 + \mathsf{steps}(t)(\mathsf{true} :: bs) + \mathsf{steps}(t)(\mathsf{false} :: bs) \\
= \quad & \text{(rewrite } 2 = n - |bs| + 1) \\
& 9 + 2^{n-|bs|+1} + \mathsf{steps}(t)(\mathsf{true} :: bs) + \mathsf{steps}(t)(\mathsf{false} :: bs) \\
= \quad & \text{(multiply by 1)} \\
& 9 * (2^{n-|bs|} - 1) + 2^{n-|bs|+1} + \mathsf{steps}(t)(\mathsf{true} :: bs) + \mathsf{steps}(t)(\mathsf{false} :: bs) \\
= \quad & \text{(rewrite binary sum)} \\
& 9 * (2^{n-|bs|} - 1) + 2^{n-|bs|} + \sum_{bs' \in \mathbb{B}^*}^{1 \le |bs'| \le n-|bs|} \mathsf{steps}(t)(bs' \mathbin{+\!\!+} bs) \\
= \quad & \text{(definition of } T) \\
& T(bs, n)
\end{aligned}
$$

□

The following theorem is a copy of Theorem 5.10.

Theorem C.6. *For all $n > 0$ and any n-standard predicate P it holds that*

(1) *The program* effcount *is a generic counting program*
(2) *The runtime complexity of* effcount P *is given by the following formula:*

$$\sum_{bs \in \mathbb{B}^*}^{|bs| \le n} \mathsf{steps}(\mathcal{T}(P))(bs) + O(2^n)$$

Proof. The proof begins by direct calculation.

$$
\begin{array}{ll}
& \langle \mathsf{effcount}\ P \mid \emptyset \mid [([], \chi_{id})]\rangle \\
= & \quad \text{(definition of residual)} \\
& \langle \mathsf{effcount}\ P \mid \emptyset \mid \mathsf{residual}(P, [], t, c)\rangle \\
\longrightarrow & \quad (\text{M-App}, [\![\mathsf{effcount}]\!]\emptyset = (\emptyset, \lambda pred. \cdots)) \\
& \langle \mathbf{handle}\ pred\ (\lambda\_.\mathbf{do}\ \mathsf{Branch}\ \langle\rangle)\ \mathbf{with}\ H_{\mathsf{count}} \mid \gamma \mid \mathsf{residual}(P, [])\rangle \\
\multicolumn{2}{l}{\text{where } \gamma = \mathsf{env}^{\perp}(P)} \\
\longrightarrow & \quad (\text{M-Handle}) \\
& \langle pred\ (\lambda\_.\mathbf{do}\ \mathsf{Branch}\ \langle\rangle) \mid \gamma \mid ([], (\gamma, H_{\mathsf{count}})) :: \mathsf{residual}(P, [])\rangle \\
= & \quad (\text{definition of } \chi_{\mathsf{count}}) \\
& \langle pred\ (\lambda\_.\mathbf{do}\ \mathsf{Branch}\ \langle\rangle) \mid \gamma \mid ([], \chi_{\mathsf{count}}(P)) :: \mathsf{residual}(P, [])\rangle \\
\longrightarrow^{steps(t)([])} & \quad \text{(by Lemma C.4)} \\
& \langle(\lambda\_.\mathbf{do}\ \mathsf{Branch}\ \langle\rangle)\ j \mid \gamma' \mid (\sigma, \chi_{\mathsf{count}}(P)) :: \mathsf{residual}(P, [])\rangle \\
\multicolumn{2}{l}{\text{where } \gamma' = \mathsf{env}(t)([]), \sigma = \mathsf{pure}(t)(bs) \text{ and } ?j = \mathsf{labs}(t)(bs)} \\
= & \quad \text{(definition of arrive)} \\
& \mathsf{arrive}(P, []) \\
\longrightarrow^{T([], n)} & \quad \text{(by Lemma C.5)} \\
& \mathsf{depart}(P, []) \\
= & \quad \text{(definition of depart)} \\
& \langle \mathbf{return}\ m \mid \gamma \mid \mathsf{residual}(P, [])\rangle \\
\multicolumn{2}{l}{\text{where } \gamma = \mathsf{env}^{\perp}(P) \text{ and } m = c([]) \leq 2^{n-|bs|} = 2^n} \\
= & \quad \text{(definition of residual)} \\
& \langle \mathbf{return}\ m \mid \gamma \mid [([], \chi_{id})]\rangle \\
\longrightarrow & \quad (\text{M-Handle-Ret}, H^{\mathsf{val}}_{id} = \{\mathbf{val}\ x \mapsto \mathbf{return}\ x\}) \\
& \langle \mathbf{return}\ x \mid \emptyset[x \mapsto m] \mid []\rangle
\end{array}
$$

Analysis. The machine yields the value m. By Lemma C.5 it follows that $m \leq 2^{n-|bs|} = 2^{n-|[]|} = 2^n$. Furthermore, the total number of transitions used were

$$
\begin{aligned}
& 5 + \mathsf{steps}(t)([]) + T([], n) \\
= \quad & \text{(definition of } T) \\
& 5 + \mathsf{steps}(t)([]) + 9 * 2^n + 2^{n+1} + \sum_{bs' \in \mathbb{B}^*}^{1 \leq |bs'| \leq n} \mathsf{steps}(t)(bs') \\
= \quad & \text{(simplify)} \\
& 5 + \mathsf{steps}(t)([]) + 9 * 2^n + 2^{n+1} + \sum_{bs' \in \mathbb{B}^*}^{1 \leq |bs'| \leq n} \mathsf{steps}(t)(bs') \\
= \quad & \text{(reorder)} \\
& 5 + \left(\sum_{bs' \in \mathbb{B}^*}^{1 \leq |bs'| \leq n} \mathsf{steps}(t)(bs') \right) + \mathsf{steps}(t)([]) + 9 * 2^n + 2^{n+1} \\
= \quad & \text{(rewrite as unary sum)} \\
& 5 + \left(\sum_{bs' \in \mathbb{B}^*}^{1 \leq |bs'| \leq n} \mathsf{steps}(t)(bs') + \sum_{bs' \in \mathbb{B}^*}^{0 \leq |bs'| \leq 0} \mathsf{steps}(t)(bs') \right) + 9 * 2^n + 2^{n+1} \\
= \quad & \text{(merge sums)} \\
& 5 + \left(\sum_{bs' \in \mathbb{B}^*}^{0 \leq |bs'| \leq n} \mathsf{steps}(t)(bs') \right) + 9 * 2^n + 2^{n+1} \\
= \quad & \text{(definition of } \mathcal{O}) \\
& \left(\sum_{bs' \in \mathbb{B}^*}^{0 \leq |bs'| \leq n} \mathsf{steps}(t)(bs') \right) + \mathcal{O}(2^n)
\end{aligned}
$$

□

D PROOF DETAILS FOR THE NO SHORTCUTS LEMMA

The proof of Lemma 5.11 relies on the fact that any n-standard predicate has a canonical form. Section D.1 disseminates canonical predicates, whilst Section D.2 proves Lemma 5.11.

D.1 Canonical Predicates

The decision tree model (Definition 5.2) captures the interaction between a given predicate P and its point p. The interior nodes correspond to those places where P queries p, whilst the leaves represent answers ultimately conferred from the dialogue between the predicate and its point.

The abstract nature of the decision tree model means that concrete syntactic structure of the predicate is lost. Thus we cannot hope to reconstruct a particular predicate from its model. Indeed many syntactically distinct predicates may share the same model. However, we can construct *some* predicate from a given model, namely, the *canonical predicate*. Intuitively, the canonical predicate P' of P is a predicate which exhibits the same dialogue as P for every (valid) point.

Let $\mathcal{U}(P) := bs \mapsto \mathcal{T}(P)(bs).1$ denote the procedure for constructing an *untimed decision tree* of a given predicate P.

Definition D.1 (Canonical predicate). A canonical predicate P' of an n-standard predicate P is itself an n-standard predicate whose body (syntactically) consists entirely of **let**-bindings of point applications and whose continuation is either another **let**-expression of the same form or **return** b for some boolean b. Moreover, P' exhibits the same dialogue as P, that is for all $bs \in \mathbb{B}^*$ such that

$|bs| \leq n$ that

$$\mathcal{U}(P)(bs) = \mathcal{U}(P')(bs)$$

Next we define a procedure for constructing canonical predicate of any given n-standard predicate.

Definition D.2 (Normalisation procedure for predicates). The meta-procedure norm takes as input an n-standard untimed decision tree, and outputs a program whose type is Point $\to$ Bool, which is exactly the type of predicates. The procedure makes use of an auxiliary procedure body to generate the predicate body.

$$\begin{array}{lll}
\mathsf{norm} & : & (\mathbb{B}^* \rightharpoonup \mathsf{Lab}) \to \mathsf{Val} \\
\mathsf{norm}(t) & := & \lambda p^{\mathsf{Point}}.\mathsf{body}(t, [], p) \\[1ex]
\mathsf{body} & : & (\mathbb{B}^* \rightharpoonup \mathsf{Lab}) \times \mathbb{B}^* \times \mathsf{Val} \to \mathsf{Comp} \\
\mathsf{body}(t, bs, p) & := & \begin{cases} \mathbf{return}\ b & t(bs) = !b \\ \mathbf{let}\ b \leftarrow p\ i\ \mathbf{in} & \\ \mathbf{if}\ b\ \mathbf{then}\ \mathsf{body}(t, \mathsf{true} :: bs, p) & \text{if } t(bs) = ?i \\ \mathbf{else}\ \mathsf{body}(t, \mathsf{false} :: bs, p) & \end{cases}
\end{array}$$

As convenient notation we write norm(P) to mean norm($bs \mapsto \mathcal{U}(P)(bs)$). Next we show that the meta-procedure norm produces canonical predicates.

Lemma D.3. *Suppose P is an n-standard predicate then $P' :=$ norm(P) is an n-standard predicate such that for all $bs \in \mathbb{B}^*$, $|bs| \leq n$*

$$\mathcal{U}(P)(bs) = \mathcal{U}(P)(bs')$$

Proof. By induction on n and body. □

Lemma D.4. *The procedure* norm *generates canonical predicates.*

Proof. First observe that the syntax produced by the body procedure of norm conforms with the syntactic restrictions of canonical predicates (Definition D.1). The rest follows as by Lemma D.3. □

D.2 No Shortcuts

We now have the necessary machinery to show that every n-count program in λ_b has at least exponential time complexity. The following lemma is a copy of Lemma 5.11.

Lemma D.5. *If C is an n-count program and P is an n-standard predicate, then C 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 $\mathcal{E}[P\ p]$ appearing in the small-step reduction of $C\ P$ such that p is a closed value (hence an n-point) and $\mathbb{P}[\![p]\!] = \pi$.*

Proof. Suppose C and P are as above, and suppose for contradiction that π is some semantic n-point such that no corresponding application $P\ p$ ever arises in the course of computing $C\ P$. Let t be the untimed decision tree for P. Now consider the leaf node in t corresponding to the point π, and let t' be the tree obtained from t' by simply negating the boolean value at this leaf node, that is

$$t' := bs' \mapsto \begin{cases} \neg b & \text{if } bs = bs' \\ \mathcal{U}(P)(bs') & \text{otherwise} \end{cases}$$

Then $P' =$ norm(t') constructs a canonical predicate, and as the numbers of true-leaves in t and t' differ by 1, it follows that their count at the leaf node in question differ by 1, i.e.

$$|C(P')(bs) - C(P)(bs)| = 1.$$

Taking $bs = []$, we get that the values ultimately returned by $C\ P$ and $C\ P'$ differ by 1, i.e.

$$|C(P')([]) - C(P)([])| = 1.$$

There are two cases to consider:

(1) If $C\ P = C\ P'$ then C cannot be an n-count program, because $C(P)([]) \neq C(P')([])$, which contradicts the assumption.

(2) If $C\ P \neq C\ P'$ then we have to argue that if the computation of $C\ P$ never actually 'visits' the leaf node in question, then C is unable to detect any difference between P and P'. To establish our argument we make use of a variation of Milner's *context lemma* for PCF. Specifically, we have to show the following by induction on length of reduction sequences:

Lemma D.6. *Let $\mathcal{F}[-]$ be any multi-hole context in C such that $\mathcal{F}[P] = C\ P$ and the type of $\mathcal{F}[P]$ is either* Nat *or* Bool. *If $\mathcal{F}[P] \rightsquigarrow^m$ **return** V then $\mathcal{F}[P'] \rightsquigarrow^*$ **return** V where the type of V is either* Nat *or* Bool.

Proof. Proof by induction on the length of the reduction sequence, m.

Base step We have that $m = 0$ which implies $\mathcal{F}[P] \rightsquigarrow^0$ **return** V from which it follows that $\mathcal{F}[-]$ is simply **return** V, thus it follows immediately that $\mathcal{F}[P'] \rightsquigarrow^0$ **return** V.

Induction step We have that $m = 1 + m'$. The induction hypothesis is

$$\forall \mathcal{F}.\mathcal{F}[P] \rightsquigarrow^{m'} \textbf{return}\ V \quad \text{implies} \quad \mathcal{F}[P'] \rightsquigarrow^* \textbf{return}\ V.$$

There are two cases to consider depending on whether applications of P occur in $\mathcal{F}$.

Case $\mathcal{F}[P]$ is not an application of P. By assumption there is at least one reduction step, unroll this step to obtain

$$\mathcal{F}[-] \rightsquigarrow \mathcal{F}'[-] \rightsquigarrow^{m'} \textbf{return}\ V$$

Now plug in P' and then the result follows by a single application of the induction hypothesis.

Case $\mathcal{F}[P]$ is an application of P. It must be that P is applied to values of type Point. Moreover by assumption, we know that denotation of those values are distinct from the critical point p_c. Now write $\mathcal{F}[P] = \mathcal{G}[P, P\ p[P]]$ such that the first component of $\mathcal{G}$ tracks residuals of P and the second component focuses on the expression in evaluation position, which in our particular case is an application of P to some point p in which P may occur again. We need to show that

$$\mathcal{G}[P, P\ p[P]] \rightsquigarrow \mathcal{G}[P, \textbf{return}\ W] \rightsquigarrow \textbf{return}\ V$$

for some W : Bool. Looking at the reduction sequence modulo $\mathcal{G}[P, -]$, we have that

$$P\ p[P] \rightsquigarrow^+ \mathcal{F}_0[p[P]\ i_0] \rightsquigarrow \mathcal{F}_0[\textbf{return}\ V_0] \rightsquigarrow^+ \mathcal{F}_1[p[P]\ i_1] \rightsquigarrow \cdots \rightsquigarrow^+ \textbf{return}\, W,$$

where each reduction step is justified by the untimed decision tree model of P. From this we can deduce that

$$\mathcal{G}[P, P\ p[P]] \rightsquigarrow^+ \mathcal{G}[P, \textbf{return}\ W] \rightsquigarrow^* \textbf{return}\ V$$

where the last step follows by the induction hypothesis and V : Bool. Now, we argue that the above reduction sequence is tracked by $\mathcal{G}[P', -]$. The n-standardness of P' guarantees that it contains n queries, and moreover, since the decision tree model for P' is the same as P except for at one leaf, we know that the queries appear the in same order, so by appeal to the decision tree for P' we obtain that

$$P'\ p[P'] \rightsquigarrow^+ \mathcal{F}'_0[p[P']\ i_0]$$

The term in evaluation position corresponds exactly to the first query node in the decision tree model. Now we can apply the induction hypothesis to obtain

$$\mathcal{F}_0'[p[P']\ i_0] \rightsquigarrow^* \mathcal{F}_0'[\mathbf{return}\ V_0]$$

The value V_0 is exactly the same answer to $p\ i_0$ as P obtained. Now there are two cases to consider depending on the value of n. If $n = 1$ then by the 1-standardness of P' we know that there will be no further queries, and it ultimately yields the same W as $P\ p$, because by assumption $\mathbb{P}[\![p]\!] \neq \pi$. Otherwise if $n > 1$ then there must be further queries, and in particular, those queries must occur in the same order as those of P. Thus by the n-standardness of P' we get

$$\mathcal{F}_0'[\mathbf{return}\ V_0] \rightsquigarrow^+ \mathcal{F}_1'[p[P']\ i_1]$$

Yet again we find ourselves in a position where we can again apply the induction hypothesis to obtain an answer. By repeating this argument n times, we get that $P'\ p$ eventually yields W, we can lift this back into the outer context to obtain

$$\mathcal{G}[P', P'\ p[P']] \rightsquigarrow^+ \mathcal{G}[P', \mathbf{return}\ W]$$

and by the induction hypothesis, we get that

$$\mathcal{G}[P', \mathbf{return}\ W] \rightsquigarrow^* \mathbf{return}\ V.$$

□

Recall that $C\ P \neq C\ P'$, but by the Context Lemma D.6 both $C\ P$ and $C\ P'$ reduce to the same value which contradicts the initial assumption.

□