Asymptotic Improvement with Effect Handlers

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

Abstract

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 effect handlers. 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.

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 [13] 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., [23, Section 2.6]), it is broadly taken as read that such models are equally applicable whatever programming language we are working with, and moreover that all respectable languages can represent all algorithms of interest; thus, one does not expect the best achievable asymptotic run-time for a typical algorithm (say in number theory or graph theory) to be sensitive to the choice of programming language, except perhaps in marginal cases.

The situation changes radically, however, if we consider *higher-order* operations: programmable operations whose inputs may themselves be programmable operations. Here it turns out that both what is computable and the efficiency with which it can be computed can be highly sensitive to the selection of language features present. This is in fact true more widely for *abstract data types*, of which higher-order types can be seen as a special case: a higher-order value will 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 of the 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 [35]. 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 [9]. Such operations are 'non-functional' in the sense that their output is not determined solely by the extension of their input ; however, there are also programs with 'functional' behaviour that can be implemented with control or local state but not without them [28]. 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 [30], and there are natural operations computable by (low-order) recursion but not by (even high-order) iteration [29]. 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 [31].

Relatively few results of this character have so far been established on the complexity side. Pippenger [33] 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. [8] who showed that the same speedup can be achieved in a pure language by using lazy evaluation. Jones [20] explores the approach of manifesting expressivity and efficiency differences between certain languages by artificially restricting attention to 'cons-free' programs; in this setting, the classes of representable first-order functions for the various languages are found to coincide with some well-known complexity classes.

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

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

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 [5] achieves the effect of 'pruned search' where the predicate admits this (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.

In order to make this efficiency difference stand out as clearly as possible, we have resorted to some slightly artificial constraints in the way we set up our scenario. Of course, even one illustration of this phenomenon suffices in principle to establish the existence of the efficiency gap in question; however, it will also be clear from our analysis that, in spite of the technical restrictions, the phenomenon we are exhibiting is actually quite general. For instance, if the problem of counting all solutions to P is replaced by that of returning the first solution found (if one exists), it will be clear that an order n speedup can still typically be expected.

The idea behind the speedup is easily explained. Suppose for example $n = 4$, and suppose that the predicate P always inspects the components of its argument in the order $0, 1, 2, 3$. A naive implementation of count_4 might start by applying the given P to $p_0 = (\mathsf{true}, \mathsf{true}, \mathsf{true}, \mathsf{true})$, and then to $p_1 = (\mathsf{true}, \mathsf{true}, \mathsf{true}, \mathsf{false})$. 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-use 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* $(\mathsf{Nat} \to \mathsf{Bool}) \to \mathsf{Bool}$.

The above idea will already be familiar, at least informally, to many who have worked with effect handlers or explicit continuations, and was explicitly presented in a closely related context by Bauer [1]; but our contribution here is to formulate and prove a precise mathematical theorem that pins down the efficiency difference in question. A general mathematical theory of the expressive power of effect handlers would be perhaps best articulated within the framework of game semantics; since in the present paper our focus is on one specific example of the difference, we shall work concretely and operationally with the languages themselves. In the first instance, we formulate our results as a comparison between a purely functional base language (a version of call-by-value PCF [35]) and an extension of this with effect handlers; we then easily observe that our results are unaffected if the base language is augmented with other features such as local mutable store. As regards the runtime estimates, we work with a CEK-style abstract machine model for our languages which, we claim, offers a realistic model of program execution time for typical real-world implementations.

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.
- Section 5 formally states and proves the complexity of generic search in λ_b ($\Omega(n2^n)$) and λ_h ($O(2^n)$).
- Section 6 outlines how the result scales to extensions of the base language with features such as state.
- Section 7 empirically 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 variants of previous work — 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 [36, 37]. Subsequently they have emerged as a practical programming abstraction for modular effectful programming [3, 11, 18, 21, 22, 24, 27]. 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 [38], and as an introduction to the mathematical foundations of handlers, we refer the reader to the founding paper by Plotkin and Pretnar [37] and the excellent tutorial paper by Bauer [2].

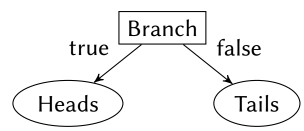
Viewed through the lens of universal algebra, an algebraic effect is given by a signature Σ of finitary *operation symbols* defined over some nonempty carrier set A, along with an equational theory that describes the properties of the operations [36]. An example of an algebraic effect is *nondeterminism*, 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 an introductory programmatic example, we will showcase a use of the operation Branch by modelling a coin toss. Suppose we have a data type $\mathsf{Toss} := \mathsf{Heads} \mid \mathsf{Tails}$, then in our programming notation (introduced formally in Section 3.2) we may implement a coin toss as follows.

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

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 (with the appropriate injections). Systems such as Frank [27], Helium [7], Koka [24], and Links [18] include type-and-effect systems which track the use of effectful operations. Whilst current iterations of systems such as Eff [3] and Multicore OCaml [11] elect not to include an effect system. Our language is closer to the latter two.

We may, in the style of Lindley [26], view an effectful computation as a tree, where the interior nodes correspond to operation invocations and the leaves correspond to return values. The computation tree for toss is as follows.



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

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

The **handle**-construct generalises the exceptional syntax of Benton and Kennedy [4]. A handler has a *success* clause and an *operation* clause. The success clause determines how to interpret the return value of toss. 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 [**inl** 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 [**inr** Tails], which is concatenated with the result of the true branch; hence the handler ultimately returns [**inl** Heads, **inr** 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 [25] variation of PCF [35]. Fine-grain call-by-value is similar to A-normal form [16] in that every intermediate computation is named, but unlike A-normal form is closed under reduction.

The types of λ_b are given by the following grammar.

$$A, B, C, D ::= \mathsf{Nat} \mid 1 \mid A \to B \mid A \times B \mid A + B$$

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.

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.

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

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

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 assume an efficient representation of naturals (e.g. naturals occupy a machine word) such that constants ($c \in \{+, -, =\}$) have efficient realisations $\ulcorner c \urcorner$. 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.

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 [16]. For example, assuming f, g, h are functions, and a is a bound variable, then the expression $(f\ (h\ a) + g\ \langle\rangle)$ is syntactic sugar for:

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

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 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.

We give a small-step operational semantics for $\lambda_b$with *evaluation contexts* in the style of Felleisen [12].

$$\begin{array}{rcl}
(\lambda x^A.\, M)V & \rightsquigarrow & M[V/x] \\
(\mathbf{rec}\ f^A\ x.\, M)V & \rightsquigarrow & M[(\mathbf{rec}\ f^A\ x.\, M)/f, V/x] \\
c\ V & \rightsquigarrow & \mathbf{return}\ (\ulcorner c \urcorner (V)) \\
\mathbf{let}\ \langle x; y \rangle = \langle V; W \rangle\ \mathbf{in}\ N & \rightsquigarrow & N[V/x, W/y] \\
\begin{array}{r}\mathbf{case}\ (\mathbf{inl}\ V)^B\ \{\mathbf{inl}\ x \mapsto M; \\ \mathbf{inr}\ y \mapsto N\}\end{array} & \rightsquigarrow & M[V/x] \\
\begin{array}{r}\mathbf{case}\ (\mathbf{inr}\ V)^A\ \{\mathbf{inl}\ x \mapsto M; \\ \mathbf{inr}\ y \mapsto N\}\end{array} & \rightsquigarrow & N[V/y] \\
\mathbf{let}\ x \leftarrow \mathbf{return}\ V\ \mathbf{in}\ N & \rightsquigarrow & N[V/x]
\end{array}$$

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

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

We write $M[V/x]$ for M with V substituted for x. The reduction relation $\rightsquigarrow$ is defined on computation terms. The statement $M \rightsquigarrow N$ reads: term M reduces to term N in one step. We write R^+ for the transitive closure of relation R.

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.

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

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 [38], we assume a global signature for every program.

The typing rules for λ_h are those of λ_b (Figure 1) plus three additional rules for operations, handling, and handlers given in Figure 2. The T-Op rule ensures that an operation invocation is only well-typed if the operation ℓ appears in the

Values

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

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

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

Computations

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

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

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

Figure 1. Typing Rules for λ_{b}

Computations

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

Handlers

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

Figure 2. Additional Typing Rules for λ_{h}

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_{ℓ} (A_{ℓ}) and resumption r_{ℓ} ($B_{\ell} \to D$) in the operation clause H^{ℓ} is determined by the signature for ℓ. The return type of r_{ℓ} is D, as the body of the resumption will itself be handled by H. We write H^{ℓ} and H^{val} for projecting success and operation clauses.

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

We extend the operational semantics to λ_{h}.

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

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

$$\begin{array}{lrl} \text{Handler contexts} & \mathcal{H} ::= & [\,] \mid \mathbf{handle}\ \mathcal{H}\ \mathbf{with}\ H \\ & \mid & \mathbf{let}\ x \leftarrow \mathcal{H}\ \mathbf{in}\ N \end{array}$$

The first rule invokes the success clause. The second rule handles an operation via the corresponding operation clause. The third rule replaces the corresponding lifting rule for λ_{b}. Rather than augmenting evaluation contexts from λ_{b}, we introduce handler contexts. The separation between pure evaluation contexts $\mathcal{E}$ and handler contexts $\mathcal{H}$ guarantees the second rule is deterministic, as otherwise it could pick an arbitrary handler in scope. With this separation, the second rule always picks 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 [14] for λ_{b}based on that of Hillerström and Lindley [18]. 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.

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

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

The abstract machine semantics is given in Figure 3. The transition function is given by the $\longrightarrow$ relation, which also depends on an interpretation function $[\![-]\!]$ for value terms and machine values. The machine is initialised by placing a term in a configuration alongside the empty environment ($\emptyset$) and identity 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 continuation with let bindings. The (M-RetCont) rule binds a returned value if there is a pure continuation in the current continuation frame. 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. We now make the correspondence between operational semantics and abstract machine more precise.

Correctness The abstract machine faithfully simulates the operational semantics; most transitions correspond directly to β-reductions, but the M-Let-rule performs an administrative step to bring the computation M into evaluation position. We define an extension of the transition function $\longrightarrow$ to capture administrative steps.

Definition 4.1 (Auxiliary reduction relations)**.** We write $\longrightarrow_a$ for administrative steps, $\longrightarrow_\beta$ for β-steps, and $\Longrightarrow$ for a sequence of steps of the form $\longrightarrow_a^* \longrightarrow_\beta$.

Theorem 4.2. *There is a one-to-one mapping between reduction relation $\leadsto$ and transition function $\Longrightarrow$.*

The proof is by induction on $M \leadsto N$, relying on an inverse map $(\!|-|\!)$ from configurations to terms [18].

4.2 Handler Machine

We now enrich the λ_b machine to a λ_h machine. To support handlers 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 as the continuation component is now occupied by a generalised continuation $\kappa \in \mathsf{Cont}$ [18], meaning a machine continuation is now a list of pairs containing a pure continuation (as in the base machine) and a handler closure (χ). A handler closure consists of an environment and a handler definition, where the former binds the free variables that occur in the latter. The identity continuation is an empty pure continuation paired with the identity handler closure, i.e. $\kappa_0 := [([], (\emptyset, \{\mathbf{val}\ x \mapsto x\}))]$. The machine values are augmented to contain 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 4 depicts the new and modified rules. The rule (M-Handle) pushes a the handler closure along with an empty pure continuation onto the continuation stack. The (M-Let) and (M-RetCont) are dual rules, as the former grows the pure continuation of the topmost continuation frame, and the latter shrinks the pure continuation. If the pure continuation is empty, then the (M-RetHandler) rule applies, which transfers control to the success clause of the current handler. If an operation is invoked, then the (M-Handle-Op) rule transfers control to the corresponding 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 Theorem 4.2 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.

4.3 Realisability and Asymptotic Complexity

The machine structures are readily realisable using standard persistent functional data structures. The pure and generalised continuations can be implemented using lists, which makes their augmentation operation (_ :: _) have time complexity $O(1)$. This also holds true for pure continuations on the handler machine as augmenting the current pure continuation only requires reaching under the topmost handler closure. Environments, γ, can be realised using a map, making the complexity of extension and lookup be $O(\log |\gamma|)$ [32].

Transition function

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

Value interpretation

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

Figure 3. Abstract Machine Semantics for λ_{b}

Transition function

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

Figure 4. Abstract Machine Semantics for λ_{h}

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

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

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}. Since addition of effect handlers is the only difference between the two languages, we obtain as a corollary that a PCF-like programming language with effect handlers exhibits fundamentally more efficient programs than a pure PCF programming language. To obtain this result it suffices to find *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*.

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. A variety of problems can be cast as instances of generic search; classic examples include solving Sudoku puzzles and n-Queens, whilst more esoteric examples include problems from game theory, graph theory, and exact real number integration [10, 39].

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 $\mathbb{N}_n := \{0, \ldots, n-1\}$. We often omit n indexes when clear from context; in particular they do 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 higher-order function which acts on points

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

where n is a natural number and a point is a first-order function taking bounded natural numbers to boolean values:

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

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

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

where $\bot := \textbf{rec } f\ i.f\ i$ 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.

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

Similarly we can define a variation, tt_2, which inspects two components of its point, but nevertheless 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

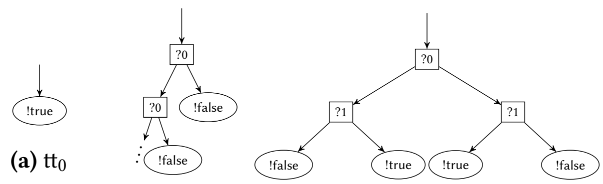
$$\mathsf{div}_1 := \textbf{rec } \mathit{div}\, p.\textbf{if } p\,0 \textbf{ then } \mathit{div}\, p \textbf{ else } \mathsf{false}$$

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

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

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

Predicate Models In essence a predicate is a decision procedure, which participates in a 'dialogue' with a supplied point $p : \mathsf{Point}_n$. 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 the left subtree corresponds to $p\ i$ being true and the right subtree 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 bit of computation that occurs in between the application of a predicate to p and the first query or answer.



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

Figure 5. Example Decision Tree Models

Figure 5 depicts models of some of the example predicates given above. The model of tt_0 is simply an unrooted leaf (Figure 5a). The model of div_1 is an infinite left-branching tree (Figure 5b). The model of odd_2 is a complete binary tree (Figure 5c). A further example is the unconditionally divergent predicate $\mathsf{div} := \textbf{rec } f\ p.f\ p$ whose model is empty.

Restrictions 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_0 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 the conditions are tt_2 and odd_n. Predicates satisfying 1 and 2 are exactly those whose models form complete binary trees (as in Figure 5c), 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 non-deterministic computation that we gave in Section 2. The main idea is to implement points as non-deterministic 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 $\Sigma := \{\mathsf{Branch} : 1 \to \mathsf{Bool}\}$. The algorithm is then expressed as follows.

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

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 $(\lambda\_.\textbf{do}\ \mathsf{Branch}\ \langle\rangle)$ 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. We formalise the concepts using the operational semantics and abstract machine for the base language λ_{b}. The reason being that it makes little sense to compare the runtime complexity of predicates which makes use effectful operations as they cannot be run on the base machine.

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

$$
\begin{aligned}
\mathcal{T}(\langle \textbf{return}\ \mathsf{true} \mid \gamma \mid []\rangle)\,[] &= (!\mathsf{true}, 0) \\
\mathcal{T}(\langle \textbf{return}\ \mathsf{false} \mid \gamma \mid []\rangle)\,[] &= (!\mathsf{false}, 0) \\
\mathcal{T}(\langle p\ V \mid \gamma \mid \sigma\rangle)\,[] &= (?[\![V]\!]\gamma, 0)
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{T}(\langle p\ V \mid \gamma \mid \sigma\rangle)\,(b :: bs) &\simeq \mathcal{T}(\langle \textbf{return}\ b \mid \gamma \mid \sigma\rangle)\, bs \\
\mathcal{T}(\langle M \mid \gamma \mid \sigma\rangle)\, bs &\simeq \mathcal{I}(\mathcal{T}(\langle M' \mid \gamma' \mid \sigma'\rangle)\, bs), \\
&\quad \text{if } \langle M \mid \gamma \mid \sigma\rangle \longrightarrow \langle M' \mid \gamma' \mid \sigma'\rangle
\end{aligned}
$$

where $\mathcal{I}(\ell, s) = (\ell, s{+}1)$ 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, i.e. $\mathcal{T}(M) := \mathcal{T}(\langle M, \emptyset[p \mapsto p], \kappa_0\rangle)$. The decision tree of a predicate P is $\mathcal{T}(P\,p)$. Since p is a parametric variable, we shall omit p and simply write $\mathcal{T}(P)$ to mean $\mathcal{T}(P\,p)$.

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 $b \in \mathbb{B}$.
- 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 $n > 0$).

Notation. We write $(\!|-|\!) : \mathbb{N} \to \mathsf{Nat}$ for the injection of natural numbers into value terms and $\mathbb{N}[\![-]\!] : \mathsf{Nat} \to \mathbb{N}$ for its inverse. Similarly, we write $\mathbb{B}[\![-]\!] : \mathsf{Bool} \to \mathbb{B}$ for the denotation function for boolean value terms.

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

$$\forall i \in \mathbb{N}_n . p\,(\!|i|\!) \leadsto^* \mathbf{return}\ W.$$

Semantically, we can think of any n-point as a total first-order function of type $\mathbb{N}_n \to \mathbb{B}$. In fact, we shall take this function type to be the model of n-points. Since an n-point terminates on its defined domain, we can easily compute a model of it using the operational semantics. Definition 5.7 provides a procedure for computing the model of any n-point.

Definition 5.7. The denotation of an n-point p is the realisation of its operational behaviour

$$\begin{aligned} \mathbb{P}[\![-]\!] &: (\mathsf{Nat}_n \to \mathsf{Bool}) \to (\mathbb{N}_n \to \mathbb{B}) \\ \mathbb{P}[\![p]\!] &:= j \in \mathbb{N}_n \mapsto \mathbb{B}[\![p(\!|j|\!)]\!] \end{aligned}$$

Definition 5.8. Any two n-points p_0 and p_1 are *distinct* if their denotations differ, i.e. $\exists j \in \mathbb{N}_n . \mathbb{P}[\![p_0]\!]\ j \neq \mathbb{P}[\![p_1]\!]\ j$.

5.4 Specification of Generic Counting

We now formally define generic counting.

Definition 5.9. 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 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 5.10. The counting function for a configuration is defined by the following equations.

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

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 5.11 (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

$$C\ P \leadsto^+ \mathbf{return}\ V, \text{ such that } [\![V]\!] = C(P)([])$$

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 B.

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

1. *The program* effcount *is a generic counting program:*

   $\mathsf{effcount}\ P \leadsto^+ \mathbf{return}\ V$ *such that* $\mathbb{N}[\![V]\!] = C(P)([]) \leq 2^n$
2. *The runtime complexity of* effcount P *is given by:*

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

5.6 Pure Generic Counting

We have shown that there exists an implementation of count in λ_{h} with a particular runtime bound. We now show that no implementation of count in λ_{b} can match this bound. To do so we exhibit two properties of the decision model:

1. there are no shortcuts, i.e. every leaf must be visited;
2. there can be no sharing of work amongst branches.

Together these two properties imply that every count program has least time complexity $\Omega(n2^n)$, because it must construct 2^n points, one for each leaf in the model, and apply the predicate once to each point. Due to the lack of sharing, each application of the predicate performs some redundant work as the path to two neighbouring leaves share n edges in the model. We formalise the first property in Section 5.7 and the second in Section 5.8. First, we give an example of a pure generic count program and discuss better alternatives.

The following is a direct implementation of count in λ_{b}.

$$\begin{array}{l} \mathsf{purecount}_n : ((\mathsf{Nat}_n \to \mathsf{Bool}) \to \mathsf{Bool}) \to \mathsf{Nat} \\ \mathsf{purecount}_n := \lambda P.\mathsf{count}'\ n \perp \\ \quad \mathbf{where}\ \ \begin{array}[t]{ll} \mathsf{count}'\ 0 & p := \mathbf{if}\ P\ p\ \mathbf{then}\ 1\ \mathbf{else}\ 0 \\ \mathsf{count}'\ (1 + n')\ p := & \\ \quad\quad \mathsf{count}'\ n'\ (\lambda i.\mathbf{if}\ i = n'\ \mathbf{then}\ \mathsf{true}\ \mathbf{else}\ p\ i) & \\ \quad + \mathsf{count}'\ n'\ (\lambda i.\mathbf{if}\ i = n'\ \mathbf{then}\ \mathsf{false}\ \mathbf{else}\ p\ i) & \\ \perp\ \_ & := \mathbf{rec}\ f\ i.f\ i \end{array} \end{array}$$

The implementation materialises 2^n points which are encoded using a standard functional linked list representation. The auxiliary function count′ exhibits a recursion pattern reminiscent of the classic recursive definition of the Fibonacci function. The function is initially applied to the divergent function $\perp$, which is partly used to seed the list generation, but also used to respond to queries $i \geq n$ such that they diverge. The base case of count′ applies the predicate P to the generated point.

At this point the reader may wonder why we cannot simply use known continuation passing style (CPS) [19] or monadic [24] transforms of effect handlers, or implement an interpreter for effect handlers [18] in λ_{b} to achieve the sharing of computation. Such global implementation techniques are ruled out in our setting, because they would change the type of count. For example, as any predicate P is a higher-order function (supplied externally to count), we cannot even CPS transform count locally as the interface of P would be incompatible with the CPS interface. Many such transforms are possible in a first-order setting, but they are not an option for us due to the inherent higher-order nature of our setting.

5.7 No Shortcuts

We sketch the idea behind the proof of the fact that any n-count program in λ_{b} must construct 2^n points. Full details are in Appendix C.2. The proof makes use of the observation that the decision tree model encodes the canonical structure of any n-standard predicate. We can reify a (semantic) n-standard model as a (syntactic) n-standard predicate. This procedure provides a means for converting any n-standard predicate P into a canonical form (first compute the model, then reify, a la normalisation by evaluation [6]).

Lemma 5.13. *Suppose P is an n-standard predicate and C is an n-count program, then C applies P to at least 2^n distinct n-points.*

The lemma guarantees that any n-count program constructs an n-point corresponding to each leaf in the model of any given n-standard predicate.

5.8 No Sharing

We now show that distinct predicate applications cannot share computation. In order to do so, we introduce the notion of *threads*. Intuitively, a thread corresponds to a path in a decision tree model. Each thread of an n-standard model is composed of $n + 1$ *sections*, where each corresponds to an edge in the model. Thus we can identify the start and end of any section by looking for point queries and responses. Definition 5.15 makes use of an auxiliary reduction relation $\twoheadrightarrow$ to define threads and sections. Intuitively, this reduction relation ensures that we cannot inadvertently "step over" an application of a point.

Definition 5.14 ($\twoheadrightarrow$). $\mathcal{E}[M] \twoheadrightarrow \mathcal{E}[N]$ iff $\mathcal{E}[M] \rightsquigarrow \mathcal{E}[N]$ and M is not of the form $p\ V$ where p is a point.

Definition 5.15 (Sections and threads). A section is a pair of computations, where the first component marks the start of the section and the second marks the end. For an n-standard predicate P, a thread of P consists of $n + 1$ sections. Given a denotation, f, of a concrete n-point, and taking p to be a distinguished free variable, a single thread for P can be computed as follows

$$
\begin{array}{l}
\mathsf{Th} : \mathsf{Comp} \times (\mathbb{N}_n \to \mathbb{B}) \rightharpoonup [(\mathsf{Comp}, \mathsf{Comp})] \\
\mathsf{Th}(P\ p, f) := (P\ p, \mathcal{E}[p\ V]) :: \mathsf{Th}(\mathcal{E}[\mathbf{return}\ (\!|b|\!)], f), \\
\qquad \text{where } \mathsf{Sec}(P\ p) = \mathcal{E}[p\ V] \text{ and } b = f(\mathbb{N}[\![V]\!]) \\
\mathsf{Th}(\mathcal{E}[\mathbf{return}\ W], f) := \\
\quad (\mathcal{E}[\mathbf{return}\ W], \mathcal{E}'[p\ V]) :: \mathsf{Th}(\mathcal{E}[\mathbf{return}\ (\!|b|\!)], f) \\
\qquad \text{where } \mathsf{Sec}(\mathcal{E}[\mathbf{return}\ W]) = \mathcal{E}'[p\ V] \text{ and } b = f(\mathbb{N}[\![V]\!]) \\
\mathsf{Th}(\mathcal{E}[\mathbf{return}\ W], f) := (\mathcal{E}[\mathbf{return}\ W], \mathbf{return}\ V) :: [] \\
\quad \text{where } \mathsf{Sec}(\mathcal{E}[\mathbf{return}\ W]) = \mathbf{return}\ V
\end{array}
$$

The auxiliary procedure Sec computes the end of a section from the start.

$$
\mathsf{Sec}(\mathcal{E}[M]) := \begin{cases} \mathcal{E}'[p\ V] & \text{if } \mathcal{E}[M] \twoheadrightarrow^{+} \mathcal{E}'[p\ V] \\ \mathbf{return}\ V & \text{if } \mathcal{E}[M] \twoheadrightarrow^{*} \mathbf{return}\ V \end{cases}
$$

Now we show that every predicate application gives rise to a corresponding thread via $\mathsf{Th}(-,-)$.

Lemma 5.16. *Suppose P is an n-standard predicate, p is an n-point, and $f = \mathbb{P}[\![p]\!]$, then*

$$
\begin{array}{c}
P\ p \twoheadrightarrow^{+} \mathcal{E}_1[p\ V_1] \rightsquigarrow^{+} \mathcal{E}_1[\mathbf{return}\ (\!|f\ (\mathbb{N}[\![V_1]\!])|\!)] \twoheadrightarrow^{+} \cdots \\
\twoheadrightarrow^{+} \mathcal{E}_n[p\ V_n] \rightsquigarrow^{+} \mathcal{E}_n[\mathbf{return}\ (\!|f\ (\mathbb{N}[\![V_n]\!])|\!)] \twoheadrightarrow^{*} \mathbf{return}\ W
\end{array}
$$

if and only if

$$
\mathsf{Th}(P\ p, f) = \left[\begin{array}{l}
(P\ p, \mathcal{E}_1[p\ V_1]), \\
(\mathcal{E}_1[\mathbf{return}\ (\!|f\ (\mathbb{N}[\![V_1]\!])|\!)], \mathcal{E}_2[p\ V_2]), \\
\vdots \\
(\mathcal{E}_n[p\ V_n], \mathcal{E}_n[\mathbf{return}\ (\!|f\ (\mathbb{N}[\![V_n]\!])|\!)]), \\
(\mathcal{E}_n[\mathbf{return}\ (\!|f\ (\mathbb{N}[\![V_n]\!])|\!)], \mathbf{return}\ W)
\end{array}\right]
$$

The lemma tells us that every predicate application has an associated thread and vice versa. By Lemma 5.13 we know that any n-count program must construct at least 2^n distinct threads. To establish the desired result, we need some way of characterising disjointness of threads.

Definition 5.17. Let C denote an n-count program and P an n-standard predicate. Any two threads T_0 and T_1 arising from distinct applications of P in C are said to be disjoint if every section computation of T_0 is distinct from every section computation of T_1, where the section computations of a thread comprise the set of all start and end computations of the sections in that thread.

Now we may conclude that no two distinct predicate applications can share computation, or in other words every section of their associated threads must be evaluated.

Lemma 5.18. *Suppose P is an n-standard predicate and C is an n-count program, and let p_0 and p_1 be distinct n-points, then the predicate applications $P\ p_0$ and $P\ p_1$ within C have disjoint threads.*

	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. Speedup of the Effectful Implementation

5.9 Complexity of Pure Generic Counting

Now we can plug together the formal machinery developed in the previous sections to state and prove the complexity result for pure generic counting programs.

Theorem 5.19. *For all $n > 0$ and every n-count program* count $\in \lambda_b$, *and n-standard predicate the runtime of* count P *is at least*

$$\sum_{bs \in \mathbb{B}^*}^{|bs| \leq n} 2^{n-|bs|}\mathrm{steps}(t)(bs) + \Omega(n2^n)$$

where $t = \mathcal{T}(P)$.

Proof. By composing Lemmas 5.13 and 5.18 we obtain the result in terms of the operational semantics. To lift this result to the abstract machine semantics we apply Theorem 4.2. □

6 Robustness

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

Beyond n-standard Predicates The n-standard restriction ensures that the models of predicates are complete binary trees. It is possible to relax the restriction at the expense of a more complicated analysis. The restriction serves to make the result as crisp as possible. Nevertheless, we may augment the effcount program with internal state to keep track of the queries raised by predicates and to remember answers of previous queries. State is definable in λ_h using a standard technique known as parameter-passing [38].

Mutable State Mutable state is a staple ingredient of many practical programming languages. To support mutable state, the base language is endowed with a heap (giving rise to a CESK machine [12], where S means *store*). Modulo heap bookkeeping, the analysis is the same.

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 variation of the purecount program;
- Pruned: a generic search procedure with space pruning based on Longley's technique [28] (uses local state);
- Berger: a lazy pure functional generic search procedure based on Berger's algorithm [5].

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 much slower than the bespoke implementation.

Exact Real Integration The integration benchmarks are adapted from Simpson [39]. 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.

The result extends to other control operators by appeal to existing results on interdefinability of handlers and other control operators [17, 34]. 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.

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

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

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

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

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

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

[6] Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. 1998. Normalisation by Evaluation. In *Prospects for Hardware Foundations (Lecture Notes in Computer Science)*, Vol. 1546. Springer, 117–137.

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

[8] 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.

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

[10] 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

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

[12] 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.

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

[14] 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.

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

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

[17] Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. 2017. On the expressive power of user-defined effects: effect handlers, monadic reflection, delimited control. *PACMPL* 1, ICFP (2017), 13:1–13:29.

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

[19] 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.

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

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

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

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

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

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

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

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

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

[29] John Longley. 2018. Bar recursion is not computable via iteration. To appear in Computability. Available at arxiv.org/abs/1804.07277. (2018).

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

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

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

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

[34] 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.

[35] Gordon Plotkin. 1997. LCF considered as a programming language. *Theor. Comput. Sci.* 5, 3 (1997), 223–255.

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

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

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

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

Configurations

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

Continuations

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

Computation terms

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

Value terms and values

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

Figure 6. Mapping from Base Machine Configurations to Terms

A Proof Details for Correctness of the Base Abstract Machine

Correctness We now show that the 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 6 describes such a mapping $(\!|-|\!)$ from configurations to terms via a collection of mutually recursive functions defined on configurations, continuations, computation terms, value terms, and machine values. The mapping makes use of two operations on environments, 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 $(\!|-|\!)$ function enables us to classify the abstract machine reduction rules according to how they relate to the operational semantics. The rule (M-Let) is administrative in the sense that $(\!|-|\!)$ is invariant under this rule. This leaves the 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 $\longrightarrow_a$ for administrative steps, $\longrightarrow_\beta$ for β-steps, and $\Longrightarrow$ for a sequence of steps of the form $\longrightarrow_a^* \longrightarrow_\beta$.

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 $(\!|C|\!) = M$, then if $M \leadsto N$ there exists C' such that $C \Longrightarrow C'$ and $(\!|C'|\!) = N$, or if $M \not\leadsto$ then $C \not\Longrightarrow$.*

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

The correspondence here is rather strong: there is a one-to-one mapping between $\leadsto$ and $\Longrightarrow$. 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 (Simulation)**.** *If $\vdash M : A$ and $M \leadsto^+ N$ such that N is normal, then $\langle M \mid \emptyset \mid []\rangle \longrightarrow^+ C$ such that $(\!|C|\!) = N$, or $M \not\leadsto$ then $\langle M \mid \emptyset \mid []\rangle \not\longrightarrow$.*

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

B Proof Details for the Complexity of Effectful Generic Counting

In this appendix we give proof details and artefacts for Theorem 5.12. Throughout this section we let H_{count} denote the handler definition of count, that is

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

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

$$\begin{aligned} \mathcal{D} &: \mathsf{Conf} \rightharpoonup \mathsf{DT} \\ \mathcal{D}(\langle \mathbf{return}\ \mathsf{true} \mid \gamma \mid [] \rangle)\, [] &= ((!\mathsf{true}, 0), (\gamma, [])) \\ \mathcal{D}(\langle \mathbf{return}\ \mathsf{false} \mid \gamma \mid [] \rangle)\, [] &= ((!\mathsf{false}, 0), (\gamma, [])) \\ \mathcal{D}(\langle p\, V \mid \gamma \mid \sigma \rangle)\, [] &= ((?[\![V]\!]\gamma, 0), (\gamma, \sigma)) \end{aligned}$$

$$\begin{aligned} \mathcal{D}(\langle p\, V \mid \gamma \mid \sigma \rangle)\, (b :: bs) &\simeq \mathcal{D}(\langle \mathbf{return}\ b \mid \gamma \mid \sigma \rangle)\, bs \\ \mathcal{D}(\langle M \mid \gamma \mid \sigma \rangle)\, bs &\simeq \mathcal{I}(\mathcal{D}(\langle M' \mid \gamma' \mid \sigma' \rangle)\, bs), \\ &\quad \text{if } \langle M \mid \gamma \mid \sigma \rangle \longrightarrow \langle M' \mid \gamma' \mid \sigma' \rangle \end{aligned}$$

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 $\chi_{\mathsf{count}}(V) := (\emptyset[\mathit{pred} \mapsto [\![V]\!]\emptyset], H_{\mathsf{count}})$ 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 B.3 (computing machine configurations)**.** For any given n-standard predicate P and a list of booleans bs, such that $|bs| \le n$, 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.

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

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

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

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.

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

- The function purecont computes the pure continuation.

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

- The function $\mathrm{env}^{\perp}$ computes the initial environment of the handler. The family of functions $\mathrm{env}^{\downarrow}_{b\in\mathbb{B}}$ 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 $\mathrm{env}^{\uparrow}_{b\in\mathbb{B}}$ describe how to compute the environment after *ascending* from the resumptive exploration of a branch.

$$\begin{aligned}\mathrm{env}^{\perp} &: \mathrm{Val} \to \mathrm{Env}\\ \mathrm{env}^{\perp}(P) &:= \emptyset[pred \mapsto [\![P]\!]\emptyset]\\[1ex] \mathrm{env}^{\downarrow}_{\mathrm{true}} &: \mathbb{B}^* \times \mathrm{Val} \rightharpoonup \mathrm{Env}\\ \mathrm{env}^{\downarrow}_{\mathrm{true}}(bs, P) &:= \mathrm{env}^{\perp}(V)[r \mapsto (\sigma, \chi_{\mathrm{count}}(P))],\\ &\qquad \text{where } \sigma = \mathrm{pure}(t)(bs)\\[1ex] \mathrm{env}^{\downarrow}_{\mathrm{false}} &: \mathbb{B}^* \times \mathrm{Val} \rightharpoonup \mathrm{Env}\\ \mathrm{env}^{\downarrow}_{\mathrm{false}}(bs, P) &:= \gamma[x \mapsto i],\\ &\text{where } \gamma = \mathrm{env}^{\downarrow}_{\mathrm{true}}(bs, P) \text{ and } i = c(\mathrm{true} :: bs)\\[1ex] \mathrm{env}^{\uparrow}_{\mathrm{false}} &: \mathbb{B}^* \times \mathrm{Val} \rightharpoonup \mathrm{Env}\\ \mathrm{env}^{\uparrow}_{\mathrm{false}}(bs, P) &:= \gamma[y \mapsto j],\\ &\text{where } \gamma = \mathrm{env}^{\downarrow}_{\mathrm{false}}(bs, P) \text{ and } j = c(\mathrm{false} :: bs)\end{aligned}$$

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

Lemma B.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 $V : \mathrm{Bool}$ and $b \in \mathbb{B}$, then the base machine and handler machine transition in lock-step in either way*

1. *If* $|bs| = []$, *then*

$$\langle P\ p \mid \gamma \mid []\rangle \longrightarrow_{\mathsf{steps}(t)([])} \langle p\ (\!|i|\!) \mid \gamma' \mid \sigma\rangle,$$

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*

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

2. *For* $bs = b :: bs'$ *we have the following two subcases*
   - *If* $|bs| < n$, *then*

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

   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*

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

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

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

   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*

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

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

Let $\mathsf{control} : \mathsf{Conf} \rightharpoonup \mathsf{Val}$ 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 5.12.

Lemma B.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 $\mathsf{control}(\mathsf{depart}(bs, P)) \leq 2^{n-|bs|}$ *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.

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

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

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

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

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

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

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 $|bs| < n$

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

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

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

We now use Lemma B.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.

$$
\begin{array}{l}
\longrightarrow^{\mathrm{steps}(t)(\mathsf{true}::bs)} \quad \text{(by Lemma B.4)} \\
\quad \langle V\ j \mid \gamma'' \mid (\sigma', \chi_{\mathsf{count}}(P)) :: \mathrm{residual}(\mathsf{true} :: bs, P)\rangle \\
\qquad \qquad \text{where } ?j = \mathrm{labs}(t)(\mathsf{true} :: bs), \gamma'' = \mathrm{env}(t)(\mathsf{true} :: bs), \sigma' = \mathrm{pure}(t)(\mathsf{true} :: bs) \\
\qquad \qquad \text{and } [\![V]\!]\gamma'' = (\mathrm{env}^{\perp}(P), \lambda\_.\mathbf{do}\ \mathsf{Branch}\ \langle\rangle) \\
= \quad \text{(definition of arrive when } 1 + |bs| < n) \\
\quad \mathrm{arrive}(\mathsf{true} :: bs, P) \\
\longrightarrow^{T(\mathsf{true}::bs,\, n)} \quad \text{(induction hypothesis)} \\
\quad \mathrm{depart}(\mathsf{true} :: bs, P) \\
= \quad \text{(definition of depart when } 1 + |bs| < n) \\
\quad \langle \mathbf{return}\ i \mid \gamma \mid \mathrm{residual}(\mathsf{true} :: bs, P)\rangle \\
\qquad \qquad \text{where } i = c(\mathsf{true} :: \mathsf{true} :: bs) + c(\mathsf{false} :: \mathsf{true} :: bs) \text{ and } \gamma = \mathrm{env}^{\uparrow}_{\mathsf{false}}(\mathsf{true} :: bs, 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}}) :: \mathrm{purecont}(bs, P), \chi_{id})]\rangle \\
\qquad \qquad \text{where } \gamma' = \mathrm{env}^{\downarrow}_{\mathsf{true}}(bs, P) \\
\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'' \mid [(\mathrm{purecont}(bs, P), \chi_{id})]\rangle \\
\qquad \qquad \text{where } \gamma'' = \gamma'[x_{\mathsf{true}} \mapsto [\![i]\!]\gamma'] \\
\longrightarrow \quad (\text{M-Let}) \\
\quad \langle r\ \mathsf{false} \mid \gamma'' \mid [((\gamma'', x_{\mathsf{false}}, x_{\mathsf{true}} + x_{\mathsf{false}}) :: \mathrm{purecont}(bs, P), \chi_{id})]\rangle \\
= \quad \text{(definition of purecont and residual)} \\
\quad \langle r\ \mathsf{false} \mid \gamma'' \mid \mathrm{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)) :: \mathrm{residual}(\mathsf{false} :: bs, P)\rangle \\
\qquad \qquad \text{where } \sigma = \mathrm{pure}(t)(bs) \\
\longrightarrow^{\mathrm{steps}(t)(\mathsf{false}::bs)} \quad \text{(by Lemma B.4 and assumption } |\mathsf{false} :: bs| < n) \\
\quad \langle V\ j \mid \gamma \mid (\sigma, \chi_{\mathsf{count}}(P)) :: \mathrm{residual}(\mathsf{false} :: bs, P)\rangle \\
\text{where } ?j = \mathrm{labs}(t)(\mathsf{false} :: bs), \sigma = \mathrm{pure}(t)(\mathsf{false} :: bs), \gamma = \mathrm{env}(t)(\mathsf{false} :: bs) \\
\quad \text{and } [\![V]\!]\gamma = (\mathrm{env}^{\perp}(P), \lambda\_.\mathbf{do}\ \mathsf{Branch}\ \langle\rangle) \\
= \quad \text{(definition of arrive when } 1 + |bs| < n) \\
\quad \mathrm{arrive}(\mathsf{false} :: bs, P) \\
\longrightarrow^{T(\mathsf{false}::bs,\, n)} \quad \text{(induction hypothesis)} \\
\quad \mathrm{depart}(\mathsf{false} :: bs, P) \\
= \quad \text{(definition of depart when } 1 + |bs| < n) \\
\quad \langle \mathbf{return}\ j \mid \gamma \mid \mathrm{residual}(\mathsf{false} :: bs, P)\rangle \\
\qquad \qquad \text{where } j = c(\mathsf{true} :: \mathsf{false} :: bs) + c(\mathsf{false} :: \mathsf{false} :: bs) \text{ and } \gamma = \mathrm{env}^{\uparrow}_{\mathsf{false}}(\mathsf{false} :: bs, 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}}) :: \mathrm{purecont}(bs, P), \chi_{id})]\rangle \\
\longrightarrow \quad (\text{M-RetCont}) \\
\quad \langle x_{\mathsf{true}} + x_{\mathsf{false}} \mid \gamma''[x_{\mathsf{false}} \mapsto [\![j]\!]\gamma''] \mid \mathrm{residual}(bs, P)\rangle \\
\longrightarrow \quad (\text{M-Plus}) \\
\quad \langle \mathbf{return}\ m \mid \gamma''[x_{\mathsf{false}} \mapsto [\![j]\!]\gamma''] \mid \mathrm{residual}(bs, P)\rangle \\
\text{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) \\
\qquad \quad = c(\mathsf{true} :: bs) + c(\mathsf{false} :: bs) = c(bs) \leq 2^{n-|bs|} \\
= \quad \text{(definition of depart when } |bs| < n) \\
\quad \mathrm{depart}(bs, P)
\end{array}
$$

Step analysis The total amount of machine transitions is given by

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

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

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

Step analysis The total amount of machine transitions is given by

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

□

The following theorem is a copy of Theorem 5.12.

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

1. *The program* effcount *is a generic counting program, that is:*

$$\mathrm{effcount}\ P \rightsquigarrow^{+} \mathbf{return}\ V, \ \textit{such that}\ \mathbb{N}[\![V]\!] = C(P)([]) \le 2^n$$

2. *The runtime complexity of* effcount P *is given by the following formula:*

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

Proof. The proof begins by direct calculation.

$$
\begin{aligned}
& \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 (\textsc{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 \\
& \text{where } \gamma = \mathsf{env}^{\perp}(P) \\
\longrightarrow\ & \quad (\textsc{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^{\mathit{steps}(t)([])}\ & \quad (\text{by Lemma B.4}) \\
& \langle (\lambda\_.\mathbf{do}\ \mathsf{Branch}\ \langle\rangle)\ j \mid \gamma' \mid (\sigma, \chi_{\mathsf{count}}(P)) :: \mathsf{residual}(P, []) \rangle \\
& \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 B.5}) \\
& \mathsf{depart}(P, []) \\
=\ & \quad (\text{definition of depart}) \\
& \langle \mathbf{return}\ m \mid \gamma \mid \mathsf{residual}(P, []) \rangle \\
& \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 (\textsc{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{aligned}
$$

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

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

□

C Proof Details for the No Shortcuts Lemma

The proof of Lemma 5.13 rely on the fact that any n-standard predicate has a canonical form. Section C.1 disseminate canonical predicates, whilst Section C.2 proves Lemma 5.13.

C.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 C.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 C.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.

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

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 C.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 C.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 C.1). The rest follows as by Lemma C.3. □

C.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 5.13.

Lemma C.5. *Let P be an n-standard predicate. Suppose C is an n-count program, then C must apply P to at least 2^n distinct n-points.*

Proof. Proof by contradiction. Pick a boolean sequence $bs \in \mathbb{B}^n$. Suppose there exists an n-count program C which does not construct the critical point p_c corresponding to bs. Let b be the answer yielded by $P\ p$. Now construct a predicate P' which yields the same answers as P except that at p_c it yields $\neg b$. Such a predicate can be constructed by negating b in the untimed decision tree model of P, i.e.

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

Then $P' = \mathrm{norm}(t')$ constructs a canonical predicate, whose count is either one less or one more than that of P, that is at bs we have

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

because $[] \sqsubset bs$ we further obtain that $|C(P')([]) - C(P)([])| = 1$. Now 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 continue to reason about the length of the reduction sequences arising from applications of P and P'.

Lemma C.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 $\mathcal{F}[P] \leadsto^m \mathbf{return}\ V$ then $\mathcal{F}[P'] \leadsto^* \mathbf{return}\ V$ 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 $\mathcal{F}[P] \leadsto^0 \mathbf{return}\ V$ from which it follows that $\mathcal{F}[-]$ is simply $\mathbf{return}\ V$, thus it follows immediately that $\mathcal{F}[P'] \leadsto^0 \mathbf{return}\ V$.

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

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

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

$$\mathcal{F}[-] \leadsto \mathcal{F}'[-] \leadsto^{m'} \mathbf{return}\ V$$

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

$$\mathcal{G}[P, P\ p[P]] \leadsto \mathcal{G}[P, \mathbf{return}\ W] \leadsto \mathbf{return}\ V$$

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

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

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

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

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

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

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

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

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 $p \neq p_c$. 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

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

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 C.6 both $C\ P$ and $C\ P'$ reduce to the same value which contradicts the initial assumption.

□

D Proof Details for the No Sharing Lemma

The following lemma is a copy of Lemma 5.18.

Lemma D.1. *Suppose P is an n-standard predicate and C is an n-count program, and let p_0 and p_1 be distinct n-points, then the predicate applications $P\ p_0$ and $P\ p_1$ within C have disjoint threads.*

Proof. Let $T_0 = \mathsf{Th}(P\ p_0, \mathbb{P}[\![p_0]\!])$ and $T_1 = \mathsf{Th}(P\ p_1, \mathbb{P}[\![p_1]\!])$ be the threads arising from the two distinct predicate applications. Suppose, without loss of generality, that P is applied to p_0 before p_1, that is $C\ P \rightsquigarrow^{+} \mathcal{E}_0[P\ p_0] \rightsquigarrow^{+} \mathcal{E}_1[P\ p_1] \rightsquigarrow^{+} \cdots$ which by Lemma 5.16 implies that T_0 starts before T_1. There are now two possible cases to consider.

1. T_0 finishes before T_1 starts. It follows immediately that T_0 and T_1 are disjoint.
2. T_1 starts in between the sections of T_0. We now argue that T_1 must finish before evaluation of T_0 can continue. Suppose for any $i < n$ that the i-th query q_i starts T_1, i.e

$$\mathcal{E}_i[p_0\ q_i] \rightsquigarrow \mathcal{E}_i[\mathcal{E}'[P\ p_1]]$$

then by the 'n-ness' of C, P, and p_1 and since the reduction relation $\rightsquigarrow$ is deterministic it follows that $\mathcal{E}'$ reduces to a boolean value W which is plugged into the continuation of $\mathcal{E}_i[-]$

$$\mathcal{E}_i[p_0\ q_i] \rightsquigarrow \mathcal{E}_i[\mathcal{E}'[P\ p_1]] \rightsquigarrow^{+} \mathcal{E}_i[\mathbf{return}\ W]$$

Thus, T_1 must finish executing before evaluation of T_0 can resume.

□