Asymptotic Improvement with Effect Handlers

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

As Filinski first showed in the 1990s, first-class 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.

In this paper we study the fundamental efficiency of effect handlers. Specifically, we show the presence of effect handlers enables an asymptotic improvement in runtime complexity for a certain class of programs. To obtain this result, we consider the *generic search* problem and define a pure PCF-like base language λ_b and an extension thereof λ_h with effect handlers. We show that λ_h admits an implementation of generic search that is asymptotically more efficient than any implementation of generic search in λ_b.

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

1 INTRODUCTION

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

One can interpret this question in various ways. For instance, Felleisen [1991] considers the question of whether a language $\mathcal{L}$ admits a translation into a sublanguage $\mathcal{L}'$ in a way which respects not only the behaviour of programs but also aspects of their (global or local) syntactic structure. If the translation of some $\mathcal{L}$-program into $\mathcal{L}'$ requires a complete global reorganisation of its structure, 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 some type A that are programmable in $\mathcal{L}$ but are 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 point of advantage of $\mathcal{L}$ over $\mathcal{L}'$.

If the 'operations' we are asking about are ordinary first-order functions — that is, both their inputs and outputs are of ground type (strings, arbitrary-size integers etc.) — then the situation is easily summarised. At such types, all reasonable languages give rise to the same class of programmable functions, namely the Church-Turing computable ones. As for complexity, the runtime of a program is typically analysed with respect to some cost model for basic instructions (e.g. one unit of time per array access). Although the realism of such cost models in the asymptotic limit can be questioned (see, e.g., [Knuth 1997, Section 2.6]), it is broadly taken as read that such models are equally applicable whatever programming language we are working with, and moreover that all respectable languages can represent all algorithms of interest; thus, one does not expect the best achievable asymptotic run-time for a typical algorithm (say in number theory or graph theory) to be sensitive to the choice of programming language, except perhaps in marginal cases. (It should be admitted, however, that proving *general theorems* to this effect may be harder than one might suppose: see for example Section 1 of [Pippenger 1996].)

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

Most 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 [Plotkin 1997]. It is also well known that the presence of control features or local state enables observational distinctions that cannot be made in a purely functional setting: for instance, there are programs involving 'call/cc' that detect the order in which a (call-by-name) '+' operation evaluates its arguments [Cartwright and Felleisen 1992]. Such operations are 'non-functional' in the sense that their output is not determined solely by the extension of their input (seen as a mathematical function $\mathbb{N}_\perp \times \mathbb{N}_\perp \rightarrow \mathbb{N}_\perp$); however, there are also programs with 'functional' behaviour that can be implemented with control or local state but not without them [Longley 1999]. More recent results have exhibited differences lower down in the language expressivity spectrum: for instance, in a purely functional setting *à la* Haskell, the expressive power of *recursion* increases strictly with its type level [Longley 2018b], and there are natural operations computable by (low-order) recursion but not by (even high-order) iteration [Longley 2018a]. Much of this territory, including the mathematical theory of some of the natural notions of higher-order computability that arise in this way, is mapped out by Longley and Normann [2015].

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

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

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

A naive implementation strategy, supported by any reasonable language, is simply to apply P to each of the 2^n vectors in turn. A much less obvious, but still purely 'functional', approach due to Berger [1990] achieves the effect of 'pruned search' where the predicate 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 $O(2^n)$ for these simple predicates: an asymptotic gain of a factor of n.

In order 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 [2011]; 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; however, 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) 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. We also touch on the prospects for formulating a more general version of our results that would apply to *any* reasonable execution model for the base language.

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}.
- Section 4 augments λ_{b} with effect handlers yielding the language λ_{h}.
- Section 5 defines an abstract machine for λ_{b} and an extension thereof for λ_{h}.
- Section 6 formally states and proves the complexity of generic search in λ_{b} (namely $\Omega(n2^n)$) and λ_{h} (namely $O(2^n)$).
- Section 7 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.

2 EFFECT HANDLERS PRIMER

Effect handlers were originally studied as a theoretical means to provide a semantics for exception handling in the setting of algebraic effects [Plotkin and Power 2001; Plotkin and Pretnar 2013]. Subsequently they have emerged as a practical programming abstraction for modular effectful programming [Bauer and Pretnar 2015; Dolan et al. 2015; Hillerström and Lindley 2016; Kammar et al. 2013; Kiselyov et al. 2013; Leijen 2017; Lindley et al. 2017]. In this section we will provide a minimal introduction to effect handlers. It is minimal in the sense that it provides the necessary vocabulary and insight to understand the main result of this paper. For a thorough introduction to programming with effect handlers, we recommend the tutorial by Pretnar [2015], and as an introduction to the mathematical foundations of handlers, we refer the reader to the founding paper by Plotkin and Pretnar [2013] and the excellent tutorial paper by Bauer [2018].

Viewed through the lens of universal algebra, an algebraic effect is a signature Σ of finitary *operation symbols* defined over some nonempty carrier set A, along with an equational theory that describes the properties of the operations [Plotkin and Power 2001]. The operations represent the sources of effects, for example for input/output the operations should be *read* and *write*. A classic example of an algebraic effect is *nondeterminism*, whose signature consists of a single nondeterministic choice operation.

$$\Sigma \overset{\text{def}}{=} \{\mathsf{Branch} : 1 \to \mathsf{Bool}\}$$

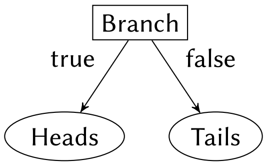
The operation takes a single parameter of type unit and ultimately produces a boolean value. The pragmatic programmatic view of algebraic effects differ slightly from the original development. For example, as of yet no practical implementation accounts for equations over the operations, and as such, the algebra underlying the implementations is the free algebra.

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 $\mathsf{Toss} \overset{\text{def}}{=} \mathsf{Heads} \mid \mathsf{Tails}$, then in our programming notation (introduced formally in Section 4) we may implement a coin toss as follows.

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

The computation toss is thunked, because our language is in essence a variation of call-by-value PCF. From the type signature it is clear that the computation returns a value of type Toss. It is not clear from looking at the signature of toss whether it is 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 [Lindley et al. 2017], Helium [Biernacki et al. 2019], Koka [Leijen 2017], and Links [Hillerström and Lindley 2016] include type-and-effect systems which track the use of effectful operations. Whilst current iterations of systems such as Eff [Bauer and Pretnar 2015] and Multicore OCaml [Dolan et al. 2015] elect not to include an effect system. Our language is closer to the latter two.

Returning to the example of toss, we may, in the style of Lindley [2014], view an effectful computation as a tree, where the interior nodes correspond to operation invocations and the leaves correspond to return values. The computation toss induces a particular simple computation tree.



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

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

The **handle**-construct generalises the exceptional syntax of Benton and Kennedy [2001]. The handler definition consists of two clauses, a *success* clause, and an *operation* clause. The former clause determines how to interpret the return value of toss, or equivalently how to interpret the leaves of the computation tree above. In this particular instance we lift 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. As a result it returns Heads, which causes the success clause of the handler to be invoked, and 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 gets concatenated with the result of the true branch, hence the ultimately the handler returns [**inl** Heads, **inr** Tails].

Program evaluation is a dialogue between a computation and its handler. When the computation invokes an operation, control transfers to the corresponding operation clause of the handler. When the handler invokes the resumption, control transfers back to the computation, which may then invoke further operations starting the cycle again. If the computation evaluates to a value then control transfers to the return clause which yields a final result.

3 A FINE-GRAIN CALL-BY-VALUE BASE CALCULUS

In this section, we present our base language λ_b along with its contextual small-step operational semantics. The underlying formalism is fine-grain call-by-value [Levy et al. 2003], which is like A-normal form [Flanagan et al. 1993] in that every intermediate computation is named, but unlike A-normal form is closed under reduction. Our language amounts to a fine-grained variant of call-by-value PCF [Plotkin 1997].

$$\begin{array}{lrl}
\text{Types} & A, B, C, D & ::= \mathsf{Nat} \mid 1 \mid A \to B \mid A \times B \mid A + B \\
\text{Type environments} & \Gamma & ::= \cdot \mid \Gamma, x : A
\end{array}$$

Fig. 1. Types and Environments

$$\begin{array}{lrl}
\text{Naturals} & n \in \mathbb{N} & \\
\text{Variables} & x, y, z, \dots & \\
\text{Values} & V, W \in \mathrm{Val} & ::= x \mid \langle\rangle \mid n \mid \mathsf{Eq} \mid \mathsf{Plus} \mid \mathsf{Minus} \mid \lambda x^A.\, M \mid \mathbf{rec}\, f^A\, x.M \\
& & \mid\ \ \langle V, W\rangle \mid (\mathbf{inl}\, V)^B \mid (\mathbf{inr}\, W)^A \\
\text{Computations} & M, N \in \mathrm{Comp} & ::= V\, W \\
& & \mid\ \ \mathbf{let}\ \langle x, y\rangle = V\ \mathbf{in}\ N \mid \mathbf{case}\ V\ \{\mathbf{inl}\ x \mapsto M; \mathbf{inr}\ y \mapsto N\} \\
& & \mid\ \ \mathbf{return}\ V \mid \mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N
\end{array}$$

Fig. 2. Term Syntax

Our primary reason for choosing fine-grain call-by-value over plain call-by-value is that the former yields a considerably simpler semantics as only the rule for let bindings admits a continuation whereas in ordinary call-by-value each congruence rule admits a continuation.

3.1 Types

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

3.2 Terms

The terms are given in Figure 2. We let n range over natural number constants. We will 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 syntax partitions terms into values and computations. Value terms comprise variables (x), the unit value ($\langle\rangle$), naturals (n), primitive functions on naturals (Eq, Plus, Minus), 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 elect to keep lambda abstractions in the core calculus, although, they are strictly speaking redundant in the presence of a recursion operator. We assume an efficient representation of naturals (e.g. naturals occupy a machine word) such that the three primitive functions Eq, Plus, and Minus have efficient realisations.

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 using the case construct ($\mathbf{case}\ V\ \{\mathbf{inl}\ x \mapsto M; \mathbf{inr}\ y \mapsto N\}$), which evaluates the computation M if the tag of V matches **inl**. Otherwise V must have been tagged with **inr** in which case it evaluates the computation 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.

Values

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

$$\textsc{T-\{Plus,Minus\}}\ \frac{\star \in \{\mathsf{Plus}, \mathsf{Minus}\}}{\Gamma \vdash \star : \langle \mathsf{Nat}, \mathsf{Nat}\rangle \to \mathsf{Nat}} \qquad \textsc{T-Eq}\ \frac{}{\Gamma \vdash \mathsf{Eq} : \langle \mathsf{Nat}, \mathsf{Nat}\rangle \to 1 + 1}$$

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

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

Computations

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

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

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

Fig. 3. Typing Rules

Syntactic sugar. Fine-grain call-by-value mandates that every intermediate computation is let-bound, meaning that it is not particularly pleasant for writing programs. As we shall see subsequently, we basically trade syntactic verbosity for semantic simplicity. For examples, we often use direct-style call-by-value with the understanding that this denotes the left-to-right call-by-value translation into fine-grain call-by-value [Flanagan et al. 1993]. For example, assuming f, g, h are functions, and a is a bound variable, then the expression $(f\,(h\,a) + g\,\langle\rangle)$ denotes

$$[\![f\,(h\,a) + g\,\langle\rangle]\!] = \mathbf{let}\ x \leftarrow h\,a\ \mathbf{in}\ \mathbf{let}\ y \leftarrow f\,x\ \mathbf{in}\ \mathbf{let}\ z \leftarrow g\,\langle\rangle\ \mathbf{in}\ \mathsf{Plus}\,\langle y, z\rangle$$

We often write the standard infix operators =, +, and − in place of Eq, Plus, and Minus.

3.3 Static semantics

The typing rules are given in Figure 3. We require two typing judgements: one for values and the other for computations. We capture both in one notation: the judgement $\Gamma \vdash \square : A$ states that a $\square$-term has type A under type environment Γ, where $\square$ can be either a value term (V) or a computation term (M). The typing rules are mostly standard. We encode booleans as sums of type $1 + 1$. In Section 3.5 we spell out the derived forms of λ_{b}.

$$\begin{array}{lll}
\textsc{S-App} & (\lambda x^A.\, M)V \leadsto M[V/x] & \\
\textsc{S-App-Rec} & (\mathbf{rec}\, f^A\, x.\, M)V \leadsto M[(\mathbf{rec}\, f^A\, x.\, M)/f, V/x] & \\
\textsc{S-Plus} & \mathsf{Plus}\,\langle i, j\rangle \leadsto \mathbf{return}\ n, & \text{where } i + j = n \\
\textsc{S-Minus} & \mathsf{Minus}\,\langle i, j\rangle \leadsto \mathbf{return}\ n, \quad \text{where } n = \begin{cases} 0 & \text{if } j \geq i \\ i - j & \text{otherwise} \end{cases} & \\
\textsc{S-Eq} & \mathsf{Eq}\,\langle i, j\rangle \leadsto \mathbf{return}\ x, \text{where } x = \begin{cases} \mathbf{inl}\ \langle\rangle & \text{if } i = j \\ \mathbf{inr}\ \langle\rangle & \text{otherwise} \end{cases} & \\
\textsc{S-Split} & \mathbf{let}\ \langle x; y\rangle = \langle V; W\rangle\ \mathbf{in}\ N \leadsto N[V/x, W/y] & \\
\textsc{S-Case-inl} & \begin{array}{r}\mathbf{case}\ (\mathbf{inl}\ V)^B\ \{\mathbf{inl}\ x \mapsto M; \\ \mathbf{inr}\ y \mapsto N\}\end{array} \leadsto M[V/x] & \\
\textsc{S-Case-inr} & \begin{array}{r}\mathbf{case}\ (\mathbf{inr}\ V)^A\ \{\mathbf{inl}\ x \mapsto M; \\ \mathbf{inr}\ y \mapsto N\}\end{array} \leadsto N[V/y] & \\
\textsc{S-Let} & \mathbf{let}\ x \leftarrow \mathbf{return}\ V\ \mathbf{in}\ N \leadsto N[V/x] & \\
\textsc{S-Lift} & \mathcal{E}[M] \leadsto \mathcal{E}[N], & \text{if } M \leadsto N
\end{array}$$

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

Fig. 4. Contextual Small-Step Operational Semantics

3.4 Dynamic semantics

We present a small-step operational semantics for $\lambda_b$using *evaluation contexts* in the style of Felleisen [1987]. Figure 4 presents the operational rules. We write $M[V/x]$ for the substitution of the value V for the variable x in the computation term M. The reduction relation $\leadsto$ is defined on computation terms. The statement $M \leadsto N$ reads: term M reduces to term N in a single step. We write R^+ for the transitive closure of relation R.

Definition 3.1 (Computation normal form). A computation M is a *normal* if it is of the form **return** V for some value $V \in \mathrm{Val}$.

Subject reduction and type soundness for λ_b are standard.

Theorem 3.2 (Subject Reduction). *If $\Gamma \vdash M : A$ and $M \leadsto M'$, then $\Gamma \vdash M' : A$.*

Theorem 3.3 (Type Soundness). *If $\vdash M : A$, then either there exists $\vdash N : A$, such that $M \leadsto^+ N$ and N is normal, or M diverges.*

3.5 Derived Forms

As mentioned in Section 3.3 we encode booleans along with their operations from the core of λ_b. In the following we present the derived forms which we will make use of later in Section 6.

We define the type of booleans Bool as a sum of units, that is

$$\mathsf{Bool} \overset{\text{def}}{=} 1 + 1$$

and correspondingly we define the two boolean constants true and false as the left and right injections, respectively.

$$\mathsf{true} \overset{\text{def}}{=} \mathbf{inl}\ \langle\rangle \quad \text{and} \quad \mathsf{false} \overset{\text{def}}{=} \mathbf{inr}\ \langle\rangle$$

We define conditional branching in terms of case splits.

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

We shall write S-If-tt as a synonym for S-Case-Inl when the scrutinee has type Bool, and in similar fashion we write S-If-ff in lieu of S-Case-Inr.

We also define some lightweight, but convenient, syntactic sugar for patterns. For suspended computations we write

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

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

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

We elide type annotations when clear from context.

4 A FINE-GRAIN CALCULUS WITH EFFECT HANDLERS

We now endow λ_{b} with effects and yielding the calculus λ_{h}. Specifically, we add two new computation term forms that introduce and eliminate effects. The syntax of value terms will be unchanged. First we define notation for operation symbols, signatures, and handler types.

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

We assume the existence of 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 $A \twoheadrightarrow B$ denotes an operation that takes an argument of type A and returns a result of type B. We write $dom(\Sigma) \subseteq \mathcal{L}$ for the set of operation symbols in a signature Σ. An effect handler type $C \Rightarrow D$ classifies effect handlers that transform computations of type C into computations of type D. Following Pretnar [2015], we shall assume a global signature for every program. We now have the basic machinery to present the two new computation forms.

4.1 Performing Effectful Operations

We give the syntax and static semantics for the introduction form for effectful operations, and defer presenting its dynamic semantics formally until we have introduced the elimination form.

$$\text{Computations} \quad M, N \in \text{Comp} ::= \cdots \mid \boxed{\mathbf{do}\ \ell\ V}$$

Informally, the behaviour of the new computation term ($\mathbf{do}\ \ell\ V$) is that it invokes an operation ℓ with a value argument V. The typing rule for **do** relies on the global signature Σ.

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

An operation invocation is only well-typed if the operation ℓ appears in the effect signature Σ, and the argument type A matches the type of the provided argument V. The result type B determines the type of the invocation. In richer type systems with effect tracking such as that of Hillerström and Lindley [2016] or Leijen [2017] signatures are an integral part of the type structure.

4.2 Handling of Effectful Operations

We now present the elimination form for effectful operations.

$$\begin{array}{lrl} \text{Computations} & M, N \in \text{Comp} ::= & \cdots \mid \boxed{\mathbf{handle}\ M\ \mathbf{with}\ H} \\ \text{Handlers} & H ::= & \{\mathbf{val}\ x \mapsto M\} \\ & \mid & \{\ell\ p\ r \mapsto N\} \uplus H \end{array}$$

The handle construct ($\mathbf{handle}\ M\ \mathbf{with}\ H$) is the counterpart to **do**. It runs computation M using handler H. A handler H consists of a value clause $\{\mathbf{val}\ x \mapsto M\}$ and a possibly empty set of operation clauses $\{\ell\ p\ r \mapsto N_\ell\}_{\ell \in \mathcal{L}}$. The value clause $\{\mathbf{val}\ x \mapsto M\}$ defines how to interpret a final return value of the handled computation. The variable x is bound to the final return value in the body M. Each operation clause $\{\ell\ p\ r \mapsto N_\ell\}_{\ell \in \mathcal{L}}$ defines how to interpret an invocation of a particular operation ℓ. The variables p and r are bound in the body N_ℓ: p binds the argument carried by the operation and resumption r binds the resumption of the invocation site of ℓ.

An appealing feature of effect handlers is *operation forwarding*, which intuitively means that if some handler H has no matching operation clause for some operation ℓ, then H silently forwards ℓ to its nearest enclosing handler. In programming practice, forwarding is crucial for modularity as it provides a basis for composing a collection of fine-grained, specialised effect handlers to interpret a larger effect signature [Hillerström and Lindley 2016; Kammar et al. 2013]. The handlers in our calculus do not support forwarding directly; rather, we follow Plotkin and Pretnar [2013] and adopt the convention that a handler with omitted operation clauses (with respect to global signature Σ) is syntactic sugar for one in which all missing clauses perform forwarding explicitly:

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

By omitting forwarding, we obtain a slightly simpler metatheory without altering the expressive power of the system. (Of course, if we were to add a type-and-effect system then this approach would lose important information.)

Given a handler H, we often wish to refer to the clause for a particular operation or the value clause; for these purposes we define two convenient projections on handlers in the metalanguage.

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

The H^ℓ projection yields the singleton set consisting of the operation clause in H that handles the operation ℓ, whilst H^{val} yields the singleton set containing the value clause of H.

4.2.1 Static semantics. The typing of effect handlers is slightly more involved than the typing of the **do**-construct.

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

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

The T-Handle rule is straightforward. Most of the work happens in the T-Handler rule. It ensures that the bodies of the value 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.

Deep Versus Shallow. An alternative would be to set the return type of r_ℓ to be C. Then the programmer would have to explicitly reinvoke the effect handler as desired. Our current rule gives rise to *deep handlers* whereas the alternative rule gives rise to *shallow handlers* [Hillerström and Lindley 2018; Kammar et al. 2013].

4.2.2 Dynamic semantics. We add two new reduction rules to the operational semantics: one for handling return values and the other for handling operations.

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

The rule S-Ret invokes the return clause of a handler. The rule S-Op handles an operation via the corresponding operation clause. Rather than augmenting the notion of evaluation contexts from the base language, we introduce a new context for handlers.

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

$$\textsc{S-Lift-H}\quad \mathcal{H}[M] \rightsquigarrow \mathcal{H}[N], \qquad \text{if } M \rightsquigarrow N$$

The separation between pure evaluation contexts $\mathcal{E}$ and handler contexts $\mathcal{H}$ guarantees the reduction rules remain deterministic, as otherwise (S-Op) can pick an arbitrary handler in scope. But with this separation, the rule must pick the innermost handler.

The metatheoretic properties of λ_{b} transfer to λ_{h} with little extra effort, alas we have to amend the definition of computation normal forms as there are now two ways in which a computation term can terminate: successfully returning a value or getting stuck on an unhandled operation.

Definition 4.1 (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 $\textbf{return}\ V$, or $\mathcal{E}[\textbf{do}\ \ell\ W]$.

Theorem 4.2 (Subject Reduction). *If $\Gamma \vdash M : C$ and $M \rightsquigarrow M'$, then $\Gamma \vdash M' : C$.*

Theorem 4.3 (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.*

5 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 *contextual small-step operational semantics* which uses a substitution model for evaluation. Whilst the substitution model is semantically pleasing, it falls short of providing a realistic account of, say, practical computation as performed by a computer. The reason being that substitution is a fairly expensive operation asymptotically. For this reason we develop an alternative model that is a little closer to practical computation, namely an *abstract machine semantics.*

5.1 Base Machine

In this section we present a *CEK*-style abstract machine semantics [Felleisen and Friedman 1987] for λ_{b}. The machine is adapted from Hillerström and Lindley [2016]. 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 last component contains the continuation which instructs the machine what to do once it is done evaluating the current computation. The syntax of abstract machine states is given in Figure 5. 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 $\textbf{let}\ x = [\,]\ \textbf{in}\ N$ over environment γ. Continuations are stacks of continuation frames. We write $[]$ for an empty continuation and $\phi :: \sigma$ for the result of pushing the frame ϕ on top of σ. We use pattern matching to deconstruct continuations

The abstract machine semantics is given in Figure 6. 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

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

Fig. 5. Abstract Machine Syntax

Initial state (for some computation M) $\langle M \mid \emptyset \mid [] \rangle$

Transition function

$$\begin{array}{lll}
\text{M-APP} & \langle V\ W \mid \gamma \mid \sigma \rangle \longrightarrow \langle M \mid \gamma'[x \mapsto [\![W]\!]\gamma] \mid \sigma \rangle, & \text{if } [\![V]\!]\gamma = (\gamma', \lambda x^A.\, M) \\
\text{M-APP-REC} & \langle V\ W \mid \gamma \mid \sigma \rangle \longrightarrow \langle M \mid \gamma'[f \mapsto (\gamma', \mathbf{rec}\, f\, x.M), x \mapsto [\![W]\!]\gamma] \mid \sigma \rangle, & \\
& & \text{if } [\![V]\!]\gamma = (\gamma', \mathbf{rec}\, f\, x^A.M) \\
\text{M-EQ} & \langle \mathsf{Eq}\ \langle V, W \rangle \mid \gamma \mid \sigma \rangle \longrightarrow \langle \mathbf{return}\ V' \mid \gamma \mid \sigma \rangle, & \text{where } V' = \begin{cases} \mathbf{inl}\ \langle\rangle & \text{if } [\![V]\!]\gamma = [\![W]\!]\gamma \\ \mathbf{inr}\ \langle\rangle & \text{otherwise} \end{cases} \\
\text{M-PLUS} & \langle \mathsf{Plus}\ \langle V, W \rangle \mid \gamma \mid \sigma \rangle \longrightarrow \langle \mathbf{return}\ V' \mid \gamma \mid \sigma \rangle, & \text{where } V' = [\![V]\!]\gamma + [\![W]\!]\gamma \\
\text{M-MINUS} & \langle \mathsf{Minus}\ \langle V, W \rangle \mid \gamma \mid \sigma \rangle \longrightarrow \langle \mathbf{return}\ V' \mid \gamma \mid \sigma \rangle, & \text{where } V' = \begin{cases} 0 & \text{if } [\![W]\!]\gamma \geq [\![V]\!]\gamma \\ [\![V]\!]\gamma - [\![W]\!]\gamma & \text{otherwise} \end{cases}
\end{array}$$

$$\begin{array}{lrcl}
\text{M-SPLIT} & \langle \mathbf{let}\ \langle x, y \rangle = V\ \mathbf{in}\ N \mid \gamma \mid \sigma \rangle & \longrightarrow & \langle N \mid \gamma[x \mapsto v, y \mapsto w] \mid \sigma \rangle, \quad \text{if } [\![V]\!]\gamma = \langle v; w \rangle \\
\text{M-CASE} & \begin{array}{r}\langle \mathbf{case}\ V\ \{\mathbf{inl}\, x \mapsto M; \\ \mathbf{inr}\, y \mapsto N\} \mid \gamma \mid \sigma \rangle\end{array} & \longrightarrow & \begin{cases} \langle M \mid \gamma[x \mapsto v] \mid \sigma \rangle, & \text{if } [\![V]\!]\gamma = \mathbf{inl}\, v \\ \langle N \mid \gamma[y \mapsto v] \mid \sigma \rangle, & \text{if } [\![V]\!]\gamma = \mathbf{inr}\, v \end{cases} \\
\text{M-LET} & \langle \mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N \mid \gamma \mid \sigma \rangle & \longrightarrow & \langle M \mid \gamma \mid (\gamma, x, N) :: \sigma \rangle \\
\text{M-RETCONT} & \langle \mathbf{return}\ V \mid \gamma \mid (\gamma', x, N) :: \sigma \rangle & \longrightarrow & \langle N \mid \gamma'[x \mapsto [\![V]\!]\gamma] \mid \sigma \rangle
\end{array}$$

Final state (yielding some value $[\![V]\!]\gamma$) $\langle \mathbf{return}\ V \mid \gamma \mid [] \rangle$

Value interpretation

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

Fig. 6. Abstract Machine Semantics

and identity continuation. The rules (M-APP), (M-SPLIT), and (M-CASE) enact the elimination of values. Observe that (M-APP) handles application of both closures and of captured continuations. The (M-LET) rule extends the current continuation with let bindings and handlers respectively. The rule (M-RETCONT) binds a returned value if there is a pure continuation in the current continuation frame. Assuming the input is 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 the operational semantics and the abstract machine more precise.

5.1.1 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 7

Configurations

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

Continuations

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

Computation terms

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

Value terms and values

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

Fig. 7. Mapping from Base Machine Configurations to Terms

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 5.1. 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 5.2 (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 5.3. *Suppose M is a computation and C is configuration such that $(\!|C|\!) = M$, then if $M \rightsquigarrow N$ there exists C' such that $C \Longrightarrow C'$ and $(\!|C'|\!) = N$, or if $M \not\rightsquigarrow$ then $C \not\Longrightarrow$.*

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

The correspondence here is rather strong: there is a one-to-one mapping between $\rightsquigarrow$ and $\Longrightarrow$. The inverse of the lemma is straightforward as the semantics is deterministic. Notice that Lemma 5.3 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 5.4 (Simulation). *If $\vdash M : A$ and $M \rightsquigarrow^+ N$ such that N is normal, then $\langle M \mid \emptyset \mid []\rangle \longrightarrow^+ C$ such that $(\!|C|\!) = M$, or $M \not\rightsquigarrow$ then $\langle M \mid \emptyset \mid []\rangle \not\longrightarrow$.*

Proof. By repeated application of Lemma 5.3. □

5.2 Handler Machine

We now enrich the machine for λ_{b} to support effect handlers, resulting in an abstract machine for λ_{h}. As the two machines share much of the syntax and transition rules, we choose to only discuss the difference between the two. Figure 8 contains the required changes to the syntax for accommodating effect handlers. The notion of configurations changes slightly as the continuation component is now occupied by $\kappa \in \mathsf{Cont}$, which is a generalised continuation in the sense of Hillerström and Lindley [2016]. A machine continuation is now a list of pairs consisting of a pure continuation (as in the base machine) and a handler closure. The handler closure acts as the delimiter for its pure continuation. A handler closure is a pair of an environment and a handler definition, where the former binds the free variables that occur in the latter. The machine values are augmented to contain handler closures, the reason being that 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 lack of implicit effect forwarding in λ_{h} means it suffices to bind only the topmost handler closure. Systems with effect forwarding need to reify potentially multiple consecutive handler closures, thus in such systems the resumption would typically be bound to a sub-part of the machine continuation.

The handler machine not only adds more transition rules to account for handlers, but also changes the M-Let and M-AppCont rules from the base machine which is necessary to account for the richer continuation structure. Figure 9 displays the new and modified transition rules. The initial machine continuation, κ_0, is a singleton containing the handler closure of the identity handler. The rule (M-Handle) pushes a the handler closure along with an empty pure continuation

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

Fig. 8. Abstract Machine Syntax for Handlers

Identity continuation

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

Initial state (for some computation M) $\quad \langle M \mid \emptyset \mid \kappa_0 \rangle$

Transition function

$$
\begin{array}{lrcl}
\textsc{M-Resume} & \langle V\ W \mid \gamma \mid \kappa \rangle & \longrightarrow & \langle \mathbf{return}\ W \mid \gamma \mid (\sigma, \chi) :: \kappa \rangle, \\
 & & & \qquad \text{if } [\![V]\!]\gamma = (\sigma, \chi)^A \\[1ex]
\textsc{M-Let} & \langle \mathbf{let}\ x \leftarrow M\ \mathbf{in}\ N \mid \gamma \mid (\sigma, \chi) :: \kappa \rangle & \longrightarrow & \langle M \mid \gamma \mid ((\gamma, x, N) :: \sigma, \chi) :: \kappa \rangle \\
\textsc{M-Handle} & \langle \mathbf{handle}\ M\ \mathbf{with}\ H \mid \gamma \mid \kappa \rangle & \longrightarrow & \langle M \mid \gamma \mid ([], (\gamma, H)) :: \kappa \rangle \\
\textsc{M-AppCont} & \langle \mathbf{return}\ V \mid \gamma \mid ((\gamma', x, N) :: \sigma, \chi) :: \kappa \rangle & \longrightarrow & \langle N \mid \gamma'[x \mapsto [\![V]\!]\gamma] \mid (\sigma, \chi) :: \kappa \rangle \\
\textsc{M-RetHandler} & \langle \mathbf{return}\ V \mid \gamma \mid ([], (\gamma', H)) :: \kappa \rangle & \longrightarrow & \langle M \mid \gamma'[x \mapsto [\![V]\!]\gamma] \mid \kappa \rangle, \\
 & & & \qquad \text{if } H^{\mathrm{val}} = \{\mathbf{val}\ x \mapsto M\} \\
\textsc{M-Handle-Op} & \langle \mathbf{do}\ \ell\ V \mid \gamma \mid (\sigma, (\gamma', H)) :: \kappa \rangle & \longrightarrow & \langle M \mid \gamma'[p \mapsto [\![V]\!]\gamma, r \mapsto (\sigma, (\gamma', H))] \mid \kappa \rangle, \\
 & & & \qquad \text{if } \ell : A \to B \in \Sigma \text{ and } H^{\ell} = \{\ell\ p\ r \mapsto M\}
\end{array}
$$

Final states (yielding some value $[\![V]\!]\gamma$) $\quad \langle \mathbf{return}\ V \mid \gamma \mid [] \rangle$

(stuck on some operation ℓ) $\quad \langle \mathbf{do}\ \ell\ V \mid \gamma \mid [] \rangle$

Fig. 9. Abstract Machine Semantics for Handlers

Configurations

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

Continuations

$$(\!|[]|\!)M = M$$
$$(\!|(\sigma, \chi) :: \kappa|\!)M = (\!|\kappa|\!)((\!|\chi|\!)((\!|\sigma|\!)(M)))$$

Handler Closures and Definitions

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

Computation Terms and Machine Values

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

Fig. 10. Mapping from Handler Machine Configurations to Terms

onto the continuation stack. The (M-Let) rule grows the pure continuation by pushing a new frame onto the pure continuation of the topmost continuation frame. Dually, the (M-AppCont) rule shrinks the pure continuation of the topmost continuation frame. 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. The correctness result for the base machine can mostly be repurposed for the handler machine as we need only recheck the cases for (M-Let) and (M-AppCont) and check the cases for handlers. Figure 10 shows the necessary changes to the $(\!|-|\!)$ function.

Lemma 5.5. *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 $M \leadsto N$. □

Theorem 5.6 (Simulation). *If $\vdash M : A$ and $M \leadsto^+ N$ such that N is normal, then $\langle M \mid \emptyset \mid \kappa_0 \rangle \longrightarrow^+ C$ such that $(\!|C|\!) = M$, or $M \not\leadsto$ then $\langle M \mid \emptyset \mid \kappa_0 \rangle \not\longrightarrow$.*

Proof. By repeated application of Lemma 5.3. □

5.3 Asymptotic Complexity of Machine Operations

In this section we briefly detail the asymptotic complexity of the abstract machine operations. Both machines have three operations in common: continuation augmentation, environment extension, and environment lookup. An overview of their asymptotic complexities is given in Table 1. In the second column of the table we use wildcard pattern matching notation (_) to denote that the operands are irrelevant for the complexity of their operation. Since continuations are lists the complexity of pushing a single frame on top of a continuation is always constant. We assume environments are implemented using a functional map data structure, and thus admit a logarithmic worst-case complexity in the size of their environments [Okasaki 1999]. The handler machine arguably has one more operation, namely, pure continuation augmentation. However, time complexity-wise this operation happens to coincide with the augmentation operation for the base machine as augmenting the pure continuation only requires reaching under the topmost handler closure, which is a constant time operation.

Operation	Syntax	Complexity
Continuation augmentation	$\_ :: \_$	$O(1)$
Environment extension	$\gamma[\_ \mapsto \_]$	$O(\log \lvert\gamma\rvert)$
Environment lookup	$\gamma(\_)$	$O(\log \lvert\gamma\rvert)$

Table 1. Asymptotic Complexity of the Abstract Machine Operations

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 $[\![-]\!]$ is defined structurally on values, but it also takes an environment as input. Its worst-case time complexity is exhibited by a nesting of pairs of variables $[\![\langle x_1, \ldots, x_n\rangle]\!]\gamma$ which has complexity $O(n \log |\gamma|)$.

6 EFFICIENT GENERIC SEARCH

We now turn to the main crux of this paper. In this section we will 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* to be our 'one' program.

A generic search procedure is a modular search procedure that finds solutions to a given search problem P. Generic search is entirely 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 [Daniels 2016; Simpson 1998].

To simplify the presentation, we compute the number of solutions of any given search problem (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 Nat $\to$ 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 $\mathbb{N}_n \overset{\text{def}}{=} \{0, \ldots, n-1\}$. We shall often omit the n indexes when they are clear from context, and in particular they do not appear explicitly in the types of our programs because 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.

6.1 Predicates and Points

Higher-order functions are the key feature that enable us to give a modular formulation of generic search. We define a predicate of size n as a higher-order function that acts on points.

$$\mathsf{Predicate}_n \overset{\text{def}}{=} \mathsf{Point}_n \to \mathsf{Bool}$$

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

$$\mathsf{Point}_n \overset{\text{def}}{=} \mathsf{Nat}_n \to \mathsf{Bool}$$

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, in fact, the points that we will concern ourselves with will typically be partial functions.

Examples. Let us consider some simple, to some extent even trivial, examples of predicates and points. The examples are intended to illustrate the possible variety of dynamic properties that predicates and points may exhibit. As a first example consider the constant true point, which is defined as $\mathsf{p}_{\mathsf{true}} \overset{\text{def}}{=} \lambda\_.\mathsf{true}$. It is clearly total on the whole of Nat. A slightly more interesting point is the one which maps 0 to true and 1 to false and otherwise is undefined, it is given by

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

where $\bot \overset{\text{def}}{=} \mathbf{rec}\ f\ i.f\ i$ is the divergent point. The point p_2 is partial as it is only defined on Nat_2.

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.

$$\mathsf{tt}_0 \overset{\text{def}}{=} \lambda p.\mathsf{true}$$

In similar fashion we can define a variation hereof, tt_2, which inspects two components of its point, but nevertheless returns true.

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

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 also inspect the same component of its point more than once

$$\mathsf{red}_1 \overset{\text{def}}{=} \lambda p.p\,0; p\,0$$

which effectively means that the predicate performs redundant work. Another class of predicates are divergent predicates such as

$$\mathsf{div}_1 \overset{\text{def}}{=} \mathbf{rec}\ \mathit{div}\,p.\mathbf{if}\ p\,0\ \mathbf{then}\ \mathit{div}\,p\ \mathbf{else}\ \mathsf{false}$$

which diverges whenever the zero-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.

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

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 because the point only has one true component, whereas applying it to $\mathsf{p}_{\mathsf{true}}$ yields false.

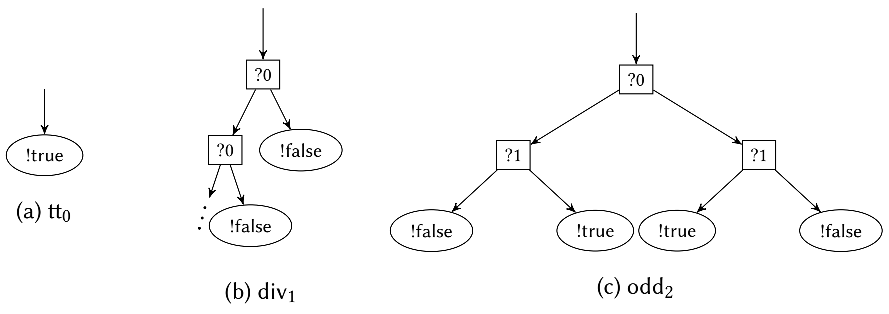


Fig. 11. Example Decision Tree Models

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). Each query has exactly two possible responses, because p returns a boolean value. 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, such that each interior node is labelled with a query $?i$ (for $i \in \mathbb{N}_n$) for which 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 decision trees are unrooted to account for the initial bit of computation which occurs in between the application of a predicate to p and the first query or answer.

Figure 11 depicts models of some of the example predicates given above. The model of tt_0 is simply an unrooted leaf (Figure 11a). The model of div_1 is an infinite left-branching tree (Figure 11b). The model of odd_2 is a complete binary tree (Figure 11c). A further example is the unconditionally divergent predicate $\mathsf{div} \overset{\text{def}}{=} \mathbf{rec}\ f\ p.f\ p$ whose model is empty.

Restrictions. In order to obtain a meaningful complexity result, we must constrain the predicates of interest. At one extreme, counting the number of solutions 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 do exhibit some interesting computational characteristics, because they at least inspect their provided point. However, we shall not concern ourselves with predicates like red_1 which performs redundant work, because any such work could be optimised away via common subexpression elimination (providing we guarantee that points are referentially transparent). 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 we have considered so far, the only ones that satisfy the above conditions are tt_2 and odd_n. In fact, the only predicates which satisfies these conditions are those predicates whose models form complete binary trees (as in Figure 11c). We call such predicates *n-standard.* We provide a rigorous definition of n-standard predicates in Section 6.3. To satisfy the first condition above, we also require that points always terminate on their defined domain Nat_n. We call a point that is defined on $0 < i < n$ an n-point.

6.2 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 $\Sigma \overset{\text{def}}{=} \{\mathsf{Branch} : 1 \twoheadrightarrow \mathsf{Bool}\}$ to be a singleton. The algorithm is then expressed as follows.

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

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

6.3 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 6.1 (label set). The label set Lab consists of queries parameterised by a natural number and answers parameterised by a boolean.

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

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 6.2 ((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.

As our goal is to reason about the time complexity of generic counting programs and their predicates it is helpful to decorate decision trees with timing data that records the number of machine steps taken.

Definition 6.3 (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 seek a way of relating predicates to decision trees. We do so by first defining an interpretation of configurations as decision trees.

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

Definition 6.4. The timed decision tree of a configuration is defined by the following equations

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

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

We can use $\mathcal{T}$ to define a construction procedure, $\mathcal{U}$, for untimed decision trees (Definition 6.2) as follows: $\mathcal{U}(P) \overset{\text{def}}{=} bs \mapsto \mathcal{T}(P)(bs).1$.

The following definition makes precise the notion of n-standard predicates.

Definition 6.5 (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.

$$dom(t) = \{bs : \mathbb{B}^* \mid |bs| \leq n\}$$

- Every leaf node in t is an answer node, i.e.

$$\text{For all } bs \in dom(t) \text{ if } |bs| = n \text{ then } t(bs) = !b \quad \text{for some } b \in \mathbb{B}$$

- 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 6.1 n-standard decision tree models are exactly those that form a complete binary tree such that each path contains no repeated queries. Note that the third condition in the definition requires only that there are no repeated queries along any path in the model, it does not, however, impose a particular ordering on those queries, meaning that a predicate like tt_2 from Section 6.1, which queries the second component of its point first, is an n-standard.

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 6.1, 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 denotation function for boolean value terms.

Definition 6.6 (n-point). For any $n > 0$ a closed value $p : \mathsf{Point}_n$ is said to be an n-point if

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

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 6.7 provides a procedure for computing the model of any n-point.

Definition 6.7. The denotation of an n-point p is the realisation of its operational behaviour

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

Using the model of n-points we define what it means for any two points to be *distinct*.

Definition 6.8. We say that any two n-points p_0 and p_1 are distinct if their denotations differ, i.e.

$$\exists j \in \mathbb{N}_n . \mathbb{P}[\![p_0]\!] j \neq \mathbb{P}[\![p_1]\!] j$$

6.4 Specification of Generic Counting

Now we have most of the prerequisites in place to be able to provide a rigorous specification of generic counting. The only missing piece of machinery is a way to determine the correct 'count' of an n-standard predicate, that is how many times it yields true.

Definition 6.9. 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. Except that 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 6.10. The counting function for a configuration is defined by the following equations.

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

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

We can now give a precise definition of generic count programs.

Definition 6.11 (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

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

There is a convenient correspondence between the size, n, of an n-standard predicate P and its maximum count value $\mathcal{C}(P)([])$. In the extreme case P answers true for every point. Since the model of P has height n it must have 2^n leaves, thus it follows that $\mathcal{C}(P)([]) \leq 2^n$. In fact, this correspondence generalises for all boolean lists bs of at most length n, that is $\mathcal{C}(P)(bs) \leq 2^{n-|bs|}$.

6.5 Complexity of Effectful Generic Counting

In this section we sketch the proof of correctness and asymptotic runtime bound for running the effectful generic counting program effcount on a predicate P. Full details are in Appendix A.

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

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

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

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

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

Proof. Proof by downwards induction on the length of bs and alternating between intensional and extensional reasoning as needed. □

6.6 Pure Generic Counting

Thus far we have shown that there exists an implementation of count in λ_h with a particular asymptotic runtime bound. Now, we are going to show that no implementation of count in λ_b can achieve this runtime bound. To obtain this result we exhibit two properties of the decision model

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

Together these two properties imply that every count program must have at 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, because in the worst case the path to two neighbouring leaves share exactly n edges in the model. We formalise the first property in Section 6.7 and the second in Section 6.8. First, we give an example of a pure generic count program and discuss better possible implementations.

The following program is a direct implementation of count in λ_b.

$$
\begin{aligned}
&\mathsf{purecount}_n : ((\mathsf{Nat}_n \to \mathsf{Bool}) \to \mathsf{Bool}) \to \mathsf{Nat}\\
&\mathsf{purecount}_n \overset{\text{def}}{=} \lambda P.\mathsf{count}'\ n\ \bot\\
&\qquad\qquad \mathbf{where}\ \mathsf{count}'\ 0 \qquad\quad p \overset{\text{def}}{=} \mathbf{if}\ P\,p\ \mathbf{then}\ 1\ \mathbf{else}\ 0\\
&\qquad\qquad\qquad\quad \mathsf{count}'\ (1+n')\ p \overset{\text{def}}{=} \quad \mathsf{count}'\ n'\ (\lambda i.\mathbf{if}\ i = n'\ \mathbf{then}\ \mathsf{true}\ \mathbf{else}\ p\ i)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad + \mathsf{count}'\ n'\ (\lambda i.\mathbf{if}\ i = n'\ \mathbf{then}\ \mathsf{false}\ \mathbf{else}\ p\ i)\\
&\qquad\qquad\qquad\quad \bot\ \_ \overset{\text{def}}{=} \mathbf{rec}\ f\ i.f\ i
\end{aligned}
$$

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 $\bot$, 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.

As mentioned in Section 1 more clever implementation is a counting variation of the search algorithm due to Berger [1990], which takes advantage of the observation that when a predicate queries a point, the point may reinvoke the predicate and use the result to determine the response to the initial query. We do not provide an example implementation here, but the interested reader may consult Escardó [2007], who demonstrates various refinements of Berger's algorithm in order to speed up the search procedure, though his setting is slightly different from ours as he considers exhaustive search on infinite sets.

At this point the reader may wonder why we cannot simply use known continuation passing style (CPS) [Hillerström et al. 2017] or monadic [Leijen 2017] transforms of effect handlers, or implement an interpreter for effect handlers [Hillerström and Lindley 2016] 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.

6.7 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 B.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 [Berger et al. 1998]).

Lemma 6.13. *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 any leaf in the model of P and construct its corresponding cursor $bs \in \mathbb{B}$. Suppose there exists an n-count program C which does not construct a point p_c for the leaf pointed to by bs. Moreover, let b denote the answer yielded by $P\ p_c$ if it were applied. Now negate b in the model of P and construct a canonical predicate P' using the modified model. We have that P and P' yield the same answers except at p_c, where $P'\ p_c = \neg b$. Moreover, it follows that $|\mathcal{C}(P')(bs) - \mathcal{C}(P)(bs)| = 1$, and because $[] \sqsubset bs$ we further obtain that $|\mathcal{C}(P')([]) - \mathcal{C}(P)([])| = 1$. Now there are two cases to consider:

(1) If $C\ P = C\ P'$ then C cannot be an n-count program, because $\mathcal{C}(P)([]) \neq \mathcal{C}(P')([])$, which contradicts the assumption.
(2) If $C\ P \neq C\ P'$ then we invoke a variation of Milner's Context Lemma [Milner 1977] which gives us that for any multi-hole context $\mathcal{F}[-]$ such that $\mathcal{F}[P] = C\ P$, the reduction sequence $\mathcal{F}[P] \rightsquigarrow^+ \textbf{return}\ V$ can be tracked by $\mathcal{F}[P'] \rightsquigarrow^* \textbf{return}\ V$. The Context Lemma tells us that $C\ P$ and $C\ P'$ both reduce to the same value, which contradicts the assumption.

□

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.

6.8 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 6.15 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 6.14 ($\twoheadrightarrow$). $\mathcal{E}[M] \twoheadrightarrow \mathcal{E}[N]$ iff $\mathcal{E}[M] \leadsto \mathcal{E}[N]$ and M is not of the form $p\ V$ where p is a point.

Definition 6.15 (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

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

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

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

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

Lemma 6.16. *Suppose P is an n-standard predicate, p is an n-point, and $f = \mathbb{P}[\![p]\!]$, then*

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

if and only if

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

Proof. For both directions the proof is by induction on n. □

The lemma tells us that every predicate application has an associated thread and vice versa. By Lemma 6.13 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 6.17. 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 are in a position to argue that no two distinct predicate applications can share computation, or in other words every section of their associated threads must be evaluated.

Lemma 6.18. *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.*

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 6.16 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 $\mathcal{E}_i[-]$

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

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

□

6.9 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 6.19. *For all $n > 0$ and every n-count program* $\mathsf{count} \in \lambda_b$*, and n-standard predicate the runtime of* $\mathsf{count}\ P$ *is at least*

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

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

Proof. By Lemma 6.13 we know that P is invoked at least 2^n times with distinct n-points. Moreover, by Lemma 6.18 we know that every reduction sequence arising from an application of P must be realised independently, which in terms of the model t of P means that in the worst case n edges are revisited. Taking $bs \in \mathbb{B}^*$ such that $|bs| \leq n$ to be any cursor into the model, we obtain that an edge is revisited $2^{n-|bs|}$ times.

Lemmas 6.13 and 6.18 are stated in terms of the operational semantics, so we invoke the correspondence Theorem 5.3 between $\rightsquigarrow$ and $\longrightarrow$ to obtain the result in terms of the abstract machine. □

7 CONCLUSIONS AND FUTURE WORK

In this paper we have taken a variation of call-by-value PCF and extended it with effect handlers. We have shown that this extension exhibits an asymptotically more efficient implementation of generic search than any possible implementation in the base language. Thus establishing that there exists a complexity gap between the two languages due to the presence of effect handlers. This result also extends to other control operators by appealing to existing results on interdefinability of handlers and other control operators [Forster et al. 2017; Piróg et al. 2019].

Our theorems no longer apply directly if we add an effect type system to λ_h, as the handler implementation of the counting program then requires the type of predicates to reflect the ability to perform effectful operations, whereas the pure PCF implementation does not. In future we plan to investigate how to account for effect type systems as well as further testing the robustness of our results by endowing the base language with primitives such as mutable state and exceptions.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. nnnnnnn and Grant No. mmmmmmm. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

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/

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

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

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

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

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.

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

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.

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

Robbie Daniels. 2016. *Efficient Generic Searches and Programming Language Expressivity.* Master's thesis. School of Informatics, the University of Edinburgh, Scotland.

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

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

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

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

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

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

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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

A PROOF DETAILS FOR THE COMPLEXITY OF EFFECTFUL GENERIC COUNTING

In this appendix we give proof details and artefacts for Theorem 6.12. Throughout this section we let H_{count} denote the handler definition of count, that is

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

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 A.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 $\mathsf{DT} \overset{\text{def}}{=} \mathbb{B}^* \rightharpoonup (\mathsf{Lab} \times \mathsf{Nat}) \times (\mathsf{Env} \times \mathsf{PureCont})$.

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 $\mathsf{env}(t) \overset{\text{def}}{=} bs \mapsto t(bs).2.1$ projects the environment, whilst the second $\mathsf{pure}(t) \overset{\text{def}}{=} bs \mapsto t(bs).2.2$ 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 6.4.

Definition A.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 $\mathcal{I}((\ell, s), (\gamma, \sigma)) \overset{\text{def}}{=} ((\ell, s+1), (\gamma, \sigma))$ 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 $\chi_{\mathsf{count}}(V) \overset{\text{def}}{=} (\emptyset[\mathit{pred} \mapsto [\![V]\!]\emptyset], H_{\mathsf{count}})$ 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 A.3 (computing machine configurations). For any given n-standard predicate P and a list of booleans bs, such that $|bs| \leq n$, we can compute machine configurations at points of interest during evaluation of count P.

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

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

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

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

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

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

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

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

- The function purecont computes the pure continuation.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The following lemma performs most of the heavy lifting for the proof of Theorem 6.12.

Lemma A.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*

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

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.

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

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

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

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

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

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

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

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

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

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

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

$$\begin{array}{cl}
 & \mathsf{arrive}(bs, P) \\
= & \text{(definition of arrive when } n < |bs|\text{)} \\
 & \langle V\ j \mid \gamma \mid (\sigma, \chi_{\mathsf{count}}(P)) :: \mathsf{residual}(bs, P)\rangle \\
\multicolumn{2}{l}{\text{where } ?j = \mathsf{labs}(t)(bs), \gamma = \mathsf{env}(t)(bs), \sigma = \mathsf{pure}(t)(bs), \text{ and } V = (\mathsf{env}^{\perp}(P), \lambda\_.\mathbf{do}\ \mathsf{Branch}\ \langle\rangle)} \\
\longrightarrow & \text{(M-App)} \\
 & \langle \mathbf{do}\ \mathsf{Branch}\ \langle\rangle \mid \gamma'[\_ \mapsto [\![j]\!]\gamma'] \mid (\sigma, \chi_{\mathsf{count}}(P)) :: \mathsf{residual}(bs, P)\rangle \\
 & \hfill \text{where } \gamma' = \mathsf{env}^{\perp}(P) \\
\longrightarrow & \text{(M-Handle-Op, } \chi_{\mathsf{count}}(P)^{\mathsf{Branch}} = \{\mathsf{Branch}\ \langle\rangle\ r \mapsto \cdots\}\text{)} \\
 & \left\langle \begin{array}{l} \mathbf{let}\ x_{\mathsf{true}} \leftarrow r\ \mathsf{true}\ \mathbf{in} \\ \mathbf{let}\ x_{\mathsf{false}} \leftarrow r\ \mathsf{false}\ \mathbf{in} \\ x_{\mathsf{true}} + x_{\mathsf{false}} \end{array} \mid \gamma[r \mapsto [\![(\sigma, \chi_{\mathsf{count}}(P))]\!]\gamma] \mid \mathsf{residual}(bs, P) \right\rangle \\
 & \hfill \text{where } \gamma = \mathsf{env}^{\perp}(P) \\
= & \text{(definition of } [\![-]\!] \text{ (1 value step))} \\
 & \left\langle \begin{array}{l} \mathbf{let}\ x_{\mathsf{true}} \leftarrow r\ \mathsf{true}\ \mathbf{in} \\ \mathbf{let}\ x_{\mathsf{false}} \leftarrow r\ \mathsf{false}\ \mathbf{in} \\ x_{\mathsf{true}} + x_{\mathsf{false}} \end{array} \mid \gamma' \mid \mathsf{residual}(bs, P) \right\rangle \\
 & \hfill \text{where } \gamma' = \gamma[r \mapsto (\sigma, \chi_{\mathsf{count}}(P))] \\
\longrightarrow & \text{(M-Let, definition of residual)} \\
 & \langle r\ \mathsf{true} \mid \gamma' \mid \mathsf{residual}(\mathsf{true} :: bs, P)\rangle \\
\longrightarrow & \text{(M-Resume, } [\![r]\!]\gamma' = (\sigma, \chi_{\mathsf{count}}(P)) \text{ (log } |\gamma'| = 1 \text{ environment operations))} \\
 & \langle \mathbf{return}\ \mathsf{true} \mid \gamma' \mid (\sigma, \chi_{\mathsf{count}}(P)) :: \mathsf{residual}(\mathsf{true} :: bs, P)\rangle
\end{array}$$

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

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

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

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

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

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

$$
\begin{array}{ll}
\longrightarrow^{\mathsf{steps}(t)(\mathsf{true}::bs)} & \text{(by Lemma A.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-AppCont)} \\
& \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 A.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-AppCont)} \\
& \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

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

□

The following theorem is a copy of Theorem 6.12.

Theorem A.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:*

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

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

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

Proof. The proof begins by direct calculation.

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

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

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

□

B PROOF DETAILS FOR THE NO SHORTCUTS LEMMA

The proof of Lemma 6.13 rely on the fact that any n-standard predicate has a canonical form. Section B.1 disseminate canonical predicates, whilst Section B.2 proves Lemma 6.13.

B.1 Canonical Predicates

The decision tree model (Definition 6.2) 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 $\mathcal{U}(P) \overset{\text{def}}{=} bs \mapsto \mathcal{T}(P)(bs).1$ denote the procedure for constructing an *untimed decision tree* of a given predicate P.

Definition B.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 B.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 → Bool, which is exactly the type of predicates. The procedure makes use of an auxiliary procedure body to generate the predicate body.

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

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 B.3. *Suppose P is an n-standard predicate then $P' \overset{\text{def}}{=}$ 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 B.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 B.1). The rest follows as by Lemma B.3. □

B.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 6.13.

Lemma B.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.

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

Then $P' =$ norm(t') constructs a canonical predicate, 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 B.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 $\mathcal{F}[P] \leadsto^m$ **return** V then $\mathcal{F}[P'] \leadsto^*$ **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 $\mathcal{F}[P] \leadsto^0$ **return** V from which it follows that $\mathcal{F}[-]$ is simply **return** V, thus it follows immediately that $\mathcal{F}[P'] \leadsto^0$ **return** V.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

and by the induction hypothesis, we get that

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

□

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

□