Cost Effects and Phase Distinctions*

Robert Harper

Spring 2024

1 Introduction

The lax framework for languages with effects used in Harper (2016) distinguishes *expressions*, which are effect-free, from *computations*, which may incur effects. The distinction is not based on types, but rather a way to resolve the tension between functional and effectful programming. This distinction was present in Algol-60, and has been refined and developed in more modern languages such as Haskell. Alternatively, the distinction can be eliminated, integrating evaluation and execution, as is done in the ML family of languages. The lax formulation may be considered prior in the sense that the integrated formulation (with no such distinction) may be obtained from it by consolidating expressions and computations, rendering them all as computations that may have effects.

The subject of this note is problem of assessing the *cost* of a program, a measure of the resources required to execute it. The concept of a resource is abstract, but a very typical usage is to count the number of critical operations taken by a computation, with the idea to facilitate a cost comparison between programs with the same behavior. The classic example is given by sorting the elements of a sequence according to some ordering. Here the figure of merit is *the number of comparisons* undertaken by a given sorting algorithm. As is well-known algorithms vary in this regard: insertion sort takes quadratically many comparisons for a given input, whereas merge sort takes a polylogarithmic number.

Textbook accounts of cost tend to emphasize machine models, with the cost being the number of instructions executed, or perhaps the amount of memory used, and are relatively loose about the relation between the high-level notation one actually uses and the machine code that it stands for. As the high-level notation becomes more sophisticated, for example including higher-order functions, the connection between the two levels becomes rather obscure, requiring a detailed understanding of a complex compiler to be fully accurate. Moreover, the machine model offers no support for abstraction. But if the figure of merit is not the use of a machine instruction, how is one to define the count?

The alternative considered here is to take a linguistic approach that naturally supports abstract cost measures. The idea is simple: equip a language with a step-counting computation that is integrated into the program to be analyzed, and define cost in terms of the number of steps (in the abstract sense) taken in a complete execution. Thus, in the case of sorting, each comparison is instrumented with a step count, so that the number of steps is the number of comparisons (in whatever sense may be relevant). Then the cost and outcome of a computation is defined by saying that it is equal to $\mathsf{step}^c(\mathsf{ret}(V))$ for some step count c and result value V.

*Copyright © Robert Harper. All Rights Reserved

Such equations are helpful for speaking about cost, but what if we wish to prove that two sorting algorithms are behaviorally equivalent, independently of their cost? The instrumentation of programs with step counting interferes with the expected equations, rendering insertion and merge sort inequivalent. What is needed is a means of distinguishing *extensional* (cost-insensitive) from *intensional* (cost-sensitive) behavior, so that any two correct sorting algorithms are extensionally equivalent, even though they may be intensionally distinct. This is achieved by drawing a *phase distinction* governing equations that expresses the desired distinction. Concretely, there are two phases, EXT and $\top$, that are to be thought of as propositions, the former asserting the extensional phase, and the latter, trivially true proposition, asserting the default phase. As the terminology suggests, the two phases are ordered by entailment, with $\mathsf{EXT} \vdash \top$, but not conversely.

Finally, step counting is a form of *effect*, incrementing a step count, which therefore is confined to the computation level. Besides signaling the imperative nature of step counting, the confinement of profiling to computations ensures that it is clear when the count is increased, a necessity for precise analysis.[1] When computations are not modally separated from expressions, as in the ML family of languages, then in effect even expressions are forms of computations, with explicit sequencing of evaluation order, and hence a well-defined cost.

2 Cost Accounting

For the sake of connecting with later work the statics of the lax language is formulated by two judgments:

1. $\Gamma \vdash M : A$: expression M with variables Γ evaluates to a value of type A;
2. $\Gamma \vdash E \mathrel{\dot{\sim}} A$: computation E with variable Γ returns a value of type A.

A variables is an expression standing for an unknown value of its declared type. The insistence on variables ranging over values implies that function application is by-value, which is required for the sake of analyzability. This is especially important in view of the fact that the type $\mathsf{comp}(A)$ classifies encapsulated computations that are executed by the bind computation: it is important to understand how and when that computation value is determined for the sake of analysis.

For present purposes the type structure of expressions is not specified explicitly, but one may assume it to include sums, products, functions, and inductive types such as the natural numbers and lists for the sake of formulating specific algorithms. To handle non-trivial patterns of recursion, such as are found in merge sort, it is necessary to introduce a "program counter" that bounds the number of recursive calls permitted in a given computation. For a given input, there is always a sufficiently large bound to achieve a final outcome, which is the same from that point onward. When the bound is too small, the algorithm prematurely terminates with no result, and is thus of an option type appropriate to the problem.

The language is parameterized by a monoid, $\mathbb{C}$, of costs that includes 0 and is closed under addition. The abstract syntax of computations includes the computation $\mathsf{step}^c(E)$ governed by the following statics:

$$\textsc{step}\quad \frac{\Gamma \vdash E \mathrel{\dot{\sim}} A \qquad (c \in \mathbb{C})}{\Gamma \vdash \mathsf{step}^c(E) \mathrel{\dot{\sim}} A}$$

[1]Of course there are even very small pieces of code whose cost bound defies analysis, the Collatz Function being one such example—no one know if it terminates, let alone how many steps it takes on a given input!

$$\textsc{ret}\ \frac{M \Downarrow V}{\mathsf{ret}(M) \longmapsto \mathsf{ret}(V)} \qquad \textsc{bnd-arg}\ \frac{M_1 \Downarrow \mathsf{comp}(E_1) \qquad E_1 \xmapsto{+c} E_1'}{\mathsf{bnd}(M_1;x.E_2) \xmapsto{+c} \mathsf{bnd}(\mathsf{comp}(E_1');x.E_2)} \qquad \textsc{bnd-ret}\ \frac{M_1 \Downarrow \mathsf{comp}(\mathsf{ret}(V_1))}{\mathsf{bnd}(M_1;x.E_2) \longmapsto [V_1/x]E_2}$$

$$\textsc{step}\ \frac{}{\mathsf{step}^c(E) \xmapsto{+c} E}$$

Figure 1: Cost Dynamics of Commands

That is, any computation may be wrapped with a step command that adds a specified count to the running total, without otherwise affecting the command's behavior.

The dynamics of step-counting may be given by transition rules of the form $E \xmapsto{+c} E'$, where $c \in \mathbb{C}$, indicating a transition step with associated cost c. (When c is omitted, it is understood to be 0.) Evaluation of expressions is given directly by the judgment $M \Downarrow V$, stating that M evaluates to the value V, with no cost accounting involved. The dynamics of computations is given in Figure 1. The use of labelled transitions is meant to convey that stepping engenders an effect, namely to increment the cost by a given amount.

Define complete execution of computation E with cost c, written $E \Downarrow^c V$, by the following rules:

$$\textsc{val}\ \frac{M \Downarrow V}{\mathsf{ret}(M) \Downarrow^0 V} \qquad \textsc{step}\ \frac{E \xmapsto{+c} E' \qquad E' \Downarrow^{c'} V}{E \Downarrow^{c+c'} V}$$

Thus, c is the cumulative cost defined by adding up the cost of the individual computation steps leading to the result. Finally, $E \Downarrow V$ is defined to mean $E \Downarrow^c V$ for some cost c, which is thereby being disregarded.

3 Equations

Phases are propositions ordered by entailment; $\psi \vdash \phi$ means that ψ being in phase ψ entails being in phase ϕ. Equations between computations are conditioned by the phase: the judgment $\Gamma \vdash^\phi E \equiv E' \mathrel{\dot{\sim}} A$ states that the two computations of type A are equal in phase ϕ. As a general principle, if $\psi \vdash \phi$, then any equation derivable in phase ϕ is also derivable in phase ψ, but not, in general, conversely. The intuition is that stronger phases impose more equations than would hold in the weaker. For present purposes there are two phases, EXT and $\top$, ordered by $\mathsf{EXT} \vdash \top$, with EXT indicating *extensional phase*, and $\top$, the "background phase," governs the *intensional phase*.

Some representative equations between expressions and computations are given in Figure 2. These include equations to consolidate stepping computations and govern their interaction with sequencing. Rule STEP-EXT states that costs are disregarded in the extensional phase, thereby isolating "pure behav-

$$\frac{\Gamma \vdash^{\phi} E \equiv E' \div A}{\Gamma \vdash^{\phi} \mathsf{comp}(E) \equiv \mathsf{comp}(E') : \mathsf{comp}(A)}\,\textsc{comp} \qquad \frac{\Gamma \vdash M : A \qquad \Gamma, x : A \vdash E \div B}{\Gamma \vdash^{\phi} \mathsf{bnd}(\mathsf{comp}(\mathsf{ret}(M));x.E) \equiv [M/x]E \div B}\,\textsc{bnd-ret}$$

$$\frac{\Gamma \vdash E_1 \div A_1 \qquad \Gamma, x : A_1 \vdash E_2 \div A_2 \qquad (c \in \mathbb{C})}{\Gamma \vdash^{\phi} \mathsf{bnd}(\mathsf{comp}(\mathsf{step}^c(E_1));x.E_2) \equiv \mathsf{step}^c(\mathsf{bnd}(\mathsf{comp}(E_1);x.E_2)) \div A_2}\,\textsc{bnd-step}$$

$$\frac{\Gamma \vdash M_1 : \mathsf{comp}(A_1) \qquad \Gamma, x_1 : A_1 \vdash E_2 \div A_2 \qquad \Gamma, x_1 : A_1, x_2 : A_2 \vdash E_3 \div A_3}{\Gamma \vdash^{\phi} \mathsf{bnd}(\mathsf{comp}(\mathsf{bnd}(M_1;x_1.E_2));x_2.E_3) \equiv \mathsf{bnd}(M_1;x_1.\,\mathsf{bnd}(\mathsf{comp}(E_2);x_2.E_3)) \div A_3}\,\textsc{bnd-bnd}$$

$$\frac{\Gamma \vdash E \div A}{\Gamma \vdash^{\phi} \mathsf{step}^0(E) \equiv E \div A}\,\textsc{step-zero} \qquad \frac{\Gamma \vdash E \div A \qquad (c, c' \in \mathbb{C})}{\Gamma \vdash^{\phi} \mathsf{step}^c(\mathsf{step}^{c'}(E)) \equiv \mathsf{step}^{c+c'}(E) \div A}\,\textsc{step-step} \qquad \frac{\Gamma \vdash E \div A}{\Gamma \vdash^{\mathsf{EXT}} \mathsf{step}^c(E) \equiv E \div A}\,\textsc{step-ext}$$

Figure 2: Some Equations

ior" independent of any cost accounting. Omitted equations include specifying that equality is reflexive, symmetric, and transitive, and that it is compatible with the constructs of the language.

Adding (eager) products to the language is straightforward, as they are formulated entirely at the pure term level, rather than as computations. Sums and inductive types are more interesting in that they involve cost accounting.

Exercise 1. *Suppose that (eager) sum types are added to the language, with their elimination forms for both values and computations. What equations would you expect to govern these constructs? Similarly, suppose that the type of natural numbers is added, with primitive recursion for both values and computations. What equations govern this construct?*

Exercise 2. *The equations given in Figure 2 are intended to hold regardless of what other forms of effect may be present in the language. Explain why it is not reasonable to add the following equation to the theory:*

$$\frac{\Gamma \vdash M : \mathit{comp}(A) \qquad \Gamma, x : A \vdash E \div B \qquad (c \in \mathbb{C})}{\Gamma \vdash^{\phi} \mathit{bnd}(M;x.\,\mathit{step}^c(E)) \equiv \mathit{step}^c(\mathit{bnd}(M;x.E)) \div B}\,\textsc{bnd-body}$$

Hint: *Consider that there may be an error command that aborts execution.*

4 Semantics

The equations given in Section 3 may be justified using Kripke-style binary logical relations to define semantic equality of expressions and computations by induction on the structure of their types. The Kripke worlds are phases, ordered by entailment, which in the cost setting means that equations true in intensional phase hold also in the extensional phase, but not *vice-versa*.

Semantic equality of expressions and computations is expressed by the following families of relations indexed by phases:

1. $M \doteq M' \in A\ [\phi]$: logical equivalence of expressions of type A in phase ϕ.
2. $E \doteq E' \in \tilde{A}\ [\phi]$: logical equivalence of computations returning type A in phase ϕ.

At the expression level the definitions of the expression relations follows the general principles of Kripke logical relations.[2] In the case of the computation type, $\mathsf{comp}(A)$, the definition of logical equivalence is as follows:

$$M \doteq M' \in \mathsf{comp}(A)\ [\phi] \quad \text{iff} \quad M \Downarrow \mathsf{comp}(E),\ M' \Downarrow \mathsf{comp}(E'),\ \text{and}\ E \doteq E' \in \tilde{A}\ [\phi].$$

The clauses for product, sum, function, and answer types may be adapted from those given in Harper (2024), including especially the treatment of function types considering all "future" worlds.

Exercise 3. *Give the definition of semantic equality for answer, product, and function types, and extend these to nullary and binary sum types and the type of natural numbers.*

Lemma 1 (Head Expansion of Expressions)**.** *Semantic equality of expressions is closed under head expansion: if $M \doteq M' \in A\ [\phi]$, then if $N \longmapsto M$, then $N \doteq M' \in A\ [\phi]$, and if $N' \longmapsto M'$, then $M \doteq N' \in A\ [\phi]$.*

Proof. Straightforward, given that the relations are characterized by evaluation of M and M'. □

At the computation level logical equivalence is phase-sensitive, and defined as follows:

$$E \doteq E' \in \tilde{A}\ [\phi] \quad \text{iff} \quad E \Downarrow^{c} V,\ E' \Downarrow^{c'} V',\ V \doteq V' \in A\ [\phi]\ \text{and}\ (c = c'\ \text{or}\ \phi \vdash \mathsf{EXT})$$

Thus, in extensional phase, cost has no influence on semantic equality. The condition on costs could be rephrased as stating $\phi \nvdash \mathsf{EXT}$ implies $c = c'$—outside of the extensional phase, the costs must coincide.

Lemma 2 (Anti-Monotonicity)**.** *Suppose that $\psi \vdash \phi$. Then for all types A, if $M \doteq M' \in A\ [\phi]$, then $M \doteq M' \in A\ [\psi]$, and if $E \doteq E' \in \tilde{A}\ [\phi]$, then $E \doteq E' \in \tilde{A}\ [\psi]$.*

Thus, every intensional semantic equality is also an extensional one, but the converse need not be true because in the intensional phase steps are taken into account.

Exercise 4. *Prove the anti-monotonicity lemma for computations, and for expressions of function type, by induction on the structure of A.*

Semantic equality is extended to open terms by *functionality*, regarding such terms as functions of their free variables. First, semantic equality of expressions is extended to substitutions by defining $\gamma \doteq \gamma' \in \Gamma\ [\phi]$ to means $\gamma(x) \doteq \gamma'(x) \in \Gamma(x)\ [\phi]$ for each variable x declared in Γ. Using this, semantic equality of open expressions and open computations is defined as follows:

$$\Gamma \gg^{\phi} M \doteq M' \in A \ \text{iff}\ \gamma \doteq \gamma' \in \Gamma\ [\phi]\ \text{implies}\ \hat{\gamma}(M) \doteq \widehat{\gamma'}(M') \in A\ [\phi]$$
$$\Gamma \gg^{\phi} E \doteq E' \in \tilde{A} \ \text{iff}\ \gamma \doteq \gamma' \in \Gamma\ [\phi]\ \text{implies}\ \hat{\gamma}(E) \doteq \widehat{\gamma'}(E') \in \tilde{A}\ [\phi].$$

Theorem 3 (Reflexivity)**.** *1. If $\Gamma \vdash M : A$, then $\Gamma \gg^{\phi} M \doteq M \in A$.*

[2]As presented in Harper (2024), albeit for the unary case.

2. *If $\Gamma \vdash E \stackrel{\cdot}{\sim} A$, then $\Gamma \gg^{\phi} E \doteq E' \in \tilde{A}$.*

Proof. These are proved simultaneously by induction on derivations. For example, consider the introduction rule for the type $\mathsf{comp}(A)$. Let ϕ be a phase, and suppose that $\gamma \doteq \gamma' \in \Gamma\ [\phi]$, with the intent to show that $\mathsf{comp}(\hat{\gamma}(E)) \doteq \mathsf{comp}(\widehat{\gamma'}(E')) \in \mathsf{comp}(A)\ [\phi]$. It suffices to show $\hat{\gamma}(E) \doteq \widehat{\gamma'}(E') \in \tilde{A}\ [\phi]$, which follows directly from the induction hypothesis.

As another example, consider the rule STEP given above. Fixing ϕ and $\gamma \doteq \gamma' \in \Gamma\ [\phi]$, the goal is to show that $\mathsf{step}^c(\hat{\gamma}(E)) \doteq \mathsf{step}^c(\widehat{\gamma'}(E')) \in \tilde{A}\ [\phi]$. By the inductive hypothesis we have $\hat{\gamma}(E) \doteq \widehat{\gamma'}(E') \in \tilde{A}\ [\phi]$, which means that

1. $\hat{\gamma}(E) \Downarrow^{d} V$ for some cost d and value V;
2. $\widehat{\gamma'}(E') \Downarrow^{d'} V'$ for some cost d' and value V';
3. $V \doteq V' \in A\ [\phi]$.
4. either $d = d'$ or $\phi \vdash \mathsf{EXT}$.

Now

1. $\mathsf{step}^c(\hat{\gamma}(E)) \Downarrow^{c+d} V$;
2. $\mathsf{step}^c(\widehat{\gamma'}(E')) \Downarrow^{c+d'} V'$.

If $\phi \vdash \mathsf{EXT}$, the result is immediate, otherwise note that if $d = d'$, then $c + d = c + d'$. □

Exercise 5. *Give the cases of the proof of Theorem 3 for bnd and ret.*

The derivable equations are validated by the Fundamental Theorem of Logical Relations, which states that derivable equations are semantically valid.

Theorem 4 (Fundamental Theorem). *1. If $\Gamma \vdash^{\phi} M \equiv M' : A$, then $\Gamma \gg^{\phi} M \doteq M' \in A$.*

2. *If $\Gamma \vdash^{\phi} E \equiv E' \stackrel{\cdot}{\sim} A$, then $\Gamma \gg^{\phi} M \doteq M' \in \tilde{A}$.*

Exercise 6. *Prove the cases of Theorem 4 for the rules given in Figure 2. Where does your proof attempt break down in extending your argument to the equation considered in Example 2?*

One consequence of the FTLR is *non-interference*, which states that extensional behavior is not affected by intensional distinctions.

Theorem 5 (Non-Interference). *If $\vdash^{\top} F : \mathsf{comp}(\mathsf{unit}) \to \mathsf{comp}(\mathsf{ans})$, then F is extensionally a constant function.*

Proof. Suppose that $\vdash^{\top} F : \mathsf{comp}(\mathsf{unit}) \to \mathsf{comp}(\mathsf{ans})$. Then by reflexivity

$$F \doteq F \in \mathsf{comp}(\mathsf{unit}) \to \mathsf{comp}(\mathsf{ans})\ [\top],$$

and so by anti-monotonicity,

$$F \doteq F \in \mathsf{comp}(\mathsf{unit}) \to \mathsf{comp}(\mathsf{ans})\ [\mathsf{EXT}].$$

Now $E \doteq E' \in \tilde{\mathsf{unit}}\ [\mathsf{EXT}]$ for any E, E', so

$$\mathsf{ap}(F; \mathsf{comp}(E)) \doteq \mathsf{ap}(F; \mathsf{comp}(E')) \in \mathsf{comp}(\mathsf{ans})\ [\mathsf{EXT}].$$

But then $\mathsf{ap}(F; \mathsf{comp}(E)) \Downarrow \mathsf{comp}(E_0)$, $\mathsf{ap}(F; \mathsf{comp}(E')) \Downarrow \mathsf{comp}(E_0')$, and $E_0 \doteq E_0' \in \tilde{\mathsf{ans}}\ [\mathsf{EXT}]$, which is to say that both sides evaluate to the same answer, yes or no as the case may be. Thus, F is extensionally equal to the constant function $\lambda(u.\, \mathsf{comp}(\mathsf{ret}(\mathsf{yes})))$ or $\lambda(u.\, \mathsf{comp}(\mathsf{ret}(\mathsf{no})))$. □

References

Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, Cambridge, England, Second edition, 2016.

Robert Harper. Kripke-style logical relations for normalization. Unpublished lecture note, Spring 2024. URL `https://www.cs.cmu.edu/~rwh/courses/atpl/pdfs/kripke.pdf`.

Yue Niu, Jonathan Sterling, Harrison Grodin, and Robert Harper. A cost-aware logical framework. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–31, January 2022. ISSN 2475-1421. doi: 10.1145/3498670. URL `https://dl.acm.org/doi/10.1145/3498670`.