CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Repository@Nottingham

Call-by-Need Is Clairvoyant Call-by-Value

JENNIFER HACKETT, University of Nottingham, UK
GRAHAM HUTTON, University of Nottingham, UK

Call-by-need evaluation, also known as lazy evaluation, provides two key benefits: compositional programming and infinite data. The standard semantics for laziness is Launchbury's natural semantics [1993], which uses a heap to memoise the results of delayed evaluations. However, the stateful nature of this heap greatly complicates reasoning about the operational behaviour of lazy programs. In this article, we propose an alternative semantics for laziness, *clairvoyant evaluation*, that replaces the state effect with nondeterminism, and prove this semantics equivalent in a strong sense to the standard semantics. We show how this new semantics greatly simplifies operational reasoning, admitting much simpler proofs of a number of results from the literature, and how it leads to the first *denotational cost semantics* for lazy evaluation.

CCS Concepts: • **Software and its engineering** → **Functional languages**; • **Theory of computation** → **Operational semantics**; **Denotational semantics**.

Additional Key Words and Phrases: lazy evaluation

ACM Reference Format:
Jennifer Hackett and Graham Hutton. 2019. Call-by-Need Is Clairvoyant Call-by-Value. *Proc. ACM Program. Lang.* 3, ICFP, Article 114 (August 2019), 23 pages. https://doi.org/10.1145/3341718

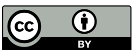
1 INTRODUCTION

Lazy evaluation [Henderson and Morris 1976] is an evaluation strategy that combines on-demand evaluation with memoisation. When a variable is bound, it is stored in the environment in an unevaluated form called a "thunk", with the evaluation being delayed until the result is required to proceed. When the thunk is evaluated, it is replaced by the result of its evaluation, avoiding any repetition of work if the value of the variable were to be needed again.

Laziness gives us two key benefits. Firstly, it supports a more *compositional* programming style than strict (non-lazy) evaluation, with chains of function compositions being computable in a pipelined manner, without the need to ever construct the entire intermediate results [Hughes 1989]. And secondly, on-demand evaluation allows us to work with *infinite data* [Turner 1982] and *circular definitions* [Bird 1984], both of which would lead to non-termination in a strict language.

Unfortunately, these benefits of lazy evaluation come at the price of tractable cost analysis, and one of the main reasons given in favour of strict functional languages is that their operational behaviour is easier to understand. Whereas in a strict program terms are evaluated more-or-less when they appear, in a lazy program terms are evaluated only when they are required, making it hard to predict exactly how much time a given program will take to run. This has led to a number of different approaches to analysing lazy programs, such as [Bjerner and Holmström 1989; Jost

Authors' addresses: Jennifer Hackett, School of Computer Science, University of Nottingham, UK, jennifer.hackett@nottingham.ac.uk; Graham Hutton, School of Computer Science, University of Nottingham, UK, graham.hutton@nottingham.ac.uk.



This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2019 Copyright held by the owner/author(s).
2475-1421/2019/8-ART114
https://doi.org/10.1145/3341718

et al. 2010; Moran and Sands 1999a; Okasaki 1999]. However, many of these theories are rather involved, and it is hard for them to compete with the simplicity of strict evaluation.

The key issue with understanding lazy evaluation is that while the external interface provided to the programmer is declarative and pure, the internal implementation is stateful. Evaluating a term can result in updates to pre-existing environment variables, meaning that the environment is no longer just an input to the evaluation, but an *output* as well. As a result, modelling lazy evaluation requires us to thread state through everything we do, vastly complicating our reasoning.

In this article, we show how to make a more theoretically tractable model of laziness that uses a limited form of nondeterminism in its evaluation. We call this model *clairvoyant* evaluation, based on the loanword "clairvoyance" of French origin, meaning "foresight". In clairvoyant evaluation, variables are evaluated when they are bound, as in a strict language, but only if they will be *needed* in the current evaluation. If a given variable will not be used in the evaluation, it will be discarded. This allows us to do away with messy state-threading and work in a much cleaner setting, while still remaining faithful to efficiency. Specifically, we make the following contributions:

- We present an alternative characterisation of lazy evaluation, called *clairvoyant call-by-value*, that replaces the internal sharing with an *external nondeterminism*, proving this alternate semantics cost-equivalent to the standard call-by-need semantics. This cost-equivalence is what we mean when we say that call-by-need *is* clairvoyant call-by-value.
- We use clairvoyant call-by-value as a stepping stone to create a sound and adequate *denotational* semantics for laziness that avoids any explicit heap-passing. As far as we are aware, this is the first denotational cost semantics that faithfully reflects the cost of sharing.
- We demonstrate the utility of our theory on a number of examples from the literature, including a much simplified proof of the time-safety of common subexpression elimination.

This work is directly inspired by that of Maraist et al. [1999], who showed that call-by-need is equivalent to a variant of call-by-value that allows for discarding. However, their work is based on an axiomatic presentation of both call-by-value and call-by-need that does not correspond directly to an evaluation strategy. Our work builds upon theirs by proving the correspondence on the level of *costed big-step semantics*, showing that the equivalence extends to an equivalence of program costs. This allows us to apply our results to matters of program efficiency rather than just correctness, as well as to develop our denotational cost semantics.

Section 2 covers the necessary background. Section 3 introduces the intuition behind the idea of clairvoyant evaluation, and shows how the idea can be implemented by means of nondeterminism. Section 4 proves the cost-equivalence of clairvoyant call-by-value with the standard call-by-need semantics due to Launchbury [1993]. Section 5 gives the denotational semantics for clairvoyant call-by-value along with soundness and adequacy theorems. Section 6 demonstrates various ways in which our work can be applied. Section 7 discusses how the minimal language we consider can be extended with more practical programming features. Section 8 covers related work, and Section 9 summarises and discusses possible avenues for future work.

2 BACKGROUND

2.1 Launchbury's Natural Semantics

The standard operational semantics for laziness is due to Launchbury [1993]. It is given in the form of a big-step, or natural semantics, with a reduction relation $\Downarrow^N$ defined over pairs consisting of a heap Γ and a term e. The semantics is defined for the following language:

$$\begin{aligned} x &\in Var \\ e &\in Exp \quad ::= \quad x \end{aligned}$$

$$\Gamma : \lambda x.e \Downarrow_0^N \Gamma : \lambda x.e \qquad \text{(LAM)}$$

$$\frac{\Gamma : e \Downarrow_k^N \Delta : \lambda y.e' \quad \Delta : e'[x/y] \Downarrow_l^N \Theta : z}{\Gamma : e\ x \Downarrow_{k+l+1}^N \Theta : z} \qquad \text{(APP)}$$

$$\frac{\Gamma, x \mapsto e_1 : e_2 \Downarrow_k^N \Delta : z}{\Gamma : \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \Downarrow_{k+1}^N \Delta : z} \qquad \text{(LET)}$$

$$\frac{\Gamma : e \Downarrow_k^N \Delta : z}{\Gamma, x \mapsto e : x \Downarrow_{k+1}^N \Delta, x \mapsto z : \hat{z}} \qquad \text{(VAR)}$$

Fig. 1. Launchbury's Natural Semantics for Lazy Evaluation

$$\begin{array}{ll} | & \lambda x.e \\ | & e\ x \\ | & \textbf{let}\ x = e\ \textbf{in}\ e \end{array}$$

This language differs from the standard untyped lambda-calculus in two ways. Firstly, we include a recursive let-binding construct. And secondly, we only permit functions to be applied to variables. The effect of both of these changes is that heap allocation *only* occurs at a let-binding, and that programs can be circular as in lazy languages like Haskell.

The reduction relation $\Downarrow^N$ is defined inductively by the rules in Figure 1. This presentation deviates from the standard presentation in two ways. Firstly, for simplicity we disallow simultaneous let-binding. Secondly, and more significantly, we have added a cost index to our relation, which in this case is given simply by the size of the derivation tree, ignoring the leaves. In this manner, we read $\Gamma : e \Downarrow_k^N \Delta : z$ as "the term e in heap Γ reduces in k steps to the value z in heap Δ".

We adopt the following notational conventions: we omit the superscript (such as N) specifying the semantics when it is clear from context; we write $\Gamma : e \Downarrow_k$ to mean that there exists some Δ, z such that $\Gamma : e \Downarrow_k \Delta, z$; we omit empty heaps, writing $e \Downarrow_k$ to mean $\emptyset : e \Downarrow_k$.

The rules themselves can be understood as follows:

- The LAM rule states that an abstraction is already in value form.
- The APP rule states that to evaluate an application, we must first evaluate the function, then apply beta reduction, and finally evaluate the result.
- The LET rule states that to evaluate a let binding, we move the binding into the heap and then evaluate the body. Note that we do not evaluate the bound expression at this point.
- The VAR rule states that to evaluate a variable, we look it up in the heap, evaluate it, and then store the result back in the heap. We then proceed with that same result, alpha-renamed so as to avoid unwanted name capture.

This operational semantics captures both *on-demand evaluation* (in that let-bound terms are not evaluated until it is necessary) and *sharing* (in that work is not duplicated unnecessarily). This is the model of evaluation used by lazy languages such as Haskell. Note the each memory cell in the heap can be written to *twice*: once when the cell is allocated, and again when it is read for the first time. However, despite this apparent statefulness, the language itself is pure; the statefulness is essentially an implementation detail that is hidden from the user by an abstraction barrier.

$$\Gamma : \lambda x.e \Downarrow_0^V \Gamma : \lambda x.e \quad \text{(Lam)}$$

$$\frac{\Gamma : e \Downarrow_k^V \Delta : \lambda y.e' \quad \Delta : e'[x/y] \Downarrow_l^V \Theta : z}{\Gamma : e\ x \Downarrow_{k+l+1}^V \Theta : z} \quad \text{(App)}$$

$$\Gamma, x \mapsto z : x \Downarrow_1^V \Gamma, x \mapsto z : \hat{z} \quad \text{(Var')}$$

$$\frac{\Gamma : e_1 \Downarrow_k^V \Delta : z_1 \quad \Delta, x \mapsto z_1 : e_2 \Downarrow_l^V \Theta : z_2}{\Gamma : \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \Downarrow_{k+l+1}^V \Theta : z_2} \quad \text{(Let')}$$

Fig. 2. Call-By-Value Semantics for the Same Language

2.2 Call-By-Value

The contrasting evaluation strategy to lazy evaluation is that of *strict* evaluation, also known as call-by-value. This is the model of evaluation that is used by the vast majority of programming languages, in which no evaluation is delayed except under a lambda abstraction. The rules in Figure 2 define call-by-value evaluation for our language. We still utilise a heap in this semantics, but now the heap only contains *evaluated* terms; this invariant is maintained in the modified Let' rule, where the let-bound term is evaluated before proceeding to the body. The Var' rule is now significantly simplified, as no longer has to deal with the case where the variable is bound to an unevaluated term. The Lam and App rules do not change, as neither involves the heap.

Because this strategy evaluates eagerly, there is a potential slowdown as compared to lazy evaluation, as call-by-value will evaluate terms that may not actually be required for the purposes of producing the final result. This same property means that call-by-value is *less terminating* than call-by-need, as there could be some diverging subterm that a call-by-need strategy would avoid evaluating. However, despite these disadvantages, call-by-value is typically favoured by language designers because it is much easier to understand and reason about.

2.3 Laziness: The Good, the Bad and the Ugly

Lazy evaluation brings a number of advantages. In addition to the fact that it is more terminating and potentially faster than call-by-value, it also allows for programs to manipulate potentially infinite data structures. This is because an infinite object will only be present in memory as far as it is *needed* to produce a result. The three benefits of better termination, faster evaluation and infinite data structures together represent the *good* of laziness.

On the other hand, there are also disadvantages to consider. The chief drawback of laziness is that it is hard to reason about the efficiency of lazy programs, because it is often unclear precisely when a particular term will be evaluated. As a consequence of this, while there have been attempts to develop general theories of efficiency for lazy languages (for example, see Bjerner and Holmström [1989]), these theories are typically far more involved than their call-by-value counterparts. This difficulty of reasoning about efficiency represents the *bad* of laziness.

There is another undesirable characteristic of laziness that only becomes apparent when we move to a *denotational* setting. In order to fully capture sharing in our semantics, the state of the heap must be threaded through every operation, effectively forcing us to work within an ambient state monad; for an example, see Josephs [1989]. The need to give a stateful interpretation to an ostensibly pure language is not just bad – it is downright *ugly*!

2.4 Improvement Theory

Because the cost of evaluating lazy programs can be unintuitive, it is generally useful to have some formal or semi-formal theory to appeal to when reasoning about cost. One approach to this is *improvement theory* [Moran and Sands 1999a]. Improvement theory is an *inequational* theory of efficiency that allows for algebraic proofs of efficiency relationships between call-by-need programs. This theory is based on the notion of *contextual improvement*, a cost-aware analogue of contextual equivalence à la Morris [1969], which is defined as follows:

$$e \gtrsim e' \;\Leftrightarrow\; \forall \mathbb{C}.\mathbb{C}[e] \Downarrow_k \Rightarrow \mathbb{C}[e'] \Downarrow_{\leq k}$$

That is, a term e is *improved* by e', written as $e \gtrsim e'$, if in all contexts the cost of evaluating e' is at most the cost of evaluating e. This relation tells us when it is safe to replace e with e' without fear of degrading performance in terms of relative execution costs. We say that e is *cost-equivalent* to e', written as $e \lessgtr e'$, if we have both $e \gtrsim e'$ and $e' \gtrsim e$.

This theory has more applications than simply efficiency; it can also be used to demonstrate the *total correctness* of program transformations performed in the fold-unfold style [Sands 1997]. Improvement theory was first developed by Sands [1991] in a call-by-name setting, and later extended to call-by-need time cost by Moran and Sands [1999a]. Gustavsson and Sands [1999, 2001] developed a corresponding theory for space costs.

In more recent years, improvement theory has been used, along with related techniques, to verify the time-safety of a number of well-known program optimisation schemes, including short-cut fusion [Hackett and Hutton 2018; Seidel and Voigtländer 2011] and the worker-wrapper transformation [Hackett and Hutton 2014]. The theory has also been further developed by Schmidt-Schauß and Sabel [2015a,b], who develop an improvement theory for the LR-calculus — an extended higher-order lambda calculus with call-by-need evaluation that models the core language of Haskell — and use it to verify the time-safety of common subexpression elimination. A corresponding theory for space cost was developed by Schmidt-Schauß and Dallmeyer [2018].

3 CLAIRVOYANT EVALUATION

As we noted above, the primary disadvantage of lazy evaluation is that it is hard to reason about with respect to efficiency concerns. The reason for this is simple: while the statefulness of the heap is hidden from the programmer by an abstraction barrier, when we consider the issue of efficiency we see that the abstraction is *leaky*. In the case of correctness, we can identify an unevaluated term with its result, but for efficiency this distinction is important, as whether we evaluate a term or not tells us whether we need to consider its cost or not.

On the other hand, while it matters *whether* a let-bound term is evaluated, it doesn't actually matter *when*, at least as far as time efficiency is concerned: a term will take as long to evaluate today as it will tomorrow. This means that if we somehow knew which terms were going to be needed, we could evaluate them ahead of time without incurring any extra total cost — this is the intuition behind strictness analysis [Wadler and Hughes 1987]. In other words, if we could look into the future and ask "do we need to know the result of this computation?", we would be able to evaluate anything necessary eagerly and discard everything else, and the resulting time cost would be the same. We call this evaluation strategy *clairvoyant call-by-value*.

3.1 Realising Clairvoyance with Nondeterminism

On the face of it, clairvoyant call-by-value is impossible, because in general we don't know when terms will be evaluated prior to evaluation. However, we can still realise a version of the idea by borrowing a trick from automata theory. In automata theory, a nondeterministic automaton differs

$$\Gamma : \lambda x.e \Downarrow_0^{CV} \Gamma : \lambda x.e \quad \text{(Lam)}$$

$$\frac{\Gamma : e \Downarrow_k^{CV} \Delta : \lambda y.e' \quad \Delta : e'[x/y] \Downarrow_l^{CV} \Theta : z}{\Gamma : e\ x \Downarrow_{k+l+1}^{CV} \Theta : z} \quad \text{(App)}$$

$$\Gamma, x \mapsto z : x \Downarrow_1^{CV} \Gamma, x \mapsto z : \hat{z} \quad \text{(Var')}$$

$$\frac{\Gamma : e_1 \Downarrow_k^{CV} \Delta : z_1 \quad \Delta, x \mapsto z_1 : e_2 \Downarrow_l^{CV} \Theta : z_2}{\Gamma : \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \Downarrow_{k+l+1}^{CV} \Theta : z_2} \quad \text{(Let')}$$

$$\frac{\Gamma : e_2 \Downarrow_k^{CV} \Delta : z_2}{\Gamma : \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \Downarrow_{k+1}^{CV} \Delta : z_2} \quad \text{(Skip)}$$

Fig. 3. Clairvoyant Call-By-Value Semantics

from a deterministic automaton in that a machine state may have several potential successor states, rather than just one. This allows the automaton to make choices during its execution, resulting in a *tree* of states. A nondeterministic automaton is said to accept its input if there is some sequence of choices that lead it to an accepting state. This can be seen as a limited form of clairvoyance, where the automaton always picks the choice that leads to an accepting state, if there is one; that is, it is a machine that is equipped with an oracle that can answer questions of the form "will making this choice result in evaluation getting stuck at some point in the future?".

Using this idea, we can implement clairvoyant call-by-value by means of a nondeterministic *semantics*, in which a nondeterministic choice takes place at every let binding. We take the call-by-value semantics from Figure 2 and extend it with a Skip rule that allows for the discarding of a binding, resulting in the semantics in Figure 3. The reduction relation will hold whenever there is *some* tree that derives it, much like a nondeterministic automaton.

To interpret this reduction relation algorithmically as an evaluation function, we should imagine that our evaluator is equipped with a limited oracle that allows it to ask questions of the form "will this evaluation fail if I discard this binding?". If the answer is "yes", then it is forced to apply the Let' rule; otherwise, it uses Skip. (Note that this is not a local nondeterminism: it really makes the globally best choice.) This process will then result in the derivation tree with the minimum cost, which we can think of as the "maximally lazy" computation cost.

4 COST-EQUIVALENCE WITH LAUNCHBURY'S SEMANTICS

Given that our clairvoyant call-by-value semantics differs quite significantly from the standard semantics for call-by-need evaluation, it is necessary to formally relate the two in some way. Clearly, the two semantics cannot be perfectly equivalent, as clairvoyant call-by-value can result in multiple possible results, whereas call-by-need cannot. However, there is a close relationship between the two semantics, as captured by the following theorem:

Theorem 4.1 (Moral Equivalence of CBN and Clairvoyant CBV). *Given a term e and a heap* Γ *that consists only of values, we have the following two properties:*

(i) If $\Gamma : e \Downarrow_k^N$ *then* $\Gamma : e \Downarrow_k^{CV}$

(ii) If $\Gamma : e \Downarrow_k^{CV}$ *then there is some* $k' \leq k$ *such that* $\Gamma : e \Downarrow_{k'}^N$

Proof.

(i) We prove the stronger statement that if $\Gamma : e \Downarrow_k^N \Delta : z$ then $\Gamma : e \Downarrow_k^{CV} \Delta' : z$, where Δ' is a version of Δ with all the bindings to non-values removed. We proceed by induction on the size of the proof of $\Gamma : e \Downarrow_k^N \Delta : z$, applying a case analysis on the last step of the proof:
 - If the last step in the proof is Lam, the result is trivially true.
 - If the last step in the proof is App, then e must be of the form $f\,y$. We know that there must be some Δ, e' such that $\Gamma : f \Downarrow_{k_1}^N \Theta : \lambda y'.e'$ and $\Theta : e'[y/y'] \Downarrow_{k_2}^N \Delta : z$, where $k = k_1 + k_2 + 1$. We now construct a new heap Θ' based on the following rules:
   1. If Θ maps x to a value v, then Θ' will map x to the same value.
   2. If Θ maps x to an unevaluated term e, and x is not evaluated in the evaluation of $\Theta : e'[y/y']$, then Θ' will contain no binding for x.
   3. If Θ maps x to an unevaluated term e, and x is evaluated in the evaluation of $\Theta : e'[y/y']$ to a value z, then Θ' will map x to z.

   It is straightforward to see that $\Theta' : e'[y/y'] \Downarrow_{k_2 - l}^N$, where l is the cost of the variables evaluated in the original sub-derivation. Furthermore, Θ' is a heap consisting only of values, so we can apply the induction hypothesis to obtain $\Theta' : e'[y/y'] \Downarrow_{k_2 - l}^{CV}$.

   Next, we apply the induction hypothesis to the derivation of $\Gamma : f \Downarrow_{k_1}^N \Theta : \lambda y'.e'$ to obtain a derivation of $\Gamma : f \Downarrow_{k_1}^{CV} \Theta'' : \lambda y'.e'$, where Θ'' is a version of Θ that omits bindings to non-values. In this derivation, therefore, we can conclude that any binding that appears in Θ' but not in Θ'' must have been the subject of a Skip rule. By changing these rules to Let' and including the sub-evaluations we removed from the evaluation of $\Theta : e'[y/y']$ (after applying the induction hypothesis to these evaluations), we can construct a derivation of $\Gamma : f \Downarrow_{k_1 + l}^{CV} \Theta' : \lambda y'.e'$. Moving sub-evaluations in this way is safe because the heap in question contains only values, so neither can result in any changes to the heap besides the addition of new bindings. Finally, we apply the App rule and we are done.
 - If the last step in the proof is Var, then we know that $e = x$ for some x bound in Γ. Because Γ contains only bindings to variables, we can conclude that $k = 1$ and $\Delta = \Gamma$, and simply apply the Var' rule to obtain our result.
 - If the last step in the proof is Let, then e is of the form **let** $x = e_1$ **in** e_2. We proceed by case analysis on whether x is evaluated in the derivation. If it is, we find the subproof that evaluates it and splice it into an application of the Let' rule; this splicing is safe because the evaluation of e_1 will not touch anything in the heap that is introduced during the rest of the evaluation of e_2, as these parts of the heap are out of scope. If x is not evaluated however, we know that removing it from the heap is safe, so $\Gamma : e_2 \Downarrow_{k-1}^N$. We then use the induction hypothesis to obtain $\Gamma : e_2 \Downarrow_{k-1}^{CV}$ and apply Skip.

(ii) Similar to the above, but in reverse. Note that now we may end up discarding the first premise of an application of Let', so we obtain an inequality.

□

The above proof can be read algorithmically, as it provides a mechanism for transforming derivation trees of one semantics into those of the other. Together, the two parts of Theorem 4.1 imply that the call-by-need cost will be the minimum clairvoyant call-by-value cost, justifying the intuition from the previous section. Note that there may be free variables in the final term that are not bound in the heap, which will be precisely those that are not forced under lazy evaluation.

While the proof of this theorem does provide us with a relationship between the results of evaluation in the two semantics, this is not needed anywhere beyond the proof itself. This is because even if we ignore those results, we have an *observational* equivalence between the different semantics: if there is some term t that gives one result in one semantics and an observably different

$$
\begin{aligned}
[\![\lambda x.e]\!]_\rho &= (0, \lambda v.[\![e]\!]_{\rho \cup \{x \mapsto v\}}) \quad \text{for } x \notin \mathrm{Dom}(\rho) \\
[\![x]\!]_\rho &= \begin{cases} (1, \rho(x)) & \text{if } \rho(x) \neq \bot \\ \bot & \text{otherwise} \end{cases} \\
[\![e\ x]\!]_\rho &= 1 \triangleright [\![e]\!]_\rho \mathbin{\$} \rho(x) \\
[\![\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2]\!]_\rho &= \begin{cases} (1 + k \triangleright [\![e_2]\!]_{\rho \cup \{x \mapsto v\}}) \sqcup (1 \triangleright [\![e_2]\!]_\rho) & \text{if } \{\!\{x \mapsto e_1\}\!\}_\rho = (k, v) \\ 1 \triangleright [\![e_2]\!]_\rho & \text{if } \{\!\{x \mapsto e_1\}\!\}_\rho = \bot \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
\{\!\{x \mapsto e\}\!\}_\rho &= \mathsf{lfp} f \\
\text{where} \quad f(\bot) &= [\![e]\!]_\rho \\
f((c, v)) &= [\![e]\!]_{\rho \cup \{x \mapsto v\}}
\end{aligned}
$$

Fig. 4. Denotational Semantics for Clairvoyant Call-By-Value

result in the other, there must be some context that we can place t within to create a term that diverges in one semantics but not in the other. (Working with this kind of observational equivalence is a standard technique in semantics; for example, see Pitts [2000].)

In fact, we can give an even stronger correspondence between the semantics, which is captured by the following theorem:

Theorem 4.2 (Contextual Cost-Equivalence of CBN and Clairvoyant CBV). *Given a pair of terms e and e', the following two statements are equivalent:*

(i) For all contexts $\mathbb{C}$, $\mathbb{C}[e] \Downarrow^N_k \;\Rightarrow\; \mathbb{C}[e'] \Downarrow^N_{\leq k}$
(ii) For all contexts $\mathbb{C}$, $\mathbb{C}[e] \Downarrow^{CV}_k \;\Rightarrow\; \mathbb{C}[e'] \Downarrow^{CV}_{\leq k}$

(Note that both statements should be read existentially, i.e. "if it is possible to reduce $\mathbb{C}[e]$ in k steps to some value z, then there must exist some value z' that $\mathbb{C}[e']$ reduces to in at most k steps".)

Proof.

- (i) $\Rightarrow$ (ii): Assume (i), and let $\mathbb{C}$ be an arbitrary context. Further assume that $\mathbb{C}[e] \Downarrow^{CV}_k$. By part (ii) of Theorem 4.1, we then know that there is some $k' \leq k$ such that $\mathbb{C}[e] \Downarrow^N_{k'}$, and so by our assumption we know that $\mathbb{C}[e'] \Downarrow^N_{\leq k'}$ and hence by part (i) of Theorem 4.1 we have $\mathbb{C}[e'] \Downarrow^{CV}_{\leq k'}$. Since $k' \leq k$, this implies our result.
- (ii) $\Rightarrow$ (i): Similarly, if we assume that $\mathbb{C}[e] \Downarrow^N_k$ then by part (i) of Theorem 4.1 we obtain $\mathbb{C}[e] \Downarrow^{CV}_k$. By assumption, this gives us $\mathbb{C}[e'] \Downarrow^{CV}_{k'}$ for some $k' \leq k$, and hence by part (ii) of Theorem 4.1 we have $\mathbb{C}[e] \Downarrow^N_{\leq k'}$, which again implies our result.

□

This theorem states that in order to prove a performance relationship between two call-by-need programs, it is sufficient to prove the same relationship for their clairvoyant call-by-value counterparts, and vice-versa. In other words, call-by-need and clairvoyant call-by-value have the same *contextual improvement relation*. We therefore use the same symbol $\trianglerighteq$ for this relation, regardless of whether we are referring to call-by-need or clairvoyant call-by-value.

5 A DENOTATIONAL COST SEMANTICS FOR LAZY EVALUATION

We can use the operational semantics for clairvoyant call-by-value as inspiration for a new *denotational* semantics for lazy evaluation that avoids explicit heap threading. We interpret terms in a complete lattice D that solves the following recursive domain equations:

$$\begin{aligned} D &\simeq (\omega^{op} \times V)_\perp \\ V &\simeq V_\perp \to D \end{aligned}$$

Here, ω is the natural numbers with the reverse of their usual order, $\times$ is the Cartesian product of lattices, $\to$ is the function space of monotone functions, and $-_\perp$ is the lifting operator that adds a bottom element to a lattice. We use complete lattices rather than the usual dcpo-based approach as our semantics makes essential use of both least fixed points and the join operation.

The lattice D has two types of element. First of all, there are pairs of the form (k, v), where k is a natural number and v is a monotone function of type V. These represent computations that will produce the value v after k steps of evaluation. And secondly, there is the extra bottom element $\perp$, which represents all computations that will never produce a result. We introduce two operations on these elements, $\triangleright$ and \$, which are defined as follows:

$$\begin{aligned} k \triangleright (k', v) &= (k + k', v) \\ k \triangleright \perp &= \perp \\ (k, f)\,\$\,x &= k \triangleright f(x) \\ \perp \$\, x &= \perp \end{aligned}$$

The first operation, $\triangleright$, is used to add k units of cost to elements of the domain. It has no effect on $\perp$, as adding steps to a divergent computation has no observable effect. The second operation, \$, is an application operator that extracts the function from a value and applies it, preserving the costs.

The interpretation function of our semantics takes two arguments, a term and a heap. We read $[\![e]\!]_\rho$ as "the interpretation of term e in (semantic) heap ρ". These semantic heaps are functions from the set of variables to $V_\perp$; we abuse notation slightly by treating these as partial mappings where convenient. The interpretation function itself is defined in Figure 4, where:

- The clause for an abstraction $\lambda x.e$ states that a lambda abstraction has zero cost (as it is already in normal form) and that its value is a function that maps values v to the interpretation of the body e in an extended heap where x maps to v.
- The clause for a variable x states that looking up a variable has cost 1 and results in the value x is bound to in the heap, if it exists.
- The clause for an application ex states that evaluating a function call adds one to the cost of the function itself at its argument.
- The clause for **let** $x = e_1$ **in** e_2 first elaborates the binding $x = e_1$ into the heap, then takes the join of two versions of the interpretation of e_2, one where the binding is evaluated and one where it is discarded. If the bound term e_1 diverges, only one of these is considered.

The final clause depends on an auxiliary definition, $\{\!\{x \mapsto e\}\!\}_\rho$, that elaborates a binding into a heap by taking the fixed point of its interpretation. This is necessary as the binding $x = e$ could be recursive, in which case we must have the value of e bound to x while we interpret e itself. We take our choice of notation from Nakata [2010].

It is worth noting that while our semantics uses the join operation to achieve nondeterminism, this is done in a fairly restrictive way. On both sides of $\sqcup$ we interpret the term e_2; the only difference is that on one side, we add a binding for x to the heap, while on the other we do not. If we disregard the costs we can see that one side will always be above the other, due to the monotonicity of

interpretations. The nondeterminism only becomes apparent once we consider costs, which is as it should be; if we forget this information, we obtain a standard semantics for call-by-name.

We illustrate the above denotational semantics by calculating the denotation of the term **let** $x = fx$ **in** $g\,x$ in the setting of the heap $\rho = \{f \mapsto v_1, g \mapsto v_2\}$. First, because the outermost construct in the term is a let-binding, we must calculate $\{\!\{x \mapsto fx\}\!\}_\rho$:

$$\{\!\{x \mapsto fx\}\!\}_\rho = \mathsf{lfp}\ h \quad \text{where} \quad h(\bot) = [\![fx]\!]_\rho, \quad h((c,z)) = [\![fx]\!]_{\rho \cup \{x \mapsto z\}}$$

Using the definitions in the denotational semantics, we can show that $h(\bot) = 2 \triangleright v_1(\bot)$ and $h((c,z)) = 2 \triangleright v_1(z)$, and hence that $\mathsf{lfp}\ h = 2 \triangleright \mathsf{lfp}\ (v_1 \cdot \pi_2)$ (where $\pi_2(\bot) = \bot$ and $\pi_2((c,z)) = z$ is a second-projection-like function that discards the cost information). In other words, the denotation of this binding is the cost of both variable lookups added to the fixed point of the denotation of the function f. In order to calculate the denotation of our full term, we must now inspect this least fixed point to see if it is $\bot$. If it is, then we have the following:

$$\begin{aligned}
& [\![\mathbf{let}\ x = f\ x\ \mathbf{in}\ g\ x]\!]_\rho \\
= \quad & \{\text{interpretation of let-binding}\} \\
& 1 \triangleright [\![g\ x]\!]_\rho \\
= \quad & \{\text{interpretation of function application}\} \\
& 2 \triangleright [\![g]\!]_\rho(\bot) \\
= \quad & \{\text{interpretation of variables}\} \\
& 3 \triangleright \rho(g)\$\bot = 3 \triangleright v_2(\bot)
\end{aligned}$$

That is, we simply add three units of cost to the result of the function v_2 at bottom. This reflects the intuitive understanding that, when the binding for x diverges, we can still evaluate $g\ x$, which may still give a useful result if g is not strict. If the least fixed point of $v_1 \cdot \pi_2$ is non-bottom, the situation is more interesting. Calling the least fixed point (c,z), we calculate as follows:

$$\begin{aligned}
& [\![\mathbf{let}\ x = f\ x\ \mathbf{in}\ g\ x]\!]_\rho \\
= \quad & \{\text{interpretation of let-binding}\} \\
& (1 + c \triangleright [\![g\ x]\!]_{\rho \cup \{x \mapsto z\}}) \sqcup (1 \triangleright [\![g\ x]\!]_\rho) \\
= \quad & \{\text{interpretation of function application}\} \\
& (2 + c \triangleright [\![g]\!]_{\rho \cup \{x \mapsto z\}}\$z) \sqcup (2 \triangleright [\![g]\!]_\rho\$\bot) \\
= \quad & \{\text{interpretation of variables}\} \\
& (3 + c \triangleright v_2(z)) \sqcup (3 \triangleright v_2(\bot)) \\
= \quad & \{\text{factoring}\} \\
& 3 \triangleright (c \triangleright v_2(z)) \sqcup v_2(\bot)
\end{aligned}$$

In this case, the result has all of the information available in $v_2(z)$ at cost $3 + c$, but the information in $v_2(\bot)$ is available at cost 3. Thus, the extra cost c of evaluating the binding to x need only be paid if we demand certain parts of the result, as we expect from call-by-need semantics.

The rest of this section covers the relationship between our denotational and operational semantics. To make such a relationship precise, we need to be able to interpret syntactic heaps as

semantic heaps. We extend the interpretation to heaps as follows:

$$[\![\Gamma]\!] \;=\; \mathsf{lfp}(\lambda\rho.(\lambda x.\pi_2[\![\Gamma(x)]\!]_\rho))$$

The idea here is that a (syntactic) heap can be seen as a system of recursive equations, with each binding giving one such equation. Here we simply take the least solution to this system, which is guaranteed to exist due to properties of lattices. Note that we lose no information by taking the second projection, as the interpretation of a value will always have cost 0.

We now prove two useful lemmas about the interpretation of heaps.

Lemma 5.1. *If x is in the domain of Γ, then* $[\![\Gamma(x)]\!]_{[\![\Gamma]\!]} = (0, [\![\Gamma]\!](x))$

Proof. Clear from the definition. □

This next states that $\{\!\{x \mapsto z\}\!\}$ is the correct way to extend a heap with a single binding.

Lemma 5.2. *If x does not occur in Γ and $\{\!\{x \mapsto z\}\!\}_{[\![\Gamma]\!]} = (0, v)$, then $[\![\Gamma]\!] \cup \{x \mapsto v\} = [\![\Gamma, x \mapsto z]\!]$.*

Proof. As these are functions, we need only compare how they act on arguments. We proceed by case analysis. Given argument x, we have:

$$\begin{array}{ll}
 & ([\![\Gamma]\!] \cup \{x \mapsto v\})(x) \\
= & \quad \{\text{function application}\} \\
 & v \\
= & \quad \{\text{assumption}\} \\
 & \mathsf{lfp}(\lambda v'.\pi_2[\![z]\!]_{[\![\Gamma]\!]\cup\{x\mapsto v\}}) \\
= & \quad \{\text{interpretation of heaps, properties of least fixed points}\} \\
 & \pi_1\mathsf{lfp}(\lambda(v,\rho).(\pi_2[\![z]\!]_{\rho\cup\{x\mapsto v\}}, \lambda x.\pi_2[\![\Gamma(x)]\!]_\rho) \\
= & \quad \{x \text{ not in } \Gamma\} \\
 & \pi_1\mathsf{lfp}(\lambda(v,\rho).(\pi_2[\![z]\!]_{\rho\cup\{x\mapsto v\}}, \lambda x.\pi_2[\![\Gamma(x)]\!]_{\rho\cup\{x\mapsto v\}}) \\
= & \quad \{\text{combining } v \text{ and } \rho \text{ into one argument}\} \\
 & \mathsf{lfp}(\lambda\rho.\lambda y.\mathbf{if}\; x = y \;\mathbf{then}\; \pi_2[\![z]\!]_{\rho\cup\{x\mapsto v\}} \;\mathbf{else}\; \pi_2[\![\Gamma(x)]\!]_{\rho\cup\{x\mapsto v\}})(x) \\
= & \quad \{\text{definition of interpretation of heaps}\} \\
 & [\![\Gamma, x \mapsto z]\!](x)
\end{array}$$

Otherwise $y \neq x$, and we have:

$$\begin{array}{ll}
 & ([\![\Gamma]\!] \cup \{x \mapsto v\})(y) \\
= & \quad \{y \neq x\} \\
 & [\![\Gamma]\!](y) \\
= & \quad \{x \text{ not in } \Gamma\text{, so extending heap with } x \text{ should not change result}\} \\
 & [\![\Gamma, x \mapsto z]\!](y)
\end{array}$$

This completes the proof. □

Now we can prove the soundness and adequacy of our denotational semantics. The trick here is to pick the right statement of each theorem. Soundness states that the denotational semantics of a term approximates the result of evaluating in the operational semantics.

Theorem 5.3 (Soundness of the Denotational Semantics).

If $\Gamma : e \Downarrow^{CV}_k \Delta, z$ *then* $[\![e]\!]_{[\![\Gamma]\!]} \geq k \triangleright [\![z]\!]_{[\![\Delta]\!]}$.

Proof. By induction on the derivation of $\Gamma : e \Downarrow^{CV}_k \Delta, z$, by case analysis on the last step:

- The cases for Lam and Var' are trivial.
- If the last step is App, then e must be of the form $e' x$. By the premises of the App rule and the induction hypothesis, we know that $[\![e']\!]_{[\![\Gamma]\!]} \geq k' \triangleright [\![\lambda y.e'']\!]_{[\![\Theta]\!]} = (k', \lambda v.[\![e'']\!]_{[\![\Theta]\!] \cup \{y \mapsto v\}})$ and $[\![e''[x/y]]\!]_{[\![\Theta]\!]} \geq l \triangleright [\![z]\!]_{[\![\Delta]\!]}$, where $k = k' + l + 1$. Since all variables that are in Θ but not Γ can be assumed to be fresh, we know that $[\![\Theta]\!](x) = [\![\Gamma]\!](x)$. Therefore:

$$
\begin{aligned}
& [\![e' x]\!]_{[\![\Gamma]\!]} \\
= \quad & \{\text{interpretation of application}\} \\
& 1 \triangleright [\![e']\!]_{[\![\Gamma]\!]}([\![\Gamma]\!](x)) \\
\geq \quad & \{\text{first part of inductive hypothesis}\} \\
& (1 + k', \lambda v.[\![e'']\!]_{[\![\Theta]\!] \cup \{y \mapsto v\}}) \,\$\, [\![\Gamma]\!](x) \\
= \quad & \{\text{definition of } \$\} \\
& 1 + k' \triangleright [\![e'']\!]_{[\![\Theta]\!] \cup \{y \mapsto [\![\Gamma]\!](x)\}} \\
= \quad & \{x \text{ and } y \text{ will map to same thing and } \Theta \text{ does not contain } y\text{, so interchangeable}\} \\
& 1 + k' \triangleright [\![e''[x/y]]\!]_{[\![\Theta]\!]} \\
\geq \quad & \{\text{second part of inductive hypothesis}\} \\
& 1 + k' + l \triangleright [\![z]\!]_{[\![\Delta]\!]}
\end{aligned}
$$

- If the last step is Let', then e must be of the form $\textbf{let } x = e_1 \textbf{ in } e_2$. By the inductive hypothesis, we know that $[\![e_1]\!]_{[\![\Gamma]\!]} \geq k' \triangleright [\![z_1]\!]_{[\![\Theta]\!]}$ and $[\![e_2]\!]_{[\![\Theta, x \mapsto z_1]\!]} \geq l \triangleright [\![z]\!]_{[\![\Delta]\!]}$, where $k = k' + l + 1$. By the first part of the inductive hypothesis, we know that $\{\!\{x \mapsto e_1\}\!\}_{[\![\Gamma]\!]} \geq [\![e]\!]_{[\![\Gamma]\!]} \geq k' \triangleright [\![z_1]\!]_{[\![\Theta]\!]}$. Letting $[\![z_1]\!]_{[\![\Theta]\!]} = (0, v)$, we have:

$$
\begin{aligned}
& [\![\textbf{let } x = e_1 \textbf{ in } e_2]\!]_{[\![\Gamma]\!]} \\
\geq \quad & \{\text{interpretation of let-binding}\} \\
& 1 + k' \triangleright [\![e_2]\!]_{[\![\Gamma]\!] \cup \{x \mapsto v\}} \\
\geq \quad & \{\text{interpretation is monotone in heaps}\} \\
& 1 + k' \triangleright [\![e_2]\!]_{[\![\Theta]\!] \cup \{x \mapsto v\}} \\
= \quad & \{\text{lemma 5.2}\} \\
& 1 + k' \triangleright [\![e_2]\!]_{[\![\Theta, x \mapsto z_1]\!]} \\
\geq \quad & \{\text{second part of inductive hypothesis}\} \\
& 1 + k' + l \triangleright [\![z]\!]_{[\![\Delta]\!]}
\end{aligned}
$$

- If the last step is Skip, then again e must be of the form **let** $x = e_1$ **in** e_2. By the inductive hypothesis, we know that $[\![e_2]\!]_{[\![\Gamma]\!]} \geq k - 1 \triangleright [\![z]\!]_{[\![\Delta]\!]}$. We have:

$$
\begin{array}{ll}
& [\![\textbf{let } x = e_1 \textbf{ in } e_2]\!]_{[\![\Gamma]\!]} \\
\geq & \quad \{\text{interpretation of let-binding}\} \\
& 1 \triangleright [\![e_2]\!]_{[\![\Gamma]\!]} \\
\geq & \quad \{\text{inductive hypothesis}\} \\
& k \triangleright [\![z]\!]_{[\![\Delta]\!]}
\end{array}
$$

This completes the proof. □

Note the inequality in the statement of the soundness theorem. This cannot be strengthened to an equality, because there may be multiple potential costs k for a given term. In turn, the adequacy theorem states that if the denotational semantics of a term terminates with cost k, there must be some potential evaluation with cost that is no more than k.

Theorem 5.4 (Adequacy of the Denotational Semantics).

If $[\![e]\!]_{[\![\Gamma]\!]} = (k, f)$ *then* $\Gamma : e \Downarrow^{CV}_{\leq k}$.

Proof. By induction on e.

- If e is a lambda abstraction, the result is trivial.
- If $e = x$, then it follows from the premise that $x \in \mathrm{Dom}(\Gamma)$. Therefore, by the Var' rule, we know that $\Gamma : e \Downarrow^{CV}_{1}$. From the definition of our interpretation function we can see that k cannot be 0, so we are done.
- If $e = e'x$, then $[\![e]\!]_{[\![\Gamma]\!]} = 1 \triangleright [\![e']\!]_{[\![\Gamma]\!]} \$ [\![\Gamma]\!](x)$. Hence we know that for some k and l such that $k + l + 1 = k$, and some g, $[\![e']\!]_{[\![\Gamma]\!]} = (k, g)$ and $g([\![\Gamma]\!](x)) = (l, f)$. We then obtain from the inductive hypothesis that $\Gamma : e' \Downarrow^{CV}_{\leq k} \Delta : \lambda y.e''$ for some Δ, e'' and fresh y, satisfying the first premise of the App rule, and then by soundness we know $(k, g) = [\![e']\!]_{[\![\Gamma]\!]} \geq k \triangleright [\![\lambda y.e'']\!]_{[\![\Delta]\!]}$. We reason as follows:

$$
\begin{array}{ll}
& (l, f) \\
= & \quad \{\text{above}\} \\
& g([\![\Gamma]\!](x)) \\
\geq & \quad \{\text{above}\} \\
& [\![e]\!]_{[\![\Delta]\!] \cup \{y \mapsto [\![\Gamma]\!](x)\}} \\
= & \quad \{x \text{ and } y \text{ will map to same thing, so interchangeable}\} \\
& [\![e[y/x]]\!]_{[\![\Delta]\!]}
\end{array}
$$

Therefore we know that $[\![e[y/x]]\!]_{[\![\Delta]\!]} = (l', f')$ for some $l' \leq l$ (because the ordering on the natural numbers has been reversed) and some $f' \geq f$, so by the inductive hypothesis we conclude that $\Gamma : e \Downarrow^{CV}_{\leq l}$, satisfying the second premise of the App rule.
- If $e = \textbf{let } x = e_1 \textbf{ in } e_2$, then we case split on $\{\!\{x \mapsto e_1\}\!\}_{[\![\Gamma]\!]}$.

If $\{\!\{x \mapsto e_1\}\!\}_{[\![\Gamma]\!]} = \bot$, then we have:

$$
\begin{aligned}
& (k, f) \\
= \quad & \{\text{assumption}\} \\
& [\![\textbf{let } x = e_1 \textbf{ in } e_2]\!]_{[\![\Gamma]\!]} \\
= \quad & \{\text{definition of interpretation}\} \\
& 1 \triangleright [\![e_2]\!]_{[\![\Gamma]\!]}
\end{aligned}
$$

So we have $[\![e_2]\!]_{[\![\Gamma]\!]} = (k-1, f)$. By the inductive hypothesis we can conclude that $\Gamma : e_2 \Downarrow^{CV}_{\leq k-1}$ and hence by the Skip rule that $\Gamma : \textbf{let } x = e_1 \textbf{ in } e_2 \Downarrow^{CV}_{\leq k}$.
If $\{\!\{x \mapsto e_1\}\!\}_{[\![\Gamma]\!]} = (k', v)$, then we have:

$$
\begin{aligned}
& (k, f) \\
= \quad & \{\text{assumption}\} \\
& [\![\textbf{let } x = e_1 \textbf{ in } e_2]\!]_{[\![\Gamma]\!]} \\
= \quad & \{\text{definition of interpretation}\} \\
& (1 + k' \triangleright [\![e_2]\!]_{[\![\Gamma]\!] \cup \{x \mapsto v\}}) \sqcup (1 \triangleright [\![e_2]\!]_{[\![\Gamma]\!]})
\end{aligned}
$$

Therefore, at least one of $1+k' \triangleright [\![e_2]\!]_{[\![\Gamma]\!] \cup \{x \mapsto v\}}$ and $1 \triangleright [\![e_2]\!]_{[\![\Gamma]\!]}$ is non-$\bot$ with cost k. If $1 \triangleright [\![e_2]\!]_{[\![\Gamma]\!]}$ is non-$\bot$, then we can repeat the same reasoning as in the previous case, albeit with an inequality rather than an equality. If $1 \triangleright [\![e_2]\!]_{[\![\Gamma]\!]}$ is $\bot$, then we know that $1 + k' \triangleright [\![e_2]\!]_{[\![\Gamma]\!] \cup \{x \mapsto v\}}$ is not, and hence that $[\![e_2]\!]_{[\![\Gamma]\!] \cup \{x \mapsto v\}} = (k - (1 + k'), v')$ for some v'. From $\{\!\{x \mapsto e_1\}\!\}_{[\![\Gamma]\!]} = (k', v)$ we know that $[\![e_1]\!]_{[\![\Gamma]\!]}$ will also have cost k', and so from the inductive hypothesis we know that there is some Δ, z_1 such that $\Gamma : e \Downarrow^{CV}_{\leq k} \Delta : z_1$. We now reason as follows:

$$
\begin{aligned}
& (k - (1 + k'), v') \\
= \quad & \\
& [\![e_2]\!]_{[\![\Gamma]\!] \cup \{x \mapsto v\}} \\
\geq \quad & \\
& [\![e_2]\!]_{[\![\Delta]\!] \cup \{x \mapsto v\}} \\
= \quad & \\
& [\![e_2]\!]_{[\![\Delta, x \mapsto z_1]\!]}
\end{aligned}
$$

So there must be some $l \leq k - (1 + k')$ and v'' such that $[\![e_2]\!]_{[\![\Delta, x \mapsto z_1]\!]} = (l, v'')$, and so by the inductive hypothesis we know $\Delta, x \mapsto z_1 : e_2 \Downarrow^{CV}_{\leq l}$. Applying Let' now gives the result.
This completes the proof. □

The inequality here is a consequence of the inequality in the soundness theorem. As a corollary, we also obtain soundness and adequacy with respect to the call-by-need semantics. We also note that our semantics implies that contexts act monotonically on the interpretation of terms:

Lemma 5.5 (Compositionality). *Given two terms M and N, if for any heap ρ, $[\![M]\!]_\rho \leq [\![N]\!]_\rho$, then for any context $\mathbb{C}$, and heap ρ', $[\![\mathbb{C}[M]]\!]_{\rho'} \leq [\![\mathbb{C}[N]]\!]_{\rho'}$.*

Proof. By induction on contexts. □

As a consequence of all of this, we get the following theorem:

Theorem 5.6. *Given two terms M and N, if for any heap ρ, $[\![M]\!]_\rho \leq [\![N]\!]_\rho$, then $M \gtrsim N$.*

Proof. Let $\mathbb{C}$ be an arbitrary program context. We reason as follows:

$$
\begin{aligned}
& \mathbb{C}[M] \Downarrow_k^{CV} \Delta : V \\
\Rightarrow\quad & \{\text{soundness}\} \\
& [\![\mathbb{C}[M]]\!]_\emptyset \geq k \triangleright [\![V]\!]_{[\![\Delta]\!]} \\
\Rightarrow\quad & \{\text{compositionality of interpretations implies that } [\![\mathbb{C}[M]]\!]_\emptyset \leq [\![\mathbb{C}[N]]\!]_\emptyset\} \\
& [\![\mathbb{C}[N]]\!]_\emptyset \geq k \triangleright [\![V]\!]_{[\![\Delta]\!]} \\
\Rightarrow\quad & \{\text{properties of the domain}\} \\
& \exists k', f. [\![\mathbb{C}[N]]\!]_\emptyset = (k', f) \quad \wedge \quad k' \leq k \\
\Rightarrow\quad & \{\text{adequacy}\} \\
& \exists k'. \mathbb{C}[N] \Downarrow_{\leq k'}^{CV} \quad \wedge \quad k' \leq k \\
\Rightarrow\quad & \{\text{transitivity of the order}\} \\
& \mathbb{C}[N] \Downarrow_{\leq k}^{CV}
\end{aligned}
$$

□

This theorem essentially states that the ordering in our denotational semantics is sound for improvements. In other words, in order to prove an operational improvement, it is enough to prove that the same relationship holds in the *denotational* realm. Note, however, that the converse is not true: proving an operational improvement does not imply that the corresponding denotational relationship holds; this would require a fully-abstract semantics à la Hyland and Ong [2000].

6 APPLICATIONS

In this section, we demonstrate the practical utility of our theory on a range of examples. In particular, we show how it can be used to verify a number of basic results from improvement theory, and how it can be used to demonstrate that common subexpression elimination is an improvement. In each case, the proofs using our theory are significantly simpler than previously.

6.1 Tick Algebra

Moran and Sands [1999a] gave a number of rules for proving improvements and cost equivalences between call-by-need programs based on an operator, ✓, that represents one unit of time cost. We can add such an operator to our language with the following operational semantics:

$$\frac{\Gamma : e \Downarrow_k \Delta : v}{\Gamma : \checkmark e \Downarrow_{k+1} \Delta : v} \quad (\textsc{Tick})$$

We can prove the validity of some of these "tick algebra" rules in the context of clairvoyant call-by-value. These proofs are significantly simpler than those in Moran and Sands' work.

Value-β. The value-β rule states that if v is a value, it is safe to inline it. Assuming $\mathbb{C}$ does not capture x or any of the free variables in v, we have the following equivalence:

$$\textbf{let } x = v \textbf{ in } \mathbb{C}[x] \;\; \triangleleft\!\triangleright\!\!\!\!\sim \;\; \textbf{let } x = v \textbf{ in } \mathbb{C}[\checkmark v]$$

This seems like an obvious property, but justifying it formally for call-by-need requires the use of a number of auxiliary results which are themselves nontrivial, as well as a modified notion of context [Moran and Sands 1999a]. In contrast, for clairvoyant call-by-value it is straightforward: simply take the derivation tree for the evaluation of $\mathbb{D}[\textbf{let } x = v \textbf{ in } \mathbb{C}[x]]$ and replace any application of the VAR' rule for x with an application of the LAM rule for v plus one application of TICK. This process is clearly reversible, so the improvement holds in both directions.

Var-β. The var-β rule states that it is safe to de-alias a variable. Assuming $\mathbb{C}$ is a context captures neither x nor y, we have the following improvement:

$$\textbf{let } x = y \textbf{ in } \mathbb{C}[x] \trianglerighteq \textbf{let } x = y \textbf{ in } \mathbb{C}[y]$$

To prove this, let $\mathbb{D}$ be an arbitrary context, and consider a derivation tree for the evaluation of $\mathbb{D}[\textbf{let } x = y \textbf{ in } \mathbb{C}[x]]$ with cost k. If the hole in $\mathbb{D}$ is not evaluated, this derivation tree can be easily made into a derivation tree for $\mathbb{D}[\textbf{let } x = y \textbf{ in } \mathbb{C}[y]]$ of the same cost. Otherwise, consider a subtree where $\textbf{let } x = y \textbf{ in } \mathbb{C}[x]$ is evaluated. This subtree will be of one of the following two forms:

$$\dfrac{\dfrac{\vdots}{\Gamma : \mathbb{C}[x] \Downarrow_k \Delta : z}}{\Gamma : \textbf{let } x = y \textbf{ in } \mathbb{C}[x] \Downarrow_{k+1} \Delta : z}\ \textsc{Skip}$$

$$\dfrac{\dfrac{}{\Gamma, y = v : y \Downarrow_1 \Gamma, y = v : v}\ \textsc{Var} \quad \dfrac{\vdots}{\Gamma, y = v, x = v : \mathbb{C}[x] \Downarrow_k \Delta : z}}{\Gamma, y = v : \textbf{let } x = y \textbf{ in } \mathbb{C}[x] \Downarrow_{k+2} \Delta : z}\ \textsc{Let}'$$

In the first case, we know the variable x is not used, so can be replaced by anything, in particular y. In the second case, x and y are bound to the same thing in the subproof, so it is safe to replace x with y without affecting the cost. By applying this process to all evaluations of $\textbf{let } x = y \textbf{ in } \mathbb{C}[x]$, we obtain a complete derivation of the evaluation of $\mathbb{D}[\textbf{let } x = y \textbf{ in } \mathbb{C}[y]]$ with the same cost k. Note that it is not necessarily safe to reverse this process by replacing y with x if the SKIP rule is applied to the binding of x, so the improvement will only hold in one direction.

(Remark: we originally presented this proof purely in words, but in more complex cases this becomes unwieldy. We find that the derivation tree representation serves as a useful visual aid.)

Let-let. The let-let rule tells us the cost effect of flattening let-bindings:

$$\checkmark\textbf{let } x = \textbf{let } y = e_1 \textbf{ in } e_2 \textbf{ in } e_3 \ \trianglelefteq\!\trianglerighteq\ \textbf{let } y = e_1 \textbf{ in let } x = \checkmark e_2 \textbf{ in } e_3$$

Once again, we take $\mathbb{C}$ to be an arbitrary context and consider a derivation tree showing $\mathbb{C}[\checkmark\textbf{let } x = \textbf{let } y = e_1 \textbf{ in } e_2 \textbf{ in } e_3] \Downarrow_n$, considering subtrees where $\checkmark\textbf{let } x = \textbf{let } y = e_1 \textbf{ in } e_2 \textbf{ in } e_3$ is evaluated. These subtrees will be of one of the following forms:

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma : e_3 \Downarrow_k \Delta : z}}{\Gamma : \textbf{let } x = \textbf{let } y = e_1 \textbf{ in } e_2 \textbf{ in } e_3 \Downarrow_{k+1} \Delta : z}\ \textsc{Skip}}{\Gamma : \checkmark\textbf{let } x = \textbf{let } y = e_1 \textbf{ in } e_2 \textbf{ in } e_3 \Downarrow_{k+2} \Delta : z}\ \textsc{Tick}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Gamma : e_2 \Downarrow_k \Theta : v}}{\Gamma : \textbf{let } y = e_1 \textbf{ in } e_2 \Downarrow_{k+1} \Theta : v}\textsc{Skip} \qquad \dfrac{\vdots}{\Theta, x = v : e_3 \Downarrow_l \Delta : z}}{\Gamma : \textbf{let } x = \textbf{let } y = e_1 \textbf{ in } e_2 \textbf{ in } e_3 \Downarrow_{k+l+2} \Delta : z}\textsc{Let'}}{\Gamma : \checkmark\textbf{let } x = \textbf{let } y = e_1 \textbf{ in } e_2 \textbf{ in } e_3 \Downarrow_{k+l+3} \Delta : z}\textsc{Tick}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Gamma : e_1 \Downarrow_k \Phi : w} \quad \dfrac{\vdots}{\Phi, y = w : e_2 \Downarrow_l \Theta : v}}{\Gamma : \textbf{let } y = e_1 \textbf{ in } e_2 \Downarrow_{k+l+1} \Theta : v}\textsc{Let'} \qquad \dfrac{\vdots}{\Theta, x = v : e_3 \Downarrow_m \Delta : z}}{\Gamma : \textbf{let } x = \textbf{let } y = e_1 \textbf{ in } e_2 \textbf{ in } e_3 \Downarrow_{k+l+m+2} \Delta : z}\textsc{Let'}}{\Gamma : \checkmark\textbf{let } x = \textbf{let } y = e_1 \textbf{ in } e_2 \textbf{ in } e_3 \Downarrow_{k+l+m+3} \Delta : z}\textsc{Tick}$$

Each of these can be rearranged into the following respective forms, so that the entire derivation tree will now show that $\mathbb{C}[\textbf{let } y = e_1 \textbf{ in let } x = \checkmark e_2 \textbf{ in } e_3] \Downarrow_n$:

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma : e_3 \Downarrow_k \Delta : z}}{\Gamma : \textbf{let } x = \checkmark e_2 \textbf{ in } e_3 \Downarrow_{k+1} \Delta : z}\textsc{Skip}}{\Gamma : \textbf{let } y = e_1 \textbf{ in let } x = \checkmark e_2 \textbf{ in } e_3 \Downarrow_{k+2} \Delta : z}\textsc{Skip}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Gamma : e_2 \Downarrow_k \Theta : v}}{\Gamma : \checkmark e_2 \Downarrow_{k+1} \Theta : v}\textsc{Tick} \qquad \dfrac{\vdots}{\Theta, x = v : e_3 \Downarrow_l \Delta : z}}{\Gamma : \textbf{let } x = \checkmark e_2 \textbf{ in } e_3 \Downarrow_{k+l+2} \Delta : z}\textsc{Let'}}{\Gamma : \textbf{let } y = e_1 \textbf{ in let } x = \checkmark e_2 \textbf{ in } e_3 \Downarrow_{k+l+3} \Delta : z}\textsc{Skip}$$

$$\dfrac{\dfrac{\vdots}{\Gamma : e_1 \Downarrow_k \Phi : w} \qquad \dfrac{\dfrac{\dfrac{\vdots}{\Phi, y = w : e_2 \Downarrow_l \Theta : v}}{\Phi, y = w : \checkmark e_2 \Downarrow_{l+1} \Theta : v}\textsc{Tick} \qquad \dfrac{\vdots}{\Theta, x = v : e_3 \Downarrow_m \Delta : z}}{\Phi, y = w : \textbf{let } x = \checkmark e_2 \textbf{ in } e_3 \Downarrow_{l+m+2} \Delta : z}\textsc{Let'}}{\Gamma : \textbf{let } y = e_1 \textbf{ in let } x = \checkmark e_2 \textbf{ in } e_3 \Downarrow_{k+l+m+3} \Delta : z}\textsc{Let'}$$

Since this transformation can clearly be reversed, the improvement holds both ways. (Again, we use the derivation trees as a visual aid to the proof.)

We have proven the validity of these rules for clairvoyant call-by-value, rather than call-by-need. However, Theorem 4.2 tells us that these same results will be applicable to the standard call-by-need semantics, and hence also applicable to real-world lazy programs.

6.2 Common Subexpression Elimination

Common subexpression elimination is a widely-used optimisation in which an expression that appears several times in a piece of code is abstracted out into a binding. This allows the repeated subexpression to be computed once, rather than several times, which saves time. The time-safety

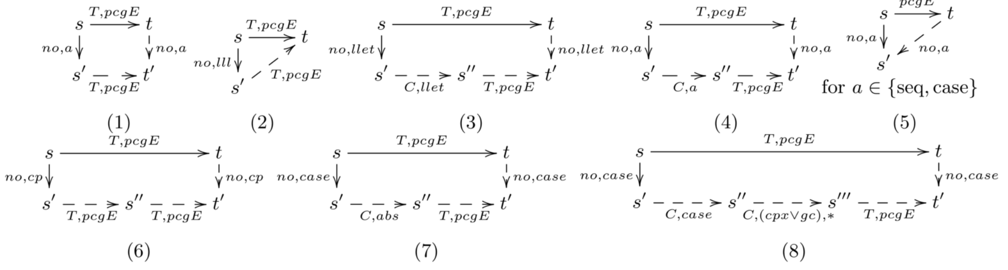


Fig. 5. Forking diagrams from Schmidt-Schauß and Sabel [2015a]

of this optimisation is captured by the following improvement:

$$\checkmark\mathbb{C}[\checkmark e] \;\gtrsim\!\!\!\triangleright\; \textbf{let}\ x = e\ \textbf{in}\ \mathbb{C}[x]$$

We assume here that x does not appear in the context $\mathbb{C}$ or the term e. Note that the presence of ticks on the left-hand side state that there are some situations in which this is not an outright improvement, but that the slowdown is bounded by a constant.

Despite the simplicity of this rule, it is has been difficult to prove. Moran and Sands [1999a] presented the rule as the consequence of their *β-expansion* conjecture, but could not provide a satisfactory proof of the conjecture. No proof was found until 16 years later, when Schmidt-Schauß and Sabel [2015a] gave a proof of the time-safety of common subexpression elimination. However, their proof is for a very different semantics and is highly technical, taking several pages to explain and relying on detailed “forking diagrams”; for an example, see Figure 5. By contrast, our theory allows for a much shorter and simpler proof via our new denotational semantics.

We introduce ticks into our interpretation by setting $[\![\checkmark e]\!]_\rho = 1 \triangleright [\![e]\!]_\rho$. It is then straightforward to extend our soundness and adequacy theorems to cover ticks. Let $\mathbb{C}$ be an arbitrary context not containing x, and ρ be an arbitrary heap and f be the function such that $[\![\mathbb{C}[e]]\!]_\rho = f([\![e]\!]_{\rho\cup\tau})$ (for some τ not dependent on e). We can derive this function by induction on the context $\mathbb{C}$, applying the rules of $[\![-]\!]$. Furthermore, this function has the property that it is either constant, or $f(k \triangleright v) = k \triangleright f(v)$; this can also be proved by induction on the rules of $[\![-]\!]$.

We proceed by case analysis on the value of $[\![e]\!]$. Suppose $[\![e]\!]_\rho = (k, v)$. Because x is not free in e, we also have that $\{\!\{x \mapsto e\}\!\}_\rho = (k, v)$. We reason as follows:

$$\begin{aligned}
& [\![\textbf{let}\ x = e\ \textbf{in}\ \mathbb{C}[x]]\!]_\rho \\
= \quad & \{\text{definition of interpretation, definition of } v\} \\
& (1 + k \triangleright [\![\mathbb{C}[x]]\!]_{\rho\cup\{x\mapsto v\}}) \sqcup (1 \triangleright [\![\mathbb{C}[x]]\!]_\rho) \\
= \quad & \{x \text{ free in } \mathbb{C}, \text{ definition of } f\} \\
& (1 + k \triangleright f([\![x]\!]_{\rho\cup\tau\cup\{x\mapsto v\}})) \sqcup (1 \triangleright f([\![x]\!]_{\rho\cup\tau})) \\
= \quad & \{\text{definition of interpretation}\} \\
& (1 + k \triangleright f((1, v))) \sqcup (1 \triangleright f(\bot)) \\
\geq \quad & \{\text{property of } f\} \\
& 1 \triangleright f((k + 1, v)) \\
= \quad & \{\text{definition of } v\} \\
& 1 \triangleright f(1 \triangleright [\![e]\!]_\rho)
\end{aligned}$$

$$
\begin{aligned}
= \quad & \{\text{definition of ticks, } f \text{ and interpretation}\} \\
& [\![\checkmark \mathbb{C}[\checkmark e]]\!]_\rho
\end{aligned}
$$

The case for when $[\![e]\!]_\rho = \bot$ is similar. Therefore, in every case we have shown that the improvement holds in the denotational semantics, so we simply apply Theorem 5.6 and we are done.

7 EXTENDING THE LANGUAGE

Our theory up to this point has been developed for a core language comprising the lambda calculus and singly-recursive let-bindings. In this section, we discuss how to extend the language with two extra features, datatypes and mutual recursion, that are provided in more practical languages.

7.1 Datatypes

The first feature we will extend our language with is datatypes. First, we fix a set of constructor symbols c_i indexed by some set I. We also require a function $ar : I \longrightarrow \mathbb{N}$ that assigns an *arity* to each constructor. We now extend the syntax of our language as follows:

$$
\begin{aligned}
e \quad ::= \quad & \ldots \\
\mid \quad & c_i\ x_1 \ldots x_{ar(i)} \\
\mid \quad & \textbf{case}\ e\ \textbf{of}\ \{\ldots;\ c_i\ x_1 \ldots x_{ar(i)} \mapsto e\ ;\ldots\}
\end{aligned}
$$

Note that we require that all the arguments of a constructor be variables. Implementing these constructs in the operational semantics is straightforward. Constructors are considered a value, and **case** first reduces the scrutinee to a constructor and then selects the appropriate branch:

$$
\Gamma : c_i x_1 \ldots x_{ar(i)} \Downarrow_0 \Gamma : c_i x_1 \ldots x_{ar(i)} \qquad \text{(Con)}
$$

$$
\frac{\Gamma : e \Downarrow_k \Delta : c_i\ y_1 \ldots y_{ar(i)} \quad \Delta : e_i[y_1/x_1, \ldots, y_{ar(i)}/x_{ar(i)}] \Downarrow_l \Theta : z}{\Gamma : \textbf{case}\ e\ \textbf{of}\ \{\ldots;\ c_i\ x_1 \ldots x_{ar(i)} \mapsto e_i\ ;\ldots\} \Downarrow_{k+l+1} \Theta : z} \qquad \text{(Case)}
$$

Because constructors may only be applied to variables, the difference between call-by-need and clairvoyant call-by-value semantics is entirely captured by the existing rules. Therefore, the new rules will be the same for both semantics and so extending the proof of cost-equivalence is trivial. Extending the denotational semantics requires a little more work, as we must extend our lattice V with elements corresponding to constructor applications, but this is also fairly straightforward, and the proofs can be adapted without much difficulty.

7.2 Mutual Recursion

Extending our language with mutual recursion requires a little more work. The syntactic change is simple, as we need only alter the existing **let** construct to allow for a sequence of definitions:

$$
\begin{aligned}
e \quad ::= \quad & \ldots \\
\mid \quad & \textbf{let}\ \overrightarrow{x = e}\ \textbf{in}\ e
\end{aligned}
$$

Extending the semantics for call-by-need is also fairly straightforward, and in fact this extension was present in Launchbury's [1993] original presentation:

$$
\frac{\Gamma, x_1 \mapsto e_1, \ldots, x_n \mapsto e_n : e \Downarrow_k^N \Delta : z}{\Gamma : \textbf{let}\ x_1 = e_1, \ldots, x_n = e_n\ \textbf{in}\ e \Downarrow_{k+n}^N \Delta : z} \qquad \text{(Let)}
$$

Note that the cost of a let-binding is set to be linear in the number of bindings created. Firstly, this seems realistic, as allocating more cells on the heap would logically take more time. And secondly, this matches the natural cost behaviour of the clairvoyant call-by-value semantics.

However, a difficulty arises with the clairvoyant call-by-value semantics. When there was only one binding, the only choice is whether to evaluate it or not. Now, we must also consider the *order* of evaluating bindings: any variable x_i could require the value of some other variable x_j before it can be itself reduced to a value, and so long as these dependencies are not cyclic, there may be an order of evaluation that is terminating in the standard semantics. Therefore, we extend the Let' rule so as to allow for *any* of the bindings to be selected for evaluation first:

$$\frac{\Gamma : e_i \Downarrow_k^{CV} \Delta : z_i \qquad \Delta, x_i \mapsto z_i : \textbf{let } x_1 = e_1, \ldots, x_{i-1} = e_{i-1}, x_{i+1} = e_{i+1}, \ldots, x_n = e_n \textbf{ in } e \Downarrow_l^{CV} \Theta : z}{\Gamma : \textbf{let } x_1 = e_1, \ldots, x_n = e_n \textbf{ in } e \Downarrow_{k+l+1}^{CV} \Theta : z} \quad (\textsc{Let'}_i)$$

The new Skip rule allows for multiple bindings to be discarded, but only once all the bindings that are to be evaluated have actually be evaluated:

$$\frac{\Gamma : e \Downarrow_k^{CV} \Delta : z}{\Gamma : \textbf{let } x_1 = e_1, \ldots, x_n = e_n \textbf{ in } e \Downarrow_{k+n}^{CV} \Delta : z} \quad (\textsc{Skip})$$

This limits the degree of nondeterminism possible, but not in a way that impacts the results.

Proving cost-equivalence between call-by-need and clairvoyant call-by-value has much the same structure as before, except now we must pick an order of evaluation in the clairvoyant case. We do this by checking the call-by-need derivation not only for *if* the bound variables are evaluated, but in what order, and then replicating that order by choosing the appropriate sequence of $\textsc{Let'}_i$ rules. Reversing this process is now less straightforward, as it may involve reordering of evaluation. However, if we consider this process stepwise, each time moving the evaluation of the last-evaluated binding to its first use site, then this will only delay evaluation, which is always time-safe.

Modifying the denotational semantics is more work, as we must now introduce a generalised version of the elaboration operator $\{\!\{-\}\!\}$ for mutually-defined bindings. Similarly, we must extend the interpretation of let-binding to allow for the 2^n possible cases of which bindings are evaluated. This naturally complicates the corresponding cases in the proofs of soundness and adequacy.

8 RELATED WORK

Maraist et al. [1999] relate the three reduction strategies of call-by-name, call-by-value and call-by-need to the linear and affine lambda calculi, demonstrating that call-by-need can be viewed as an affine variant of call-by-value that allows for discarding. This work is the direct inspiration of ours, but differs from our approach in a number of key ways. First of all, we do not use linear or affine calculi of the basis of our approach, instead working solely with a lambda calculus without linearity restrictions. Secondly, Maraist et al.'s work does not address questions of cost, so is unable to capture the full effect of sharing. And finally, their work is purely operational, whereas we also provide a denotational cost semantics for our language.

Braßel et al. [2007] pursue a similar avenue, relating call-by-need evaluation to a nondeterministic version of call-by-value that can be made deterministic by providing trace information from a call-by-need evaluation. This allows for simpler debugging tools for lazy functional languages. This work is based on a big-step semantics, like ours, but their basic lazy semantics is not quite the same as ours. No cost correspondence is proved in their work, and this is again a purely operational approach, with no applications to denotational semantics explored.

There have been many other approaches to providing a semantics for call-by-need. Besides the natural semantics of Launchbury [1993] that we have discussed, there area also the axiomatic semantics of Ariola et al. [1995], Ariola and Felleisen [1997] and Maraist et al. [1998]. These semantics are not totally deterministic, in that there may be multiple potential reductions of a given term, but they are confluent and admit a *standard* reduction that will always produce a result, if there is one. By contrast, clairvoyant call-by-value can admit no straightforward notion of standard reduction, as it relies essentially on the nondeterminism of the semantics.

In denotational cost semantics, Ghica [2005] presents a quantitative semantics for a call-by-name language in the form of *slot* games, an extension of standard game semantics to include extra "token" actions that represent resource usage. Hope [2008] presents a calculational approach to the problem, deriving an abstract machine from an evaluator, augmenting with cost information and then reversing the original derivation to produce a cost-instrumented evaluator. Rosendahl [1989] demonstrates how the standard denotational semantics for first-order Lisp can be augmented with cost information, and Seidel and Voigtländer [2011] use a similar technique to produce a costed semantics that validates *free theorems* relating to resource usage. Finally, Danner et al. [2015] define an elaboration from a source language into a *complexity* language from which cost recurrences can be extracted. With the exception of Ghica's work, all of these approaches focus on strict evaluation. As far as we are aware, this paper represents the first comparable work for call-by-need.

Finally, Sansom and Peyton Jones [1997] provide an *operational* cost semantics for lazy programs, allowing for the attribution of cost to cost centers, and prove that several standard program transformations respect the attribution of cost without transferring cost erroneously from one cost center to another. This forms the basis of the profiler built into the Glasgow Haskell Compiler.

9 CONCLUSION AND FURTHER WORK

We have presented an alternative operational semantics for lazy evaluation based on the idea of *clairvoyant evaluation*, and proven it to be cost-equivalent to the more standard semantics of Launchbury [1993]. This new semantics has enabled us to produce a clean *denotational* cost semantics for lazy evaluation. We have used our results to validate a number of theorems from the literature with significantly simpler proofs than previously possible.

This work has addressed the issue of time cost for call-by-need programs. However, *space* cost is also an important concern, and avoiding space leaks is an important issue when working in a lazy language. It is already known how to produce improvement theories for call-by-need space costs [Gustavsson and Sands 1999, 2001]; a natural next step for this work would be to see if a similar technique can be applied in this case. We know that we cannot apply the exact same technique, as we relied on the fact that it doesn't matter *when* we evaluate subterms, and this assumption is no longer valid for space. However, it may be possible to apply the more general pattern of "moving internal effects to the external theory" to address space concerns.

The theory that was developed in this paper is entirely untyped. Previous work has shown that information from the type system can assist with cost analyses [Hackett and Hutton 2018; Jost et al. 2010; Simões et al. 2012], so it is natural to ask if a type system would similarly augment our work. A logical starting point would be modifying the denotational semantics to apply to a typed lambda calculus such as System F; this may provide a way to replicate the cost parametricity results of Hackett and Hutton [2018] in a denotational setting.

The transformation of state to nondeterminism we use here was enabled by the fact that the standard semantics for call-by-need uses state in a fairly disciplined way: each memory cell is updated at most once, and only with something that can be calculated from its original contents. Similarly, the nondeterminism in the new semantics is also used in a disciplined way, having the property that all successful evaluations will produce the same final result. It may be possible to

extract general principles from this, abstracting away from the details of the exact effects employed to develop an implementation-generic theory of laziness.

Perhaps the most interesting application of this work is in education. One of the most problematic aspects of Haskell for programmers new to the language is the behaviour of laziness, which can be hard to predict even for seasoned Haskellers. By developing alternative models of what laziness *is*, we may be able to create more intuitive ways to explain the behaviour of lazy programs, leading to better learning outcomes. Perhaps laziness can be simple, after all.

ACKNOWLEDGEMENTS

We would like to thank the referees for many useful comments and suggestions. This work was funded by the Engineering and Physical Sciences Research Council (EPSRC) grant EP/P00587X/1, *Mind the Gap: Unified Reasoning About Program Correctness and Efficiency*.

REFERENCES

Zena M Ariola and Matthias Felleisen. 1997. The Call-By-Need Lambda Calculus. *Journal of Functional Programming* 7, 3 (1997).

Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. 1995. A Call-by-Need Lambda Calculus. In *Principles of Programming Languages*.

Richard S. Bird. 1984. Using Circular Programs to Eliminate Multiple Traversals of Data. *Acta Informatica* 21, 3 (1984).

Bror Bjerner and Sören Holmström. 1989. A Composition Approach to Time Analysis of First Order Lazy Functional Programs. In *International Conference on Functional Programming Languages and Computer Architecture*.

Bernd Braßel, Michael Hanus, Sebastian Fischer, Frank Huch, and Germán Vidal. 2007. Lazy Call-By-Value Evaluation. In *ACM SIGPLAN Notices*, Vol. 42. ACM, 265–276.

Norman Danner, Daniel R. Licata, and Ramyaa Ramyaa. 2015. Denotational Cost Semantics for Functional Languages with Inductive Types. In *International Conference on Functional Programming*.

Dan R. Ghica. 2005. Slot Games: A Quantitative Model of Computation. In *Principles of Programming Languages*.

Jörgen Gustavsson and David Sands. 1999. A Foundation for Space-Safe Transformations of Call-by-Need Programs. *Electronic Notes on Theoretical Computer Science* 26 (1999).

Jörgen Gustavsson and David Sands. 2001. Possibilities and Limitations of Call-by-Need Space Improvement. In *International Conference on Functional Programming*.

Jennifer Hackett and Graham Hutton. 2014. Worker/Wrapper/Makes It/Faster. In *International Conference on Functional Programming*.

Jennifer Hackett and Graham Hutton. 2018. Parametric Polymorphism and Operational Improvement. *Proceedings of the ACM on Programming Languages* 2, ICFP, Article 68 (2018).

Peter Henderson and James H Morris. 1976. A Lazy Evaluator. In *Principles on Programming Languages*.

Catherine Hope. 2008. *A Functional Semantics for Space and Time*. Ph.D. Dissertation. University of Nottingham.

John Hughes. 1989. Why Functional Programming Matters. *Comput. J.* 32, 2 (1989).

J. M. E. Hyland and C.-H.L. Ong. 2000. On Full Abstraction for PCF: I, II, and III. *Information and Computation* 163, 2 (2000).

Mark B. Josephs. 1989. The Semantics of Lazy Functional Languages. *Theoretical Computer Science* 68, 1 (1989).

Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. 2010. Static Determination of Quantitative Resource Usage for Higher-Order Programs. In *ACM SIGPLAN Notices*, Vol. 45.

John Launchbury. 1993. A Natural Semantics for Lazy Evaluation. In *Principles of Programming Languages*.

John Maraist, Martin Odersky, David N. Turner, and Philip Wadler. 1999. Call-By-Name, Call-By-Value, Call-By-Need and the Linear Lambda Calculus. *Theoretical Computer Science* 228, 1-2 (1999).

John Maraist, Martin Odersky, and Philip Wadler. 1998. The Call-by-Need Lambda Calculus. *Journal of Functional Programming* 8, 3 (1998).

Andrew Moran and David Sands. 1999a. Improvement in a Lazy Context: An Operational Theory for Call-by-Need. (1999). Extended version of [Moran and Sands 1999b], available at http://tinyurl.com/ohuv8ox.

Andrew Moran and David Sands. 1999b. Improvement in a Lazy Context: An Operational Theory for Call-by-Need. In *Principles of Programming Languages*.

James Hiram Morris. 1969. *Lambda-Calculus Models of Programming Languages*. Ph.D. Dissertation. MIT.

Keiko Nakata. 2010. Denotational Semantics for Lazy Initialization of letrec. In *Fixed Points in Computer Science*.

Chris Okasaki. 1999. *Purely Functional Data Structures*. Cambridge University Press.

Andrew M Pitts. 2000. Operational Semantics and Program Equivalence. In *International Summer School on Applied Semantics*. Springer, 378–412.

Mads Rosendahl. 1989. Automatic Complexity Analysis. In *Functional Programming Languages and Computer Architecture*.

David Sands. 1991. Operational Theories of Improvement in Functional Languages. In *Glasgow Workshop on Functional Programming*.

David Sands. 1997. Improvement Theory and its Applications. In *Higher Order Operational Techniques in Semantics, Publications of the Newton Institute*. Cambridge University Press.

Patrick M. Sansom and Simon L. Peyton Jones. 1997. Formally Based Profiling for Higher-Order Functional Languages. *ACM Transactions on Programming Languages and Systems* 19, 2 (1997).

Manfred Schmidt-Schauß and Nils Dallmeyer. 2018. Space Improvements and Equivalences in a Functional Core Language. *arXiv preprint arXiv:1802.06498* (2018).

Manfred Schmidt-Schauß and David Sabel. 2015a. Improvements in a Functional Core Language with Call-By-Need Operational Semantics. In *Principles and Practice of Declarative Programming*.

Manfred Schmidt-Schauß and David Sabel. 2015b. Sharing-Aware Improvements in a Call-by-Need Functional Core Language. In *Implementation and Application of Functional Programming Languages*.

Daniel Seidel and Janis Voigtländer. 2011. Improvements for Free. In *Quantitative Aspects of Programming Languages*.

Hugo Simões, Pedro Vasconcelos, Mário Florido, Steffen Jost, and Kevin Hammond. 2012. Automatic Amortised Analysis of Dynamic Memory Allocation for Lazy Functional Programs. In *International Conference on Functional Programming*.

David A Turner. 1982. Recursion Equations as a Programming Language. In *Functional Programming and its Applications: An Advanced Course*. Cambridge University Press.

Philip Wadler and John Hughes. 1987. Projections for Strictness Analysis. In *Functional Programming Languages and Computer Architecture*.