Improvement in a Lazy Context: An Operational Theory for Call-By-Need

Andrew Moran
Oregon Graduate Institute
and
David Sands
Chalmers University of Technology

The standard implementation technique for lazy functional languages is call-by-need, which ensures that an argument to a function in any given call is evaluated at most once. A significant problem with call-by-need is that it is difficult — even for compiler writers — to predict the effects of program transformations. The traditional theories for lazy functional languages are based on call-by-name models, and offer no help in determining which transformations do indeed optimize a program.

We present an operational theory for call-by-need, based upon an improvement ordering on programs: M is improved by N if in all program-contexts $\mathbb{C}$, when $\mathbb{C}[M]$ terminates then $\mathbb{C}[N]$ terminates at least as cheaply.

We show that this improvement relation satisfies a "context lemma", and supports a rich inequational theory, subsuming the call-by-need lambda calculi of Ariola *et al.* [Ariola et al. 1995]. The reduction-based call-by-need calculi are inadequate as a theory of lazy-program transformation since they only permit transformations which speed up programs by at most a constant factor (a claim we substantiate); we go beyond the various reduction-based calculi for call-by-need by providing powerful proof rules for recursion, including syntactic continuity — the basis of fixed-point-induction style reasoning, and an improvement theorem, suitable for arguing the correctness and safety of recursion-based program transformations.

1. INTRODUCTION

Call-by-need optimises call-by-name by ensuring that when evaluating a given function application, arguments are evaluated at most once. All serious compilers for lazy functional languages implement call-by-need evaluation. Lazy functional languages are believed to be well-suited to high-level program transformations, and some state-of-the-art compilers take advantage of this by applying a myriad of transformations and analyses during compilation [Peyton Jones and Santos 1998]. However, it is notoriously difficult, even for those with extremely solid intuitions

Name: Andrew Moran
Affiliation: Department of Computer Science and Engineering, Oregon Graduate Institute of Science and Technology
Address: moran@cse.ogi.edu
Name: David Sands
Affiliation: Department of Computing Science, Chalmers University of Technology and University of Göteborg
Address: dave@cs.chalmers.se

about call-by-need, to predict the effects of a program transformation on the running time. Since traditional theories for lazy languages are based upon call-by-name models, they give no assurance that a given transformation doesn't lead to an asymptotic slow-down.

Call-by-need Calculi. The call-by-need lambda calculi [Ariola et al. 1995; Ariola and Felleisen 1997; Maraist et al. 1998] offer a solution to some of these problems. By permitting fewer equations than call-by-name, these calculi enable term-level reasoning without ignoring the key implementation issues underpinning call-by-need. However, they do have some serious limitations. All of the equations in the calculi are, by definition, symmetric. This means that certain useful local transformations cannot be present. In fact, the call-by-need calculi are limited to transformations which change running-times by at most a constant-factor (see section 7), independent of the context in which the programs are used. Even within the confines of constant-factor transformations there are significant shortcomings, since none of the calculi have proof rules for recursion; we believe that, as a consequence, almost no interesting equivalences between recursive programs — such as the fusion of recursive functions (*e.g.* via deforestation) — can be justified in the calculi.

Our Approach. We aim to go beyond these limitations by refining the notion of *observational approximation* between terms, and by establishing algebraic laws (containing the laws of the call-by-need calculi as theorems) and recursion principles for that approximation relation. A key result of [Ariola et al. 1995] is that the standard observational equivalence and approximation relations, in which one only observes termination, cannot distinguish call-by-need evaluation from call-by-name. To obtain an operational theory which retains the computational distinctions between name and need, we also observe the *cost* of evaluation, in terms of a high-level model of computation steps. Our observational approximation relation, *improvement*, is defined with respect to a fixed operational semantics by saying that: M is improved by N if in all program-contexts $\mathbb{C}$, when $\mathbb{C}[M]$ terminates then $\mathbb{C}[N]$ terminates at least as fast.

Summary of Results. We develop an operational theory for a call-by-need lambda calculus with recursive lets, constructors, and case expressions. The theory is based upon an abstract machine semantics for call-by-need, and is cost-sensitive, and therefore reflects the computational distinctions between call-by-name and call-by-need. We show that the improvement relation has a rich inequational theory, validating the reduction rules of the call-by-need calculi. Most importantly, it supports powerful induction principles for recursive programs. Some specific original results are:

—A *context lemma* for call-by-need, meaning we can establish improvement by considering just computation in a restricted class of contexts, the evaluation contexts;
—A rich inequational theory, the *tick algebra*, which subsumes the call-by-need calculi;
—A *syntactic continuity* property which characterises improvement of a recursive function in terms of its finite unwindings, and forms the basis of fixed-point

induction style proofs, and

—Two powerful proof techniques, the *improvement theorem* and *improvement induction*, which are particularly well-suited to inferring the correctness *and* safety of recursion-based program transformations which proceed by local improvements.

—A general method for establishing laws, properties, and proof rules which generalises the context lemma, known as *open uniform computation.*

Overview. The paper may be split into two separate parts. The first half presents the operational theory and contains all of the major results, mostly stated without proof. The second half presents the technical machinery behind those results, and proves them.

We begin the first half of the paper with a discussion of related work in section 2. Section 3 then presents the operational semantics (Sestoft's "mark 1" abstract machine for laziness). A discussion of the complexity of computation follows in section 4, where we show that the number of heap accesses during a computation is a reasonable measure of cost. This is used as the basis for a contextual definition of improvement and cost equivalence, and the context lemma is stated.

The inequational theory, known as the tick algebra, is then presented in section 6, and the relative power of the algebra and the call-by-need calculi is discussed in section 7. Syntactic continuity is presented in section 8 and used to show that an unwinding fixed-point combinator is improved (up to a constant factor) by a knot-tying fixed-point combinator. We also present a syntactic variant of fixed-point fusion for call-by-need, which can be established via syntactic continuity. The improvement theorem is introduced in section 9, along with improvement induction and examples of their use. A more substantial example is presented in section 10.

The second half of the paper is contained in section 11. We generalise the notion of program contexts to configurations, and extend reduction to open configuration contexts. This allows us to establish *open uniform computation*, a general technique used to prove not only the context lemma, but also many of the more difficult algebraic laws, and the various induction rules.

Finally, section 12 concludes, and we discuss of future avenues of research.

2. RELATED WORK

Improvement theory and the improvement theorem were originally developed in the call-by-name setting [Sands 1991; Sands 1996], and generalised to a variety of call-by-name and call-by-value languages in [Sands 1997]. Whether this programme could be carried out in a call-by-need setting has long been an open question. An inspiration which gave us confidence in the possibility of a tractable improvement theory for call-by-need is the *call-by-need lambda calculus* presented by Ariola and Felleisen, and Maraist, Odersky and Wadler [Ariola et al. 1995; Ariola and Felleisen 1997; Maraist et al. 1998]. For us, the significance of the call-by-need calculi is that they are based on reduction (and hence equations) between terms in the source language (see figure 7), rather than, say, term-graphs, abstract-machine configurations, or terms plus explicit substitutions. The reduction rules are confluent, and enjoy a deterministic notion of standard reduction. Related concepts appear in other approaches, in particular in the study of so-called optimal reductions *e.g.*, [Field 1990; Maranget 1991; Yoshida 1993].

One limitation of the original work by Ariola *et al.* is in the treatment of recursive *cycles*; naïve extension of the calculi to deal with recursive lets leads to a loss of confluence [Jeffrey 1993; Ariola and Klop 1997]. The original call-by-need calculus considers recursive lets only briefly. To recover confluence, one can simply disallow reductions under cycles, as in *e.g.*, [Benaissa et al. 1996; Niehren 1996]. Ariola and Blom give a full study of cyclic recursion in [Ariola and Blom 1997; Ariola and Blom 1998], and show that an approximation to confluence can be obtained by equating terms with the same infinite normal-form. Their λ_{OSHARE} calculus can be seen as the natural successor to the call-by-need calculi.

In general, reduction calculi appear to be a good vehicle for exploring the language design space with regard to call-by-need-like features. Rose's work *e.g.* [Rose 1996; Benaissa et al. 1996] exemplifies this approach in an elegant combination of explicit substitution and combinatory reduction systems. Our view is complementary to the rewriting approaches: once a particular operational semantics (reduction strategy) has been fixed, one can go beyond the confines of the calculi by developing an operational theory.

Apart from the rewriting-based approaches, there have been a few attempts to give a high-level semantics to call-by-need *e.g.* [Josephs 1989; Jeffrey 1994; Seaman and Purushothaman Iyer 1996; Launchbury 1993; Sestoft 1997]. Launchbury's natural semantics, and Sestoft's abstract machine(s) have been adopted by a number of researchers as the formal definition of call-by-need *e.g.* [Turner et al. 1995; Hughes and Moran 1995; Sansom and Peyton Jones 1997; Gustavsson 1998]. Since it appears to be a non-controversial choice, we adopt Sestoft's machine — essentially a Krivine-machine [Curien 1991] with updating of the heap — as the operational model underpinning our theory. As others have observed (*e.g.* [Pitts 1997a]), working with an abstract machine rather than an inductive semantics also has benefits in proofs about computations (examples of this may be found in section 11).

The techniques used in this paper, open uniform computation in particular, have proven quite robust. They have been applied successfully to a non-deterministic call-by-need language [Moran et al. 1999], and in the development of an algebra for showing when transformations are *space-safe* optimisations in the presence of sharing [Gustavsson and Sands 1999].

3. THE OPERATIONAL SEMANTICS

Our language is an untyped lambda calculus with recursive lets, structured data, and case expressions. We work with a restricted syntax in which arguments to functions (including constructors) are always variables:

$$\begin{array}{rcl} x, y, z & \in & \mathit{Var} \\ c & \in & \mathit{Con} \\ L, M, N & ::= & x \\ & | & \lambda x.M \\ & | & M\,x \\ & | & \mathsf{let}\ \{\vec{x} = \vec{M}\}\ \mathsf{in}\ N \\ & | & c\,\vec{x} \\ & | & \mathsf{case}\ M\ \mathsf{of}\ \{c_i\,\vec{x}_i \to N_i\} \\ V, W & ::= & \lambda x.M \\ & | & c\,\vec{x} \end{array}$$

The syntactic restriction is now rather standard, following its use in core language of the Glasgow Haskell compiler, *e.g.*, [Peyton Jones et al. 1996; Peyton Jones and Santos 1998], and in [Launchbury 1993; Sestoft 1997].

All constructors have a fixed arity, and are assumed to be saturated. By $c\,\vec{x}$ we mean $c\,x_1 \cdots x_n$. The only values are lambda expressions and fully-applied constructors. Throughout, x, y, z, and w will range over variables, c over constructor names, and V and W over values. We will write

$$\mathsf{let}\ \{\vec{x} = \vec{M}\}\ \mathsf{in}\ N$$

as a shorthand for

$$\mathsf{let}\ \{x_1 = M_1, \ldots, x_n = M_n\}\ \mathsf{in}\ N$$

where the $\vec{x}$ are distinct, the order of bindings is not syntactically significant, and the $\vec{x}$ are considered bound in N *and* the $\vec{M}$ (so our lets are recursive). Similarly we write

$$\mathsf{case}\ M\ \mathsf{of}\ \{c_i\,\vec{x}_i \to N_i\}$$

for

$$\mathsf{case}\ M\ \mathsf{of}\ \{c_1\,\vec{x}_1 \to N_1 | \cdots | c_m\,\vec{x}_m \to N_m\}.$$

where each $\vec{x}_i$ is a vector of distinct variables, and the c_i are distinct constructors. In addition, we will sometimes write *alts* as an abbreviation for case alternatives $\{c_i\,\vec{x}_i \to N_i\}$.

For examples, working with a restricted syntax can be cumbersome, so it is sometimes useful to lift the restriction. Where we do this it should be taken that

$$M\,N \equiv \mathsf{let}\ \{x = N\}\ \mathsf{in}\ M\,x, \quad x \text{ fresh}$$

whenever N is not a variable. Similarly for constructor expressions.

The only kind of substitution that we consider is *variable for variable*, with σ ranging over such substitutions. The simultaneous substitution of one vector of variables for another will be written $M[\vec{y}/\vec{x}]$, where the $\vec{x}$ are assumed to be distinct (but the $\vec{y}$ need not be).

3.1 The Abstract Machine

The semantics presented in this section is essentially Sestoft's "mark 1" abstract machine for laziness [Sestoft 1997]. In that paper, he proves his abstract machine

$$\langle\, \Gamma\{x = M\},\ x,\ S \,\rangle \rightarrow \langle\, \Gamma,\ M,\ \#x : S \,\rangle \qquad (\textit{Lookup})$$

$$\langle\, \Gamma,\ V,\ \#x : S \,\rangle \rightarrow \langle\, \Gamma\{x = V\},\ V,\ S \,\rangle \qquad (\textit{Update})$$

$$\langle\, \Gamma,\ M\,x,\ S \,\rangle \rightarrow \langle\, \Gamma,\ M,\ x : S \,\rangle \qquad (\textit{Unwind})$$

$$\langle\, \Gamma,\ \lambda x.M,\ y : S \,\rangle \rightarrow \langle\, \Gamma,\ M[{}^{y}\!/\!_{x}],\ S \,\rangle \qquad (\textit{Subst})$$

$$\langle\, \Gamma,\ \mathsf{case}\ M\ \mathsf{of}\ \textit{alts},\ S \,\rangle \rightarrow \langle\, \Gamma,\ M,\ \textit{alts} : S \,\rangle \qquad (\textit{Case})$$

$$\langle\, \Gamma,\ c_j\,\vec{y},\ \{c_i\,\vec{x}_i \rightarrow N_i\} : S \,\rangle \rightarrow \langle\, \Gamma,\ N_j[{}^{\vec{y}}\!/\!_{\vec{x}_j}],\ S \,\rangle \qquad (\textit{Branch})$$

$$\langle\, \Gamma,\ \mathsf{let}\ \{\vec{x} = \vec{M}\}\ \mathsf{in}\ N,\ S \,\rangle \rightarrow \langle\, \Gamma\{\vec{x} = \vec{M}\},\ N,\ S \,\rangle \quad \vec{x} \between \mathrm{dom}(\Gamma, S) \qquad (\textit{Letrec})$$

Fig. 1. The abstract machine semantics for call-by-need.

semantics sound and complete with respect to Launchbury's natural semantics, and we will not repeat those proofs here.

Transitions are over configurations consisting of a heap, containing bindings, the expression currently being evaluated, and a stack. The heap is a partial function from variables to terms, and denoted in an identical manner to a collection of let-bindings. The stack may contain variables (the arguments to applications), case alternatives, or *update markers* denoted by $\#x$ for some variable x. Update markers ensure that a binding to x will be recreated in the heap with the result of the current evaluation; this is how sharing is maintained in the semantics.

We write $\langle\, \Gamma,\ M,\ S \,\rangle$ for the abstract machine configuration with heap Γ, expression M, and stack S. We denote the empty heap by $\emptyset$, and the addition of a group of bindings $\vec{x} = \vec{M}$ to a heap Γ by juxtaposition: $\Gamma\{\vec{x} = \vec{M}\}$. The stack written $b : S$ will denote the a stack S with b pushed on the top. The empty stack is denoted by ϵ, and the concatenation of two stacks S and T by ST (where S is on top of T).

We will refer to the set of variables bound by Γ as $\mathrm{dom}\,\Gamma$, and to the set of variables marked for update in a stack S as $\mathrm{dom}\,S$. Update markers should be thought of as binding occurrences of variables. A configuration is *well-formed* if $\mathrm{dom}\,\Gamma$ and $\mathrm{dom}\,S$ are disjoint. We write $\mathrm{dom}(\Gamma, S)$ for their union. For a configuration $\langle\, \Gamma,\ M,\ S \,\rangle$ to be closed, any free variables in Γ, M, and S must be contained in $\mathrm{dom}(\Gamma, S)$. For sets of variables P and Q we will write $P \between Q$ to mean that P and Q are disjoint, *i.e.*, $P \cap Q = \emptyset$. The free variables of a term M will be denoted $\mathsf{FV}\,(M)$; for a vector of terms $\vec{M}$, we will write $\mathsf{FV}\,(\vec{M})$.

The abstract machine semantics is presented in figure 3.1; we implicitly restrict the definition to well-formed configurations. There are seven rules, which can grouped as follows. Rules (*Lookup*) and (*Update*) concern evaluation of variables. To begin evaluation of x, we remove the binding $x = M$ from the heap and start evaluating M, with x, marked for update, pushed onto the stack. Rule (*Update*) applies when this evaluation is finished, and we may update the heap with the new binding for x.

Rules (*Unwind*) and (*Subst*) concern function application: rule (*Unwind*) pushes an argument onto the stack while the function is being evaluated; once a lambda expression has been obtained, rule (*Subst*) retrieves the argument from the stack and substitutes it into the body of that lambda expression.

Rules (*Case*) and (*Branch*) govern the evaluation of case expressions. Rule (*Case*) initiates evaluation of the case expression, with the case alternatives pushed onto the stack. Rule (*Branch*) uses the result of this evaluation to choose one of the branches of the case, performing substitution of the constructor's arguments for the branch's pattern variables.

Lastly, rule (*Letrec*) adds a set of bindings to the heap. The side condition ensures that no inadvertent name capture occurs, and can always be satisfied by a local α-conversion.

3.2 Relating Terms and Configurations

We can translate between configurations to terms straightforwardly, by induction over the stack:

$$\begin{aligned}
\mathsf{trans}\langle\, \emptyset,\ M,\ \epsilon\,\rangle &= M\\
\mathsf{trans}\langle\, \{\vec{x} = \vec{M}\},\ N,\ \epsilon\,\rangle &= \mathsf{let}\ \{\vec{x} = \vec{M}\}\ \mathsf{in}\ N\\
\mathsf{trans}\langle\, \Gamma,\ M,\ x : S\,\rangle &= \mathsf{trans}\langle\, \Gamma,\ M\,x,\ S\,\rangle\\
\mathsf{trans}\langle\, \Gamma,\ M,\ \#x : S\,\rangle &= \mathsf{trans}\langle\, \Gamma\{x = M\},\ x,\ S\,\rangle\\
\mathsf{trans}\langle\, \Gamma,\ M,\ \mathit{alts} : S\,\rangle &= \mathsf{trans}\langle\, \Gamma,\ \mathsf{case}\ M\ \mathsf{of}\ \mathit{alts},\ S\,\rangle
\end{aligned}$$

The operational semantics tells us how to translate terms into configurations. In the following lemma, $\mathbb{C}$ is a *program context* containing zero or more holes. $\mathbb{C}[M]$ denotes the insertion of M into those holes, yielding another term. (Contexts will be introduced in more detail in section 5.)

Lemma 3.1. (Translation) *For all Γ, $\mathbb{C}$, S, there exists $k \geqslant 0$ such that for any M, $\langle\, \emptyset,\ \mathsf{trans}\langle\, \Gamma,\ \mathbb{C}[M],\ S\,\rangle,\ \epsilon\,\rangle \rightarrow^k \langle\, \Gamma,\ \mathbb{C}[M],\ S\,\rangle$.*

Proof. Simple induction on the size of S. □

3.3 Convergence

An operational theory relies upon having a useful notion of an *observable*, that is, a property of closed progams which may be observed. The simplest observable is termination, or convergence.

Definition 1. (Convergence) *For closed configurations $\langle\, \Gamma,\ M,\ S\,\rangle$,*

$$\begin{aligned}
\langle\, \Gamma,\ M,\ S\,\rangle{\Downarrow^n} &\overset{\text{def}}{=} \exists \Delta, V. \langle\, \Gamma,\ M,\ S\,\rangle \rightarrow^n \langle\, \Delta,\ V,\ \epsilon\,\rangle,\\
\langle\, \Gamma,\ M,\ S\,\rangle{\Downarrow} &\overset{\text{def}}{=} \exists n. \langle\, \Gamma,\ M,\ S\,\rangle{\Downarrow^n},\\
\langle\, \Gamma,\ M,\ S\,\rangle{\Downarrow^{\leqslant n}} &\overset{\text{def}}{=} \exists m. \langle\, \Gamma,\ M,\ S\,\rangle{\Downarrow^m} \wedge m \leqslant n.
\end{aligned}$$

Closed configurations which do not converge are of three types: they either reduce indefinitely, get stuck because of a type error, or get stuck because of a *black-hole* (a self-dependent expression as in $\mathsf{let}\ x = x\ \mathsf{in}\ x$). All non-converging configurations will be semantically identified.

We will also write $M{\Downarrow}$, $M{\Downarrow^n}$ and $M{\Downarrow^{\leqslant n}}$, identifying closed M with the initial configuration $\langle\, \emptyset,\ M,\ \epsilon\,\rangle$.

4. COMPLEXITY OF COMPUTATION

The cost of computation is what distinguishes call-by-name from call-by-need. Our strategy for building an operational theory which respects this distinction is to observe this cost when comparing terms. Before developing this theory, the question which remains is how one should measure cost. In an attempt to predict actual running times, one might assign implementation specific constants to each abstract machine step. Even if this were possible (we are doubtful, since most compilers perform a myriad of optimisations), it would lead to a very fine-grained and implementation-specific theory. Instead we work with a more abstract measure of cost, and aim for a non implementation-specific theory.

In an earlier version of this work [Moran and Sands 1999; Moran and Sands 1998] for simplicity we chose simply to count the number of abstract machine steps as our measure. It would be unrealistic to assume that abstract machine steps could reveal information about actual running times, given that we are working with such a high-level machine. For whatever cost measure we choose, the bottom line is whether it is sufficient to describe the *complexity* of computation. In other words, the measure should be within a constant factor of "actual cost". A reasonable question is whether each step of the abstract machine can be considered implementable in constant time; we defer discussion of this point to appendix A.

We now move to an even leaner notion of cost than abstract machine steps. The aim is to make the notion of cost as simple as possible, but without sacrificing our bottom line — namely that the measure of cost should be within a program-size dependent constant factor of running-time. It is sufficient to measure cost in terms of the number of times the lookup rule is applied. This claim is proven in appendix A.

Let us now define the cost of computation.

Definition 2. *For closed configurations* $\langle \Gamma,\ M,\ S \rangle$,

$$\langle \Gamma,\ M,\ S \rangle{\downarrow}^{n} \stackrel{\text{def}}{=} \langle \Gamma,\ M,\ S \rangle{\Downarrow} \textit{ with } n \textit{ occurrences of } (\textit{Lookup})$$
$$\langle \Gamma,\ M,\ S \rangle{\downarrow}^{\leqslant n} \stackrel{\text{def}}{=} \exists m. \langle \Gamma,\ M,\ S \rangle{\downarrow}^{m} \wedge m \leqslant n.$$

As with $\Downarrow$, we will identify closed M with the initial configuration $\langle \emptyset,\ M,\ \epsilon \rangle$, writing $M{\downarrow}^{n}$, and $M{\downarrow}^{\leqslant n}$.

To demonstrate the soundness of our cost measure, we argue that

(1) the number of abstract-machine steps is within a program-size dependent constant factor of actual running time of an implementation based on the abstract machine, and
(2) the number of lookup steps is within a program-size-specific constant factor of the number of abstract machine steps.

The former is discussed in appendix A, and the latter is formalised in the following theorem, the proof of which may be found in the same appendix.

Theorem 4.1. *For all* $s > 0$*, there exists a linear function* f *such that for all closed terms* M *of size* s,

$$M{\downarrow}^{m} \implies M{\Downarrow}^{\leqslant f(m)}.$$

This justifies the use of the number of lookups as a measure of cost. We can now define *improvement*, which will be based on this measure.

5. IMPROVEMENT

The starting point for an operational theory is usually an approximation and an equivalence defined in terms of *program contexts.* Program contexts are generally introduced as "programs with holes", the intention being that an expression is to be "plugged into" all of the holes in the context. The central idea is that to compare the behaviour of two terms one should compare their behaviour in all program contexts.

We will use contexts of the following form:

$$\begin{aligned}
\mathbb{C}, \mathbb{D} ::= {} & [\cdot] \\
\mid {} & x \\
\mid {} & \lambda x.\mathbb{C} \\
\mid {} & \mathbb{C}\, x \\
\mid {} & \mathsf{let}\ \{\vec{x} = \vec{\mathbb{D}}\}\ \mathsf{in}\ \mathbb{C} \\
\mid {} & c\, \vec{x} \\
\mid {} & \mathsf{case}\ \mathbb{C}\ \mathsf{of}\ \{c_i\, \vec{x}_i \rightarrow \mathbb{D}_i\} \\
\mathbb{V}, \mathbb{W} ::= {} & \lambda x.\mathbb{C} \\
\mid {} & c\, \vec{x}.
\end{aligned}$$

Our contexts may contain zero or more occurrences of the hole, and as usual the operation of filling a hole with a term can cause variables in that term to become captured.

We define observational approximation and equivalence via contexts in the standard way [Abramsky and Ong 1993].

Definition 3. (Observational Approximation) *We say that M* observationally approximates *N, written $M \precsim N$, if for all $\mathbb{C}$ such that $\mathbb{C}[M]$ and $\mathbb{C}[N]$ are closed,*

$$\mathbb{C}[M]{\Downarrow} \implies \mathbb{C}[N]{\Downarrow}.$$

We say that M and N are *observationally equivalent*, written $M \cong N$, when $M \precsim N$ and $N \precsim M$.

We know that $\cong$ coincides with its call-by-name counterpart, so this tells us nothing new. We need to incorporate more intensional information if we are to build an operational theory that retains the distinction between name and need. Since call-by-need may be thought of as an optimisation of call-by-name, a natural intensional property to compare is how many reduction steps are required for termination. However, theorem 4.1tells us that counting lookups is in fact sufficient. Recall that we will write

$$M{\downarrow}^n$$

to mean that M converges with a cost of n, where n is the number of lookups that occur during the evaluation of M.

Definition 4. (Improvement) *We say that M is* improved by *N, written*

$M \trianglerighteq\!\!\!\!\sim N$, *if for all* $\mathbb{C}$ *such that* $\mathbb{C}[M]$ *and* $\mathbb{C}[N]$ *are closed,*

$$\mathbb{C}[M]{\downarrow}^{n} \implies \mathbb{C}[N]{\downarrow}^{\leqslant n}.$$

We say that M and N are *cost equivalent*, written $M \mathrel{\underset{\sim}{\lhd\!\rhd}} N$, when $M \mathrel{\underset{\sim}{\trianglerighteq}} N$ and $N \mathrel{\underset{\sim}{\trianglerighteq}} M$.

This definition suffers from the same problem as any contextual definition: to prove that two terms are related requires one to examine their behaviour in *all* contexts. For this reason, it is common to seek to prove a *context lemma* [Milner 1977] for an operational semantics: one tries to show that to prove M observationally approximates N, one only need compare their behaviour with respect to a more tractable set of contexts.

We have established the following context lemma for call-by-need:

Lemma 5.1. (Context Lemma) *For all terms M and N, if for all Γ, S, and n, such that $\langle\,\Gamma,\ M,\ S\,\rangle$ and $\langle\,\Gamma,\ N,\ S\,\rangle$ are closed,*

$$\langle\,\Gamma,\ M,\ S\,\rangle{\downarrow}^{n} \implies \langle\,\Gamma,\ N,\ S\,\rangle{\downarrow}^{\leqslant n}$$

then $M \mathrel{\underset{\sim}{\trianglerighteq}} N$.

It says that we need only consider configuration contexts of the form $\langle\,\Gamma,\ [\cdot],\ S\,\rangle$ where the hole $[\cdot]$ appears only once. This corresponds exactly to a subset of term contexts called *evaluation contexts*, in which the hole is the subject of evaluation. We shall make this correspondence precise in the section 6.2.

Note that the context lemma applies to open terms M and N. It is more common to restrict one's attention to closed terms, and then show that the preorder in question is closed under (general) substitution.

5.1 Strong Improvement

The improvement relation, like the notion of operational approximation which it refines, also increases the termination of programs, so if $M \mathrel{\underset{\sim}{\trianglerighteq}} N$ then N may also terminate "more often" than M. In the context of compiler optimisations it is natural to ask for a stronger notion of improvement which does not permit any change in termination behaviour.

Definition 5. (Strong Improvement) *We say that M is* strongly improved *by N, written $M \trianglerighteq N$, if*

$$M \mathrel{\underset{\sim}{\trianglerighteq}} N \land N \mathrel{\underset{\sim}{\sqsubseteq}} M.$$

M is strongly improved by N if it is improved by N, and N has identical termination behaviour (note that we need only have $N \mathrel{\underset{\sim}{\sqsubseteq}} M$ in the definition since $M \mathrel{\underset{\sim}{\trianglerighteq}} N \implies M \mathrel{\underset{\sim}{\sqsubseteq}} N$).

For simplicity of presentation we emphasise improvement rather than strong improvement. However, almost all the laws and proof rules presented in subsequent sections also hold for strong improvement, notable exceptions being the "strictness laws" concerning Ω, the divergent term. The syntactic continuity proof principle is sound for strong improvement, but degenerates to a trivial rule.

The following Hasse-diagram illustrates the relationships between the various approximations and equivalences introduced in this section:

$$\begin{tikzcd}
 & & \Lambda \times \Lambda \arrow[dl, dash] \arrow[dr, dash] & & \\
 & \underset{\sim}{\sqsupset} \arrow[dl, dash] \arrow[dr, dash] & & \underset{\sim}{\sqsubset} \arrow[dl, dash] \arrow[dr, dash] & \\
\underset{\sim}{\lhd} \arrow[dr, dash] & & \cong \arrow[dl, dash] \arrow[dr, dash] & & \underset{\sim}{\rhd} \arrow[dl, dash] \\
 & \trianglelefteq \arrow[dr, dash] & & \trianglerighteq \arrow[dl, dash] & \\
 & & \underset{\sim}{\lhd\!\rhd} & &
\end{tikzcd}$$

The diagram is a $\cap$-semi-lattice of relations on terms. In other words, the greatest lower bound of any two relations in the diagram is equal to their set-intersection.

6. THE TICK ALGEBRA

Consider the following improvement:

$$\text{let } \{x = V\} \text{ in } x \underset{\sim}{\rhd} \text{let } \{x = V\} \text{ in } V \qquad (*)$$

Clearly, for any Γ and S:

$$\begin{aligned}\langle \Gamma,\ \text{let } \{x = V\} \text{ in } x,\ S \rangle &\rightarrow \langle \Gamma\{x = V\},\ x,\ S \rangle \\ &\rightarrow \langle \Gamma,\ V,\ \#x : S \rangle \\ &\rightarrow \langle \Gamma\{x = V\},\ V,\ S \rangle\end{aligned}$$

and

$$\langle \Gamma,\ \text{let } \{x = V\} \text{ in } V,\ S \rangle \rightarrow \langle \Gamma\{x = V\},\ V,\ S \rangle$$

so $(*)$ follows from the context lemma. But we can say more: let $\{x = V\}$ in x *always* takes exactly two more steps to converge than let $\{x = V\}$ in V. More importantly, one of those two steps is always a lookup, incurring cost.

If we had some syntactic way of introducing cost to the right-hand side, $(*)$ could be written as a cost equivalence, which would be preferable, since it is a more informative statement. This motivates the introduction of the "tick", written $\checkmark$, which we will use to add a unit of cost to a computation. Now we can write $(*)$ as

$$\text{let } \{x = V\} \text{ in } x \underset{\sim}{\lhd\!\rhd} {}^{\checkmark}\text{let } \{x = V\} \text{ in } V$$

We introduce the tick as a new syntactic construct[1], with the following transition rule:

$$\langle \Gamma,\ {}^{\checkmark}M,\ S \rangle \rightarrow \langle \Gamma,\ M,\ S \rangle \qquad (\textit{Tick})$$

[1]In earlier work, the tick was defined within the language. To do so here, we could introduce a spurious indirection, *i.e.* ${}^{\checkmark}M$ would be defined by let $\{x = M\}$ in x. However, this needlessly complicates proofs, since it changes the heap.

with the further stipulation that we count occurrences of both (*Lookup*) and the (*Tick*) transitions when calculating the cost of a compuation.

By definition, ✓ adds one unit to the cost of evaluating M without otherwise changing its behaviour. Note that:

$$M{\Downarrow} \iff {}^{\checkmark}M{\Downarrow}$$
$$M{\downarrow}^{n} \iff {}^{\checkmark}M{\downarrow}^{n+1}$$

We will write ${}^{k\checkmark}M$ to mean that M has been slowed down by k ticks. The following inference rule and axiom, known collectively as "tick elimination" are crucial when establishing improvement or cost equivalence.

$$\frac{{}^{\checkmark}M \trianglerighteq_{\sim} {}^{\checkmark}N}{M \trianglerighteq_{\sim} N} \qquad {}^{\checkmark}M \trianglerighteq_{\sim} M \qquad (\checkmark\text{-}elim)$$

Their validity follows from the definition of $\trianglerighteq_{\sim}$.

We can easily prove a number of improvements and cost equivalences modulo tick, and we present a selection of the more useful ones in the following sections. Throughout, we will follow the standard convention that all bound variables in the statement of a law are distinct, and that they are disjoint from the free variables. Together with (✓*-elim*), the laws presented in figures 2, 3, 4, 5, and figure 6 are known collectively as the *tick algebra*.

6.1 Beta Laws

The first set of laws, presented in figure 2, are important in that they allow us to mimic evaluation within the algebra. (β) is the familiar law for call-by-need beta reduction; (*case-*β) is the analogous law for case expressions. To see the validity of (β), note that, for all Γ and S

$$\begin{aligned}\langle\, \Gamma,\ (\lambda y.M)\,x,\ S\,\rangle &\to \langle\, \Gamma,\ \lambda y.M,\ x : S\,\rangle\\ &\to \langle\, \Gamma,\ M[{}^{x}/_{y}],\ S\,\rangle\end{aligned}$$

Since $(\lambda y.M)\,x$ always reduces to $M[{}^{x}/_{y}]$ in two *zero cost* steps, irrespective of Γ and S, the context lemma tells us that they are cost equivalent. Many of the laws in this section are this easily established.

In (*value-*β), one may replace occurrences of a variable, which is bound to some value V, with ${}^{\checkmark}V$. The tick reflects the fact that by replacing x with its value, we are short-circuiting a lookup step.

(*var-*β) is a version of (*value-*β) where x is instead bound to another variable z. It is an improvement only, because the speedup achieved can vary. It can be reversed if we compensate for the indirection, as in (*var-abs*). (*var-subst*) and (*var-expand*) are slight variations on (*var-*β) and (*var-abs*), respectively, that allow us to replace x with z even in argument positions (not allowed in (*var-*β) due to the use of contexts). The proofs of validity of (*value-*β), (*var-*β), (*var-abs*), (*var-subst*) and (*var-expand*) rely upon general techniques that are outlined in section 11.

There are also two derived beta laws, corresponding to unrestricted versions of (β) and (*case-*β). We can derive the following cost equivalence:

$$(\lambda x.M)\,N \mathrel{\triangleleft\!\triangleright_{\sim}} \mathsf{let}\ \{x = N\}\ \mathsf{in}\ M \qquad (\beta')$$

Laws of the Tick Algebra

Throughout, we follow the standard convention that all bound variables in the statement of a law are distinct, and that they are disjoint from the free variables.

$$(\lambda x.M)\,y \mathrel{\underset{\sim}{\lhd\!\rhd}} M[{}^{y}\!/\!_{x}] \qquad (\beta)$$

$$\mathsf{case}\ c_j\,\vec{y}\ \mathsf{of}\ \{c_i\,\vec{x}_i \to M_i\} \mathrel{\underset{\sim}{\lhd\!\rhd}} M_j[{}^{\vec{y}}\!/\!_{\vec{x}_j}] \qquad (\textit{case-}\beta)$$

$$\mathsf{let}\ \{x = V, \vec{y} = \vec{\mathbb{D}}[x]\}\ \mathsf{in}\ \mathbb{C}[x] \mathrel{\underset{\sim}{\lhd\!\rhd}} \mathsf{let}\ \{x = V, \vec{y} = \vec{\mathbb{D}}[{}^{\checkmark}V]\}\ \mathsf{in}\ \mathbb{C}[{}^{\checkmark}V] \qquad (\textit{value-}\beta)$$

$$\mathsf{let}\ \{x = z, \vec{y} = \vec{\mathbb{D}}[x]\}\ \mathsf{in}\ \mathbb{C}[x] \mathrel{\underset{\sim}{\rhd}} \mathsf{let}\ \{x = z, \vec{y} = \vec{\mathbb{D}}[z]\}\ \mathsf{in}\ \mathbb{C}[z] \qquad (\textit{var-}\beta)$$

$${}^{\checkmark}\mathsf{let}\ \{x = z, \vec{y} = \vec{\mathbb{D}}[z]\}\ \mathsf{in}\ \mathbb{C}[z] \mathrel{\underset{\sim}{\rhd}} \mathsf{let}\ \{x = z, \vec{y} = \vec{\mathbb{D}}[x]\}\ \mathsf{in}\ \mathbb{C}[x] \qquad (\textit{var-abs})$$

$$\mathsf{let}\ \{x = z, \vec{y} = \vec{M}[{}^{x}\!/\!_{w}]\}\ \mathsf{in}\ N[{}^{x}\!/\!_{w}] \mathrel{\underset{\sim}{\rhd}} \mathsf{let}\ \{x = z, \vec{y} = \vec{M}[{}^{z}\!/\!_{w}]\}\ \mathsf{in}\ N[{}^{z}\!/\!_{w}] \qquad (\textit{var-subst})$$

$${}^{\checkmark}\mathsf{let}\ \{x = z, \vec{y} = \vec{M}[{}^{z}\!/\!_{w}]\}\ \mathsf{in}\ N[{}^{z}\!/\!_{w}] \mathrel{\underset{\sim}{\rhd}} \mathsf{let}\ \{x = z, \vec{y} = \vec{M}[{}^{x}\!/\!_{w}]\}\ \mathsf{in}\ N[{}^{x}\!/\!_{w}] \qquad (\textit{var-expand})$$

Fig. 2. Beta laws for call-by-need.

$$\mathbb{E}[{}^{\checkmark}M] \mathrel{\underset{\sim}{\lhd\!\rhd}} {}^{\checkmark}\mathbb{E}[M] \qquad (\checkmark\text{-}\mathbb{E})$$

$$\mathbb{E}[\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to N_i\}] \mathrel{\underset{\sim}{\lhd\!\rhd}} \mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to \mathbb{E}[N_i]\} \qquad (\textit{case-}\mathbb{E})$$

$$\mathbb{E}[\mathsf{let}\ \{\vec{x} = \vec{M}\}\ \mathsf{in}\ N] \mathrel{\underset{\sim}{\lhd\!\rhd}} \mathsf{let}\ \{\vec{x} = \vec{M}\}\ \mathsf{in}\ \mathbb{E}[N] \qquad (\textit{let-}\mathbb{E})$$

$$\mathsf{let}\ \{x = M\}\ \mathsf{in}\ \mathbb{E}[x] \mathrel{\underset{\sim}{\lhd\!\rhd}} \mathbb{E}[{}^{\checkmark}M], \quad \text{if } x \notin \mathsf{FV}\,(M, \mathbb{E}) \qquad (\textit{inline-}\mathbb{E})$$

Fig. 3. Laws for evaluation contexts.

$$\mathsf{let}\ \{\vec{x} = \vec{M}\}\ \mathsf{in}\ N \mathrel{\underset{\sim}{\lhd\!\rhd}} N, \quad \text{if } \vec{x} \notin \mathsf{FV}\,(N) \qquad (\textit{gc})$$

$$\mathsf{let}\ \{\vec{x} = \vec{L}\}\ \mathsf{in}\ \mathsf{let}\ \{\vec{y} = \vec{M}\}\ \mathsf{in}\ N \mathrel{\underset{\sim}{\lhd\!\rhd}} \mathsf{let}\ \{\vec{x} = \vec{L}, \vec{y} = \vec{M}\}\ \mathsf{in}\ N \qquad (\textit{let-flatten})$$

$$\mathsf{let}\ \{x = \mathsf{let}\ \{\vec{y} = \vec{L}, \vec{z} = \vec{M}\}\ \mathsf{in}\ N\}\ \mathsf{in}\ N' \mathrel{\underset{\sim}{\lhd\!\rhd}} \mathsf{let}\ \{x = \mathsf{let}\ \{\vec{z} = \vec{M}\}\ \mathsf{in}\ N, \vec{y} = \vec{L}\}\ \mathsf{in}\ N' \qquad (\textit{let-let})$$

$$\mathbb{C}[\mathsf{let}\ \{\vec{y} = \vec{V}\}\ \mathsf{in}\ M] \mathrel{\underset{\sim}{\lhd\!\rhd}} \mathsf{let}\ \{\vec{y} = \vec{V}\}\ \mathsf{in}\ \mathbb{C}[M] \qquad (\textit{let-float-val})$$

$$\mathsf{let}\ \{\vec{x} = \vec{V}\sigma_1, \vec{y} = \vec{V}\sigma_2, \vec{z} = \vec{M}\}\ \mathsf{in}\ N \mathrel{\underset{\sim}{\lhd\!\rhd}} \mathsf{let}\ \{\vec{x} = \vec{V}\sigma_2\sigma_3, \vec{z} = \vec{M}\sigma_3\}\ \mathsf{in}\ N\sigma_3,$$
$$\sigma_1 = [{}^{\vec{y}}\!/\!_{\vec{w}}], \sigma_2 = [{}^{\vec{x}}\!/\!_{\vec{w}}], \sigma_3 = [{}^{\vec{x}}\!/\!_{\vec{y}}], \qquad (\textit{value-copy})$$

Fig. 4. Laws for dealing with lets.

$$\Omega \mathrel{\underset{\sim}{\rhd}} M \qquad (\Omega)$$

$$M \mathrel{\underset{\sim}{\rhd}} \Omega, \quad \text{iff } M \cong \Omega \qquad (\textit{imp-}\Omega)$$

$$M \cong \Omega, \quad \text{iff } M \mathrel{\underset{\sim}{\rhd}} {}^{\checkmark}M \qquad (\textit{diverge})$$

$$\mathsf{let}\ \{x = \Omega, \vec{y} = \vec{\mathbb{D}}[x]\}\ \mathsf{in}\ \mathbb{C}[x] \mathrel{\underset{\sim}{\lhd\!\rhd}} \mathsf{let}\ \{x = \Omega, \vec{y} = \vec{\mathbb{D}}[\Omega]\}\ \mathsf{in}\ \mathbb{C}[\Omega] \qquad (\Omega\text{-}\beta)$$

$$\mathbb{C}[\mathsf{let}\ \{y = \Omega\}\ \mathsf{in}\ M] \mathrel{\underset{\sim}{\lhd\!\rhd}} \mathsf{let}\ \{y = \Omega\}\ \mathsf{in}\ \mathbb{C}[M] \qquad (\textit{let-float-}\Omega)$$

$$\mathbb{C}[{}^{\checkmark}M] \mathrel{\underset{\sim}{\rhd}} {}^{\checkmark}\mathbb{C}[M], \quad \text{if } \mathbb{C} \text{ is strict} \qquad (\checkmark\textit{-float})$$

Fig. 5. Laws for Ω and strictness.

$$\mathsf{let}\ \{x = M, \vec{y} = \vec{\mathbb{D}}[{}^{\checkmark}M]\}\ \mathsf{in}\ \mathbb{C}[{}^{\checkmark}M] \mathrel{\underset{\sim}{\rhd}} \mathsf{let}\ \{x = M, \vec{y} = \vec{\mathbb{D}}[x]\}\ \mathsf{in}\ \mathbb{C}[x] \qquad (\beta\textit{-expand})$$

Fig. 6. Beta expansion conjecture.

where N is not a variable. There is a similar derived law for general case expressions.

6.2 Laws for Evaluation Contexts

An *evaluation context* is a context in which the hole is the target of evaluation; in other words, evaluation cannot proceed until the hole is filled. Evaluation contexts have the following form:

$$\begin{array}{rl}
\mathbb{E} ::= & \mathbb{A} \\
\mid & \mathsf{let}\ \{\vec{x} = \vec{M}\}\ \mathsf{in}\ \mathbb{A} \\
\mid & \mathsf{let}\ \{\vec{y} = \vec{M}, \\
& \quad x_0 = \mathbb{A}_0[x_1], \\
& \quad x_1 = \mathbb{A}_1[x_2], \\
& \quad \ldots \\
& \quad x_n = \mathbb{A}_n\} \\
& \mathsf{in}\ \mathbb{A}[x_0] \\
\mathbb{A} ::= & [\cdot] \\
\mid & \mathbb{A}\, x \\
\mid & \mathsf{case}\ \mathbb{A}\ \mathsf{of}\ \{c_i\, \vec{x}_i \to M_i\}.
\end{array}$$

$\mathbb{E}$ ranges over evaluation contexts, and $\mathbb{A}$ over what we call *applicative contexts.* Our evaluation contexts are strictly contained in those mentioned in Ariola and Felleisen's letrec extension [Ariola and Felleisen 1997] of the call-by-need calculus: there they allow $\mathbb{E}$ to appear anywhere we have an $\mathbb{A}$. Our "flattened" definition corresponds exactly to configuration contexts (with a single hole) of the form $\langle\, \Gamma,\ [\cdot],\ S\,\rangle$, as made precise by the following lemma, where $\Lambda_{\mathbb{E}}$ is the set of all evaluation contexts.

Lemma 6.1. $\Lambda_{\mathbb{E}} = \{\mathsf{trans}\langle\, \Gamma,\ [\cdot],\ S\,\rangle \mid \textit{all}\ \Gamma, S\}$.

The two laws in figure 3 are very useful indeed: they allow us to move cases and lets in and out of evaluation contexts. A common motif in proofs using the tick algebra is the use of (*case*-$\mathbb{E}$) and (*let*-$\mathbb{E}$) to expose the sub-term of interest. Their validity follows easily from a simple lemma (presented in section 11).

($\checkmark$-$\mathbb{E}$) allows us to move ticks in and out of evaluation contexts. It follows by a simple use of the context lemma and the properties of the (*Tick*) transition. Another useful law is (*inline*-$\mathbb{E}$), which allows us to inline x if it is used but once in an evaluation context. It follows by similar reasoning to ($\checkmark$-$\mathbb{E}$).

6.3 Concerning Lets

Some of the laws that allow us to manipulate lets are presented in figure 4. Law (*gc*) corresponds to garbage collection: it allows us to add or remove superfluous bindings. Laws (*let-flatten*) and (*let-let*) allow bindings to move across each other, and law (*let-float-val*) concerns the movement of value bindings in and out of general contexts (*i.e.* including across λs); along with (*let-float*) below, it forms the essence of the full-laziness transformation, as noted in [Peyton Jones et al. 1996]). The last law, (*value-copy*) says that if we have two copies of a strongly-connected component of the heap (composed solely of values), then we may remove one of them, provided we perform some renaming.

Note that in, for example, the (*let-let*) axiom, the variable convention ensures that the $\vec{z}$ do not occur free in the $\vec{L}$; in (*let-float-val*), the convention guarantees that x is not free in the $\vec{V}$.

All of the let laws except (*value-copy*) follow via similar arguments to that for (β) above. (*value-copy*) requires the use of the same general techniques needed to justify the more complex β laws (proof in section 11).

6.4 Divergence and Strictness

Let Ω denote any closed term which does not converge. For example, the "black-hole" term, $\mathsf{let}\ x = x\ \mathsf{in}\ x$, would suffice as a definition for Ω. The first three laws in figure 5 concern Ω and its relationship with $\gtrsim$. (Ω-β) and (*let-float*-Ω) are similar to (*value*-β) and (*let-float-val*) except that Ω is used in place of a value. All of these laws follow in a straightforward manner from the context lemma and the fact that call-by-name termination behaviour is preserved in the call-by-need theory.

We say that a context $\mathbb{C}$ is *strict* if and only if $\mathbb{C}[\Omega] \cong \Omega$. Given this definition, we can float ticks out of any strict context, as stated by ($\checkmark$-*float*). The proof follows by the same techniques used to prove (*value*-β).

It turns out that this tick floating property can be used as a characterisation of strictness: for all $\mathbb{C}$, if $\mathbb{C}[{}^{\checkmark}x] \trianglerighteq\!\!\!\sim {}^{\checkmark}\mathbb{C}[x]$, x fresh, then $\mathbb{C}$ is strict. This follows since, by congruence,

$$\mathsf{let}\ x = \Omega\ \mathsf{in}\ \mathbb{C}[{}^{\checkmark}x] \gtrsim \mathsf{let}\ x = \Omega\ \mathsf{in}\ {}^{\checkmark}\mathbb{C}[x]$$

which implies, by (Ω-β), and (*gc*), that $\mathbb{C}[{}^{\checkmark}\Omega] \gtrsim {}^{\checkmark}\mathbb{C}[\Omega]$. But since ${}^{\checkmark}\Omega \lessgtr \Omega$, by ($\Omega$) and (*imp*-$\Omega$), $\mathbb{C}[\Omega] \gtrsim {}^{\checkmark}\mathbb{C}[\Omega]$. Therefore, by (*diverge*), $\mathbb{C}[\Omega] \cong \Omega$.

6.5 Beta Expansion: A Conjecture

In analogy to (*value*-β), we have (β-*expand*) where values are replaced by general terms:

$$\mathsf{let}\ \{x = M, \vec{y} = \vec{\mathbb{D}}[{}^{\checkmark}M]\}\ \mathsf{in}\ \mathbb{C}[{}^{\checkmark}M] \gtrsim \mathsf{let}\ \{x = M, \vec{y} = \vec{\mathbb{D}}[x]\}\ \mathsf{in}\ \mathbb{C}[x] \quad (\beta\text{-}expand)$$

The intuition here is that the rule undoes a call-by-name computation step (a beta-reduction). This is an improvement providing we can pay for the potential gain that the computation step might have made — which is at most one tick at each occurrence of the variable which is unfolded.

Unfortunately we lack a satisfactory proof for (β-*expand*). The context lemma seems inadequate to establish this property. This seems to be linked to the fact that the axiom embodies the essential difference between call-by-name and call-by-need evaluation, and thus it may be possible to adapt techniques based on redex-marking [Maraist et al. 1998]. However, while we believe the conjecture to be an improvement as regards speed, it can lead to asymptotic worsening of space behaviour [Gustavsson and Sands 1999].

The conjecture can be used to "tie the knot" when deriving cyclic programs. This possible since we allow x to occur free in M. See the last step of the proof of proposition 8.4 for an example of the use of (β-*expand*) in this context.

$$
\begin{array}{rlr}
(\lambda x.M)\,N =_{\text{NEED}} \mathsf{let}\ x = N\ \mathsf{in}\ M & & (\mathsf{let}\text{-}I)\\
\mathsf{let}\ x = V\ \mathsf{in}\ \mathbb{C}[x] =_{\text{NEED}} \mathsf{let}\ x = V\ \mathsf{in}\ \mathbb{C}[V] & & (\mathsf{let}\text{-}V)\\
(\mathsf{let}\ x = L\ \mathsf{in}\ M)\,N =_{\text{NEED}} \mathsf{let}\ x = L\ \mathsf{in}\ M\,N & & (\mathsf{let}\text{-}C)\\
\mathsf{let}\ y = (\mathsf{let}\ x = L\ \mathsf{in}\ M)\ \mathsf{in}\ N =_{\text{NEED}} \mathsf{let}\ x = L\ \mathsf{in}\ \mathsf{let}\ y = M\ \mathsf{in}\ N & & (\mathsf{let}\text{-}A)
\end{array}
$$

Fig. 7. Axioms of the call-by-need calculus of Ariola *et al.*.

Using the conjecture, we can also establish the following:

$$^{\checkmark}(\lambda x.\mathsf{let}\ \{\vec{y} = \vec{L}, \vec{z} = \vec{M}\}\ \mathsf{in}\ N) \underset{\sim}{\trianglerighteq} \mathsf{let}\ \{\vec{y} = \vec{L}\}\ \mathsf{in}\ \lambda x.\mathsf{let}\ \{\vec{z} = \vec{M}\}\ \mathsf{in}\ N \qquad (\textit{let-float})$$

which concerns moving non-value bindings out of λs (where the variable convention ensures that x does not occur free in the $\vec{L}$). As noted above, this is an essential part of the full-laziness transformation. Another consequence of the conjecture is standard common sub-expression elimination:

$$^{\checkmark}\mathbb{C}[^{\checkmark}M] \underset{\sim}{\trianglerighteq} \mathsf{let}\ \{x = M\}\ \mathsf{in}\ \mathbb{C}[x] \qquad (\textit{cse})$$

Again, the convention ensures that any free variables of M are not captured by context $\mathbb{C}$.

7. RELATING THE TICK ALGEBRA AND THE CALCULI

We reproduce the axioms of the call-by-need calculus of [Ariola et al. 1995], in figure 7[2].

The laws collected in figures 2, 3, and 4 subsume the call-by-need lambda calculi (in both cases minus the symmetry law): each calculus rewrite rule of the form $L \to R$ turns out to be an outright improvement, *i.e.* $L \underset{\sim}{\trianglerighteq} R$.

In fact, with the exception of (let-V), they are cost equivalences, so we can reverse the improvement also. As for (let-V), we can reverse the improvement *modulo tick*. In other words, there exists an R', obtained from R by inserting ticks, such that $R' \underset{\sim}{\trianglerighteq} L$. This fact will enable us to prove that any two terms related by these calculi compute within a constant factor of each other in any program context. Thus the best (worst) speedup (resp. slowdown) program obtainable in these calculi is linear.

First it is natural to generalise the idea of improvement modulo ticks.

Definition 6. (Improvement within a Constant Factor) *We say that* M is improved by N within a constant factor*, written* $M \underset{\approx}{\trianglerighteq} N$*, if there exists a* k *such that for all* $\mathbb{C}$ *such that* $\mathbb{C}[M]$ *and* $\mathbb{C}[N]$ *are closed,*

$$\mathbb{C}[M]{\downarrow}^{n} \implies \mathbb{C}[N]{\downarrow}^{\leqslant k(n+1)}.$$

So $M \underset{\approx}{\trianglerighteq} N$ means that N is never more than a constant factor slower than M (but it might still be faster by a non-constant factor). Note that the constant factor is independent of the context of use.

It can be seen that $\underset{\approx}{\trianglerighteq}$ is a precongruence relation (to show transitivity requires a small calculation) and clearly contains the improvement relation.

[2]In the original paper V ranges over variables as well as values. In addition, Ariola and Felleisen [Ariola and Felleisen 1997] restrict $\mathbb{C}$ in (let-V) to be evaluation contexts.

Now we consider a special case of $\underset{\approx}{\triangleright}$, namely programs which only differ by ticks. Let $M \overset{\checkmark}{\to} N$ mean that N can be obtained from M by removing some ticks (from anywhere within the term), and $M \overset{\checkmark}{\sim} N$ mean that there exists an L such that $M \overset{\checkmark}{\to} L$ and $N \overset{\checkmark}{\to} L$. Clearly $\overset{\checkmark}{\to}$ is a precongruence and $\overset{\checkmark}{\sim}$ is a congruence.

Lemma 7.1. $M \overset{\checkmark}{\sim} N \implies M \underset{\approx}{\triangleright} N$.

Proof. (Sketch) Clearly $\overset{\checkmark}{\to} \subseteq \underset{\sim}{\triangleright}$, so it suffices to show that $M \overset{\checkmark}{\to} N \implies N \underset{\approx}{\triangleright} M$. First show that the nesting of ticks in a configuration never increases as computation proceeds (easy to see since the rules never substitute terms for variables). Then let k be the maximum nesting of ticks in M, and show by induction on the length of the computation that $\mathbb{C}[N]{\Downarrow}^{n}$ implies $\mathbb{C}[N]{\Downarrow}^{k(n+1)}$ (strengthening this statement to configurations). □

With this lemma we can establish the following:

Theorem 7.2. *For all terms N and M (of our restricted syntax) if $M =_{\text{NEED}} N$ then $M \underset{\approx}{\triangleright} N$.*

Proof. (Sketch) By induction on the proof of $M =_{\text{NEED}} N$. The base case requires us to show that the (oriented) equations are contained in $\underset{\approx}{\triangleright}$. This follows easily since they are all either improvements or improvements modulo tick. In the inductive cases, the congruence and transitivity rules follow from the inductive hypothesis since $\underset{\approx}{\triangleright}$ is a precongruence. The only difficult case is symmetry. It will be sufficient to prove that reversed equations are contained in $\underset{\approx}{\triangleright}$. For each equation $L =_{\text{NEED}} R$ we have from the laws of the tick algebra either that $R \underset{\sim}{\triangleright} L$, or, in the case of ($\mathsf{let}$-$V$), an R' such that $R' \overset{\checkmark}{\to} R$ and $R' \underset{\sim}{\triangleright} L$. By lemma 7.1 we know that $R \underset{\approx}{\triangleright} R'$, so $R \underset{\approx}{\triangleright} L$ follows from the fact that $\underset{\sim}{\triangleright} \subseteq \underset{\approx}{\triangleright}$ and transitivity of $\underset{\approx}{\triangleright}$. □

Corollary 7.3. *The call-by-need calculus of [Ariola et al. 1995] cannot improve (or worsen) a program by more than a constant factor.*

We are confident that this result can be extended to Ariola and Blom's sharing calculus λ_{OSHARE} [Ariola and Blom 1997] since almost all the rules are represented more or less directly in the collection of improvement laws. It is interesting to note that we assembled our collection of laws "by need", considering what was required to tackle a number of examples, and it was encouraging to find that we had already covered almost all of Ariola and Blom's rules. As it stands however, our (*value-copy*) cost equivalence is not as expressive as Ariola and Blom's value-copy rule.[3] We believe that Ariola and Blom's value-copy rule *is* a cost equivalence, but their formulation of the rule is rather indirect, so it is not obvious to us how to prove this.

8. SYNTACTIC CONTINUITY

We wish to say something meaningful about recursive functions with this theory, and a natural starting point is to attempt to mimic the fixed-point induction Scott-style denotational semantics. Examples of this kind of operational analogue to

[3]Thanks to Stefan Blom for providing an example, and to Zena Ariola for pointing out an error in the use of an earlier formulation of our value-copy rule.

Scott induction for other languages may be found in *e.g.*, [Pitts 1997b; Smith 1991; Mason et al. 1996; Sands 1997; Lassen 1998]; we present the first such result for a call-by-need semantics.

We will use the following mechanism to describe the syntactic unwindings of a recursive function. In the definition, the f_i are distinct, new variables.

Definition 7. $f \overset{0}{=} V \overset{\text{def}}{=} f_0 = \Omega,$
$f \overset{n+1}{=} V \overset{\text{def}}{=} f \overset{n}{=} V, f_{n+1} = V[{}^{f_n}/_f].$

Then, for an f defined by $\mathsf{let}\ \{f = V\}\ \mathsf{in}\ f$, we define the n^{th} unwinding as $\mathsf{let}\ \{f \overset{n}{=} V\}\ \mathsf{in}\ f_n$. If we expand the definition of $f \overset{n}{=} V$, we see that this is really

$$\begin{array}{l} \mathsf{let}\ \{f_0 = \Omega, \\ \qquad f_1 = V[{}^{f_0}/_f], \\ \qquad \ldots \\ \qquad f_n = V[{}^{f_{n-1}}/_f]\} \\ \mathsf{in}\ f_n. \end{array}$$

Note that we have restricted our attention to those f whose defining body is a value; this unwinding trick would not work for general cycles (since loss of sharing would render the exercise pointless). To extend the method to cycles would require some extension to the language, but this would lead to the problem of showing that the extension is conservative with respect to the improvement relation.

The point is that the functions $\mathsf{let}\ \{f \overset{n}{=} V\}\ \mathsf{in}\ f_n$ completely characterise the behaviour of $\mathsf{let}\ \{f = V\}\ \mathsf{in}\ f$. This is the essence of Scott induction. The main property that justifies this is a syntactic notion of continuity, which says that $\mathsf{let}\ \{f = V\}\ \mathsf{in}\ f$ is the least upper bound of chain $\{\mathsf{let}\ \{f \overset{n}{=} V\}\ \mathsf{in}\ f_n\}_{n\geqslant 0}$ and that any M which uses f preserves this property.

We first show that $\{\mathsf{let}\ \{f \overset{n}{=} V\}\ \mathsf{in}\ M[{}^{f_n}/_f]\}_{n\geqslant 0}$ does indeed form a chain with respect to $\gtrsim$, and that $\mathsf{let}\ \{f = V\}\ \mathsf{in}\ M$ is an upper bound of that chain.

Lemma 8.1. $\forall n.\, \mathsf{let}\ \{f \overset{n}{=} V\}\ \mathsf{in}\ M[{}^{f_n}/_f]$
$\quad \gtrsim \mathsf{let}\ \{f \overset{n+1}{=} V\}\ \mathsf{in}\ M[{}^{f_{n+1}}/_f]$
$\quad \gtrsim \mathsf{let}\ \{f = V\}\ \mathsf{in}\ M.$

Proof. We prove only the second improvement, that for all n,

$$\mathsf{let}\ \{f \overset{n}{=} V\}\ \mathsf{in}\ M[{}^{f_n}/_f] \gtrsim \mathsf{let}\ \{f = V\}\ \mathsf{in}\ M.$$

The first follows by a similar argument. We proceed by induction on n. The base case follows easily by (gc) and the Ω laws, and the inductive case follows by this calculation:

$$\begin{array}{lll}
& \mathsf{let}\ \{f \overset{n}{=} V, f_{n+1} = V[{}^{f_n}\!/\!_f]\}\ \mathsf{in}\ M[{}^{f_{n+1}}\!/\!_f] & \\
\underset{\sim}{\lhd\!\rhd} & \mathsf{let}\ \{f \overset{n}{=} V\}\ \mathsf{in}\ \mathsf{let}\ \{f_{n+1} = V[{}^{f_n}\!/\!_f]\}\ \mathsf{in}\ M[{}^{f_{n+1}}\!/\!_f] & (\textit{let-let}) \\
\equiv & \mathsf{let}\ \{f \overset{n}{=} V\}\ \mathsf{in}\ \mathsf{let}\ \{g = V[{}^{f_n}\!/\!_f]\}\ \mathsf{in}\ M[{}^{g}\!/\!_f] & (\text{renaming}) \\
\underset{\sim}{\rhd} & \mathsf{let}\ \{f = V\}\ \mathsf{in}\ \mathsf{let}\ \{g = V[{}^{f}\!/\!_f]\}\ \mathsf{in}\ M[{}^{g}\!/\!_f] & (\text{I.H.}) \\
\underset{\sim}{\lhd\!\rhd} & \mathsf{let}\ \{f = V, g = V\}\ \mathsf{in}\ M[{}^{g}\!/\!_f] & (\textit{let-let}) \\
\underset{\sim}{\lhd\!\rhd} & \mathsf{let}\ \{f = V\}\ \mathsf{in}\ M & (\textit{value-copy}), (\textit{gc})
\end{array}$$

□

To establish syntactic continuity, we will need the following lemma (see section 11 for the proof). It says that if $\mathsf{let}\ \{f = V\}\ \mathsf{in}\ M$ converges then there must exist some unwinding that does so with the same cost.

Lemma 8.2. (Unwinding) *For all* Γ, S, *and* n,

$$\langle \Gamma,\ \mathsf{let}\ \{f = V\}\ \mathsf{in}\ M,\ S\rangle{\downarrow^n} \implies \exists m. \langle \Gamma,\ \mathsf{let}\ \{f \overset{m}{=} V\}\ \mathsf{in}\ M[{}^{f_m}\!/\!_f],\ S\rangle{\downarrow^n}.$$

Theorem 8.3. (Syntactic Continuity) *The following is a sound proof rule:*

$$\frac{\forall n. \mathsf{let}\ \{f \overset{n}{=} V\}\ \mathsf{in}\ M[{}^{f_n}\!/\!_f]\ \underset{\sim}{\rhd}\ N}{\mathsf{let}\ \{f = V\}\ \mathsf{in}\ M\ \underset{\sim}{\rhd}\ N}$$

Proof. Assume $\langle \Gamma,\ \mathsf{let}\ \{f = V\}\ \mathsf{in}\ M,\ S\rangle{\downarrow^n}$. Then by the Unwinding lemma, there exists some m such that $\langle \Gamma,\ \mathsf{let}\ \{f \overset{m}{=} V\}\ \mathsf{in}\ M[{}^{f_m}\!/\!_f],\ S\rangle{\downarrow^n}$. By the premise, we have that $\langle \Gamma,\ N,\ S\rangle{\downarrow^{\leqslant n}}$, and the result follows by the context lemma. □

Syntactic continuity is also valid for mutually recursive functions. This proof rule is sound for strong improvement, but note that the base case of the premise requires that N be contextually equivalent to Ω. This tends to limit the applicability of the strong improvement version of syntactic continuity.

As an example of the use of syntactic continuity, we show that an unwinding fixed-point combinator is improved within a constant factor by a "knot-tying" fixed-point combinator.

Proposition 8.4. *If* (*β-expand*) *is valid, then*

$$\mathsf{let}\ \mathit{rec} = (\lambda f. \mathsf{let}\ x = \mathit{rec}\, f\ \mathsf{in}\ f\, x)\ \mathsf{in}\ \mathit{rec} \underset{\approx}{\rhd} \mathsf{let}\ \mathit{fix} = (\lambda f. \mathsf{let}\ x = f\, x\ \mathsf{in}\ x)\ \mathsf{in}\ \mathit{fix}.$$

Proof. Let $V = \lambda f. \mathsf{let}\ x = \mathit{rec}\, f\ \mathsf{in}\ {}^{\checkmark} f\, x$, and abbreviate $V[{}^{\mathit{rec}_n}\!/\!_{\mathit{rec}}]$ by V_n. We will show that for all n, $\mathsf{let}\ \mathit{rec} \overset{n}{=} V\ \mathsf{in}\ \mathit{rec}_n \underset{\sim}{\rhd} {}^{\checkmark}\lambda f. \mathsf{let}\ x = f\, x\ \mathsf{in}\ x$. Then the result will then follow by syntactic continuity, since

$$\begin{array}{lll}
& {}^{\checkmark}\lambda f. \mathsf{let}\ x = f\, x\ \mathsf{in}\ x & \\
\underset{\sim}{\lhd\!\rhd} & \mathsf{let}\ \mathit{fix} = (\lambda f. \mathsf{let}\ x = f\, x\ \mathsf{in}\ x) & (\textit{gc}) \\
& \quad \mathsf{in}\ {}^{\checkmark}\lambda f. \mathsf{let}\ x = f\, x\ \mathsf{in}\ x & \\
\underset{\sim}{\lhd\!\rhd} & \mathsf{let}\ \mathit{fix} = (\lambda f. \mathsf{let}\ x = f\, x\ \mathsf{in}\ x)\ \mathsf{in}\ \mathit{fix} & (\textit{value-}\beta)
\end{array}$$

$$\begin{array}{ll}
\mathsf{let}\ rec \overset{n}{=} V, rec_{n+1} = V_n\ \mathsf{in}\ rec_{n+1} & \\
\underset{\sim}{\triangleleft\!\triangleright}\ \mathsf{let}\ rec \overset{n}{=} V, rec_{n+1} = V_n\ \mathsf{in}\ {}^{\checkmark}V_n & (\textit{value-}\beta) \\
\underset{\sim}{\triangleleft\!\triangleright}\ \mathsf{let}\ rec \overset{n}{=} V\ \mathsf{in}\ {}^{\checkmark}\lambda f.\mathsf{let}\ x = rec_n\, f\ \mathsf{in}\ {}^{\checkmark}f\,x & (\textit{gc}),\ (\text{defn. of } V_n) \\
\underset{\sim}{\triangleleft\!\triangleright}\ {}^{\checkmark}\lambda f.\mathsf{let}\ rec \overset{n}{=} V, x = rec_n\, f\ \mathsf{in}\ {}^{\checkmark}f\,x & (\textit{let-float-val}), (\textit{let-float-}\Omega) \\
\underset{\sim}{\triangleleft\!\triangleright}\ {}^{\checkmark}\lambda f.\mathsf{let}\ x = (\mathsf{let}\ rec \overset{n}{=} V\ \mathsf{in}\ rec_n)\, f\ \mathsf{in}\ {}^{\checkmark}f\,x & (\textit{let-let}), (\textit{let-}\mathbb{E}) \\
\underset{\sim}{\triangleright}\ {}^{\checkmark}\lambda f.\mathsf{let}\ x = ({}^{\checkmark}\lambda g.\mathsf{let}\ y = g\,y\ \mathsf{in}\ y)\, f\ \mathsf{in}\ {}^{\checkmark}f\,x & (\text{I.H.}), (\text{renaming}) \\
\underset{\sim}{\triangleleft\!\triangleright}\ {}^{\checkmark}\lambda f.\mathsf{let}\ x = {}^{\checkmark}\mathsf{let}\ y = f\,y\ \mathsf{in}\ y\ \mathsf{in}\ {}^{\checkmark}f\,x & (\beta) \\
\underset{\sim}{\triangleleft\!\triangleright}\ {}^{\checkmark}\lambda f.\mathsf{let}\ x = {}^{\checkmark}y, y = f\,y\ \mathsf{in}\ {}^{\checkmark}f\,x & (\textit{let-let}) \\
\underset{\sim}{\triangleright}\ {}^{\checkmark}\lambda f.\mathsf{let}\ x = y, y = f\,y\ \mathsf{in}\ {}^{\checkmark}f\,y & (\checkmark\textit{-elim}), (\textit{var-subst}) \\
\underset{\sim}{\triangleleft\!\triangleright}\ {}^{\checkmark}\lambda f.\mathsf{let}\ x = f\,x\ \mathsf{in}\ {}^{\checkmark}f\,x & (\textit{gc}), (\text{renaming}) \\
\underset{\sim}{\triangleright}\ {}^{\checkmark}\lambda f.\mathsf{let}\ x = f\,x\ \mathsf{in}\ x & (\beta\textit{-expand})
\end{array}$$

Fig. 8. The inductive case for proposition 8.4.

We proceed via induction on n. The base case follows trivially by (*imp*-Ω) and (Ω) since $\mathsf{let}\ rec_0 = \Omega\ \mathsf{in}\ rec_0 \cong \Omega$, and the inductive case follows by the derivation in figure 8. We have $\underset{\approx}{\triangleright}$ and not $\underset{\sim}{\triangleright}$ because we use a slightly slower version of *rec*. □

The converse of the proposition is false, since the knot-tying fixed-point combinator can give asymptotically better programs.

We can also use syntactic continuity to establish the following proof rule, which is a syntactic, call-by-need version of what is called *fixed-point fusion* in [Meijer et al. 1991]. In the statement, $\mathbb{V}$ and $\mathbb{W}$ range over value contexts.

THEOREM 8.5. (IMPROVEMENT FUSION) *If* $\mathbb{C}$ *is strict, and* $\mathbb{C}[\mathbb{V}[x]] \underset{\sim}{\triangleright} \mathbb{W}[\mathbb{C}[x]]$ *where* $x \notin \mathsf{FV}(\mathbb{V},\mathbb{W},\mathbb{C}) \cup \mathsf{CV}(\mathbb{V},\mathbb{W},\mathbb{C})$*, then for all* $\mathbb{D}$ *such that* $x \notin \mathsf{FV}(\mathbb{D}) \cup \mathsf{CV}(\mathbb{D})$*,*

$$\mathsf{let}\ \{x = \mathbb{V}[x]\}\ \mathsf{in}\ \mathbb{D}[\mathbb{C}[x]] \underset{\sim}{\triangleright} \mathsf{let}\ \{x = \mathbb{W}[x]\}\ \mathsf{in}\ \mathbb{D}[x].$$

PROOF. Assume $\mathbb{C}$ is strict, and that $\mathbb{C}[\mathbb{V}[x]] \underset{\sim}{\triangleright} \mathbb{W}[\mathbb{C}[x]]$. By syntactic continuity, it suffices to show, for all n and all $\mathbb{D}$ such that $x \notin \mathsf{FV}(\mathbb{D}) \cup \mathsf{CV}(\mathbb{D})$,

$$\mathsf{let}\ \{x \overset{n}{=} \mathbb{V}[x]\}\ \mathsf{in}\ \mathbb{D}[\mathbb{C}[x_n]] \underset{\sim}{\triangleright} \mathsf{let}\ \{x = \mathbb{W}[x]\}\ \mathsf{in}\ \mathbb{D}[x].$$

The base case follows by this calculation:

$$\begin{array}{ll}
\mathsf{let}\ \{x_0 = \Omega\}\ \mathsf{in}\ \mathbb{D}[\mathbb{C}[x_0]] & \\
\underset{\sim}{\triangleleft\!\triangleright}\ \mathsf{let}\ \{x_0 = \Omega\}\ \mathsf{in}\ \mathbb{D}[\mathbb{C}[\Omega]] & (\Omega\text{-}\beta) \\
\underset{\sim}{\triangleleft\!\triangleright}\ \mathsf{let}\ \{x_0 = \Omega\}\ \mathsf{in}\ \mathbb{D}[\Omega] & (\mathbb{C}\ \text{strict}) \\
\underset{\sim}{\triangleleft\!\triangleright}\ \mathbb{D}[\Omega] & (\textit{gc}) \\
\underset{\sim}{\triangleleft\!\triangleright}\ \mathsf{let}\ \{x = \mathbb{W}[x]\}\ \mathsf{in}\ \mathbb{D}[\Omega] & (\textit{gc}) \\
\underset{\sim}{\triangleright}\ \mathsf{let}\ \{x = \mathbb{W}[x]\}\ \mathsf{in}\ \mathbb{D}[x] & (\Omega \underset{\sim}{\triangleright} x), (\text{cong.})
\end{array}$$

and the inductive case by this calculation:

$$
\begin{array}{lll}
& \mathsf{let}\ \{x \overset{n+1}{=} \mathbb{V}[x]\}\ \mathsf{in}\ \mathbb{D}[\mathbb{C}[x_{n+1}]] & \\
\underset{\sim}{\lhd\!\rhd} & \mathsf{let}\ \{x \overset{n}{=} \mathbb{V}[x]\}\ \mathsf{in}\ \mathbb{D}[\mathbb{C}[{}^{\checkmark}\mathbb{V}[x_n]]] & (\textit{value-}\beta), (\textit{gc}) \\
\underset{\sim}{\lhd\!\rhd} & \mathsf{let}\ \{x \overset{n}{=} \mathbb{V}[x]\}\ \mathsf{in}\ \mathbb{D}[{}^{\checkmark}\mathbb{C}[\mathbb{V}[x_n]]] & (\mathbb{C}\text{ strict}) \\
\underset{\sim}{\rhd} & \mathsf{let}\ \{x \overset{n}{=} \mathbb{V}[x]\}\ \mathsf{in}\ \mathbb{D}[{}^{\checkmark}\mathbb{W}[\mathbb{C}[x_n]]] & (\text{assumption}) \\
\underset{\sim}{\rhd} & \mathsf{let}\ \{x = \mathbb{W}[x]\}\ \mathsf{in}\ \mathbb{D}[{}^{\checkmark}\mathbb{W}[x]] & (\text{I.H.}) \\
\underset{\sim}{\lhd\!\rhd} & \mathsf{let}\ \{x = \mathbb{W}[x]\}\ \mathsf{in}\ \mathbb{D}[x] & (\textit{value-}\beta)
\end{array}
$$

□

Fixed-point fusion can be used to establish a number of general fusion laws. It is also central to Tullsen and Hudak's [Tullsen and Hudak 1998] approach to program transformation in Haskell.

9. THE IMPROVEMENT THEOREM

In this section we introduce a second key technique for reasoning about recursion, the improvement theorem. In [Sands 1996] a call-by-name improvement theorem was introduced as a means to prove the extensional correctness of recursion-based program transformations. In this section we show how these results carry over to the call-by-need setting.

9.1 The Problem of Transformations

As a motivation for the improvement theorem, consider the correctness problem for recursion-based program transformations such as *unfold-fold*; the correctness of such transformations does not follow from the simple fact that the basic transformation steps are equivalences. To take a simple example to illustrate the problem, consider the following "transformation by equivalence-preserving steps". Start with the recursive function *repeat* which produces the "infinite" list of its argument:

$$\textit{repeat}\, x = x : (\textit{repeat}\, x)$$

The following property can be easily deduced: $\textit{repeat}\, x \cong \mathsf{tail}(\textit{repeat}\, x)$. Now suppose that we use this "local equivalence" to transform the body of the function to obtain a new version of the function:

$$\textit{repeat}\, x = x : (\mathsf{tail}\,(\textit{repeat}\, x))$$

This definition is not equivalent to the original, since it can never produce more than first element in the list. How did equivalence-preserving local steps produce a non-equivalent function? Analysing such transformations more carefully we see that while it is true that

$$M \cong N \implies \mathsf{let}\ \{x = M\}\ \mathsf{in}\ L \cong \mathsf{let}\ \{x = N\}\ \mathsf{in}\ L \tag{9.1}$$

it is no longer the case when the transformation from M to N depends on the recursive definition of x itself:

$$
\begin{array}{l}
\mathsf{let}\ \{x = M\}\ \mathsf{in}\ M \cong \mathsf{let}\ \{x = M\}\ \mathsf{in}\ N \\
\not\Longrightarrow \mathsf{let}\ \{x = M\}\ \mathsf{in}\ L \cong \mathsf{let}\ \{x = N\}\ \mathsf{in}\ L.
\end{array}
$$

But in order to reason about "interesting" program transformations (*e.g.* unfold-fold, recursion-based deforestation, partial evaluation with memoization), inference (9.1) is simply not sufficient.

The improvement theorem comes to the rescue:

$$\frac{\mathsf{let}\ \{x = M\}\ \mathsf{in}\ M \mathrel{\underset{\sim}{\trianglerighteq}} \mathsf{let}\ \{x = M\}\ \mathsf{in}\ N}{\mathsf{let}\ \{x = M\}\ \mathsf{in}\ L \mathrel{\underset{\sim}{\trianglerighteq}} \mathsf{let}\ \{x = N\}\ \mathsf{in}\ L} \tag{9.2}$$

This is sufficient to establish the correctness of recursion-based transformations by requiring — rather naturally — that the local transformation steps are also improvements. This was proved for an improvement theory based on call-by-name, so the fact that the theorem gives "improved" programs as well as correctness is not considered to be particularly significant.

A question left open was whether the improvement theorem holds for a call-by-need improvement theory. We can now supply the answer:

Theorem 9.1. (Improvement Theorem) *The following proof rule is sound:*

$$\frac{\mathsf{let}\ \{f = V\}\ \mathsf{in}\ V \mathrel{\underset{\sim}{\trianglerighteq}} \mathsf{let}\ \{f = V\}\ \mathsf{in}\ W}{\mathsf{let}\ \{f = V\}\ \mathsf{in}\ N \mathrel{\underset{\sim}{\trianglerighteq}} \mathsf{let}\ \{f = W\}\ \mathsf{in}\ N}$$

The inference is also sound when $\underset{\sim}{\trianglerighteq}$ *is replaced throughout with* $\underset{\sim}{\triangleleft\!\triangleright}$ *(the cost equivalence theorem).*

The improvement theorem and the cost equivalence theorem can also be stated for a set of mutually recursive definitions. The proof of the theorem is in section 11.

Notation. In establishing a premise of the improvement theorem, in the context of some recursive declarations $\vec{g} = \vec{V}$, a derivation of the form

$$\begin{aligned} \mathsf{let}\ \{\vec{g} = \vec{V}\}\ \mathsf{in}\ M_1 &\mathrel{\underset{\sim}{\trianglerighteq}} \mathsf{let}\ \{\vec{g} = \vec{V}\}\ \mathsf{in}\ M_2 \\ &\mathrel{\underset{\sim}{\trianglerighteq}} \mathsf{let}\ \{\vec{g} = \vec{V}\}\ \mathsf{in}\ M_3 \ldots \end{aligned}$$

will be written in the following abbreviated form:

$$\begin{aligned} \vec{g} \vdash M_1 &\mathrel{\underset{\sim}{\trianglerighteq}} M_2 \\ &\mathrel{\underset{\sim}{\trianglerighteq}} M_3 \ldots \end{aligned}$$

when the declarations $\vec{g}$ are clear from the context. This is of course of limited use without the following congruence rule:

$$\frac{\vec{g} \vdash M \mathrel{\underset{\sim}{\trianglerighteq}} N}{\vec{g} \vdash \mathbb{C}[M] \mathrel{\underset{\sim}{\trianglerighteq}} \mathbb{C}[N]} \tag{$\vdash$-cong}$$

for all contexts $\mathbb{C}$. It says that any improvement proven in the context of some recursive definitions may be lifted to all contexts.

The following example illustrates the use of the proof rule, which shows that a representation of the standard lambda-calculus fixed-point combinator

$$Y = \lambda f.f\ ((\lambda x.f\ (x\,x))\ \lambda x.f\ (x\,x))$$

(suitably converted to the restricted syntax) is cost equivalent to the non-cyclic version *rec* from proposition 8.4.

PROPOSITION 9.2.

$$\begin{array}{l}\mathsf{let}\ Y = \lambda f.\mathsf{let}\ d = \lambda y.\mathsf{let}\ z = y\,y\ \mathsf{in}\ f\,z \\ \qquad\qquad\qquad\quad x = d\,d \\ \qquad\qquad\quad \mathsf{in}\ f\,x \\ \mathsf{in}\ Y \\ \quad \mathrel{\underset{\sim}{\lhd\!\rhd}} \mathsf{let}\ \mathit{rec} = \lambda f.\mathsf{let}\ x = \mathit{rec}\,f\ \mathsf{in}\ f\,x. \\ \qquad \mathsf{in}\ \mathit{rec}\end{array}$$

PROOF. To use the cost equivalence theorem, we are required to show that

$$\begin{array}{l}\mathsf{let}\ Y = \cdots \\ \mathsf{in}\ \lambda f.\mathsf{let}\ d = \lambda y.\mathsf{let}\ z = y\,y\ \mathsf{in}\ f\,z \\ \qquad\qquad\quad x = d\,d \\ \qquad\quad \mathsf{in}\ f\,x \\ \quad \mathrel{\underset{\sim}{\lhd\!\rhd}} \mathsf{let}\ Y = \cdots \\ \qquad \mathsf{in}\ \lambda f.\mathsf{let}\ x = Y\,f\ \mathsf{in}\ f\,x\end{array}$$

where *rec* has been renamed (without loss of generality) to Y. Using the entailment notation, we need to show:

$$\begin{array}{l}Y \vdash \lambda f.\mathsf{let}\ d = \lambda y.\mathsf{let}\ z = y\,y\ \mathsf{in}\ f\,z \\ \qquad\qquad\quad x = d\,d \\ \qquad\quad \mathsf{in}\ f\,x \\ \quad \mathrel{\underset{\sim}{\lhd\!\rhd}} \lambda f.\mathsf{let}\ x = Y\,f\ \mathsf{in}\ f\,x.\end{array}$$

By calculation, we have that:

$$\begin{array}{ll}Y \vdash \lambda f.\mathsf{let}\ d = \lambda y.\mathsf{let}\ z = y\,y\ \mathsf{in}\ f\,z & \\ \qquad\qquad\quad x = d\,d & \\ \qquad\quad \mathsf{in}\ f\,x & \\ \quad \mathrel{\underset{\sim}{\lhd\!\rhd}} \lambda f.\mathsf{let}\ d = \lambda y.\mathsf{let}\ z = y\,y\ \mathsf{in}\ f\,z & (\mathit{value}\text{-}\beta), (\beta) \\ \qquad\qquad\quad x = {}^{\checkmark}\mathsf{let}\ z = d\,d\ \mathsf{in}\ f\,z & \\ \qquad\quad \mathsf{in}\ f\,x & \\ \quad \mathrel{\underset{\sim}{\lhd\!\rhd}} \lambda f.\mathsf{let}\ x = {}^{\checkmark}\mathsf{let}\ d = \lambda y.\mathsf{let}\ z = y\,y\ \mathsf{in}\ f\,z & (\mathit{let}\text{-}\mathit{let}) \\ \qquad\qquad\qquad\qquad\quad z = d\,d & \\ \qquad\qquad\qquad\quad \mathsf{in}\ f\,z & \\ \qquad\quad \mathsf{in}\ f\,x & \\ \quad \mathrel{\underset{\sim}{\lhd\!\rhd}} \lambda f.\mathsf{let}\ x = {}^{\checkmark}(\lambda g.\mathsf{let}\ d = \lambda y.\mathsf{let}\ z = y\,y\ \mathsf{in}\ g\,z & (\beta) \\ \qquad\qquad\qquad\qquad\qquad z = d\,d & \\ \qquad\qquad\qquad\qquad \mathsf{in}\ g\,z)\,f & \\ \qquad\quad \mathsf{in}\ f\,x & \\ \quad \mathrel{\underset{\sim}{\lhd\!\rhd}} \lambda f.\mathsf{let}\ x = Y\,f\ \mathsf{in}\ f\,x & (\mathit{value}\text{-}\beta)\end{array}$$

Then the result follows by the cost equivalence theorem. □

Improvement Theorem vs. Syntactic Continuity. Suppose one wants to establish an improvement of the form

$$\mathsf{let}\ \{f = V\}\ \mathsf{in}\ N \mathrel{\underset{\sim}{\rhd}} \mathsf{let}\ \{f = W\}\ \mathsf{in}\ N.$$

If the left-hand side is non-recursive (in f) then syntactic continuity is of no help, since the unwindings (> 0) of the left-hand side will all be identical; conversely, if the right-hand side is non recursive (in f) then the improvement theorem is not immediately useful, since proving the premise amounts to directly proving the conclusion of the rule. There are, however, many examples which can be proved by both methods. In these cases the improvement theorem is often preferable since it is more calculational in style.

9.2 Improvement Induction

Finally, we mention one last proof rule which is closely allied to the improvement theorem (in the sense that a closely-related rule can be derived from the improvement theorem); this corresponds to what we called *improvement induction* in [Sands 1997], where it was established for any call-by-name or call-by-value language with SOS rules fitting a certain syntactic rule-format.

THEOREM 9.3. (IMPROVEMENT INDUCTION) *For any M, N, $\mathbb{C}$, and substitution σ, the following proof rule is sound:*

$$\frac{\vec{f} \vdash M \underset{\sim}{\triangleright} {}^{\checkmark}\mathbb{C}[M\sigma] \quad \vec{f} \vdash N \underset{\sim}{\triangleleft\!\triangleright} {}^{\checkmark}\mathbb{C}[N\sigma]}{\vec{f} \vdash M \underset{\sim}{\triangleright} N}$$

The proof is quite straightforward, and is given in section 11. A example of the proof technique is provided in section 10.

10. AN EXAMPLE PROGRAM TRANSFORMATION

In this section we consider a larger example of a program transformation — an automatic method for eliminating calls to the append function. The transfomation is something of a classic, and can be viewed as an instance of the unfold-fold scheme [Burstall and Darlington 1977]. The particular mechanisation described here is based on [Wadler 1988]. The example was used previously to illustrate the improvement theorem for call-by-name evaluation [Sands 1996]. Here we show that the correctness argument there can be strengthened to encompass a guarantee of call-by-need improvement.

10.1 The Concatenate Vanishes

The basic idea is to eliminate occurrences of the list-concatenate function:

$$\begin{array}{l} (+\!\!+) = \lambda xs.\lambda ys.\mathsf{case}\ xs\ \mathsf{of} \\ \qquad\qquad\qquad\qquad \mathsf{nil} \ \ \ \rightarrow ys \\ \qquad\qquad\qquad\qquad h : t \rightarrow h : (t +\!\!+ ys), \end{array}$$

when it occurs to the right of a function application, as in: $f\,y_1 \ldots y_n +\!\!+ z$. This is achieved by by defining and optimising a function f^+ which satisfies

$$f^+\,y_1 \ldots y_n\,z \cong (f\,y_1 \ldots y_n) +\!\!+ z.$$

We present the transformation in two phases: *initialization*, which introduces an initial definition for f^+, and *transformation*, which applies a set of rewrites to terms in the scope of these definitions. Throughout we assume that the definition of the append function is in scope. To ease the notation, we will occasionally make use of the syntactic identity for general application from section 3, and we will use an infix form of append.

Initialization. The target of the transformation is a function definition $f = \lambda x_1 \ldots \lambda x_n.M$, for which there is an occurrence of a term $(f\, y_1 \ldots y_n) \mathbin{+\!\!+} z$ in the program. The initial step is to replace the definition by the pair:

$$f = \lambda x_1 \ldots \lambda x_n.f^{+} x_1 \ldots x_n\, \mathsf{nil}$$
$$f^{+} = \lambda x_1 \ldots \lambda x_n.\lambda z.M \mathbin{+\!\!+} z$$

Transformation. Apply the following rewrite rules, in any order, to all expressions in the scope of the above definitions:

$$\mathsf{let}\ y = \mathsf{nil}\ \mathsf{in}\ \mathbb{C}[y \mathbin{+\!\!+} x] \rightarrow \mathsf{let}\ y = \mathsf{nil}\ \mathsf{in}\ \mathbb{C}[x] \tag{i}$$
$$\mathsf{let}\ w = x : y\ \mathsf{in}\ \mathbb{C}[w \mathbin{+\!\!+} z] \rightarrow \mathsf{let}\ w = x : y\ \mathsf{in}\ \mathbb{C}[x : (y \mathbin{+\!\!+} z)] \tag{ii}$$
$$(x \mathbin{+\!\!+} y) \mathbin{+\!\!+} z \rightarrow x \mathbin{+\!\!+} (y \mathbin{+\!\!+} z) \tag{iii}$$
$$(\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \rightarrow N_i\}) \mathbin{+\!\!+} z \rightarrow \mathsf{case}\ M\ \mathsf{of}\ \{pat_i \rightarrow N_i \mathbin{+\!\!+} z\} \tag{iv}$$
$$(f\, z_1 \ldots z_n) \mathbin{+\!\!+} z \rightarrow f^{+}\, z_1 \ldots z_n\, z \tag{v}$$
$$(f^{+}\, z_1 \ldots z_n\, z) \mathbin{+\!\!+} z' \rightarrow f^{+}\, z_1 \ldots z_n\ , (z \mathbin{+\!\!+} z') \tag{vi}$$
$$(\mathsf{let}\ \vec{x} = \vec{M}\ \mathsf{in}\ N) \mathbin{+\!\!+} z \rightarrow \mathsf{let}\ \vec{x} = \vec{M}\ \mathsf{in}\ N \mathbin{+\!\!+} z \tag{vii}$$

10.2 An Example Application

The classic example of this transformation is the conversion of a naïve quadratic time list reverse function into a linear time version. Suppose we have the definition

$$\begin{aligned} reverse = \lambda xs.\mathsf{case}\ xs\ \mathsf{of} & \\ & \mathsf{nil} \rightarrow \mathsf{nil} \\ & h : t \rightarrow \mathsf{let}\ z = [\,h\,]\ \mathsf{in}\ (reverse\, t) \mathbin{+\!\!+} z \end{aligned}$$

The expression $(reverse\, t) \mathbin{+\!\!+} z$ is a candidate for the transformation, so initialisation yields:

$$reverse = \lambda xs.reverse^{+}\ xs\, \mathsf{nil}$$
$$reverse^{+} = \lambda xs.\lambda z. \begin{pmatrix} \mathsf{case}\ xs\ \mathsf{of} & \\ \quad \mathsf{nil} \rightarrow \mathsf{nil} & \\ \quad h : t \rightarrow \mathsf{let}\ y = [\,h\,]\ \mathsf{in}\ (reverse\, t) \mathbin{+\!\!+} y \end{pmatrix} \mathbin{+\!\!+} z$$

Now we apply the transformation rules to the program. We will also use garbage collection to remove redundant bindings. The important part is the application to the case expression in the right hand side of the definition of $reverse^{+}$. The transformation is presented in figure 9.

10.3 Correctness

We have seen, with the standard reverse example, that the transformation can achieve asymptotic program speedups. In the remainder of this section we use the improvement theory to prove that the method described can never slow down programs by more than a constant factor.

For the correctness argument we make a simplification to the initialisation phase: we will not modify the definition of the original function f. The effect of this

$$
\begin{array}{ll}
\left(\begin{array}{l}\textsf{case}\ xs\ \textsf{of}\\ \quad \textsf{nil} \quad \to \textsf{nil}\\ \quad h : t \to \textsf{let}\ y = [\,h\,]\ \textsf{in}\ (\mathit{reverse}\,t) +\!\!+\, y\end{array}\right) +\!\!+\, z & \\[2ex]
\to \textsf{case}\ xs\ \textsf{of} & \text{(iv)}\\
\quad \textsf{nil} \quad \to \textsf{nil} +\!\!+\, z & \\
\quad h : t \to (\textsf{let}\ y = [\,h\,]\ \textsf{in}\ (\mathit{reverse}\,t) +\!\!+\, y) +\!\!+\, z & \\[2ex]
\to \textsf{case}\ xs\ \textsf{of} & \text{(i)}, \text{(vii)}, (gc)\\
\quad \textsf{nil} \quad \to z & \\
\quad h : t \to (\textsf{let}\ y = [\,h\,]\ \textsf{in}\ (\mathit{reverse}\,t) +\!\!+\, y) +\!\!+\, z & \\[2ex]
\to \textsf{case}\ xs\ \textsf{of} & \text{(vii)}, \text{(iii)}, \text{(vi)}\\
\quad \textsf{nil} \quad \to z & \\
\quad h : t \to \textsf{let}\ y = [\,h\,]\ \textsf{in}\ \mathit{reverse}^{+}\,t\,(y +\!\!+\, z) & \\[2ex]
\to \textsf{case}\ xs\ \textsf{of} & \text{(ii)}\\
\quad \textsf{nil} \quad \to z & \\
\quad h : t \to \textsf{let}\ y = [\,h\,]\ \textsf{in}\ \mathit{reverse}^{+}\,t\,(h : (\textsf{nil} +\!\!+\, z)) & \\[2ex]
\to \textsf{case}\ xs\ \textsf{of} & \text{(i)}, (gc)\\
\quad \textsf{nil} \quad \to z & \\
\quad h : t \to \mathit{reverse}^{+}\,t\,(h : z) &
\end{array}
$$

Fig. 9. Example transformation sequence.

simplification is to cause duplicated transformation work (and some duplicated code) in some examples — but is not otherwise significant. The reason for this simplification is that replacing the body of f by $f^{+}x_1 \ldots x_n\,\textsf{nil}$ is not sound in an untyped language — since it relies on the equality $x = x +\!\!+\, \textsf{nil}$. In a typed theory it would be straightforward to establish that this is a weak cost equivalence — but a typed theory is beyond the scope of the present article.

The architecture of the proof is as follows. The introduction of the new function is merely garbage-introduction, so is patently sound. The remaining steps illustrate the use of:

—basic laws to establish that the remaining laws are all improvements;

—improvement induction, to establish associativity properties of append, and

—the use of the above properties together with the improvement theorem to establish the property of the transformation as a whole.

10.4 Properties of Append

PROPOSITION 10.1.

$$
\begin{array}{r}
(+\!\!+) \vdash \textsf{let}\ y = \textsf{nil}\ \textsf{in}\ \mathbb{C}[y +\!\!+\, x] \mathrel{\underset{\sim}{\lhd\!\rhd}} \textsf{let}\ y = \textsf{nil}\ \textsf{in}\ \mathbb{C}[{}^{2\checkmark}x]\\
(+\!\!+) \vdash \textsf{let}\ w = x : y\ \textsf{in}\ \mathbb{C}[w +\!\!+\, z] \mathrel{\underset{\sim}{\lhd\!\rhd}} \textsf{let}\ w = x : y\ \textsf{in}\ \mathbb{C}[{}^{2\checkmark}x : (y +\!\!+\, z)]\\
(+\!\!+) \vdash (\textsf{case}\ M\ \textsf{of}\ \{\mathit{pat}_i \to N_i\}) +\!\!+\, z \mathrel{\underset{\sim}{\lhd\!\rhd}} \textsf{case}\ M\ \textsf{of}\ \{\mathit{pat}_i \to \textsf{let}\ y = N_i\ \textsf{in}\ y +\!\!+\, z\}\\
(+\!\!+) \vdash (\textsf{let}\ \vec{x} = \vec{M}\ \textsf{in}\ N) +\!\!+\, z \mathrel{\underset{\sim}{\lhd\!\rhd}} \textsf{let}\ \vec{x} = \vec{M}\ \textsf{in}\ N +\!\!+\, z
\end{array}
$$

PROOF. The proofs are routine calculations. We present just the proof of the case property:

$$\begin{array}{lll}
(+\!\!+) \vdash (\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to N_i\}) +\!\!+ z & & \\
\quad \equiv \mathsf{let}\ y = \mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to N_i\}\ \mathsf{in}\ y +\!\!+ z & & \\
\quad \mathrel{\underset{\sim}{\lhd\!\rhd}} \mathsf{let}\ y = \mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to N_i\}\ \mathsf{in}\ {}^{\checkmark}\mathsf{case}\ y\ \mathsf{of} & & (value\text{-}\beta) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{nil} \to z & & \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad h : t \to h : (t +\!\!+ z) & & \\
\quad \mathrel{\underset{\sim}{\lhd\!\rhd}} \mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to \mathsf{let}\ y = N_i\ \mathsf{in}\ {}^{\checkmark}\mathsf{case}\ y\ \mathsf{of} & & (case\text{-}\mathbb{E}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{nil} \to z & & \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad h : t \to h : (t +\!\!+ z)\} & & \\
\quad \mathrel{\underset{\sim}{\lhd\!\rhd}} \mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to \mathsf{let}\ y = N_i\ \mathsf{in}\ y +\!\!+ z\} & & (value\text{-}\beta)
\end{array}$$

□

Append also satisfies associativity properties, which are established below.

Proposition 10.2.

$$(+\!\!+), (\overline{+\!\!+}) \vdash (x +\!\!+ y) +\!\!+ z \mathrel{\underset{\sim}{\lhd\!\rhd}} x \,\overline{+\!\!+}\, (y +\!\!+ z)$$

where $(\overline{+\!\!+}) = \lambda xs.\lambda ys.\mathsf{case}\ xs\ \mathsf{of}$

$$\begin{array}{l}
\mathsf{nil} \to ys \\
h : t \to {}^{2\checkmark} h : (t \,\overline{+\!\!+}\, ys).
\end{array}$$

Proof. We calculate with the left and right-hand sides independently, and find a context $\mathbb{D}$ such that

$$x +\!\!+ (y \,\overline{+\!\!+}\, z) \mathrel{\underset{\sim}{\lhd\!\rhd}} {}^{\checkmark}\mathbb{D}[x \,\overline{+\!\!+}\, (y +\!\!+ z)]$$

and also that

$$(x +\!\!+ y) +\!\!+ z \mathrel{\underset{\sim}{\lhd\!\rhd}} {}^{\checkmark}\mathbb{D}[(x +\!\!+ y) +\!\!+ z]$$

and the result then follows by improvement induction.

$$
\begin{array}{lll}
\multicolumn{3}{l}{(\overline{+\!\!+}), (+\!\!+) \vdash (x +\!\!+ y) +\!\!+ z} \\
 & \equiv \mathsf{let}\ w = x +\!\!+ y\ \mathsf{in}\ w +\!\!+ z & \\
 & \mathrel{\underset{\sim}{\lhd\!\rhd}} \mathsf{let}\ w = {}^{\checkmark}\mathsf{case}\ x\ \mathsf{of} & (\textit{value-}\beta), (\textit{var-}\beta) \\
 & \qquad\qquad\qquad \mathsf{nil} \ \ \to y & \\
 & \qquad\qquad\qquad h : t \to h : (t +\!\!+ y) & \\
 & \quad \mathsf{in}\ w +\!\!+ z & \\
 & \mathrel{\underset{\sim}{\lhd\!\rhd}} {}^{\checkmark}\mathsf{case}\ x\ \mathsf{of} & \text{prop. 10.1} \\
 & \qquad \mathsf{nil} \ \ \to \mathsf{let}\ w = y\ \mathsf{in}\ w +\!\!+ z & \\
 & \qquad h : t \to \mathsf{let}\ w = h : (t +\!\!+ y)\ \mathsf{in}\ w +\!\!+ z & \\
 & \mathrel{\underset{\sim}{\lhd\!\rhd}} {}^{\checkmark}\mathsf{case}\ x\ \mathsf{of} & (\textit{var-}\beta), (\textit{gc}) \\
 & \qquad \mathsf{nil} \ \ \to {}^{\checkmark} w +\!\!+ z & \\
 & \qquad h : t \to \mathsf{let}\ w = h : (t +\!\!+ y)\ \mathsf{in}\ w +\!\!+ z & \\
 & \mathrel{\underset{\sim}{\lhd\!\rhd}} {}^{\checkmark}\mathsf{case}\ x\ \mathsf{of} & \text{prop. 10.1}, (\textit{let-let}), (\textit{let-flatten}) \\
 & \qquad \mathsf{nil} \ \ \to {}^{\checkmark} w +\!\!+ z & \\
 & \qquad h : t \to {}^{2\checkmark} h : (t +\!\!+ y) +\!\!+ z & \\
 & \equiv {}^{\checkmark}\mathsf{case}\ x\ \mathsf{of} & \text{(renaming)} \\
 & \qquad \mathsf{nil} \ \ \to {}^{\checkmark} w +\!\!+ z & \\
 & \qquad h : x \to {}^{2\checkmark} h : \underline{(x +\!\!+ y) +\!\!+ z} &
\end{array}
$$

Thus we have found a context $\mathbb{D}$ such that

$$(x +\!\!+ y) +\!\!+ z \mathrel{\underset{\sim}{\lhd\!\rhd}} {}^{\checkmark}\mathbb{D}[(x +\!\!+ y) +\!\!+ z]$$

It just remains to show that

$$x +\!\!+ (y \,\overline{+\!\!+}\, z) \mathrel{\underset{\sim}{\lhd\!\rhd}} {}^{\checkmark}\mathbb{D}[x \,\overline{+\!\!+}(y +\!\!+ z)]$$

$$
\begin{array}{lll}
(\overline{+\!\!+}), (+\!\!+) \vdash x \mathbin{\overline{+\!\!+}} (y +\!\!+ z) & & \\
\quad \equiv \mathsf{let}\ r = y +\!\!+ z\ \mathsf{in}\ x \mathbin{\overline{+\!\!+}} r & & \\
\quad \underset{\sim}{\lhd\!\rhd} \mathsf{let}\ r = y +\!\!+ z\ \mathsf{in}\ {}^{\checkmark}\mathsf{case}\ x\ \mathsf{of} & & (\textit{value-}\beta), (\textit{var-}\beta) \\
\qquad\qquad\qquad\qquad \mathsf{nil} \ \to r & & \\
\qquad\qquad\qquad\qquad h : t \to {}^{2\checkmark} h : (t \mathbin{\overline{+\!\!+}} r) & & \\
\quad \underset{\sim}{\lhd\!\rhd} \mathsf{case}\ x\ \mathsf{of} & & (\textit{case-}\mathbb{E}), (\textit{inline-}\mathbb{E}), (\textit{gc}) \\
\qquad \mathsf{nil} \ \to {}^{\checkmark} y +\!\!+ z & & \\
\qquad h : t \to {}^{2\checkmark}\mathsf{let}\ r = y +\!\!+ z\ \mathsf{in}\ h : (t \mathbin{\overline{+\!\!+}} r) & & \\
\quad \underset{\sim}{\lhd\!\rhd} {}^{\checkmark}\mathsf{case}\ x\ \mathsf{of} & & (\textit{let-flatten}) \\
\qquad \mathsf{nil} \ \to {}^{\checkmark} w +\!\!+ z & & \\
\qquad h : t \to {}^{2\checkmark}\mathsf{let}\ r = y +\!\!+ z & & \\
\qquad\qquad\qquad\quad s = t +\!\!+ r & & \\
\qquad\qquad\quad \mathsf{in}\ h : s & & \\
\quad \underset{\sim}{\lhd\!\rhd} {}^{\checkmark}\mathsf{case}\ x\ \mathsf{of} & & (\textit{let-let}) \\
\qquad \mathsf{nil} \ \to {}^{\checkmark} w +\!\!+ z & & \\
\qquad h : t \to {}^{2\checkmark}\mathsf{let}\ s = \mathsf{let}\ r = y +\!\!+ z\ \mathsf{in}\ t \mathbin{\overline{+\!\!+}} r & & \\
\qquad\qquad\quad \mathsf{in}\ h : s & & \\
\quad \equiv {}^{\checkmark}\mathsf{case}\ x\ \mathsf{of} & & (\text{renaming}) \\
\qquad \mathsf{nil} \ \to {}^{\checkmark} w +\!\!+ z & & \\
\qquad h : x \to {}^{2\checkmark} h : \underline{x \mathbin{\overline{+\!\!+}} (y +\!\!+ z)} & &
\end{array}
$$

Working note: *Should use the basic properties of append from the proposition*

□

Corollary 10.3.

$$
(x +\!\!+ y) +\!\!+ z \underset{\sim}{\rhd} x +\!\!+ (y +\!\!+ z)
$$
$$
x +\!\!+ (y +\!\!+ z) \underset{\approx}{\rhd} (x +\!\!+ y) +\!\!+ z
$$

This follows by the obvious improvement/weak improvement relation between $+\!\!+$ and $\overline{+\!\!+}$. and shows that the associativity property of append cannot, in itself, change the asymptotic time complexity of a program.

10.4.1 *The Transformation Laws*

Proposition 10.4. *The rewrite laws of the transformation are all improvements.*

Proof. Rules (i), (ii), (iv) and (vii) follow from proposition 10.1, and rule (iii) from corollary 10.3. For (vi) we have:

$$
\begin{array}{lll}
f, f^{+} \vdash (f\, z_1 \ldots z_n) +\!\!+ z & & \\
\quad \underset{\sim}{\lhd\!\rhd} {}^{\checkmark} M[\vec{z}/\vec{x}] +\!\!+ z & & (??), (??) \\
\quad \underset{\sim}{\lhd\!\rhd} f^{+}\, z_1 \ldots z_n\, z & & (??), (??)
\end{array}
$$

And lastly for (vi) we have:

$$\begin{array}{lll}
(\mathbin{+\!\!+}), f^{+} \vdash (f^{+}\, z_1 \ldots z_n\, z) \mathbin{+\!\!+} z' & & \\
\quad \mathrel{\underset{\sim}{\triangleleft\!\triangleright}} ({}^{\checkmark} M[{}^{z_1 \cdots z_n}/_{\vec{x}}] \mathbin{+\!\!+} z) \mathbin{+\!\!+} z & & (??), (??) \\
\quad \mathrel{\underset{\sim}{\triangleleft\!\triangleright}} {}^{\checkmark} M[{}^{z_1 \cdots z_n}/_{\vec{x}}] \mathbin{+\!\!+} (z \mathbin{+\!\!+} z') & & \text{cor. 10.3} \\
\quad \mathrel{\underset{\sim}{\triangleleft\!\triangleright}} f^{+}\, z_1 \ldots z_n\, (z \mathbin{+\!\!+} z') & & (??)(??), (??)
\end{array}$$

□

The main correctness argument. The improvement property of the individual steps is not the whole story, since the definition of f^{+} itself needs to be transformed.

PROPOSITION 10.5. *The transformation yields a program which is an improvement on the original.*

PROOF. Assume that the transformed (sub)program has the form

$$\begin{array}{l}
\mathsf{let}\ \mathbin{+\!\!+} = \ldots \\
\quad f = \lambda\, x_1 \ldots \lambda x_n . M \\
\quad f^{+} = \lambda\, x_1 \ldots \lambda x_n . \lambda z . M \mathbin{+\!\!+} z \\
\mathsf{in}\ N
\end{array}$$

(This is without loss of generality since by (*value-β*) we can float in the definition of append.) Now suppose that by applying the transformation rules we obtain:

$$\begin{array}{l}
\mathsf{let}\ \mathbin{+\!\!+} = \ldots \\
\quad f = \lambda\, x_1 \ldots \lambda x_n . M' \\
\quad f^{+} = \lambda\, x_1 \ldots \lambda x_n . \lambda z . M'' \\
\mathsf{in}\ N'
\end{array}$$

for some M', M'', and N'. Since the transformation rules are all improvements, we know that:

$$\begin{array}{lll}
\mathsf{let}\ \mathbin{+\!\!+} = \ldots & \mathrel{\underset{\sim}{\triangleright}} & \mathsf{let}\ \mathbin{+\!\!+} = \ldots \\
\quad f = \lambda\, x_1 \ldots \lambda x_n . M & & \quad f = \lambda\, x_1 \ldots \lambda x_n . M \\
\quad f^{+} = \lambda\, x_1 \ldots \lambda x_n . \lambda z . M \mathbin{+\!\!+} z & & \quad f^{+} = \lambda\, x_1 \ldots \lambda x_n . \lambda z . M \mathbin{+\!\!+} z \\
\mathsf{in}\ N & & \mathsf{in}\ N'.
\end{array}$$

Now we also know that

$$\begin{array}{lll}
\mathsf{let}\ \mathbin{+\!\!+} = \ldots & \mathrel{\underset{\sim}{\triangleright}} & \mathsf{let}\ \mathbin{+\!\!+} = \ldots \\
\quad f = \lambda\, x_1 \ldots \lambda x_n . M & & \quad f = \lambda\, x_1 \ldots \lambda x_n . M \\
\quad f^{+} = \lambda\, x_1 \ldots \lambda x_n . \lambda z . M \mathbin{+\!\!+} z & & \quad f^{+} = \lambda\, x_1 \ldots \lambda x_n . \lambda z . M \mathbin{+\!\!+} z \\
\mathsf{in}\ M & & \mathsf{in}\ M'
\end{array}$$

and that

$$\begin{array}{lll}
\mathsf{let}\ \mathbin{+\!\!+} = \ldots & \mathrel{\underset{\sim}{\triangleright}} & \mathsf{let}\ \mathbin{+\!\!+} = \ldots \\
\quad f = \lambda\, x_1 \ldots \lambda x_n . M & & \quad f = \lambda\, x_1 \ldots \lambda x_n . M \\
\quad f^{+} = \lambda\, x_1 \ldots \lambda x_n . \lambda z . M \mathbin{+\!\!+} z & & \quad f^{+} = \lambda\, x_1 \ldots \lambda x_n . \lambda z . M \mathbin{+\!\!+} z \\
\mathsf{in}\ M \mathbin{+\!\!+} z & & \mathsf{in}\ M''.
\end{array}$$

Thus by the Improvement Theorem we can conclude that

$$\begin{array}{ll}
\mathsf{let}\ {+\!\!+} = \ldots & \gtrsim\ \mathsf{let}\ {+\!\!+} = \ldots \\
\quad f = \lambda x_1 \ldots \lambda x_n.M & \quad f = \lambda x_1 \ldots \lambda x_n.M' \\
\quad f^+ = \lambda x_1 \ldots \lambda x_n.\lambda z.M \mathbin{+\!\!+} z & \quad f^+ = \lambda x_1 \ldots \lambda x_n.\lambda z.M'' \\
\mathsf{in}\ N' & \mathsf{in}\ N'
\end{array}$$

and by transitivity we are done. □

11. PROOFS OF MAIN THEOREMS

This section gives an outline of the technical development and proofs of the main results. Most proofs follow a direct style reasoning which is reminiscent of proofs about functional languages with effects by Mason and Talcott *et al.* [Mason and Talcott 1991; Agha et al. 1997; Talcott 1998]. In order to make this style of proof rigourous we generalise the abstract machine semantics so that it works on *configuration contexts* — configurations with holes. To ensure that transitions on configuration contexts are consistent with hole filling one must work with a more general representation of contexts. One such approach is described in [Talcott 1998]. We use an alternative approach to generalising contexts which is due to Pitts [Pitts 1994].

11.1 Substituting Contexts

Following Pitts [Pitts 1994], we use second-order syntax to represent (and generalise) the traditional definition of contexts given in section 5. We give a fuller description in [Sands 1998a]; other examples of their use are to be found in [Lassen 1998; Moran 1998]. The idea is that instead of holes $[\cdot]$ we use *second-order variables*, ranged over by ξ, applied to some vector of variables. The syntax of generalised contexts is:

$$\begin{array}{rcl}
\mathbb{C}, \mathbb{D} & ::= & \xi \cdot \vec{x} \\
& | & x \\
& | & \lambda x.\mathbb{C} \\
& | & \mathbb{C}\, x \\
& | & c\, \vec{x} \\
& | & \mathsf{let}\ \{\vec{x} = \vec{\mathbb{D}}\}\ \mathsf{in}\ \mathbb{C} \\
& | & \mathsf{case}\ \mathbb{C}\ \mathsf{of}\ \{c_i\, \vec{x}_i \rightarrow \mathbb{D}_i\}.
\end{array}$$

$\mathbb{V}$ and $\mathbb{W}$ will range over value contexts, $\mathbb{\Gamma}$ and $\mathbb{\Delta}$ over heap contexts, and $\mathbb{S}$ and $\mathbb{T}$ over stack contexts. Each "hole variable" ξ has a fixed arity, and ranges over meta-abstractions of the form $(\vec{x})M$ where the length of $\vec{x}$ is the arity of ξ. In the meta-abstraction $(\vec{x})M$, the variables $\vec{x}$ are bound in M. Hole-filling is now a general non-capturing substitution: $[{}^{(\vec{x})M}/_{\xi}]$. The effect of a substitution is as expected (remembering that the $\vec{x}$ are considered bound in $(\vec{x})M$). Coupled with the meta-abstraction is of course meta-application, written $\xi \cdot \vec{x}$. We restrict application of ξ to variables so that hole-filling cannot violate the restriction on syntax. In the definition of substitution we make the following identification:

$$(\vec{x})M \cdot \vec{y} \equiv M[{}^{\vec{y}}/_{\vec{x}}].$$

This definition of context generalises the usual definition since we can represent a traditional context $\mathbb{C}$ by $\mathbb{C}[\xi \cdot \vec{x}]$ where $\vec{x}$ is a vector of the capture-variables of $\mathbb{C}$; filling $\mathbb{C}$ with a term M is then represented by $(\mathbb{C}[\xi \cdot \vec{x}])[{}^{(\vec{x})M}/_{\xi}]$.

Example. The traditional context

$$\mathsf{let}\ x = [\cdot]\ \mathsf{in}\ \lambda y.[\cdot]$$

can be represented by

$$\mathsf{let}\ x = \xi \cdot (x, y)\ \mathsf{in}\ \lambda y.\xi \cdot (x, y).$$

Filling the hole with the term $x\,y$ is represented by:

$$\begin{aligned} &(\mathsf{let}\ x = \xi \cdot (x, y)\ \mathsf{in}\ \lambda y.\xi \cdot (x, y))[{}^{(x,y)\ x\,y}/_{\xi}] \\ &\equiv \mathsf{let}\ z = (x, y)\ x\,y \cdot (z, y)\ \mathsf{in}\ \lambda w.(x, y)\ x\,y \cdot (x, w) \\ &\equiv \mathsf{let}\ z = z\,y\ \mathsf{in}\ \lambda w.x\,w \end{aligned}$$

which is α-equivalent to what we would have obtained by the usual hole-filling with capture. Note that the generalised representation permits contexts to be identified up to α-conversion.

Henceforth we work only with generalised contexts. We will write $\mathbb{C}[(\vec{x})M]$ to mean $\mathbb{C}[{}^{(\vec{x})M}/_{\xi}]$ when $\mathbb{C}$ contains just a single hole variable ξ. We assume that the arities of hole variables are always respected.

We implicitly generalise our definitions of improvement to work with generalised contexts. This is not quite identical to the earlier definition since with generalised contexts, when placing a term in a hole we obtain a substitution instance of the term. This means in particular that improvement is now closed under substitution (variable-for-variable) by definition — a useful property. This difference is a relatively minor technicality which we will gloss over in this section.

11.2 Open Uniform Computation

The basis of our proofs will be to compute with configurations containing holes and free variables. Thanks to the capture-free representation of contexts, this means that normal reduction can be extended to contexts with ease. See [Sands 1998a] for a thorough treatment of generalised contexts and how they support generalisation of inductive definitions over terms.

Firstly, in order to fill the holes in a configuration we need to identify configurations up to renaming of the heap variables (recalling that update-markers on the stack are also binding occurrences of heap variables).

We tacitly extend the operational semantics to open configurations with holes. Note that holes can only occur in the stack within the branches of case alternatives. In what follows, θ will range over substitutions composed of variable for variable substitutions and substitutions of the form $[{}^{(\vec{x_i})M_i}/_{\xi_i}]$, and $\mathbb{\Sigma}$ range over configuration contexts.

We have the following key property.

Lemma 11.1. (Extension) *If* $\langle\, \mathbb{\Gamma},\ \mathbb{C},\ \mathbb{S} \,\rangle \to^k \langle\, \mathbb{\Delta},\ \mathbb{D},\ \mathbb{T} \,\rangle$ *then*

(i) for all $\mathbb{\Gamma}'$ *and* $\mathbb{S}'$ *such that* $\langle\, \mathbb{\Gamma}'\mathbb{\Gamma},\ \mathbb{C},\ \mathbb{S}\mathbb{S}' \,\rangle$ *is well-formed,* $\langle\, \mathbb{\Gamma}'\mathbb{\Gamma},\ \mathbb{C},\ \mathbb{S}\mathbb{S}' \,\rangle \to^k \langle\, \mathbb{\Gamma}'\mathbb{\Delta},\ \mathbb{D},\ \mathbb{T}\mathbb{S}' \,\rangle$.

(ii) for all θ, $\langle\,\mathbb{\Gamma},\ \mathbb{C},\ \mathbb{S}\,\rangle\theta \rightarrow^k \langle\,\mathbb{\Delta},\ \mathbb{D},\ \mathbb{T}\,\rangle\theta$.

Proof. (i) follows by inspection of possible open reductions over configuration contexts. (ii) amounts to the standard substitution lemma; see [Sands 1998a] for a general argument. □

The following *open uniform computation* property is central. It allows us to evaluate open configuration contexts until either the computation is finished, or we find ourselves in an "interesting" case.

Lemma 11.2. (Open Uniform Computation) *If well-formed and well-typed configuration context $\langle\,\mathbb{\Gamma},\ \mathbb{C},\ \mathbb{S}\,\rangle \rightarrow^k \mathbb{\Sigma} \not\rightarrow$, then $\mathbb{\Sigma}$ has one of the following forms:*

(i) $\langle\,\mathbb{\Delta},\ \mathbb{V},\ \epsilon\,\rangle$,

(ii) $\langle\,\mathbb{\Delta},\ \xi_i\cdot\vec{y},\ \mathbb{T}\,\rangle$, for some hole ξ_i, or

(iii) $\langle\,\mathbb{\Delta},\ x,\ \mathbb{T}\,\rangle$, $x \in \mathsf{FV}(\mathbb{\Gamma},\mathbb{C},\mathbb{S})$.

Proof. Assume $\langle\,\mathbb{\Gamma},\ \mathbb{C},\ \mathbb{S}\,\rangle \rightarrow^k \mathbb{\Sigma} \not\rightarrow$. We consider the reduction of $\langle\,\mathbb{\Gamma},\ \mathbb{C},\ \mathbb{S}\,\rangle$ and proceed by induction on k with cases on the structure of $\mathbb{C}$. We show three illustrative cases only. The others are similar.

$\mathbb{C} \equiv \xi_i\cdot\vec{y}$. This is a type (ii) context, so we are done.

$\mathbb{C} \equiv x$. Since we have termination, x must be bound in either $\mathbb{\Gamma}$ or is free in $\mathsf{FV}(\mathbb{\Gamma},\mathbb{C},\mathbb{S})$ (since if it was bound in $\mathbb{S}$, $\langle\,\mathbb{\Gamma},\ \mathbb{C},\ \mathbb{S}\,\rangle$ would diverge). In the former case, $\mathbb{\Gamma} \equiv \mathbb{\Delta}\{x = \mathbb{D}\}$. By (*Lookup*), $\langle\,\mathbb{\Delta}\{x = \mathbb{D}\},\ x,\ \mathbb{S}\,\rangle$ reduces to $\langle\,\mathbb{\Delta},\ \mathbb{D},\ \#x : \mathbb{S}\,\rangle$. By the inductive hypothesis, we know that $\langle\,\mathbb{\Delta},\ \mathbb{D},\ \#x : \mathbb{S}\,\rangle$ reduces to a configuration context of type (i), (ii), or (iii), and therefore $\langle\,\mathbb{\Delta}\{x = \mathbb{D}\},\ x,\ \mathbb{S}\,\rangle$ does also, as required. In the latter case, $\langle\,\mathbb{\Gamma},\ x,\ \mathbb{S}\,\rangle$ is a type (iii) context, and we are done.

$\mathbb{C} \equiv \mathbb{V}$. There are four sub-cases, depending upon the structure of $\mathbb{S}$; we consider only the case when $\mathbb{S} \equiv x : \mathbb{T}$. Since $\langle\,\mathbb{\Gamma},\ \mathbb{C},\ \mathbb{S}\,\rangle$ is well-typed, $\mathbb{V} \equiv \lambda y.\mathbb{D}$, and by (*Subst*), $\langle\,\mathbb{\Gamma},\ \lambda y.\mathbb{D},\ x : \mathbb{T}\,\rangle$ reduces to $\langle\,\mathbb{\Gamma},\ \mathbb{D}[{}^{x}\!/\!_{y}],\ \mathbb{T}\,\rangle$. The inductive hypothesis applies, and the result follows as above. □

Uniform reductions are clearly also uniform in cost: if $\mathbb{\Sigma} \rightarrow^k_n \mathbb{\Sigma}'$, then for all M, $\mathbb{\Sigma}[(\vec{x})M] \rightarrow^k_n \mathbb{\Sigma}'[(\vec{x})M]$.

11.3 Translation

We can extend the definition of trans to cover open configurations and configuration contexts, and can therefore extend translation thus:

Lemma 11.3. (Translation) *For all $\mathbb{D},\mathbb{\Gamma},\mathbb{C},\mathbb{S}$ such that $\mathbb{D} \equiv \mathsf{trans}\langle\,\mathbb{\Gamma},\ \mathbb{C},\ \mathbb{S}\,\rangle$, there exists $n \geqslant 0$ such that $\langle\,\emptyset,\ \mathbb{D},\ \epsilon\,\rangle \rightarrow^n \langle\,\mathbb{\Gamma},\ \mathbb{C},\ \mathbb{S}\,\rangle$.*

Proof. Simple induction on $\mathbb{S}$. □

11.4 Proof: the Context Lemma

The proof of the context lemma relies upon two lemmas, the latter of which is the most complex.

Lemma 11.4. *$M \gtrsim N$ if and only if for all $\mathbb{\Sigma}$ and n, $\mathbb{\Sigma}[(\vec{x})M]{\downarrow}^n \implies \mathbb{\Sigma}[(\vec{x})N]{\downarrow}^{\leqslant n}$.*

Proof. (Sketch) ($\Leftarrow$). Trivial; let $\mathbb{\Sigma} = \langle\, \emptyset,\ \mathbb{C},\ \epsilon\,\rangle$.

($\Rightarrow$). By a simple lexicographic induction on n and the length of transition sequences, using translation. $\square$

Lemma 11.5. *If for all Γ, S, and n*

$$\langle\, \Gamma,\ (\vec{x})M \cdot \vec{y},\ S\,\rangle{\downarrow}^{n} \implies \langle\, \Gamma,\ (\vec{x})N \cdot \vec{y},\ S\,\rangle{\downarrow}^{\leqslant n}$$

then for all $\mathbb{\Sigma}$ and n, $\mathbb{\Sigma}[(\vec{x})M]{\downarrow}^{n} \implies \mathbb{\Sigma}[(\vec{x})N]{\downarrow}^{\leqslant n}$, where $\vec{x} \supseteq \mathsf{FV}\,(M, N)$.

Proof. Assume the premise and suppose $\mathbb{\Sigma}[(\vec{x})M]{\downarrow}^{n}$ in k computation steps. We proceed via lexicographic induction on (n, k). By open uniform computation, $\mathbb{\Sigma}$ reduces in $k_0 \geqslant 0$ steps with cost n_0 to one of:

$$\textbf{(1)}\ \langle\, \mathbb{\Gamma},\ \mathbb{V},\ \epsilon\,\rangle, \qquad \textbf{(2)}\ \langle\, \mathbb{\Gamma},\ \xi \cdot \vec{y},\ \mathbb{S}\,\rangle.$$

(There are only two possibilities since $\mathbb{\Sigma}$ is closed.) In case (1), we are done. In case (2), we have

$$\mathbb{\Sigma}[(\vec{x})N] \to^{k_0}_{n_0} \langle\, \mathbb{\Gamma}[(\vec{x})N],\ N[\vec{y}/\vec{x}],\ \mathbb{S}[(\vec{x})N]\,\rangle. \tag{11.1}$$

By open uniform computation, $\langle\, \mathbb{\Gamma},\ M[\vec{y}/\vec{x}],\ \mathbb{S}\,\rangle$ reduces in $k_1 \geqslant 0$ steps with cost n_1 to one of:

$$\textbf{(2.1)}\ \langle\, \mathbb{\Delta},\ \mathbb{W},\ \epsilon\,\rangle, \qquad \textbf{(2.2)}\ \langle\, \mathbb{\Delta},\ \xi \cdot \vec{z},\ \mathbb{T}\,\rangle.$$

(Again, there are only two possibilities since $\langle\, \mathbb{\Gamma},\ M[\vec{y}/\vec{x}],\ \mathbb{S}\,\rangle$ is closed.) In case (2.1), we have that $\langle\, \mathbb{\Gamma}[(\vec{x})N],\ (\vec{x})M \cdot \vec{y},\ \mathbb{S}[(\vec{x})N]\,\rangle$ reduces in k_1 steps to $\langle\, \mathbb{\Delta}[(\vec{x})N],\ \mathbb{W}[(\vec{x})N],\ \epsilon\,\rangle$ with cost $n_1 = n - n_0$, so

$$\begin{array}{ll}
\langle\, \mathbb{\Gamma}[(\vec{x})N],\ M[\vec{y}/\vec{x}],\ \mathbb{S}[(\vec{x})N]\,\rangle{\downarrow}^{\leqslant n-n_0} & \\
\implies \langle\, \mathbb{\Gamma}[(\vec{x})N],\ N[\vec{y}/\vec{x}],\ \mathbb{S}[(\vec{x})N]\,\rangle{\downarrow}^{\leqslant n-n_0} & \text{(ass.)} \\
\implies \mathbb{\Sigma}[(\vec{x})N]{\downarrow}^{\leqslant n} & \text{(11.1)}
\end{array}$$

as required. In case (2.2), we know that $k_1 > 0$, since $M[\vec{y}/\vec{x}] \not\equiv \xi \cdot \vec{z}$. We have

$$\langle\, \mathbb{\Gamma}[(\vec{x})M],\ (\vec{x})M \cdot \vec{y},\ \mathbb{S}[(\vec{x})M]\,\rangle \to^{k_1}_{n_1} \langle\, \mathbb{\Delta}[(\vec{x})M],\ (\vec{x})M \cdot \vec{z},\ \mathbb{T}[(\vec{x})M]\,\rangle$$

and

$$\langle\, \mathbb{\Delta}[(\vec{x})M],\ (\vec{x})M \cdot \vec{z},\ \mathbb{T}[(\vec{x})M]\,\rangle{\downarrow}^{n-n_0-n_1}. \tag{11.2}$$

Furthermore,

$$\langle\, \mathbb{\Gamma}[(\vec{x})N],\ (\vec{x})M \cdot \vec{y},\ \mathbb{S}[(\vec{x})N]\,\rangle \to^{k_1}_{n_1} \langle\, \mathbb{\Delta}[(\vec{x})N],\ (\vec{x})N \cdot \vec{z},\ \mathbb{T}[(\vec{x})N]\,\rangle \tag{11.3}$$

Therefore

$$\begin{array}{ll}
\langle\, \mathbb{\Delta}[(\vec{x})M],\ M[\vec{z}/\vec{x}],\ \mathbb{T}[(\vec{x})M]\,\rangle{\downarrow}^{n-n_0-n_1} & \text{(11.2)} \\
\implies \langle\, \mathbb{\Delta}[(\vec{x})N],\ M[\vec{z}/\vec{x}],\ \mathbb{T}[(\vec{x})N]\,\rangle{\downarrow}^{\leqslant n-n_0-n_1} & \text{(I.H.)} \\
\implies \langle\, \mathbb{\Delta}[(\vec{x})N],\ N[\vec{z}/\vec{x}],\ \mathbb{T}[(\vec{x})N]\,\rangle{\downarrow}^{\leqslant n-n_0-n_1} & \text{(ass.)} \\
\implies \langle\, \mathbb{\Gamma}[(\vec{x})N],\ M[\vec{y}/\vec{x}],\ \mathbb{S}[(\vec{x})N]\,\rangle{\downarrow}^{\leqslant n-n_0} & \text{(11.3)} \\
\implies \langle\, \mathbb{\Gamma}[(\vec{x})N],\ N[\vec{y}/\vec{x}],\ \mathbb{S}[(\vec{x})N]\,\rangle{\downarrow}^{\leqslant n-n_0} & \text{(ass.)} \\
\implies \mathbb{\Sigma}[(\vec{x})N]{\downarrow}^{\leqslant n} & \text{(11.1)}
\end{array}$$

as required. □

The generalised statement of the context lemma is:

For all terms M and N, if

$$\forall \Gamma, S, \sigma, n. \langle\, \Gamma,\ M\sigma,\ S\,\rangle{\downarrow}^{n} \implies \langle\, \Gamma,\ N\sigma,\ S\,\rangle{\downarrow}^{\leqslant n}$$

then $M \underset{\sim}{\trianglerighteq} N$.

This follows from lemmas 11.4 and 11.5, and the fact that $M\sigma \equiv (\vec{x})M \cdot \vec{y}$ for $\sigma = [\vec{y}/\vec{x}]$.

11.5 Validating the Tick Algebra

We present proofs of the validity of (*value-β*) and (*value-copy*), and sketch a proof of the correspondence between evaluation contexts and configuration contexts of the form $\langle\, \Gamma,\ [\cdot],\ S\,\rangle$. The proofs of the more complex laws (*e.g.* (*var-β*), (*var-abs*), (*var-subst*), and (*✓-float*)) have a similar structure to that for (*value-β*), except they require more use of open uniform computation.

11.5.1 *Proof:* (*value-β*). Recall (*value-β*):

$$\mathsf{let}\ \{x = V, \vec{y} = \vec{\mathbb{D}}[x]\}\ \mathsf{in}\ \mathbb{C}[x] \underset{\sim}{\trianglelefteq\!\trianglerighteq} \mathsf{let}\ \{x = V, \vec{y} = \vec{\mathbb{D}}[{}^{\checkmark}V]\}\ \mathsf{in}\ \mathbb{C}[{}^{\checkmark}V].$$

Let $W \equiv {}^{\checkmark}V$ throughout. It suffices to show

$$\forall \mathbb{\Gamma}, \mathbb{S}. \langle\, \mathbb{\Gamma}[x]\{x = V\},\ \mathbb{C}[x],\ \mathbb{S}[x]\,\rangle{\downarrow}^{n} \iff \langle\, \mathbb{\Gamma}[W]\{x = V\},\ \mathbb{C}[W],\ \mathbb{S}[W]\,\rangle{\downarrow}^{n}$$

where $x \notin \mathrm{dom}(\mathbb{\Gamma}, \mathbb{S})$, and the only hole is $[\cdot]$, a non-capturing hole. We prove the forward direction only; the reverse direction is similar.

Suppose $\langle\, \mathbb{\Gamma}[x]\{x = V\},\ \mathbb{C}[x],\ \mathbb{S}[x]\,\rangle{\downarrow}^{n}$ in k computation steps. We proceed by lexicographic induction on (n, k). By open uniform computation, $\langle\, \mathbb{\Gamma},\ \mathbb{C},\ \mathbb{S}\,\rangle$ reduces in $k_0 \geqslant 0$ steps with cost n_0 to one of

$$\textbf{(1)}\ \langle\, \mathbb{\Delta},\ \mathbb{V},\ \epsilon\,\rangle,\ \textbf{(2)}\ \langle\, \mathbb{\Delta},\ [\cdot],\ \mathbb{T}\,\rangle,\ \textbf{(3)}\ \langle\, \mathbb{\Delta},\ x,\ \mathbb{T}\,\rangle.$$

In case (1), we are done. In case (2), by extension, (***Lookup***) and (***Update***), we have

$$\begin{aligned} \langle\, \mathbb{\Gamma}[x]\{x = V\},\ \mathbb{C}[x],\ \mathbb{S}[x]\,\rangle &\rightarrow^{k_0}_{n_0} \langle\, \mathbb{\Delta}[x]\{x = V\},\ x,\ \mathbb{T}[x]\,\rangle \\ &\rightarrow^{2}_{1} \langle\, \mathbb{\Delta}[x]\{x = V\},\ V,\ \mathbb{T}[x]\,\rangle, \end{aligned}$$

and by extension and the definition of W,

$$\begin{aligned} \langle\, \mathbb{\Gamma}[W]\{x = V\},\ \mathbb{C}[W],\ \mathbb{S}[W]\,\rangle &\rightarrow^{k_0}_{n_0} \langle\, \mathbb{\Delta}[W]\{x = V\},\ W,\ \mathbb{T}[W]\,\rangle \\ &\rightarrow^{2}_{1} \langle\, \mathbb{\Delta}[W]\{x = V\},\ V,\ \mathbb{T}[W]\,\rangle. \end{aligned}$$

Since $\langle\, \mathbb{\Delta}[x]\{x = V\},\ V,\ \mathbb{T}[x]\,\rangle{\downarrow}^{n-(n_0+1)}$, by the inductive hypothesis we have $\langle\, \mathbb{\Delta}[W]\{x = V\},\ V,\ \mathbb{T}[W]\,\rangle{\downarrow}^{n-(n_0+1)}$, and the result follows.

In case (3), we have $\langle\, \mathbb{\Delta}[x]\{x = V\},\ V,\ \mathbb{T}[x]\,\rangle{\downarrow}^{n-(n_0+1)}$, as above. Furthermore, by extension, (***Lookup***) and (***Update***), we have

$$\begin{aligned} \langle\, \mathbb{\Gamma}[W]\{x = V\},\ \mathbb{C}[W],\ \mathbb{S}[W]\,\rangle &\rightarrow^{k_0}_{n_0} \langle\, \mathbb{\Delta}[W]\{x = V\},\ x,\ \mathbb{T}[W]\,\rangle \\ &\rightarrow^{2}_{1} \langle\, \mathbb{\Delta}[W]\{x = V\},\ V,\ \mathbb{T}[W]\,\rangle. \end{aligned}$$

From the inductive hypothesis, we have $\langle\, \mathbb{\Delta}[W]\{x = V\},\ V,\ \mathbb{T}[W]\,\rangle{\downarrow}^{n-(n_0+1)}$, and the result follows.

11.5.2 *Proof:* (*value-copy*). Recall (*value-copy*):

$$\mathsf{let}\ \{\vec{x} = \vec{V}\sigma_1, \vec{y} = \vec{V}\sigma_2, \vec{z} = \vec{M}\}\ \mathsf{in}\ N \mathrel{\underset{\sim}{\lhd\!\rhd}} \mathsf{let}\ \{\vec{x} = \vec{V}\sigma_2\sigma_3, \vec{z} = \vec{M}\sigma_3\}\ \mathsf{in}\ N\sigma_3,$$

where $\sigma_1 = [\vec{y}/\vec{w}]$, $\sigma_2 = [\vec{x}/\vec{w}]$, and $\sigma_3 = [\vec{x}/\vec{y}]$.

It suffices to show that for all Γ, S, and n,

$$\langle\, \Gamma\{\vec{x} = \vec{V}\sigma_1, \vec{y} = \vec{V}\sigma_2\},\ N,\ S\,\rangle{\downarrow}^n \iff \langle\, \Gamma\{\vec{x} = \vec{V}\sigma_2\sigma_3\},\ N\sigma_3,\ S\,\rangle{\downarrow}^n.$$

We show only the forward direction. To show the reverse, we need only establish termination, which follows by the fact that call-by-name and call-by-need agree on termination.

Suppose $\langle\, \Gamma\{\vec{x} = \vec{V}\sigma_1, \vec{y} = \vec{V}\sigma_2\},\ N,\ S\,\rangle{\downarrow}^n$ in k computation steps. We proceed via lexicographic induction on (n, k). Consider the (hole-less) open configuration context $\langle\, \Gamma,\ N,\ S\,\rangle$, in which the $\vec{x}$ and $\vec{y}$ may appear free. By open uniform computation, this reduces in $k_0 \geqslant 0$ steps with cost n_0 to one of:

$$\textbf{(1)}\ \langle\, \Delta,\ W,\ \epsilon\,\rangle,\ \textbf{(2a)}\ \langle\, \Delta,\ x_i,\ T\,\rangle,\ \textbf{(2b)}\ \langle\, \Delta,\ y_i,\ T\,\rangle.$$

In case (1), we are done. In case (2a), by extension, (*Lookup*) and (*Update*), we have

$$\begin{aligned}\langle\, \Gamma\{\vec{x} = \vec{V}\sigma_1, \vec{y} = \vec{V}\sigma_2\},\ N,\ S\,\rangle &\rightarrow^{k_0}_{n_0} \langle\, \Delta\{\vec{x} = \vec{V}\sigma_1, \vec{y} = \vec{V}\sigma_2\},\ x_i,\ T\,\rangle \\ &\rightarrow^{2}_{1} \langle\, \Delta\{\vec{x} = \vec{V}\sigma_1, \vec{y} = \vec{V}\sigma_2\},\ V_i\sigma_1,\ T\,\rangle,\end{aligned}$$

and furthermore,

$$\langle\, \Delta\{\vec{x} = \vec{V}\sigma_1, \vec{y} = \vec{V}\sigma_2\},\ V_i\sigma_1,\ T\,\rangle{\downarrow}^{n-(n_0+1)}. \tag{11.4}$$

Similarly, by extension, (*Lookup*) and (*Update*), we have also that

$$\begin{aligned}\langle\, \Gamma\{\vec{x} = \vec{V}\sigma_2\sigma_3\},\ N\sigma_3,\ S\,\rangle &\rightarrow^{k_0}_{n_0} \langle\, \Delta\{\vec{x} = \vec{V}\sigma_2\sigma_3\},\ x_i\sigma_3,\ T\,\rangle \\ &\rightarrow^{2}_{1} \langle\, \Delta\{\vec{x} = \vec{V}\sigma_2\sigma_3\},\ V_i\sigma_2\sigma_3,\ T\,\rangle.\end{aligned}$$

By elementary properties of substitution,

$$V_i\sigma_1[\vec{x}/\vec{y}] \equiv V_i[\vec{x}/\vec{w}][\vec{x}/\vec{y}],$$

so the inductive hypothesis applies (with $N \equiv V_i\sigma_1$), yielding the desired result.

In case (2b), by extension, (*Lookup*) and (*Update*), we have

$$\begin{aligned}\langle\, \Gamma\{\vec{x} = \vec{V}\sigma_1, \vec{y} = \vec{V}\sigma_2\},\ N,\ S\,\rangle &\rightarrow^{k_0}_{n_0} \langle\, \Delta\{\vec{x} = \vec{V}\sigma_1, \vec{y} = \vec{V}\sigma_2\},\ y_i,\ T\,\rangle \\ &\rightarrow^{2}_{1} \langle\, \Delta\{\vec{x} = \vec{V}\sigma_1, \vec{y} = \vec{V}\sigma_2\},\ V_i\sigma_2,\ T\,\rangle,\end{aligned}$$

and furthermore,

$$\langle\, \Delta\{\vec{x} = \vec{V}\sigma_1, \vec{y} = \vec{V}\sigma_2\},\ V_i\sigma_2,\ T\,\rangle{\downarrow}^{n-(n_0+1)}. \tag{11.5}$$

Similarly, by extension, (*Lookup*) and (*Update*), we have also that

$$\begin{aligned}\langle\, \Gamma\{\vec{x} = \vec{V}\sigma_2\sigma_3\},\ N\sigma_3,\ S\,\rangle &\rightarrow^{k_0}_{n_0} \langle\, \Delta\{\vec{x} = \vec{V}\sigma_2\sigma_3\},\ y_i\sigma_3,\ T\,\rangle \\ &\equiv \langle\, \Delta\{\vec{x} = \vec{V}\sigma_2\sigma_3\},\ x_i,\ T\,\rangle \\ &\rightarrow^{2}_{1} \langle\, \Delta\{\vec{x} = \vec{V}\sigma_2\sigma_3\},\ V_i\sigma_2\sigma_3,\ T\,\rangle.\end{aligned}$$

The inductive hypothesis applies (with $N \equiv V_i\sigma_2$), yielding the desired result.

11.5.3 *Proof: Lemma 6.1.* Recall the statement of lemma 6.1:

$$\Lambda_{\mathbb{E}} = \{\mathsf{trans}\langle\,\Gamma,\ [\cdot],\ S\,\rangle \mid \text{all } \Gamma, S\}.$$

So we need to show that:

(i) $\forall \Gamma, S.\ \exists \mathbb{E}.\ \mathsf{trans}\langle\,\Gamma,\ [\cdot],\ S\,\rangle \equiv \mathbb{E}$, and
(ii) $\forall \mathbb{E}.\ \exists \Gamma, S.\ \mathsf{trans}\langle\,\Gamma,\ [\cdot],\ S\,\rangle \equiv \mathbb{E}$.

First note that $\Lambda_{\mathbb{A}}$ (the set of all applicative contexts) is in 1-1 correspondence to update-marker free stacks, realised by the following isomorphism (writing $[x]$ for the singleton stack):

$$\begin{aligned} [\cdot]^{\circ} &= \epsilon \\ (\mathbb{A}\,x)^{\circ} &= \mathbb{A}^{\circ}[x] \\ (\mathsf{case}\ \mathbb{A}\ \mathsf{of}\ \mathit{alts})^{\circ} &= \mathbb{A}^{\circ}\mathit{alts} \end{aligned}$$

$(\cdot)^{\circ}$ takes $\Lambda_{\mathbb{A}}$ into the set of update-marker free stacks. Its inverse is denoted by $(\cdot)^{\bullet}$ and satisfies the following equations:

$$\begin{aligned} \epsilon^{\bullet} &= [\cdot] \\ (x : S)^{\bullet} &= S^{\bullet}[[\cdot]\,x] \\ (\mathit{alts} : S)^{\bullet} &= S^{\bullet}[\mathsf{case}\ [\cdot]\ \mathsf{of}\ \mathit{alts}] \end{aligned}$$

It can easily be shown that $\langle\,\Gamma,\ \mathbb{A}[\mathbb{C}],\ \mathbb{S}\,\rangle \to^{*} \langle\,\Gamma,\ \mathbb{C},\ \mathbb{A}^{\circ}\mathbb{S}\,\rangle$ and that $\mathsf{trans}\langle\,\Gamma,\ \mathbb{A}[\mathbb{C}],\ \mathbb{S}\,\rangle$ is identical to $\mathsf{trans}\langle\,\Gamma,\ \mathbb{C},\ \mathbb{A}^{\circ}\mathbb{S}\,\rangle$ by induction on the structure of $\mathbb{A}$.

To show (i), we generalise the statement to show that for all Γ and S both $\mathsf{trans}\langle\,\Gamma,\ \mathbb{A},\ S\,\rangle$ and $\mathsf{trans}\langle\,\Gamma\{x_0 = \mathbb{A}_0[x_1], \ldots, x_n = \mathbb{A}_n\},\ \mathbb{A}[x_0],\ S\,\rangle$ are evaluation contexts. This proceeds by an easy induction on the number of update markers in S.

To show (ii), we proceed by case analysis on $\mathbb{E}$, and produce a Γ and S in each case. The difficult case is when

$$\begin{aligned} \mathbb{E} \equiv \mathsf{let}\ \{&\vec{y} = \vec{M}, \\ &x_0 = \mathbb{A}_0[x_1], \\ &x_1 = \mathbb{A}_1[x_2], \\ &\ldots \\ &x_n = \mathbb{A}_n\} \\ \mathsf{in}\ &\mathbb{A}[x_0] \end{aligned}$$

Here, we let Γ be $\{\vec{y} = \vec{M}\}$ and let S be

$$\mathbb{A}_n^{\circ}\#x_n \cdots \mathbb{A}_1^{\circ}\#x_1\mathbb{A}_0^{\circ}\#x_0\mathbb{A}^{\circ}.$$

The other cases are similar.

11.5.4 *Proof:* (*case*-$\mathbb{E}$). The following lemma will be used to validate (*case*-$\mathbb{E}$), (*let*-$\mathbb{E}$) follows by similar reasoning. $\mathsf{CV}\,(\mathbb{E})$ denotes the *capture variables* of $\mathbb{E}$.

LEMMA 11.6. *For all $\mathbb{E}$, there exist Δ, T, such that* $\mathrm{dom}(\Delta, T) \subseteq \mathsf{CV}\,(\mathbb{E})$ *and* $\forall \Gamma, S. \langle\,\Gamma,\ \mathbb{E},\ S\,\rangle \to_n^k \langle\,\Gamma\Delta,\ [\cdot],\ TS\,\rangle$, *for some k and n.*

PROOF. By lemma 6.1, there exist Δ and T such that $\mathsf{trans}\langle\, \Delta,\ [\cdot],\ T\,\rangle \equiv \mathbb{E}$, so by translation $\langle\, \emptyset,\ \mathbb{E},\ \epsilon\,\rangle \to^{*} \langle\, \Delta,\ [\cdot],\ T\,\rangle$, and thus by extension, provided $\text{dom}(\Delta,T) \subseteq \mathsf{CV}(\mathbb{E})$, $\langle\, \Gamma,\ \mathbb{E},\ S\,\rangle \to^{*} \langle\, \Gamma\Delta,\ [\cdot],\ TS\,\rangle$. □

Recall the statement of (*case*-$\mathbb{E}$):

$$\mathbb{E}[\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to N_i\}] \mathrel{\underset{\sim}{\lhd\!\rhd}} \mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to \mathbb{E}[N_i]\}.$$

By the standard bound variable convention, we know that $\mathsf{CV}(\mathbb{E}) \not\between \mathsf{CV}(pat_i)$ for all i, and that $\mathsf{FV}(M) \not\between \mathsf{BV}(\mathbb{E})$, where $\mathsf{BV}(\mathbb{E})$ denotes the let-bound variables of $\mathbb{E}$.

Assume wlog that for any Γ, we have that

$$\langle\, \Gamma,\ M,\ \epsilon\,\rangle \to^{k'}_{n'} \langle\, \Gamma'\Theta,\ c_j\,\vec{x}_j,\ \epsilon\,\rangle \tag{11.6}$$

where Γ' is the same as Γ with some possible updates, and Θ contains any bindings introduced during the evaluation. (This is valid since otherwise (*case*-$\mathbb{E}$) holds vacuously as both sides would diverge; an empty stack is sufficient by extension.)

For any Γ and S, we have that

$$\begin{array}{lr}
\langle\, \Gamma,\ \mathbb{E}[\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to N_i\}],\ S\,\rangle & \\
\to^{k}_{n} \langle\, \Gamma\Delta,\ \mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to N_i\},\ TS\,\rangle & \text{lem. 11.6} \\
\to \langle\, \Gamma\Delta,\ M,\ \{pat_i \to N_i\} : TS\,\rangle & (\mathit{Case}) \\
\to^{k'}_{n'} \langle\, \Gamma'\Delta'\Theta,\ c_j\,\vec{x}_j,\ \{pat_i \to N_i\} : TS\,\rangle & (11.6), (\text{ext.}) \\
\to \langle\, \Gamma'\Delta'\Theta,\ N_j[\vec{x}_j/\vec{y}_j],\ TS\,\rangle & (\mathit{Branch})
\end{array}$$

and

$$\begin{array}{lr}
\langle\, \Gamma,\ \mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to \mathbb{E}[N_i]\},\ S\,\rangle & \\
\to \langle\, \Gamma,\ M,\ \{pat_i \to N_i\} : S\,\rangle & (\mathit{Case}) \\
\to^{k'}_{n'} \langle\, \Gamma'\Theta,\ c_j\,\vec{x}_j,\ \{pat_i \to \mathbb{E}[N_i]\} : S\,\rangle & (11.6), (\text{ext.}) \\
\to \langle\, \Gamma'\Theta,\ \mathbb{E}[N_j][\vec{x}_j/\vec{y}_j],\ S\,\rangle & (\mathit{Branch}) \\
\equiv \langle\, \Gamma'\Theta,\ \mathbb{E}[N_j[\vec{x}_j/\vec{y}_j]],\ S\,\rangle & \mathsf{CV}(\mathbb{E}) \not\between \vec{y}_j \\
\to^{k}_{n} \langle\, \Gamma'\Delta\Theta,\ N_j[\vec{x}_j/\vec{y}_j],\ TS\,\rangle & \text{lem. 11.6}
\end{array}$$

Since $\mathsf{FV}(M) \not\between \mathsf{BV}(\mathbb{E})$, the evaluation of M cannot affect any of the bindings introduced by the evaluation $\mathbb{E}$; in other words, $\Delta' \equiv \Delta$. Therefore, the result follows by the context lemma.

11.6 Congruence of Entailment

In examples, we often want to perform calculation in the context of recursive declarations. A notation for this was introduced in section 9; a derivation of the form

$$\begin{aligned}
\mathsf{let}\ \{\vec{g} = \vec{V}\}\ \mathsf{in}\ M_1 &\mathrel{\underset{\sim}{\rhd}} \mathsf{let}\ \{\vec{g} = \vec{V}\}\ \mathsf{in}\ M_2 \\
&\mathrel{\underset{\sim}{\rhd}} \mathsf{let}\ \{\vec{g} = \vec{V}\}\ \mathsf{in}\ M_3 \ldots
\end{aligned}$$

was written:

$$\begin{aligned}\vec{g} \vdash M_1 &\gtrsim M_2\\ &\gtrsim M_3 \ldots\end{aligned}$$

when the declarations $\vec{g}$ are clear from the context. We prove the following extension of ($\vdash$-*cong*) to general contexts valid:

$$\frac{\vec{g} \vdash M \gtrsim N}{\vec{g} \vdash \mathbb{C}[(\vec{x})M] \gtrsim \mathbb{C}[(\vec{x})N]} \qquad (\vdash\text{-}cong)$$

for all contexts $\mathbb{C}$ such that $\vec{x} \not\between \mathsf{CV}(\mathbb{C})$. As usual, $\vec{x} \supseteq \mathsf{FV}(M, N)$.

To prove the validity of this rule, we require some lemmata. This next lemma is used to prove lemma 11.8.

Lemma 11.7. *For all* Γ, S, *and* n

$$\langle \Gamma\{\vec{x} = \vec{V}, \vec{y} = \vec{V}\sigma\},\ M,\ S\rangle{\downarrow}^n \iff \langle \Gamma\sigma\{\vec{y} = \vec{V}\sigma\},\ M\sigma,\ S\sigma\rangle{\downarrow}^n.$$

where $\sigma = [{}^{y}\!/_{x}]$.

Proof. (Sketch) ($\Rightarrow$) Simple induction on n, with cases of the structure of M.

($\Leftarrow$) It is sufficient to show that termination is implied. This is true for the call-by-name theory, and therefore here also. □

To prove ($\vdash$-*cong*) and improvement theorem, we will need the following lemma.

Lemma 11.8. *If* $\mathsf{let}\ \{\vec{x} = \vec{V}\}\ \mathsf{in}\ M \gtrsim \mathsf{let}\ \{\vec{x} = \vec{V}\}\ \mathsf{in}\ N$ *then for all* Γ *and* S,

$$\langle \Gamma\{\vec{x} = \vec{V}\},\ M,\ S\rangle{\downarrow}^n \implies \langle \Gamma\{\vec{x} = \vec{V}\},\ N,\ S\rangle{\downarrow}^{\leqslant n}.$$

where $\vec{x} \supseteq \mathsf{FV}(\Gamma, S)$.

Proof. By the context lemma and (*Letrec*), we have:

$\mathsf{let}\ \{\vec{x} = \vec{V}\}\ \mathsf{in}\ M \gtrsim \mathsf{let}\ \{\vec{x} = \vec{V}\}\ \mathsf{in}\ N$

$\iff \forall \Gamma, S. \langle \Gamma,\ \mathsf{let}\ \{\vec{x} = \vec{V}\}\ \mathsf{in}\ M,\ S\rangle{\downarrow}^n \implies \langle \Gamma,\ \mathsf{let}\ \{\vec{x} = \vec{V}\}\ \mathsf{in}\ N,\ S\rangle{\downarrow}^{\leqslant n}$

$\iff \forall \Gamma, S, \vec{x} \not\between \mathsf{FV}(\Gamma, S), \not\between \mathrm{dom}(\Gamma, S) \langle \Gamma\{\vec{x} = \vec{V}\},\ M,\ S\rangle{\downarrow}^n \implies \langle \Gamma\{\vec{x} = \vec{V}\},\ N,\ S\rangle{\downarrow}^{\leqslant n}.$

Letting $\sigma = [{}^{\vec{y}}\!/_{\vec{x}}]$, this implies that

$$\begin{aligned}&\forall \Gamma, S, \vec{x} \not\between \mathsf{FV}(\Gamma, S), \vec{x} \not\between \mathrm{dom}(\Gamma, S), \vec{y} \supseteq \mathsf{FV}(\Gamma, S).\\ &\qquad \langle \Gamma\{\vec{y} = \vec{V}\sigma, \vec{x} = \vec{V}\},\ M,\ S\rangle{\downarrow}^n \implies \langle \Gamma\{\vec{y} = \vec{V}\sigma, \vec{x} = \vec{V}\},\ N,\ S\rangle{\downarrow}^{\leqslant n}.\end{aligned}$$

By lemma 11.7, this is equivalent to

$\forall \Gamma, S, \vec{x} \not\between \mathsf{FV}(\Gamma, S), \vec{x} \not\between \mathrm{dom}(\Gamma, S), \vec{y} \supseteq \mathsf{FV}(\Gamma, S).$

$\langle \Gamma\sigma\{\vec{y} = \vec{V}\sigma\},\ M\sigma,\ S\sigma\rangle{\downarrow}^n \implies \langle \Gamma\sigma\{\vec{y} = \vec{V}\sigma\},\ N\sigma,\ S\sigma\rangle{\downarrow}^{\leqslant n}$

$\iff \forall \Gamma, S, \vec{y} \supseteq \mathsf{FV}(\Gamma, S). \langle \Gamma\sigma\{\vec{y} = \vec{V}\sigma\},\ M\sigma,\ S\sigma\rangle{\downarrow}^n \implies \langle \Gamma\sigma\{\vec{y} = \vec{V}\sigma\},\ N\sigma,\ S\sigma\rangle{\downarrow}^{\leqslant n}$

$\iff \forall \Gamma, S, \vec{x} \supseteq \mathsf{FV}(\Gamma, S). \langle \Gamma\{\vec{x} = \vec{V}\},\ M,\ S\rangle{\downarrow}^n \implies \langle \Gamma\{\vec{x} = \vec{V}\},\ N,\ S\rangle{\downarrow}^{\leqslant n}$

where the last step follows by renaming. □

$$\begin{array}{ll}
\quad \mathsf{let}\ \{\vec{x} = \vec{V}\}\ \mathsf{in}\ M \underset{\sim}{\triangleright} \mathsf{let}\ \{\vec{x} = \vec{V}\}\ \mathsf{in}\ N & \\
\Longrightarrow \mathsf{let}\ \{\vec{x} = \vec{V}\}\ \mathsf{in}\ M\sigma' \underset{\sim}{\triangleright} \mathsf{let}\ \{\vec{x} = \vec{V}\}\ \mathsf{in}\ N\sigma' & \sigma' = [{}^{z}\!/\!_{y}], z \text{ fresh} \\
\Longrightarrow \mathsf{let}\ \{z = V_i[{}^{z}\!/\!_{x_i}]\}\ \mathsf{in}\ \mathsf{let}\ \{\vec{x} = \vec{V}\}\ \mathsf{in}\ M\sigma' \underset{\sim}{\triangleright} \mathsf{let}\ \{z = V_i[{}^{z}\!/\!_{x_i}]\}\ \mathsf{in}\ \mathsf{let}\ \{\vec{x} = \vec{V}\}\ \mathsf{in}\ N\sigma' & (\text{cong.}) \\
\Longrightarrow \mathsf{let}\ \{z = V_i[{}^{z}\!/\!_{x_i}], \vec{x} = \vec{V}\}\ \mathsf{in}\ M\sigma' \underset{\sim}{\triangleright} \mathsf{let}\ \{z = V_i[{}^{z}\!/\!_{x_i}], \vec{x} = \vec{V}\}\ \mathsf{in}\ N\sigma' & (\textit{let-let}) \\
\Longrightarrow \mathsf{let}\ \{\vec{x} = \vec{V}\}\ \mathsf{in}\ M\sigma'\sigma'' \underset{\sim}{\triangleright} \mathsf{let}\ \{\vec{x} = \vec{V}\}\ \mathsf{in}\ N\sigma'\sigma'' & (gc), \sigma'' = [{}^{x_i}\!/\!_{z}]
\end{array}$$

Fig. 10. Calculational portion of the proof of lemma 11.9.

We will use this next lemma in the proof of ($\vdash$-*cong*).

Lemma 11.9. *Provided the $\vec{V}$ are closed, and $\vec{x} \lightning \operatorname{dom}\sigma$,*

$$\frac{\mathsf{let}\ \{\vec{x} = \vec{V}\}\ \mathsf{in}\ M \underset{\sim}{\triangleright} \mathsf{let}\ \{\vec{x} = \vec{V}\}\ \mathsf{in}\ N}{\mathsf{let}\ \{\vec{x} = \vec{V}\}\ \mathsf{in}\ M\sigma \underset{\sim}{\triangleright} \mathsf{let}\ \{\vec{x} = \vec{V}\}\ \mathsf{in}\ N\sigma}$$

Proof. It is sufficient to show this for a single renaming $[{}^{z}\!/\!_{y}]$, where $y \notin \operatorname{dom}\sigma$. Then the case when $z \notin \vec{x}$ follows from the fact that $\underset{\sim}{\triangleright}$ is closed under variable for variable substitution. So without loss of generality, let $\sigma = [{}^{x_i}\!/\!_{y}]$. By the reasoning in figure 10, we have that

$$\begin{array}{l}
\mathsf{let}\ \{\vec{x} = \vec{V}\}\ \mathsf{in}\ M \underset{\sim}{\triangleright} \mathsf{let}\ \{\vec{x} = \vec{V}\}\ \mathsf{in}\ N \\
\quad \Longrightarrow\ \mathsf{let}\ \{\vec{x} = \vec{V}\}\ \mathsf{in}\ M\sigma'\sigma'' \underset{\sim}{\triangleright} \mathsf{let}\ \{\vec{x} = \vec{V}\}\ \mathsf{in}\ N\sigma'\sigma''
\end{array}$$

But $\sigma'\sigma'' = [{}^{z}\!/\!_{y}][{}^{x_i}\!/\!_{z}] = [{}^{x_i}\!/\!_{y}] = \sigma$, since z was fresh, and we have the desired result. □

Moving to general contexts, to show ($\vdash$-*cong*), it will be sufficient to prove, under assumption of the premise, that for all $\mathbb{C}$ with a single hole variable ξ and $\vec{z}$ such that arity $\xi = |\vec{z}|$ and $\vec{z} \lightning \vec{x}$,

$$\mathsf{let}\ \{\vec{x} = \vec{V}\}\ \mathsf{in}\ \mathbb{C}[(\vec{z})M] \underset{\sim}{\triangleright} \mathsf{let}\ \{\vec{x} = \vec{V}\}\ \mathsf{in}\ \mathbb{C}[(\vec{z})N].$$

By the definition of $\underset{\sim}{\triangleright}$, it will suffice to show that for all $\mathbb{\Gamma}$ and $\mathbb{S}$,

$$\langle\, \mathbb{\Gamma}[(\vec{z})M]\{\vec{x} = \vec{V}\},\ \mathbb{C}[(\vec{z})M],\ \mathbb{S}[(\vec{z})M]\,\rangle{\downarrow^{n}} \implies$$
$$\langle\, \mathbb{\Gamma}[(\vec{z})N]\{\vec{x} = \vec{V}\},\ \mathbb{C}[(\vec{z})N],\ \mathbb{S}[(\vec{z})N]\,\rangle{\downarrow^{\leqslant n}}.$$

Suppose $\langle\, \mathbb{\Gamma}[(\vec{z})M]\{\vec{x} = \vec{V}\},\ \mathbb{C}[(\vec{z})M],\ \mathbb{S}[(\vec{z})M]\,\rangle{\downarrow^{n}}$ in k computation steps. We proceed via lexicographic induction on (n, k). Consider $\mathbb{\Sigma} = \langle\, \mathbb{\Gamma},\ \mathbb{C},\ \mathbb{S}\,\rangle$. Clearly $\mathbb{\Sigma} \rightarrow^{k_0}_{n_0} \mathbb{\Sigma}' \not\rightarrow$, so by open uniform computation, $\mathbb{\Sigma}'$ takes on one of the following forms:

$$\textbf{(1)}\ \langle\, \mathbb{\Delta},\ \mathbb{V},\ \epsilon\,\rangle, \qquad \textbf{(2)}\ \langle\, \mathbb{\Delta},\ x_i,\ \mathbb{T}\,\rangle, \qquad \textbf{(3)}\ \langle\, \mathbb{\Delta},\ \xi\cdot\vec{y},\ \mathbb{T}\,\rangle.$$

In case (1), we are done. In case (2), by (*Lookup*) and (*Update*), we have that

$$\langle\, \mathbb{\Delta}[(\vec{z})M]\{\vec{x} = \vec{V}\},\ x_i,\ \mathbb{T}[(\vec{z})M]\,\rangle \rightarrow^{2}_{1} \langle\, \mathbb{\Delta}[(\vec{z})M]\{\vec{x} = \vec{V}\},\ V_i,\ \mathbb{T}[(\vec{z})M]\,\rangle.$$

So by the inductive hypothesis,

$$\langle\, \mathbb{A}[(\vec{z})N]\{\vec{x} = \vec{V}\},\ V_i,\ \mathbb{T}[(\vec{z})N]\,\rangle{\downarrow}^{\leqslant n-(n_0+1)}$$

which in turn, by (*Lookup*) and (*Update*), implies

$$\langle\, \mathbb{A}[(\vec{z})N]\{\vec{x} = \vec{V}\},\ x_i,\ \mathbb{T}[(\vec{z})N]\,\rangle{\downarrow}^{\leqslant n-n_0}.$$

Then the desired result follows by open uniform computation.

In case (3), we have that

$$\langle\, \mathbb{A}[(\vec{z})M]\{\vec{x} = \vec{V}\},\ M[\vec{y}/\vec{z}],\ \mathbb{T}[(\vec{z})M]\,\rangle{\downarrow}^{n-n_0}. \tag{11.7}$$

By lemma 11.9, the assumption implies that

$$\mathsf{let}\ \{\vec{x} = \vec{V}\}\ \mathsf{in}\ M[\vec{y}/\vec{z}] \gtrsim \mathsf{let}\ \{\vec{x} = \vec{V}\}\ \mathsf{in}\ N[\vec{y}/\vec{z}]$$

which in turn, by lemma 11.8 and (11.7), implies

$$\langle\, \mathbb{A}[(\vec{z})M]\{\vec{x} = \vec{V}\},\ N[\vec{y}/\vec{z}],\ \mathbb{T}[(\vec{z})M]\,\rangle{\downarrow}^{\leqslant n-n_0}. \tag{11.8}$$

We are required to show instead

$$\langle\, \mathbb{A}[(\vec{z})N]\{\vec{x} = \vec{V}\},\ N[\vec{y}/\vec{z}],\ \mathbb{T}[(\vec{z})N]\,\rangle{\downarrow}^{\leqslant n-n_0}.$$

Consider $\langle\, \mathbb{A},\ N[\vec{y}/\vec{z}],\ \mathbb{T}\,\rangle$. By (11.8), this reduces in k_1 steps with cost n_1 to some $\mathbb{\Sigma} \not\twoheadrightarrow$. By open uniform computation, $\mathbb{\Sigma}$ has one of the following forms:

$$\mathbf{(3.1)}\langle\, \mathbb{A}',\ \mathbb{W},\ \epsilon\,\rangle,\ \mathbf{(3.2)}\langle\, \mathbb{A}',\ x_i,\ \mathbb{T}'\,\rangle,\ \mathbf{(3.3)}\langle\, \mathbb{A}',\ \xi\cdot\vec{w},\ \mathbb{T}'\,\rangle.$$

In case (3.1), we are done. In case (3.2), we appeal to case (2) above. In case (3.3), since $N[\vec{y}/\vec{z}] \not\equiv \xi\cdot\vec{w}$, $k_1 > 0$, so the inductive hypothesis applies, and we have

$$\langle\, \mathbb{A}'[(\vec{z})N]\{\vec{x} = \vec{V}\},\ N[\vec{w}/\vec{z}],\ \mathbb{T}'[(\vec{z})N]\,\rangle{\downarrow}^{\leqslant n-n_0-n_1}$$

and the desired result follows by open uniform computation.

11.7 Proof: the Unwinding Lemma

To prove the Unwinding lemma we will need the following lemma, which we state without proof.

Lemma 11.10. *For all M, Γ, S, V and n,*

$$\langle\, \Gamma\{x \overset{k}{=} V\},\ M,\ S\,\rangle{\downarrow}^{n} \implies \langle\, \Gamma\{x \overset{k+1}{=} V\},\ M\sigma,\ S\sigma\,\rangle{\downarrow}^{n}$$

where $\sigma = [{}^{x_{k+1}}/_{x_k}]$ *and* $\{x_i\}_{i=0}^{k+1} \nmid \mathsf{FV}(V)$.

Recall the statement of the Unwinding lemma:

For all $\Gamma, S,$ *and* n*,*

$$\langle\, \Gamma,\ \mathsf{let}\ \{f = V\}\ \mathsf{in}\ M,\ S\,\rangle{\downarrow}^{n} \implies \exists m.\langle\, \Gamma,\ \mathsf{let}\ \{f \overset{m}{=} V\}\ \mathsf{in}\ M[{}^{f_m}/_f],\ S\,\rangle{\downarrow}^{n}.$$

It suffices to prove that for all Γ, S, and n such that $\{x_i\}_{i=0}^{n} \nmid \mathsf{FV}(\Gamma, S)$,

$$\langle\, \Gamma\{x = V\},\ M,\ S\,\rangle{\downarrow}^{n} \implies \langle\, \Gamma\sigma\{x \overset{n}{=} V\},\ M\sigma,\ S\sigma\,\rangle{\downarrow}^{n}$$

where $\sigma = [{}^{x_n}/_{x}]$ (*i.e.* $m = n$). Suppose $\langle \Gamma\{x = V\},\ M,\ S\rangle{\downarrow}^n$ in k computation steps. We proceed by lexicographic induction on (n, k). By open uniform computation, $\langle \Gamma,\ M,\ S\rangle$ reduces in $k_0 \geqslant 0$ steps with cost n_0 to one of

$$\textbf{(1)}\ \langle \Delta,\ W,\ \epsilon\rangle, \qquad \textbf{(2)}\ \langle \Delta,\ x,\ T\rangle.$$

(Type (ii) cannot occur, since there is no hole involved.) By extension, the corresponding result holds for $\langle \Gamma,\ M,\ S\rangle\sigma$, and hence for $\langle \Gamma\sigma,\ M\sigma,\ S\sigma\rangle$, since x_n is free in $\langle \Gamma,\ M,\ S\rangle$.

Therefore, in case (1), by extension, $\langle \Gamma\sigma\{x = V\},\ M\sigma,\ S\sigma\rangle$ reduces in k_0 steps (with cost n_0) to $\langle \Delta\sigma\{x = V\},\ W\sigma,\ \epsilon\rangle$ and we are done, since $k_0 = k$ and $n_0 = n$.

In case (2), by extension, (*Lookup*), and (*Update*),

$$\begin{aligned}\langle \Gamma\sigma\{x = V\},\ M\sigma,\ S\sigma\rangle &\rightarrow^{k_0}_{n_0} \langle \Delta\sigma\{x = V\},\ x_n,\ T\sigma\rangle\\ &\rightarrow^{2}_{1} \langle \Delta\sigma\{x = V\},\ V\sigma,\ T\sigma\rangle.\end{aligned}$$

Similarly, $\langle \Gamma\{x = V\},\ M,\ S\rangle$ reduces in $k_0 + 2$ steps (with cost $n_0 + 1$) to $\langle \Delta\{x = V\},\ V,\ T\rangle$. By the inductive hypothesis, we know that $\langle \Delta\sigma'\{x \overset{n'}{=} V\},\ V\sigma',\ T\sigma'\rangle{\downarrow}^{n'}$ where $\sigma' = [{}^{x_{n'}}/_{x}]$ and $n' = n - (n_0 + 1)$. By repeated application of lemma 11.10, we have that $\langle \Delta\sigma\{x \overset{n}{=} V\},\ V\sigma,\ T\sigma\rangle{\downarrow}^{n'}$ and hence $\langle \Gamma\sigma\{x \overset{n}{=} V\},\ M\sigma,\ S\sigma\rangle{\downarrow}^n$ as required.

11.8 Proof: the Improvement Theorem

We prove the improvement theorem generalised to mutually-recursive definitions:

The following proof rule is sound:

$$\frac{\forall j \in I.\, \mathsf{let}\ \{f_i = V_i\}_{i\in I}\ \mathsf{in}\ V_j \mathrel{\trianglerighteq\!\!\sim} \mathsf{let}\ \{f_i = V_i\}_{i\in I}\ \mathsf{in}\ W_j}{\mathsf{let}\ \{f_i = V_i\}_{i\in I}\ \mathsf{in}\ N \mathrel{\trianglerighteq\!\!\sim} \mathsf{let}\ \{f_i = W_i\}_{i\in I}\ \mathsf{in}\ N}$$

By the context lemma it suffices to show that for all Γ, S, and n,

$$\langle \Gamma\{\vec{f} = \vec{V}\},\ N,\ S\rangle{\downarrow}^n \implies \langle \Gamma\{\vec{f} = \vec{W}\},\ N,\ S\rangle{\downarrow}^{\leqslant n}.$$

Assume the premise, and suppose that $\langle \Gamma\{\vec{f} = \vec{V}\},\ N,\ S\rangle{\downarrow}^n$ in k computation steps. We proceed by lexicographic induction on (n, k). By open uniform computation, $\langle \Gamma,\ N,\ S\rangle$ reduces in $k_0 \geqslant 0$ steps, with cost n_0, to one of

$$\textbf{(1)}\ \langle \Delta,\ V,\ \epsilon\rangle, \qquad \textbf{(2)}\ \langle \Delta,\ f_i,\ T\rangle.$$

In case (1), we have by extension that $\langle \Gamma\{\vec{f} = \vec{W}\},\ N,\ S\rangle$ reduces in k_0 steps to $\langle \Delta\{\vec{f} = \vec{W}\},\ V,\ \epsilon\rangle$ and $k_0 = k$ and $n_0 = n$, so we are done. In case (2),

$$\begin{aligned}\langle \Gamma\{\vec{f} = \vec{V}\},\ N,\ S\rangle &\rightarrow^{k_0}_{n_0} \langle \Delta\{\vec{f} = \vec{V}\},\ f_i,\ T\rangle\\ &\rightarrow^{2}_{1} \langle \Delta\{\vec{f} = \vec{V}\},\ V_i,\ T\rangle\end{aligned} \tag{11.9}$$

and

$$\begin{aligned}\langle \Gamma\{\vec{f} = \vec{W}\},\ N,\ S\rangle &\rightarrow^{k_0}_{n_0} \langle \Delta\{\vec{f} = \vec{W}\},\ f_i,\ T\rangle\\ &\rightarrow^{2}_{1} \langle \Delta\{\vec{f} = \vec{W}\},\ W_i,\ T\rangle\end{aligned} \tag{11.10}$$

so

$$\begin{array}{ll}
\langle\, \Delta\{\vec{f} = \vec{V}\},\ V_i,\ T\,\rangle{\downarrow}^{n-(n_0+1)} & (11.9)\\
\Longrightarrow \langle\, \Delta\{\vec{f} = \vec{V}\},\ W_i,\ T\,\rangle{\downarrow}^{\leqslant n-(n_0+1)} & \text{(ass., lem. 11.8)}\\
\Longrightarrow \langle\, \Delta\{\vec{f} = \vec{W}\},\ W_i,\ T\,\rangle{\downarrow}^{\leqslant n-(n_0+1)} & \text{(I.H.)}\\
\Longrightarrow \langle\, \Gamma\{\vec{f} = \vec{W}\},\ N,\ S\,\rangle{\downarrow}^{\leqslant n}. & (11.10)
\end{array}$$

11.9 Proof: Improvement Induction

We prove instead the more general version, involving entailment:

For any set of recursive declarations $\vec{f}$, terms M, N and substitution σ, the following proof rule is sound:

$$\frac{\vec{f} \vdash M \underset{\sim}{\rhd} {}^{\checkmark}\mathbb{C}[M\sigma] \qquad \vec{f} \vdash N \underset{\sim}{\lhd\!\rhd} {}^{\checkmark}\mathbb{C}[N\sigma]}{\vec{f} \vdash M \underset{\sim}{\rhd} N}$$

Furthermore, we generalise $\mathbb{C}[M\sigma]$ to $\mathbb{C}[(\vec{x})M]$. By lemma 11.8, the premises imply more general statements. For example, the first premise implies

$$\forall n, \Gamma, S.\ \langle\, \Gamma\{\vec{f} = \vec{V}\},\ M,\ S\,\rangle{\downarrow}^{n} \implies \langle\, \Gamma\{\vec{f} = \vec{V}\},\ {}^{\checkmark}\mathbb{C}[(\vec{x})M],\ S\,\rangle{\downarrow}^{\leqslant n} \qquad \text{(ass.(i))}$$

We will refer to the corresponding generalisation of the second premise as (ass.(ii)). We show instead the more general statement, that for all $\mathbb{\Sigma}$ and n,

$$\mathbb{\Sigma}[(\vec{x})M]{\downarrow}^{n} \implies \mathbb{\Sigma}[(\vec{x})N]{\downarrow}^{\leqslant n}.$$

Suppose $\mathbb{\Sigma}[(\vec{x})M]{\downarrow}^{n}$ in k computation steps. We proceed by lexicographic induction on (n, k). By open uniform computation, $\mathbb{\Sigma}$ reduces in $k_0 \geqslant 0$ to one of

$$\textbf{(1)}\ \langle\, \mathbb{\Delta},\ \mathbb{V},\ \epsilon\,\rangle, \qquad \textbf{(2)}\ \langle\, \mathbb{\Delta},\ \xi\cdot\vec{y},\ \mathbb{T}\,\rangle.$$

In case (1), we are done. In case (2), first note that, letting $\sigma = [\vec{y}/\vec{x}]$, $(\vec{x})M\cdot\vec{y} \equiv M\sigma$, and $\mathbb{C}[(\vec{x})M]\sigma \equiv \mathbb{C}\sigma[(\vec{x})M]$ since $\vec{x} \supseteq \mathsf{FV}(M)$, and similarly for N. Then we have that

$$\mathbb{\Sigma}[(\vec{x})N] \to^{k_0}_{n_0} \langle\, \mathbb{\Delta}[(\vec{x})N],\ N\sigma,\ \mathbb{T}[(\vec{x})N]\,\rangle \qquad (11.11)$$

and

$$\begin{array}{ll}
\langle\, \mathbb{\Delta}[(\vec{x})M],\ M\sigma,\ \mathbb{T}[(\vec{x})M]\,\rangle{\downarrow}^{n-n_0} & \\
\Longrightarrow \langle\, \mathbb{\Delta}[(\vec{x})M],\ {}^{\checkmark}\mathbb{C}[(\vec{x})M]\sigma,\ \mathbb{T}[(\vec{x})M]\,\rangle{\downarrow}^{\leqslant n-n_0} & \text{(ass.(i))}\\
\Longrightarrow \langle\, \mathbb{\Delta}[(\vec{x})M],\ \mathbb{C}[(\vec{x})M]\sigma,\ \mathbb{T}[(\vec{x})M]\,\rangle{\downarrow}^{\leqslant n-(n_0+1)} & (\checkmark)\\
\equiv\ \langle\, \mathbb{\Delta}[(\vec{x})M],\ \mathbb{C}\sigma[(\vec{x})M],\ \mathbb{T}[(\vec{x})M]\,\rangle{\downarrow}^{\leqslant n-(n_0+1)} & \\
\Longrightarrow \langle\, \mathbb{\Delta}[(\vec{x})N],\ \mathbb{C}\sigma[(\vec{x})N],\ \mathbb{T}[(\vec{x})N]\,\rangle{\downarrow}^{\leqslant n-(n_0+1)} & \text{(I.H.)}\\
\equiv\ \langle\, \mathbb{\Delta}[(\vec{x})N],\ \mathbb{C}[(\vec{x})N]\sigma,\ \mathbb{T}[(\vec{x})N]\,\rangle{\downarrow}^{\leqslant n-(n_0+1)} & \\
\Longrightarrow \langle\, \mathbb{\Delta}[(\vec{x})N],\ {}^{\checkmark}\mathbb{C}[(\vec{x})N]\sigma,\ \mathbb{T}[(\vec{x})N]\,\rangle{\downarrow}^{\leqslant n-n_0} & (\checkmark)\\
\Longleftrightarrow \langle\, \mathbb{\Delta}[(\vec{x})N],\ N\sigma,\ \mathbb{T}[(\vec{x})N]\,\rangle{\downarrow}^{\leqslant n-n_0} & \text{(ass.(ii))}\\
\Longrightarrow \mathbb{\Sigma}[(\vec{x})N]{\downarrow}^{\leqslant n}. & (11.11)
\end{array}$$

12. CONCLUSIONS AND FUTURE WORK

We have presented a rich operational theory for a call-by-need based on an improvement ordering on programs. The theory subsumes the (oriented) call-by-need lambda calculi of Ariola *et al.* [Ariola et al. 1995]. The most important extensions are proof techniques for reasoning about recursion. Syntactic continuity allows us to prove properties of recursive programs via a kind of fixed-point induction, without sacrificing information about intensional behaviour, like sharing. The improvement theorem and improvement induction are rules for recursion which support more calculational proofs. Both are particularly useful in proving the safety of program transformations.

An obvious further application of the theory is to formalise arguments about the running time of programs, following Sands' use of call-by-name cost equivalence for this purpose [Sands 1995; Sands 1998b].

Another direction for future work would be to consider the time-safety of a larger-scale program transformation, such as deforestation [Wadler 1990]. In such a transformation we must inevitably consider conditions under which we can unfold function calls. It is straightforward to define simple syntactic conditions on contexts which guarantee that

$$\mathsf{let}\ \{\vec{x} = \vec{M}\}\ \mathsf{in}\ \mathbb{C}[\vec{x}] \trianglerighteq\!\!\!\sim\ \mathsf{let}\ \{\vec{x} = \vec{M}\}\ \mathsf{in}\ \mathbb{C}[\vec{M}],$$

but in the case where holes occur under λ-abstractions a more global form of information is required: one needs to know that the lambda expression in question will not be applied more than once. The type system of [Turner et al. 1995] provides just such global information, so it would be interesting to prove that their system (and generalisations to full recursive lets [Gustavsson 1998]) does indeed satisfy the desired improvement property above. We saw in section 6.4 that the strictness property of a context can be characterised exactly by

$$\mathbb{C}[{}^{\checkmark}x] \trianglerighteq\!\!\!\sim {}^{\checkmark}\mathbb{C}[x],$$

where x is fresh. Could it be the case that the "used at most once" property might be semantically characterised by ${}^{\checkmark}\mathbb{C}[x] \trianglerighteq\!\!\!\sim \mathbb{C}[{}^{\checkmark}x]$?

Acknowledgements. We have benefited from numerous discussions with Jörgen Gustavsson on various aspects of this work, and we would thank him in particular for his suggestions which led to a simplification of the proof of the context lemma. Thanks also to Koen Claessen, Keith Wansbrough, and the referees for their helpful comments.

REFERENCES

Abramsky, S. and Ong, C.-H. L. 1993. Full abstraction in the lazy lambda calculus. *Information and Computation 105*, 159–267.

Agha, G. A., Mason, I. A., Smith, S. F., and Talcott, C. L. 1997. A foundation for actor computation. *Journal of Functional Programming 7*, 1–72.

Ariola, Z., Felleisen, M., Maraist, J., Odersky, M., and Wadler, P. 1995. A call-by-need lambda calculus. In *Proc. POPL'95, the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Jan. 1995), pp. 233–246. ACM Press.

Ariola, Z. M. and Blom, S. 1997. Cyclic lambda calculi. In *Proc. TACS'97*, Volume 1281 of *LNCS*, pp. 77–106. Springer-Verlag.

ARIOLA, Z. M. AND BLOM, S. 1998. Lambda calculi plus letrec. Technical report, Dept. of Computer Science, University of Oregon. Extended version of [Ariola and Blom 1997]; submitted for publication.

ARIOLA, Z. M. AND FELLEISEN, M. 1997. The call-by-need lambda calculus. *Journal of Functional Programming 7*, 3 (May), 265–301.

ARIOLA, Z. M. AND KLOP, J. W. 1997. Lambda calculus with explicit recursion. *Information and Computation 139*, 2, 154–233.

BENAISSA, Z.-E.-A., LESCANNE, P., AND ROSE, K. H. 1996. Modeling sharing and recursion for weak reduction strategies using explicit substitution. In *Proc. PLILP'96, the 8th International Symposium on Programming Languages, Implementations, Logics, and Programs*, Volume 1140 of *LNCS*, pp. 393–407. Springer-Verlag.

BURSTALL, R. M. AND DARLINGTON, J. 1977. A transformational system for developing recursive programs. *Journal of the ACM 24*, 1 (Jan.), 44–67.

CURIEN, P.-L. 1991. An abstract framework for environment machines. *Theoretical Computer Science 82*, 2 (May), 389–402.

FIELD, J. 1990. On laziness and optimality in lambda interpreters: Tools for specification and analysis. In *Proc. POPL'90, the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Jan. 1990), pp. 1–15. ACM Press.

GORDON, A. D. AND PITTS, A. M. Eds. 1998. *Higher Order Operational Techniques in Semantics*. Publications of the Newton Institute. Cambridge University Press.

GUSTAVSSON, J. 1998. A type based sharing analysis for update avoidance and optimisation. In *Proc. ICFP'98, the 3rd ACM SIGPLAN International Conference on Functional Programming* (Sept. 1998), pp. 39–50.

GUSTAVSSON, J. AND SANDS, D. 1999. A foundation for space-safe transformations of call-by-need programs. In A. D. GORDON AND A. PITTS Eds., *Proc. HOOTS III, the 3rd Workshop on Higher Order Operational Techniques in Semantics*, Volume 26 of *Electronic Notes in Theoretical Computer Science* (1999). Elsevier Science Publishers B.V.

HUGHES, J. AND MORAN, A. K. 1995. Making choices lazily. In *Proc. FPCA'95, ACM Conference on Functional Programming Languages and Computer Architecture* (June 1995), pp. 108–119. ACM Press.

JEFFREY, A. 1993. A fully abstract semantics for concurrent graph reduction. Technical Report 93:12, School of Cognitive and Computing Sciences, University of Sussex.

JEFFREY, A. 1994. A fully abstract semantics for concurrent graph reduction. In *Proc. LICS'94, the 9th IEEE Symposium on Logic in Computer Science* (July 1994), pp. 82–91. IEEE Computer Society Press.

JOSEPHS, M. B. 1989. The semantics of lazy functional languages. *Theoretical Computer Science 68*, 1 (Oct.), 105–111.

LASSEN, S. B. 1998. *Relational Reasoning about Functions and Nondeterminism*. Ph. D. thesis, Department of Computer Science, University of Aarhus.

LAUNCHBURY, J. 1993. A natural semantics for lazy evaluation. In *Proc. POPL'93, the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Jan. 1993), pp. 144–154. ACM Press.

MARAIST, J., ODERSKY, M., AND WADLER, P. 1998. The call by need lambda calculus. *Journal of Functional Programming 8*, 3 (May), 275–317.

MARANGET, L. 1991. Optimal derivations in weak lambda-calculi and in orthogonal term rewriting systems. In *Proc. POPL'91, the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Jan. 1991), pp. 255–269. ACM Press.

MASON, I. AND TALCOTT, C. 1991. Equivalence in functional languages with effects. *Journal of Functional Programming 1*, 3 (July), 287–327.

MASON, I. A., SMITH, S. F., AND TALCOTT, C. L. 1996. From operational semantics to domain theory. *Information and Computation 128*, 1 (July), 26–47.

MEIJER, E., FOKKINGA, M., AND PATERSON, R. 1991. Functional programming with bananas, lenses, envelopes and barbed wire. In J. HUGHES Ed., *Proc. FPCA'91, ACM Conference on Functional Programming Languages and Computer Architecture*, Volume 523 of *LNCS* (Aug. 1991), pp. 124–144. Springer-Verlag.

Milner, R. 1977. Fully abstract models of the typed λ-calculus. *Theoretical Computer Science 4*, 1–22.

Moran, A. K. 1998. *Call-by-name, Call-by-need, and McCarthy's Amb.* Ph. D. thesis, Department of Computing Sciences, Chalmers University of Technology, Sweden.

Moran, A. K. and Sands, D. 1998. Improvement in a lazy context: An operational theory for call-by-need (extended version). Extended version of [Moran and Sands 1999].

Moran, A. K. and Sands, D. 1999. Improvement in a lazy context: An operational theory for call-by-need. In *Proc. POPL'99, the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Jan. 1999). ACM Press.

Moran, A. K., Sands, D., and Carlsson, M. 1999. Erratic Fudgets: A semantic theory for an embedded coordination language. In *Coordination '99*, Volume 1594 of *LNCS* (April 1999). Springer-Verlag.

Niehren, J. 1996. Functional computation as concurrent computation. In *Proc. POPL'96, the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Jan. 1996), pp. 333–343. ACM Press.

Peyton Jones, S. L., Partain, W., and Santos, A. 1996. Let-floating: moving bindings to give faster programs. In *Proc. ICFP'96, the 1st ACM SIGPLAN International Conference on Functional Programming* (May 1996), pp. 1–12. ACM Press.

Peyton Jones, S. L. and Santos, A. 1998. A transformation-based optimiser for Haskell. *Science of Computer Programming 32*, 1–3, 3–47.

Pitts, A. M. 1994. Some notes on inductive and co-inductive techniques in the semantics of functional programs. Notes Series BRICS-NS-94-5 (Dec.), BRICS, Department of Computer Science, University of Aarhus.

Pitts, A. M. 1997a. Operational semantics for program equivalence. Invited talk at MFPS XIII, the 13th Conference on Mathematical Foundations of Programming Semantics, slides available at http://www.cl.cam.ac.uk/users/ap/talks/ mfps97.ps.gz.

Pitts, A. M. 1997b. Operationally-based theories of program equivalence. In P. Dybjer and A. M. Pitts Eds., *Semantics and Logics of Computation*, Publications of the Newton Institute, pp. 241–298. Cambridge University Press.

Rose, K. H. 1996. *Operational Reduction Models for Functional Programming Languages.* Ph. D. thesis, DIKU, University of Copenhagen, Denmark. available as DIKU report 96/1.

Sands, D. 1991. Operational theories of improvement in functional languages (extended abstract). In *Proc. 1991 Glasgow Functional Programming Workshop*, Workshops in Computing Series (Aug. 1991), pp. 298–311. Springer-Verlag.

Sands, D. 1995. A naïve time analysis and its theory of cost equivalence. *Journal of Logic and Computation 5*, 4, 495–541.

Sands, D. 1996. Total correctness by local improvement in the transformation of functional program. *ACM Transactions on Programming Languages and Systems (TOPLAS) 18*, 2 (March), 175–234.

Sands, D. 1997. From SOS rules to proof principles: An operational metatheory for functional languages. In *Proc. POPL'97, the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Jan. 1997). ACM Press.

Sands, D. 1998a. Computing with contexts: A simple approach. In A. D. Gordon, A. M. Pitts, and C. L. Talcott Eds., *Proc. HOOTS II, the 2nd Workshop on Higher Order Operational Techniques in Semantics*, Volume 10 of *Electronic Notes in Theoretical Computer Science* (1998). Elsevier Science Publishers B.V. at http://www.elsevier.nl/cas/tree/store/ tcs/free/noncas/pc/menu.htm.

Sands, D. 1998b. Improvement theory and its applications. Publications of the Newton Institute, pp. 275–306. Cambridge University Press.

Sansom, P. and Peyton Jones, S. L. 1997. Formally-based profiling for higher-order functional languages. *ACM Transactions on Programming Languages and Systems (TOPLAS) 19*, 1 (Jan.), 334–385.

SEAMAN, J. AND PURUSHOTHAMAN IYER, S. 1996. An operational semantics of sharing in lazy evaluation. *Science of Computer Programming 27*, 3 (Nov.), 289–322.

SESTOFT, P. 1997. Deriving a lazy abstract machine. *Journal of Functional Programming 7*, 3 (May), 231–264.

SMITH, S. F. 1991. From operational to denotational semantics. In S. BROOKES, M. MAIN, A. MELTON, M. MISLOVE, AND D. SCHMIDT Eds., *Proc. MFPS VII, the 7th Conference on Mathematical Foundations of Programming Semantics*, Volume 598 of *LNCS* (March 1991), pp. 54–76. Springer-Verlag.

TALCOTT, C. L. 1998. Reasoning about functions with effects. Publications of the Newton Institute, pp. 347–390. Cambridge University Press.

TULLSEN, M. AND HUDAK, P. 1998. An intermediate meta-language for program transformation. YALEU/DCS/RR 1154 (June), Yale University.

TURNER, D. N., WADLER, P., AND MOSSIN, C. 1995. Once upon a type. In *Proc. FPCA'95, ACM Conference on Functional Programming Languages and Computer Architecture* (June 1995), pp. 1–11. ACM Press.

WADLER, P. 1988. The concatenate vanishes. Technical report, University of Glasgow (UK). appeared as a note on an FP electronic mailing list, December 1987.

WADLER, P. 1990. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science 73*, 231–248.

YOSHIDA, N. 1993. Optimal reduction in weak-lambda-calculus with shared environments. In *Proc. FPCA'93, ACM Conference on Functional Programming Languages and Computer Architecture* (June 1993), pp. 243–254. ACM Press.

APPENDIX

A. LOOKUPS ARE ENOUGH

In this appendix we justify the use of counting lookups as our cost measure, by proving theorem 4.1.

Despite the relatively high-level nature of our abstract machine, we argue (informally) that each abstract machine step can be implemented by constant-time operations, where the constant depends on the size of the program to be executed.[4]

The following observation is crucial to our argument:

PROPOSITION A.1. *During the execution of a given program, every term appearing in an abstract machine configuration is a substitution instance (variable for variable) of some subterm of the original program.*

PROOF. By inspection of the rules. □

Given this, and assuming that the variable lookup operation is implemented in constant time, we wish to argue that every transition can be implemented as a constant time operation. This would be straightforward to argue — but for rule (*Letrec*) which requires a non-constant amount of variable renaming. Fortunately, Sestoft [Sestoft 1997] provides a slightly lower level variant of this machine, in which renaming is completely avoided by the use of *environments*. As Sestoft notes, the correctness of this modification is clear. It is also clear that all of the rules can be implemented in constant time.

[4]For actual implementations one may have be able to give much more refined bound than simply program size (*e.g.* the maximum number of free variables of any subexpression in the program).

Working note: *I'm still not completely sure about this. One needs to argue that environment size is bounded by program size. I'm sure this is true, but I don't see why right away.*

In order to prove theorem 4.1, which says that just counting lookup steps is sufficient to capture computational complexity, we first introduce a size metric on terms, stacks and term-stack pairs:

DEFINITION 8.

$$\begin{aligned}
|x| &= 1\\
|c\,\vec{x}| &= 1\\
|\lambda x.M| &= |M| + 1\\
|M\,x| &= |M| + 2\\
|\mathsf{case}\ M\ \mathsf{of}\ \{c_i\,\vec{x}_i \rightarrow N_i\}| &= |M| + 1 + \Sigma_{i=1}^{i=n}|N_i|\\
|\mathsf{let}\ \{\vec{x} = \vec{M}\}\ \mathsf{in}\ N| &= |N| + 1 + \Sigma_{i=1}^{i=n}|M_i|
\end{aligned}$$

$$\begin{aligned}
|\epsilon| &= 0\\
|x : S| &= |S| + 1\\
|\#x : S| &= |S| + 1\\
|\{c_i\,\vec{x}_i \rightarrow N_i\} : S| &= |S| + \Sigma_{i=1}^{i=n}|N_i|
\end{aligned}$$

$$|M, S| = |M| + |S|$$

With the exception of rule (*Lookup*), the combined term and stack size decreases strictly with each abstract machine transition, *i.e.* if

$$\langle \Gamma,\ M,\ S \rangle \rightarrow \langle \Gamma',\ N,\ T \rangle$$

then $|M,S| > |N,T|$. (*Letrec*) adds a group of bindings to the heap, and thus decreases the metric by an amount dependent upon the size of the bindings made plus 1; the others decrease it by exactly 1. (*Lookup*) is the exception: the metric is increased by an amount equal to the size of the term to be evaluated.

Recall the statement of theorem 4.1:

For all $s > 0$, there exists a linear function f such that for all closed terms M of size s,

$$M{\downarrow}^m \implies M{\Downarrow}^{\leqslant f(m)}.$$

PROOF. Consider some M of size s which converges in n steps to some final state

$\langle\,\Delta,\ V,\ \epsilon\,\rangle$. We partition the transition sequence thus:

$$\begin{aligned}
\langle\,\emptyset,\ M,\ \epsilon\,\rangle &\equiv \langle\,\Gamma_0,\ M_0,\ S_0\,\rangle\\
&\rightarrow^{k_0} \langle\,\Delta_0,\ N_0,\ T_0\,\rangle \rightarrow^{\#} \langle\,\Gamma_1,\ M_1,\ S_1\,\rangle\\
&\rightarrow^{k_1} \langle\,\Delta_1,\ N_1,\ T_1\,\rangle \rightarrow^{\#} \langle\,\Gamma_2,\ M_2,\ S_2\,\rangle\\
&\qquad \ldots\\
&\rightarrow^{k_{m-1}} \langle\,\Delta_{m-1},\ N_{m-1},\ T_{m-1}\,\rangle \rightarrow^{\#} \langle\,\Gamma_m,\ M_m,\ S_m\,\rangle\\
&\rightarrow^{k_m} \langle\,\Delta_m,\ N_m,\ T_m\,\rangle\\
&\equiv \langle\,\Delta,\ V,\ \epsilon\,\rangle
\end{aligned}$$

where m is the total number of instances of rule (*Lookup*) (marked by a #). We know the following facts

$$k_i \leqslant |M_i, S_i| - |N_i, T_i| \tag{A.1}$$

$$|M_i| \leqslant |M| \tag{A.2}$$

$$|M_{i+1}, S_{i+1}| - |N_i, T_i| = |M_{i+1}| \tag{A.3}$$

$$|M_i, S_i| \leqslant i|M| \tag{A.4}$$

(A.1) follows since there are k_i non-(*Lookup*) transitions in moving from $\langle\,\Gamma_i,\ M_i,\ S_i\,\rangle$ to $\langle\,\Delta_i,\ N_i,\ T_i\,\rangle$, and each transition decreases the metric by at least one. (A.2) follows from the fact any term arising during the evaluation of M must be a substitution instance of a sub-term of M, and therefore smaller than M. Since it is (*Lookup*) that takes $\langle\,\Delta_i,\ N_i,\ T_i\,\rangle$ to $\langle\,\Gamma_{i+1},\ M_{i+1},\ S_{i+1}\,\rangle$, the difference in size is exactly $|M_{i+1}|$, yielding (A.3). As for (A.4), we argue as follows. Since only lookups can increase the size of the term-stack pair, and since the increase is bounded by $|M|$, we conclude that $|M_i, S_i|$ cannot be larger than $i|M|$.

Working note: *Where do we use* (A.4)*?*

$$\begin{array}{rll}
\Sigma_{i=0}^{i=m} k_i & \leqslant \Sigma_{i=0}^{i=m} |M_i, S_i| - |N_i, T_i| & \text{(A.1)} \\
& = |M_0, S_0| - |N_0, T_0| + & \\
& \quad |M_1, S_1| - |N_1, T_1| + & \\
& \quad \ldots & \\
& \quad |M_m, S_m| - |N_m, T_m| & \\
& = |M_0, S_0| + & \\
& \quad |M_1, S_1| - |N_0, T_0| + & \\
& \quad |M_2, S_2| - |N_1, T_1| + & \\
& \quad \ldots & \\
& \quad |M_m, S_m| - |N_{m-1}, T_{m-1}| + & \\
& \quad - |N_m, T_m| & \\
& = |M_0, S_0| + & \\
& \quad \Sigma_{i=0}^{i=m-1} |M_{i+1}, S_{i+1}| - |N_i, T_i| & \\
& \quad - |N_m, T_m| & \\
& \leqslant |M_0, S_0| + \Sigma_{i=0}^{i=m-1} |M_{i+1}| & \text{(A.3)} \\
& \leqslant |M| + m|M| & \text{(A.2)}, M \equiv M_0, S_0 \equiv \epsilon
\end{array}$$

Now

$$\begin{aligned}
n &= m + \Sigma_{i=0}^{i=m} k_i \\
&\leqslant m + |M| + m|M| \\
&= m + (m+1)s.
\end{aligned}$$

This is linear in m, so we are done. □

To summarise, we have argued that

(1) the number abstract-machine steps is within a program-size dependent constant factor of actual running time of an implementation based on the abstract machine, and

(2) the number of lookup steps is within a program-size dependent constant factor of the number of abstract machine steps.

This demonstrates the soundness of using the number of lookups as a measure of cost.