Improvement in a Lazy Context:  
An Operational Theory for Call-By-Need  
(Extended Version)

Andrew Moran    David Sands
Chalmers¹

Abstract

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.

In this article 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 $C$, when $C[M]$ terminates then $C[N]$ 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. [AFM⁺95]. 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 [PJS98]. However, it is notoriously difficult, even for those with extremely solid intuitions 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 [AFM⁺95, AF97, MOW98] 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 4.6), 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 [AFM⁺95] 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 opera-

¹Department of Computing Science, Chalmers University of Technology and the University of Göteborg, S-412 96 Göteborg, Sweden: [andrew.dave]@cs.chalmers.se

tional semantics by saying that: $M$ is improved by $N$ if in all program-contexts $C$, when $C[M]$ terminates then $C[N]$ 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.

**Overview** We begin with a discussion of related work in section 2. Section 3 then presents the operational semantics (Sestoft’s “mark 1” abstract machine for laziness). 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 presented in section 4, and the relative power of the algebra and the call-by-need calculi is discussed. Syntactic continuity is presented in section 5 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 6, along with improvement induction and examples of their use. Section 7 concludes, and we discuss of future avenues of research. For the interested reader, appendix A summarises the technical development and presents the proofs of the main theorems.

**2 Related Work**

Improvement theory and the improvement theorem were originally developed in the call-by-name setting [San91, San96], and generalised to a variety of call-by-name and call-by-value languages in [San97]. 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 Marais, Odersky and Wadler [AFM+95, AF97, MO'98]. 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., [Fie90, Mar91, Yos93].

One limitation of the original work by Ariola et al. is in the treatment of recursive cycles; naive extension of the calculi to deal with recursive lets leads to a loss of confluence [Jef93, AK97]. 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., [BLR96, Nie96]. Ariola and Blom give a full study of cyclic recursion in [AB97, AB98], and show that an approximation to confluence can be obtained by equating terms with the same infinite normal-form. Their normal order calculus can be seen as the natural successor to the call-by-need calculus.

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., [Ros96, BLR96] 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 calculus 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., [Joc89, Jef94, SP96, Lau93, Ses97]. 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., [TW95, HM95, SP97, Gus98]. Since it appears to be a non-controversial choice, we adopt Sestoft’s machine — essentially a Krivine-machine [Cur91] with updating of the heap — as the operational model underpinning our theory. As others have observed (e.g., [Ph97a]), working with an abstract machine rather than an inductive semantics also has benefits in proofs about computations (examples of this may be seen in the appendix).

**3 The Operational Semantics**

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

$$L, M, N ::= \lambda x. M \mid M \times \overline{x} \mid \mathrm{let} \overline{x} = M \mid \mathrm{in} \ N \mid \mathrm{case} \ M \in \{c_i, \overline{x_i} \rightarrow N_i\}.$$  

The syntactic restriction is now rather standard, following its use in core language of the Glasgow Haskell compiler, e.g., [PJPS96, PJ98], and in [Lau93, Ses97].
All constructors have a fixed arity, and are assumed to be saturated. By 
\( c \bar{x} \) we mean \( c \, \bar{x}_1 \cdots \bar{x}_n \). The only values 
are lambda expressions and fully-applied constructors. Throughout 
\( w, y, z \), and \( w \) will range over variables, \( c \) 
over constructor names, and \( V \) and \( W \) over values. We 
will write let \( \{ \bar{x} = \bar{M} \} \) in \( N \) as a shorthand for 
let \( \{ x_1 = M_1, \ldots, x_n = M_n \} \) in \( N \)
where the \( \bar{x} \) are distinct, the order of bindings is not 
syntactically significant, and the \( \bar{x} \) are considered bound 
in \( N \) and the \( \bar{M} \) (so our lets are recursive). Similarly we 
write case \( M \) of \( \{ c_1 \, \bar{x}_1 \rightarrow N_1 \} \cdots \{ c_m \, \bar{x}_m \rightarrow N_m \} \) for 
case \( M \) of \( \{ c_1 \, \bar{x}_1 \rightarrow N_1 \} \cdots \{ c_m \, \bar{x}_m \rightarrow N_m \} \) 
where each \( \bar{x}_i \) is a vector of distinct variables, and the \( c_i \) 
are distinct constructors. In addition, we will sometimes 
write \( \text{alt} \) as an abbreviation for case alternatives \( \{ c_i \, \bar{x}_i \rightarrow N_i \} \).
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
\[ MN \equiv \text{let } \{ x = N \} \text{ in } M \, x, \quad x \text{ fresh} \]
whenever \( N \) is not a variable. Similarly for constructor expressions.
The only kind of substitution that we consider is variable 
for variable, with \( \sigma \) ranging over such substitutions. The 
simultaneous substitution of one vector of variables for 
another will be written \( M[\bar{y}/\bar{x}] \), where the \( \bar{x} \) are assumed 
 to be distinct (but the \( \bar{y} \) need not be).

### 3.1 The Abstract Machine

The semantics presented in this section is essentially 
Sestoft’s “mark 1” abstract machine for laziness [Ses97]. In 
that paper, he proves his abstract machine semantics 
sound and complete with respect to Lauchbury’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 heap 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 \( \Gamma \), expression \( M \), and stack \( S \). We denote 
the empty heap by \( \emptyset \), and the addition of a group of bindings 
\( \bar{x} = \bar{M} \) to a heap \( \Gamma \) by juxtaposition: \( \Gamma(\bar{x} = \bar{M}) \).
The stack written \( b \cdot S \) will denote the a stack \( S \) with \( b \) 
pushed on the top. The empty stack is denoted by \( \epsilon \), 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 \( \Gamma \) as 
dom \( \Gamma \), and to the set of variables marked for update in 
a stack \( S \) as \( \text{dom} \, S \). Update markers should be thought 
of as binding occurrences of variables. A configuration 
is well-formed if dom \( \Gamma \) and dom \( S \) are disjoint. We 
write dom \( (\Gamma, \, M, \, S) \) for their union. For a configuration 
\( \langle \Gamma, \, M, \, S \rangle \) to be closed, any free variables in \( \Gamma \), \( M \), 
and \( S \) must be contained in dom \( (\Gamma, \, S) \). For sets of variables 
\( P \) and \( Q \) we will write \( P \cup Q \) to mean that \( P \) and \( Q \) are 
disjoint, i.e., \( P \cap Q = \emptyset \). The free variables of a term \( M \) 
will be denoted \( \text{FV} \, (M) \); for a vector of terms \( \bar{M} \), we will 
write \( \text{FV} \, (\bar{M}) \).
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 
have 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 substi-
tution 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 
occur, and can always be satisfied by a local \( \alpha \)-
conversion.

**Definition 3.1 (Convergence)** For closed configurations 
\( \langle \Gamma, \, M, \, S \rangle \),
\[ \langle \Gamma, \, M, \, S \rangle^\psi \triangleq \exists \Delta, \, V \, . \langle \Gamma, \, M, \, S \rangle \rightarrow^* \langle \Delta, \, V, \, \epsilon \rangle, \]
\[ \langle \Gamma, \, M, \, S \rangle^\psi \triangleq \exists m \, . \langle \Gamma, \, M, \, S \rangle^\psi_m, \]
\[ \langle \Gamma, \, M, \, S \rangle^\psi \triangleq \exists m \, . \langle \Gamma, \, M, \, S \rangle^\psi_m \land m \leq n. \]
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 let \( x = x \) in \( x \)). All non-
converging configurations will be semantically identified.
We will also write \( M_1^\psi \), \( M_1^\psi \), and \( M_1^\psi \), identifying 
closed \( M \) with the initial configuration \( \langle \emptyset, \, M, \, \epsilon \rangle \).

### 3.2 Program Contexts

The starting point for an operational theory is usually 
an approximation and an equivalence defined in terms of 
program contexts. Program contexts are usually intro-
duced 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.
\[
\begin{align*}
\langle \Gamma[x = M], x, S \rangle & \rightarrow \langle \Gamma, M, \#x : S \rangle \\
\langle \Gamma, V, \#x : S \rangle & \rightarrow \langle \Gamma[x = V], V, S \rangle \\
\langle \Gamma, Mx, S \rangle & \rightarrow \langle \Gamma, M, x : S \rangle \\
\langle \Gamma, \lambda x.M, y : S \rangle & \rightarrow \langle \Gamma, M[y/x], S \rangle \\
\langle \Gamma, \text{case } M \text{ of } \text{alts}, S \rangle & \rightarrow \langle \Gamma, M, \text{alts} : S \rangle \\
\langle \Gamma, c_j \bar{x} \ {c_i \bar{x}_i \rightarrow N_i} : S \rangle & \rightarrow \langle \Gamma, N_j[y_j/x_j], S \rangle \\
\langle \Gamma, \text{let } \bar{x} = \bar{M} \text{ in } N, S \rangle & \rightarrow \langle \Gamma[\bar{x} = \bar{M}], N, S \rangle \quad \bar{x} \notin \text{dom}(\Gamma, S)
\end{align*}
\]
(\text{Lookup}) (\text{Update}) (\text{Unwind}) (\text{Subst}) (\text{Case}) (\text{Branch}) (\text{Letrec})

Figure 1: The abstract machine semantics for call-by-need.

We will use contexts of the following form:

\[
\mathcal{C}, \mathcal{D} ::= [] \mid x \mid \lambda x.\mathcal{C} \mid \mathcal{C}x \mid e\bar{x}
\]
\[\text{let } \{\bar{x} = \bar{C}\} \text{ in } \mathcal{D} \mid \text{case } \mathcal{C} \text{ of } \{c_i \bar{x}_i \rightarrow \bar{D}_i\}.\]

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 the term to become captured.

The relationship between terms and configurations is characterised by a translation function from configurations to terms, defined inductively on the stack:

\[
\begin{align*}
\text{trans}(\emptyset, M, e) &= M \\
\text{trans}(\bar{x} = \bar{M}), N, e) &= \text{let } \{\bar{x} = \bar{M}\} \text{ in } N \\
\text{trans}(\Gamma, M, x : S) &= \text{trans}(\Gamma, Mx, S) \\
\text{trans}(\Gamma, M, \#x : S) &= \text{trans}(\Gamma[x = M], x, S) \\
\text{trans}(\Gamma, M, \text{alts} : S) &= \text{trans}(\Gamma, \text{case } M \text{ of } \text{alts} : S)
\end{align*}
\]

The following lemma clarifies the relationship:

**Lemma 3.1 (Translation)** For all \( \Gamma, \mathcal{C}, S \), there exists \( k \geq 0 \) such that for any \( M \), \( \langle \emptyset, \text{trans}(\Gamma, \mathcal{C}[M], S), e \rangle \rightarrow^k \langle \Gamma, \mathcal{C}[M], S \rangle \).

**Proof.** Simple induction on the size of \( S \). \( \square \)

### 3.3 Improvement

We define observational approximation and equivalence via contexts in the standard way [AO93].

**Definition 3.2 (Observational Approximation)** We say that \( M \) observationally approximates \( N \), written \( M \sim N \), if for all \( \mathcal{C} \) such that \( \mathcal{C}[M] \) and \( \mathcal{C}[N] \) are closed,

\[\mathcal{C}[M] \Downarrow \Rightarrow \mathcal{C}[N] \Downarrow.\]

We say that \( M \) and \( N \) are observationally equivalent, written \( M \cong N \), when \( M \sim N \) and \( N \sim M \).

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.

**Definition 3.3 (Improvement)** We say that \( M \) is improved by \( N \), written \( M \triangleright N \), if for all \( \mathcal{C} \) such that \( \mathcal{C}[M] \) and \( \mathcal{C}[N] \) are closed,

\[\mathcal{C}[M] \Downarrow^\ast \Rightarrow \mathcal{C}[N] \Downarrow^\ast.\]

We say that \( M \) and \( N \) are cost equivalent, written \( M \triangleright N \), when \( M \triangleright N \) and \( N \triangleright M \).

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 [Mil77] 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 much smaller set of contexts.

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

**Lemma 3.2 (Context Lemma)** For all terms \( M \) and \( N \), if for all \( \Gamma \) and \( S \),

\[\langle \Gamma, M, S \rangle \Downarrow^\ast \Rightarrow \langle \Gamma, N, S \rangle \Downarrow^\ast\]

then \( M \triangleright N \).

It says that we need only consider configuration contexts of the form \( \langle \Gamma, [], S \rangle \) where the hole \( [] \) 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 4.2.

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.

### 3.4 Strong Improvement

The improvement relation, like the notion of operational approximation which it refines, also increases the termination of programs, so if \( M \triangleright N \) 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 3.4 (Strong Improvement) We say that $M$ is strongly improved by $N$, written $M \trianglerighteq N$, if

$$M \trianglerighteq N \land N \trianglelefteq M$$

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

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 $\Omega$, 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:

The diagram is a $\trianglerighteq$-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.

4 The Tick Algebra

Consider the following improvement:

$$(\lambda x. M) y \trianglerighteq M[y/x]$$

Clearly, for any $\Gamma$ and $S$:

$$\langle \Gamma, (\lambda x. M) y, S \rangle \rightarrow \langle \Gamma, \lambda x. M, y : S \rangle \rightarrow \langle \Gamma, M[y/x], S \rangle,$$

so (*) follows from the context lemma. But we can say more: $(\lambda x. M) y$ always takes exactly two more steps to converge than $M[y/x]$. If we had some syntactic way of slowing the right-hand side down, $(*)$ 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 $\check{\cdot}$, which we will use to add a dummy step to a computation. Now we can write $(*)$ as

$$(\lambda x. M) y \trianglerighteq \check{\cdot} M[y/x]$$

We can define $\check{\cdot}$ within the language\footnote{We could introduce a new syntactic construct instead, with the semantics of a dummy step. Since we are working with contextual approximations and equivalences, this would not necessarily be a conservative extension: we would have to prove that it could be represented in the original language. By defining $\check{\cdot}$ in the language, we neatly sidestep such considerations.} as an empty let binding, thus:

$$\check{\cdot} M \overset{\text{def}}{=} \text{let } \{\} \text{ in } M.$$  

Clearly, $\check{\cdot}$ adds one unit to the cost of evaluating $M$ without otherwise changing its behaviour. Note that:

$$M[y/x] \iff \check{\cdot} M[y/x] \iff \check{\cdot} M[y/x]^n$$

We will write $k \check{\cdot} M$ 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.

$$\frac{M \trianglerighteq N}{\check{\cdot} M \trianglerighteq \check{\cdot} N} \quad \frac{\check{\cdot} M \trianglerighteq \check{\cdot} N}{M \trianglerighteq N} \quad (\check{\cdot}-\text{elim})$$

Their validity follows from the definition of $\trianglerighteq$.

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 (\check{\cdot}-elim), the laws presented in figures 2, 3, 4, 5, and figure 6 are known collectively as the tick algebra.

4.1 Beta Laws

The first set of laws, presented in figure 2, are important in that they allow us to mimic evaluation within the algebra. We have already seen $(\beta)$; $(\text{case-}\beta)$ is the analogous law for case expressions. In $(\text{value-}\beta)$, one may replace occurrences of a variable, which is bound to some value $V$, with $x^2 \cdot V$. The ticks reflect the fact that by replacing $x$ with its value, we are short-circuiting one lookup and one update step.

The proofs of validity of $(\text{value-}\beta)$, $(\text{var-}\beta)$, $(\text{var-}\abs)$, and $(\text{var-subst})$ rely upon general techniques that are outlined in the appendix.

There are also two derived beta laws, corresponding to unrestricted versions of $(\beta)$ and $(\text{case-}\beta)$. We can derive the following cost equivalence (modulo tick):

$$(\lambda x. M) N \overset{\text{2\check{\cdot}}}{\equiv} \text{let } \{x = N\} \text{ in } M$$

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

4.2 Laws for Evaluation Contexts

An evaluation context is a context in which the hole is the target of evaluation; in other words, evaluation cannot
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.

\[(\lambda x. M) y \triangleq^\ast \ast M[y/x]\]  
\[(\beta)\]

\[\text{case } c \, \tilde{y} \text{ of } \{ c_i \, \tilde{x}_i \rightarrow M_i \} \triangleq^\ast \ast M_j[y/j] \]  
\[(\text{case-}\beta)\]

let \(x = V, \tilde{y} = \tilde{E}[x]\) in \(\mathbb{C}[x] \triangleq\) let \(x = V, \tilde{y} = \tilde{E}[V]\) in \(\mathbb{C}^\ast V\)  
\[(\text{value-}\beta)\]

let \(x = z, \tilde{y} = \tilde{E}[z]\) in \(\mathbb{C}[x] \triangleq\) let \(x = z, \tilde{y} = \tilde{E}[z]\) in \(\mathbb{C}[z]\)  
\[(\text{var-}\beta)\]

let \(x = z, \tilde{y} = \tilde{E}^{\ast\ast} [z]\) in \(\mathbb{C}^{\ast\ast} z\) \triangleq\) let \(x = z, \tilde{y} = \tilde{E}[z]\) in \(\mathbb{C}[x]\)  
\[(\text{var-abs})\]

let \(x = z, \tilde{y} = \tilde{M}\) in \(N \square\) let \(x = z, \tilde{y} = \tilde{M}[y/x]\) in \(N[y/x]\)  
\[(\text{var-subst})\]

Figure 2: Beta laws for call-by-need.

\[E[\text{case } M \text{ of } \{ \text{pat}_i \rightarrow N_i \}] \triangleq\) case \(M\) of \(\{ \text{pat}_i \rightarrow E[N_i] \}\)  
\[(\text{case-}E)\]

\[E[\text{let } \tilde{x} = M \text{ in } N] \triangleq\) let \(\tilde{x} = M\) in \(E[N]\)  
\[(\text{let-}E)\]

Figure 3: Laws for evaluation contexts.

let \(\tilde{x} = \tilde{L}, \tilde{y} = \tilde{M}\) in \(N \square\) let \(\tilde{y} = \tilde{M}\) in \(N\), \(\tilde{x} \notin \text{FV}(\tilde{M}, N)\)  
\[(\text{gc})\]

let \(\tilde{x} = \tilde{L}\) in \(N \square\) let \(\tilde{y} = \tilde{M}\) in \(N\)  
\[(\text{let-flatten})\]

let \(x = \text{let } \tilde{y} = \tilde{L}, \tilde{z} = \tilde{M} \text{ in } N') \text{ in } N \square\) let \(x = \text{let } \tilde{z} = \tilde{M} \text{ in } N, \tilde{y} = \tilde{L}\) in \(N'\)  
\[(\text{let-let})\]

\(\lambda x . \text{let } \tilde{y} = \tilde{V}, \tilde{z} = \tilde{M} \text{ in } N \square\) let \(\tilde{y} = \tilde{V}\) in \(\lambda x . \text{let } \tilde{z} = \tilde{M}\) in \(N\)  
\[(\text{let-float-val})\]

let \(x = M, \tilde{y} = \tilde{N}\) in \(x \square\) let \(x = M, \tilde{y} = \tilde{N}\) in \(\ast M, \tilde{x} \notin \text{FV}(M, \tilde{N})\)  
\[(\text{inline})\]

let \(\tilde{x} = \tilde{V}\sigma_1, \tilde{y} = \tilde{V}\sigma_2, \tilde{z} = \tilde{M}\) in \(N \square\) let \(\tilde{x} = \tilde{V}\sigma_2\sigma_3, \tilde{z} = \tilde{M}\sigma_3\) in \(N\sigma_3, \sigma_1 = [\tilde{y}/\tilde{a}], \sigma_2 = [\tilde{z}/\tilde{a}], \sigma_3 = [\tilde{y}/\tilde{a}]\)  
\[(\text{value-copy})\]

Figure 4: Laws for dealing with lets.

\(\emptyset \sqsupset M\)  
\[(\Omega)\]

\(M \sqsupset \emptyset, \text{ iff } M \cong \emptyset\)  
\[(\text{imp-}\Omega)\]

\(M \sqsupset \emptyset, \text{ iff } M \cong^\ast M\)  
\[(\text{diverge})\]

let \(x = \emptyset, \tilde{y} = \tilde{E}[x]\) in \(\mathbb{C}[x] \triangleq\) let \(x = \emptyset, \tilde{y} = \tilde{E}[\emptyset]\) in \(\mathbb{C}[\emptyset]\)  
\[(\Omega_{-}\beta)\]

\(\lambda x.\text{let } y = \emptyset, \tilde{z} = \tilde{M} \text{ in } N \sqsupset\) let \(y = \emptyset\) in \(\lambda x.\text{let } \tilde{z} = \tilde{M}\) in \(N\)  
\[(\text{let-float-}\emptyset)\]

\(\mathbb{C}[^\ast M] \sqsupset\) \(\mathbb{C}[M]\), \(\text{ if } C\text{ is strict}\)  
\[(\check{\ast}\text{-float})\]

Figure 5: Laws for \(\Omega\) and strictness.

let \(x = M, \tilde{y} = \tilde{E}[^\ast M]\) in \(\mathbb{C}[^\ast M] \sqsupset\) let \(x = M, \tilde{y} = \tilde{E}[x]\) in \(\mathbb{C}[x]\)  
\[(\beta\text{-expand})\]

Figure 6: Beta expansion conjecture.
proceed until the hole is filled. Evaluation contexts have the following form:

\[
E ::= Λ \mid \text{let } \{ \vec{y} = \vec{M} \} \text{ in } Λ \\
\mid \text{let } \{ \vec{y} = \vec{M}, x_0 = \vec{A}[x_1], x_1 = \vec{A}[x_2], \ldots, x_n = \vec{A}_n \} \text{ in } Λ[x_0]
\]

\[Λ ::= \emptyset \mid Λ x \mid \text{case } Λ \text{ of } \{ c_1, ξ, \Rightarrow M \} \]

E ranges over evaluation contexts, and Λ over what we call applicative contexts. Our evaluation contexts are strictly contained in those mentioned in the letrec extension of Ariola and Felleisen [AF97]: there they allow E to appear anywhere we have an Λ. Our “flattened” definition corresponds exactly to configuration contexts (with a single hole) of the form (Γ, [S], S), as stated by the following lemma, where A_E is the set of all evaluation contexts.

**Lemma 4.1** \(A_E = \{ \text{trans}(\Gamma, [S]) : \text{all } \Gamma, S \} \).

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-E) and (let-E) to expose the sub-term of interest. Their validity follows easily from a simple lemma (given in the appendix).

We can derive a law which allows us to move ticks in and out of evaluation contexts:

\[
E[\vec{M}] \preceq \cdot E[M] \quad (\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\
which concerns moving non-value bindings out of λs
(where the variable convention ensures that x does not
occur free in the L). As noted above, this is an es-
sential part of the full-laziness transformation. An-
other consequence of the conjecture is standard common
subexpression elimination:

\[ \forall M \models \text{let } \{ x = M \} \in C[x] \] (ese)

Again, the convention ensures that any free variables
of M are not captured by context C.

4.6 Constant Factors and the Calculi

We reproduce the axioms of the call-by-need calculus
[AFM95], in figure 7.3.

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 \rightarrow R turns out to be either an outright improve-
ment, i.e. L \succeq R, or, in the case of (let-A), an improve-
ment “modulo tick”:

\[ \forall \text{let } y = (\text{let } x = L \text{ in } M) \in N \models \text{let } x = L \]

in \text{let } y = M \text{ in } N.

This follows by (let-flatten), (let-let), and (\varnothing-elim). The
extra tick is nested in this case is because without it the
right-hand side might (in the case when y is not used)
perform one extra heap-allocation step.

What is more interesting is that in each case we can re-
verse the improvement modulo tick. In other words, there
exists an R’, obtained from R by inserting ticks, such that
R’ \succeq 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 obtain-
able in these calculi is linear.

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

Definition 4.1 (Imp. within a Constant Factor) We say
that M is improved by N within a constant factor, writ-
en M \succeq N, if there exists a k such that for all C such
that C[M] and C[N] are closed,

\[ C[M]|^{k} \Rightarrow C[N]|^{k} \in k^{(n+1)} \] .

So M \succeq 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 \succeq is a precongruence relation (to
show transitivity requires a small calculation) and clearly
contains the improvement relation.

Now we consider a special case of \succeq, namely programs
which only differ by ticks. Let M \rightarrow N mean that N
can be obtained from M by removing some ticks (from
anywhere within the term), and M \bowtie N mean that there
exists an L such that M \rightarrow L and N \rightarrow L. Clearly \rightarrow is
a precongruence and \bowtie a congruence.

Lemma 4.2 M \bowtie N \implies M \succeq N.

Proof. (Sketch) Clearly \rightarrow \subseteq \bowtie, so it suffices to show
that M \rightarrow N \implies N \succeq M. First show that the nesting
of ticks in a configuration never increases as computa-
tion 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 C[N]|^{k} implies C[N]|^{k(n+1)} (strengthen-
ing this statement to configurations).

With this lemma we can establish the following:

Theorem 4.3 For all terms N and M (of our restricted
syntax) if M \equiv N then M \succeq N.

Proof. (Sketch) By induction on the proof of M \equiv N.
The base case requires us to show that the (oriented)
equations are contained in \succeq. This follows easily since
they are all either improvements or improvements mod-
ulo tick. In the inductive cases, the congruence and tran-
sitivity rules follow from the inductive hypothesis since
\succeq is a precongruence. The only difficult case is symmetry.
It will be sufficient to prove that reversed equations are
contained in \succeq. For each equation L \equiv R we have
from the laws an R’ such that R’ \rightarrow R and R’ \succeq L. By
lemma 4.2 we know that R \succeq R’, so R \succeq L follows from
the fact that \succeq \subseteq \bowtie and transitivity of \succeq.

Corollary 4.4 The call-by-need calculus of [AFM95] 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 ASM [AB97] 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 ex-
amples, and it was encouraging to find that we had al-
ready 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.4 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.

5 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

4Thanks 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.
denotational semantics. Examples of this kind of operational analogue to Scott induction for other languages may be found in e.g., [Pit97b, Smi91, MST96, San97, Las98]; 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 fi are distinct, new variables.

**Definition 5.1**

\[
\begin{align*}
\text{let } f_0 & = \Omega, \\
\text{let } f_{n+1} & = f V, \\
\text{let } f & = V f_{n+1}
\end{align*}
\]

Then, for an f defined by let \( f V \) in f, we define the nth unwinding as let \( f V \) in fn. If we expand the definition of f V, we see that this is really

\[
\text{let } \{ f_0 = \Omega, f_1 = V f_0 / f, \ldots, f_n = V f_{n-1} / f \}
\]

in fn.

Note that we have restricted our attention to 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 let \( f V \) in fn completely characterise the behavior of let \( f V \) in f. This is the essence of Scott induction. The main property that justifies this is a syntactic notion of continuity, which says that let \( f V \) in f is the least upper bound of chain \{ let \( f V \) in fn \}n≥0 and that any M which uses f preserves this property.

We first show that \{ let \( f V \) in M[f/f] \}n≥0 does indeed form a chain with respect to ≼, and that let \( f V \) in M is an upper bound of that chain.

**Lemma 5.1** \( \forall n. \text{let } \{ f V \} \text{ in } M[f/f] \)

\[
\begin{align*}
& \supseteq \text{let } \{ f V \} \text{ in } M[f_{n+1}/f] \\
& \supseteq \text{let } f V \text{ in } M.
\end{align*}
\]

**Proof.** We prove only the second improvement, that for all n, let \( f V \) in M[f/f] \( \supseteq \) let \( f V \) in M. The first follows by a similar argument. We proceed by induction on n. The base case follows easily by (gc) and the \( \Omega \) laws, and the inductive case follows by (\( \forall \)-elim), since

\[
\text{let } f V \text{ in } M[f/f] \supseteq \text{let } f V \text{ in } M
\]

by the derivation in figure 8.

To establish syntactic continuity, we will need the following lemma (proof given in the appendix). It says that if let \( f V \) in M converges then there must exist some unwinding that does so with the same cost.

**Lemma 5.2 (Unwinding)**

For all \( \Gamma, S, \) and n,

\[
\langle \Gamma, \text{let } f V \text{ in } M, S \rangle \psi^0 \implies \\
3m. \langle \Gamma, \text{let } f V \text{ in } M[f_m/f], S \rangle \psi^0.
\]

**Theorem 5.3 (Syntactic Continuity)**

The following is a sound proof rule:

\[
\forall n. \text{let } f V \text{ in } M[f/f] \supseteq N \\
\implies \text{let } f V \text{ in } M \supseteq N
\]

**Proof.** Assume \( \langle \Gamma, \text{let } f V \text{ in } M, S \rangle \psi^0 \). Then by the Unwinding lemma, there exists some m such that \( \langle \Gamma, \text{let } f V \text{ in } M[f_m/f], S \rangle \psi^0 \). By the premise, we have that \( \langle \Gamma, N, S \rangle \psi^0 \), 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 \( \Omega \). 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 5.4**

If (\( \beta \)-expand) is valid, then

\[
\text{let } \text{rec} = (\lambda f. \text{let } x = \text{rec } f \text{ in } f x) \text{ in } \text{rec} \\
\text{let } \text{fix} = (\lambda f. \text{let } x = f x \text{ in } x) \text{ in } \text{fix}.
\]

**Proof.** Let \( V = \lambda f. \text{let } x = \text{rec } f \text{ in } f x \text{ in } f x \text{ in } x \). Then the result will then follow by syntactic continuity, since

\[3 \text{ let } \text{rec } f \text{ in } x \]

\[\supseteq \text{let } \text{fix } = (\lambda f. \text{let } x = f x \text{ in } x) \text{ in } \text{fix} \quad \text{(gc)}
\]

\[\text{let } \text{fix } = (\lambda f. \text{let } x = f x \text{ in } x) \text{ in } \text{fix} \quad \text{(value-}\beta)\]

We proceed via induction on n. The base case follows trivially by (\( \text{imp-} \Omega \)) since let \( \text{rec}_0 = \Omega \) in \( \text{rec}_0 \supseteq \Omega \), and the inductive case follows by the derivation in figure 9. We have \( \supseteq \) and not \( \supseteq \) because we use a slightly slower version of \( \text{rec} \).
\( \vdash \{ f \equiv V, f_{n+1} = V[f_{n+1}/f] \} \) in \( M[f_{n+1}/f] \)

\( \vdash \{ f \equiv V \} \) in \( \{ f_{n+1} = V[f_{n+1}/f] \} \) in \( M[f_{n+1}/f] \) (let-let)

\( \equiv \) \( \{ f \equiv V \} \) in \( \{ g = V[f_{n+1}/f] \} \) in \( M[g/f] \) (rename)

\( \vdash \{ f = V \} \) in \( \{ g = V[f_{n+1}/f] \} \) in \( M[g/f] \) (I.H.)

\( \vdash \{ f = V, g = V \} \) in \( M[g/f] \) (let-let)

\( \vdash \{ f = V \} \) in \( M \) (value-copy), (gc)

Figure 8: The inductive case for lemma 5.1.

\( \vdash \{ \text{rec} \equiv V, \text{rec}_{n+1} = V_n \in \text{rec}_{n+1} \) (value-\( \beta \))

\( \vdash \{ \text{rec} \equiv V \} \) in \( \{ \text{rec}_n \equiv f \} \) in \( \{ \text{rec}_n \equiv f \} \) (gc), (defn. of \( V_n \))

\( \vdash \{ \text{rec}_n \equiv f \} \) in \( \{ \text{rec}_n \equiv f \} \) (let-float-val), (let-float-\( \Omega \))

\( \vdash \{ \text{rec}_n \equiv f \} \) in \( \{ \text{rec}_n \equiv f \} \) (let-let), (let-E)

\( \vdash \{ \text{rec}_n \equiv f \} \) in \( \{ \text{rec}_n \equiv f \} \) (I.H.), (rename)

\( \vdash \{ \text{rec}_n \equiv f \} \) in \( \{ \text{rec}_n \equiv f \} \) (\( \beta \))

\( \vdash \{ \text{rec}_n \equiv f \} \) in \( \{ \text{rec}_n \equiv f \} \) (let-let)

\( \vdash \{ \text{rec}_n \equiv f \} \) in \( \{ \text{rec}_n \equiv f \} \) (\( \checkmark \)-elim), (var-subst)

\( \vdash \{ \text{rec}_n \equiv f \} \) in \( \{ \text{rec}_n \equiv f \} \) (gc), (rename)

\( \vdash \{ \text{rec}_n \equiv f \} \) in \( \{ \text{rec}_n \equiv f \} \) (\( \beta \)-expand)

Figure 9: The inductive case for proposition 5.4.

The converse of the proposition is false, since the knotting 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 [MFP91]. In the statement, \( \forall \) and \( \exists \) range over value contexts.

**Theorem 5.5 (Improvement Fusion)** If \( C \) is strict, and \( C[V[x]] \implies W[C[x]] \) where \( x \notin \text{fv} (V, W, C) \cup \text{cv} (V, W, C) \), then for all \( D \) such that \( x \notin \text{fv} (D) \cup \text{cv} (D) \),

\( \vdash \{ x = \forall [x] \} \) in \( D[C[x]] \implies \{ x = \forall [x] \} \) in \( D[x] \).

**Proof.** Assume \( C \) is strict, and that \( C[V[x]] \implies W[C[x]] \). By syntactic continuity, it suffices to show, for all \( n \) and all \( D \) such that \( x \notin \text{fv} (D) \cup \text{cv} (D) \),

\( \{ x = \forall [x] \} \) in \( D[C[x]] \implies \{ x = \forall [x] \} \) in \( D[x] \).

The base case follows by this calculation:

\( \vdash \{ x_0 = \text{\( \Omega \)} \} \) in \( D[C[x_0]] \)

\( \vdash \{ x_0 = \text{\( \Omega \)} \} \) in \( D[C[\text{\( \Omega \)}]] \) (\( \Omega \)-\( \beta \))

\( \vdash \{ x_0 = \text{\( \Omega \)} \} \) in \( D[\text{\( \Omega \)}] \) (C strict)

\( \vdash \{ x = \forall [x] \} \) in \( D[\text{\( \Omega \)}] \) (gc)

\( \vdash \{ x = \forall [x] \} \) in \( D[x] \) (gc)

\( \vdash \{ x = \forall [x] \} \) in \( D[x] \) (\( \Omega \)-\( \beta \)), (cong.)

and for the inductive case:

\( \{ x = x_{n+1} \} \) in \( D[C[x_{n+1}]] \)

\( \vdash \{ x = \forall [x] \} \) in \( D[C[\forall [x]]] \) (value-\( \beta \)), (gc)

\( \vdash \{ x = \forall [x] \} \) in \( D[\forall [C[x]]] \) (C strict)

\( \vdash \{ x = \forall [x] \} \) in \( D[C[x]] \) (assumption)

\( \vdash \{ x = \forall [x] \} \) in \( D[C[x]] \) (I.H.)

\( \vdash \{ x = \forall [x] \} \) in \( D[x] \) (value-\( \beta \))

Fixed-point fusion can be used to establish a number of general fusion laws. It is also central to Tullsen and Hudak’s [TH98] approach to program transformation in Haskell.

6 The Improvement Theorem

In this section we introduce a second key technique for reasoning about recursion, the improvement theorem. In [San96] 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.
6.1 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:

\[ \text{repeat } x = x : (\text{repeat } x) \]

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

\[ \text{repeat } x = x : (\text{tail } (\text{repeat } x)) \]

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

\[ M \cong N \implies \text{let } \{ x = M \} \text{ in } L \cong \text{let } \{ x = N \} \text{ in } L \]

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

\[ \text{let } \{ x = M \} \text{ in } M \cong \text{let } \{ x = M \} \text{ in } N \]

\[ \implies \text{let } \{ x = M \} \text{ in } L \cong \text{let } \{ x = N \} \text{ in } L . \]

But in order to reason about "interesting" program transformations (e.g. unfold-fold, recursion-based deforestation, partial evaluation with memoization), inference (6.1) is simply not sufficient.

The improvement theorem comes to the rescue:

\[ \text{let } \{ x = M \} \text{ in } M \not\cong \text{let } \{ x = M \} \text{ in } N \]

\[ \implies \text{let } \{ x = M \} \text{ in } L \not\cong \text{let } \{ x = N \} \text{ in } L . \]  

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

\[ \text{let } \{ f = V \} \text{ in } V \not\cong \text{let } \{ f = V \} \text{ in } W \]

\[ \text{let } \{ f = V \} \text{ in } N \not\cong \text{let } \{ f = W \} \text{ in } N \]

The inference is also sound when \( \not\cong \) is replaced throughout with \( \not\cong \) (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 appendix.

**Notation** In establishing a premise of the improvement theorem, in the context of some recursive declarations \( \tilde{g} = \tilde{V} \), a derivation of the form

\[ \text{let } \{ \tilde{g} = \tilde{V} \} \text{ in } M_1 \not\cong \text{let } \{ \tilde{g} = \tilde{V} \} \text{ in } M_2 \]

\[ \not\cong \text{let } \{ \tilde{g} = \tilde{V} \} \text{ in } M_3 \ldots \]

will be written in the following abbreviated form:

\[ \tilde{g} \vdash M_i \not\cong \text{let } \{ \tilde{g} = \tilde{V} \} \text{ in } M_3 \ldots \]

when the declarations \( \tilde{g} \) are clear from the context.

The following example illustrates the use of the proof rule, which shows that a representation of the standard lambda-calculus fixed-point combinator \( Y = \lambda f. (\lambda x. f (x x)) \lambda x. f (x x) \) (suitably converted to the restricted syntax) is cost equivalent to the non-cyclic version rec from proposition 3.4.

**Proposition 6.2**

\[ \text{let } Y = \lambda f. \text{let } d = \lambda y. \text{let } z = y y \text{ in } f z \]

\[ \quad x = d d \text{ in } f x \]

\[ \text{in } Y \]

\[ \not\cong \text{let } \text{rec } = (\lambda f. \text{let } x = \text{rec } f \text{ in } f x) \text{ in } \text{rec}. \]

**Proof.** By a simple 6-step derivation, we have that

\[ Y \vdash \lambda f. \text{let } d = \lambda y. \text{let } z = y y \text{ in } f z \]

\[ x = d d \text{ in } f x \]

\[ \not\cong \lambda f. \text{let } x = Y f \text{ in } f x \]

Then the result follows by the Cost Equivalence Theorem. \( \square \)

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

\[ \text{let } \{ f = V \} \text{ in } N \not\cong \text{let } \{ f = W \} \text{ in } N \]

If the left-hand side is non-recursive (in \( f \)) then syntactic continuity is of no help, since the unwindings \( \not\cong 0 \) 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.

6.2 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 [San97], where it was established for any call-by-name or call-by-value language with SOS rules fitting a certain syntactic rule-format.
Theorem 6.3 (Improvement Induction) For any $M$, $N$, $C$, and substitution $\sigma$, the following proof rule is sound:

$$
M \overset{\tau}{\rightarrow} C[\sigma] \quad N \overset{\tau}{\rightarrow} C[\sigma] \\
M \overset{\tau}{\rightarrow} N
$$

The proof is quite straightforward, and is given in the appendix.

Let us take a standard example to illustrate the proof technique: the associativity of append (the list-concatenation function). In order to show that it is an improvement, we need to insert a tick into the recursive branch; this is a consequence of the fact that our cost-measure is rather fine-grained. To ease the notation, we will make use of the syntactic identity for general application from section 3, and we will use an infix form of append.

Given the recursive declaration

$$
(\rightarrow) = \lambda x.s.\lambda y.s.\text{case } x \text{ of } \\
\quad \text{nil} \to y s \\
\quad h : t \to h \cdot \tau(x \rightarrow y) s
$$

then $(\rightarrow) \vdash (x \rightarrow y) \rightarrow z \geq x \rightarrow (y \rightarrow z)$. To show this, by induction on the right-hand side it is sufficient to find a context $C$ and substitution $\sigma$ such that

$$
(\rightarrow) \vdash (x \rightarrow y) \rightarrow z \geq C[(x \rightarrow y) \rightarrow z] \sigma, \\
(\rightarrow) \vdash x \rightarrow (y \rightarrow z) \geq C[(x \rightarrow (y \rightarrow z)] \sigma.
$$

The solution is to take $C$ to be

$$
\begin{align*}
\text{case } x \text{ of } \\
\quad \text{nil} \to \tau(y \rightarrow z) \\
\quad h : t \to \text{let } r \equiv [t] \ \text{in } h : r
\end{align*}
$$

and $\sigma = [h.x]$. This context is easily derived by performing a cost equivalence calculation on the right-hand side until a recurring instance of $x \rightarrow (y \rightarrow z)$ is discovered. We omit the derivations.

By using suitably slowed versions of append on the right-hand side, we can also show that this is only a linear speedup in all contexts, i.e. that

$$
(\rightarrow) \vdash x \rightarrow (y \rightarrow z) \geq x \rightarrow (y \rightarrow z).
$$

Using the above properties we have been able to prove, with the help of the improvement theorem and an adaptation of the argument given in [San96], that a mechanisable append-elimination transformation described in [Wad88] can never slow-down programs by more than a constant factor. What is interesting about this example is that it shows that the improvement theorem can obtain asymptotic speedups using only linear ones, since in particular the transformation is able to turn the naïve quadratic-time definition of reverse into a linear-time version.

7 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. [AFM+95]. 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.

Our unit of cost is the abstract machine reduction step. This choice simplifies the technical development. A drawback is that it is very fine-grained, so one must carefully track costs of optimisation steps. This bookkeeping can, in larger examples, become rather tedious. However, it should be possible to follow our programme with a much more abstract cost measure. For example, one possibility would be to count only the number of lookups\textsuperscript{8}. This should significantly simplify the tick algebra, without compromising the ability to characterise relative efficiency within a constant factor.

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 [San95, San98b].

Another direction for future work would be to consider the time-safety of a larger-scale program transformation, such as deforestation [Wad90]. 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

$$
\text{let } \{z = \bar{M} \} \text{ in } C[z] \overset{\tau}{\rightarrow} \text{let } \{z = \bar{M} \} \text{ in } C[\bar{M}],
$$

but in the case where holes occur under $\lambda$-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 [TWM95] provides such global information, so it would be interesting to prove that their system (and generalisations to full recursive lets [Gus88]) does indeed satisfy the desired improvement property above. We saw in section 4.4 that the strictness property of a context can be characterised exactly by

$$
C[\tau x] \overset{\tau}{\rightarrow} C[x],
$$

where $x$ is fresh. Could it be the case that the “used at most once” property might be semantically characterised by $C[x] \overset{\tau}{\rightarrow} C[\tau x]$?

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 and the referees for their helpful comments.

\textsuperscript{8}Of course, one would need to change the definition of $\tau$ accordingly.


A Proofs of Main Theorems

This appendix 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 [MT91, AMST97, Ta98]. In order to make this style of proof rigorous 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 [Ta98]. We use an alternative approach to generalising contexts which is due to Pitts [Pit94].

A.1 Substituting Contexts

Following Pitts [Pit94], we use second-order syntax to represent (and generalise) the traditional definition of contexts given in section 3.2. We give a fuller description in [San98a]; other examples of their use are to be found in [Las98, Mor98]. The idea is that instead of holes [ ] we use second-order variables, ranged over by \( \xi \), applied to some vector of variables. The syntax of generalised contexts is:

\[
\begin{align*}
C, D := & \quad \xi \cdot \vec{x} \\
| & \quad \lambda \cdot C \mid C \mid x \mid \text{let } \{ \vec{x} = \vec{D} \} \text{ in } C \\
| & \quad \text{case } C \text{ of } \{ c_i : \vec{z}_i \rightarrow D_i \}. 
\end{align*}
\]

\( \forall \text{ and } \exists \) will range over value contexts, \( \Gamma \) and \( \Delta \) over heap contexts, and \( S \) and \( T \) over stack contexts. Each “hole variable” \( \xi \) 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 \( \xi \). 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 \( \vec{z} \) are considered bound in \( (\vec{x})M \)). Coupled with the meta-abstraction is of course meta-application, written \( \xi : \vec{x} \). We restrict application of \( \xi \) 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 \( C \) by \( (\vec{x})C \) where \( \vec{x} \) is a vector of the capture-variables of \( C \); filling \( C \) with a term \( M \) is then represented by \( (C[\vec{x}] : \vec{z})((\vec{x})M)_\vec{z} \).

Example The traditional context \( \text{let } x = [ ] \) in \( \lambda y. \xi (x, y) \) can be represented by \( \text{let } x = \xi (x, y) \) in \( \lambda y. \xi (x, y) \). Filling the hole with the term \( xy \) is represented by:

\[
\begin{align*}
\text{let } x & = \xi (x, y) \text{ in } \lambda y. \xi (x, y) \, [x \, y]_\vec{x} \\
\equiv & \text{let } z = (x, y) \, xy \, (z, y) \text{ in } \lambda w. (x, y) \, xy \, (x, w) \\
\equiv & \text{let } z = z \, y \text{ in } \lambda w. (x, w)
\end{align*}
\]

which is \( \alpha \)-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 \( \alpha \)-conversion.

Henceforth we work only with generalised contexts. We will write \( C[(\vec{x})M] \) to mean \( C[(\vec{x})M]_\vec{z} \) when \( C \) contains just a single hole variable \( \xi \). 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 appendix.

A.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 [San98a] 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 \( \theta \) will range over substitutions composed of variable for variable substitutions and substitutions of the form \( [(\vec{x})M]_\vec{z} \).

We have the following key property.

**Lemma A.1 (Extension)** If \( \langle \Gamma, C, S \rangle \rightarrow^h \langle \Delta, D, T \rangle \) then

(i) for all \( \Gamma' \) and \( S' \) such that \( \langle \Gamma', \Gamma, C, S S' \rangle \) is well-formed, \( \langle \Gamma', \Gamma, C, S S' \rangle \rightarrow^h \langle \Gamma' \Delta, D, T S' \rangle \).

(ii) for all \( \theta \), \( \langle \Gamma, C, S \rangle \theta \rightarrow^h \langle \Delta, D, T \rangle \theta \)

**Proof.** (i) follows by inspection of open reductions over configuration contexts. (ii) amounts to the standard substitution lemma; see [San98a] for a general argument. \( \square \)

**Definition A.1** \( \langle \Gamma, C, S \rangle \theta_{\text{univ}} \equiv \exists \theta. \langle \Gamma, C, S \rangle \theta_{\text{uns}} \), where \( \theta \) is a closing substitution.

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 A.2 (Open Uniform Computation)** Given some \( \Gamma, C, S \), if there exist \( \Gamma' \) and \( S' \) such that \( \langle \Gamma', \Gamma, C, S S' \rangle \) is well-formed, and \( \langle \Gamma', \Gamma, C, S S' \rangle \theta_{\text{univ}} \) then \( \langle \Gamma, C, S \rangle \) reduces to a configuration context of one of the following forms:

**Proof.** (i) follows by inspection of open reductions over configuration contexts. (ii) amounts to the standard substitution lemma; see [San98a] for a general argument. \( \square \)
(i) \( \langle A, V, e \rangle \),
(ii) \( \langle A, \xi \cdot \bar{g}, \top \rangle \), for some hole \( \xi \), or
(iii) \( \langle A, x, \top \rangle \), \( x \in \text{dom } \Gamma' \).

\textbf{Proof.} Assume there exists some \( \theta \) such that \( \langle \Gamma', C, S^- \theta \rangle \psi \). We consider the reduction of \( \langle \Gamma', C, S \rangle \) and proceed by induction on \( n \) with cases on the structure of \( C \). We show three illustrative cases only. The others are similar.

\( C \equiv x \cdot \bar{g} \). This is a type (ii) context, so we are done.

\( C \equiv x \). Since we have termination, \( x \) must be bound in either \( \Gamma \) or \( \Gamma' \). In the former case, \( \Gamma \equiv A \{ x = D \} \). By \textbf{(Lookup)}, \( \langle \Delta(x = x), x, S \rangle \) reduces to \( \langle \Delta, D, \#x : S \rangle \), and \( \langle \Delta \Gamma', D, \#x : S^- \theta \psi_{\xi - k} \rangle \), by extension. By the inductive hypothesis, we know that \( \langle \Delta, D, \#x : S \rangle \) reduces to a configuration context of type (i), (ii), or (iii), and therefore \( \langle \Delta(x = D), x, S \rangle \) does as well, and we are done.

\( C \equiv V \). There are four sub-cases, depending upon the structure of \( S \): we consider only the case when \( S \equiv x : \top \).

Since we have termination, \( V \equiv \lambda y.D \), and by \textbf{(Subst)}, \( \langle \Gamma, \lambda y.D, x : \top \rangle \) reduces to \( \langle \Gamma, D[y/x], \top \rangle \), and \( \langle \Gamma \Gamma', D[y/x], S \Gamma' \psi_{\xi - k} \rangle \). The inductive hypothesis applies, and the result follows as above. \( \square \)

In what follows we will use \( \Sigma \) range over configuration contexts \( \langle \Gamma, C, S \rangle \).

\textbf{A.3 Translation}

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

\textbf{Lemma A.3 (Translation)} For all \( D, \Gamma, C, S \) such that \( D \equiv \text{trans}(\Gamma, C, S) \), there exists \( k \geq 0 \) such that \( \langle \emptyset, D, e \rangle \rightarrow^k \langle \Gamma, C, S \rangle \).

\textbf{Proof.} Simple induction on \( S \). \( \square \)

\textbf{A.4 Proof: the Context Lemma}

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

\textbf{Lemma A.4} \( M \triangleright N \) if and only if for all \( \Sigma \) and \( n \), \( \Sigma[[x]M] \psi^n \implies \Sigma[[x]N] \psi^n \).

\textbf{Proof.} (Sketch) \((\triangleright)\). Trivial; let \( \Sigma = \{ \emptyset, C, e \} \).

\((\Rightarrow)\). By a simple induction on \( n \), using translation. \( \square \)

Note that the proof of the next lemma does not rely upon any specific cost-measure.

\textbf{Lemma A.5} If for all \( \Gamma, \Sigma, \) and \( n \)
\[ \langle \Gamma, [x]M \cdot \bar{g}, S \rangle \psi^n \implies \langle \Gamma, [x]N \cdot \bar{g}, S \rangle \psi_{\xi - k}^n \]
then for all \( \Sigma \) and \( n \), \( \Sigma[[x]M] \psi^n \implies \Sigma[[x]N] \psi_{\xi - k}^n \), where \( \xi \geq \overline{V} (M, N) \).

\textbf{Proof.} Assume the premise and suppose \( \Sigma[[x]M] \psi^n \).

We prove by induction on \( n \).

By open uniform computation, \( \Sigma \) reduces in \( k \geq 0 \) steps to one of:

\( (1) \langle \Delta, V, e \rangle \), \( (2) \langle \Delta, \xi \cdot \bar{g}, S' \rangle \).

(There are only two possibilities since \( \langle \Gamma, C, S \rangle \) is closed.) In case (1), we are done. In case (2), we have
\[ \Sigma[[x]N] \rightarrow^k \langle \Delta[[x]N], N[[x]S'], S'[[x]N] \rangle \] (A.1)

By open uniform computation, \( \langle \Delta, M[[x]g], S' \rangle \) reduces in \( k' \geq 0 \) steps to one of:

\( (2.1) \langle \Delta', \xi', e \rangle \), \( (2.2) \langle \Delta', \xi', \xi', \top \rangle \).

(Again, there are only two possibilities since \( \langle \Delta, M[[x]g], S' \rangle \) is closed.) In case (2.1), we have
\[ \langle \Delta[[x]N], M[[x]g], S'[[x]N] \rangle \psi_{\xi - k}^n \] (ass.)
\[ \Sigma[[x]N] \psi_{\xi - k}^n \] (A.1)

as required. In case (2.2), we know that \( k' > 0 \), since \( M[[x]g] \neq \xi \cdot \xi \).

We have
\[ \langle \Delta[[x]N], M[[x]g], S'[[x]N] \rangle \rightarrow^{k'} \langle \Delta'[[[x]M], [x]M : \xi, \top[[x]M] \rangle \] (A.2)
and
\[ \langle \Delta[[x]N], [x]M : \bar{g}, S'[[x]N] \rangle \rightarrow^{k'} \langle \Delta'[[[x]N], [x]N : \bar{g}, \top[[x]N] \rangle \] (A.3)

Therefore
\[ \langle \Delta[[x]M], M[[x]g], \top[[x]M] \psi_{\xi - k}^n \] (A.2)
\[ \rightarrow \langle \Delta'[[[x]N], M[[x]g], \top[[x]N] \psi_{\xi - k}^{n - k'} \] (I.H.)
\[ \rightarrow \langle \Delta'[[[x]N], N[[x]S'], \top[[x]N] \psi_{\xi - k}^{n - k'} \] (ass.)
\[ \rightarrow \langle \Delta[[x]N], M[[x]g], S'[[x]N] \psi_{\xi - k}^{n - k} \] (A.3)
\[ \rightarrow \langle \Delta[[x]N], N[[x]S'], S'[[x]N] \psi_{\xi - k}^{n - k} \] (ass.)
\[ \rightarrow \Sigma[[x]N] \psi_{\xi - k}^n \] (A.1)

as required. \( \square \)

The generalised statement of the context lemma is:

\textbf{For all terms M and N, if}

\[ \forall V, S, \sigma, n, \langle \Gamma, M, \sigma, S \rangle \psi^n \implies \langle \Gamma, N, \sigma, S \rangle \psi_{\xi}^n \]
\textbf{then M \triangleright N.}

This follows from lemmas A.4 and A.5, and the fact that \( M \triangleright N \).

\textbf{A.5 Validating the Tick Algebra}

We present proofs of the validity of (value-\( \beta \)) and (value-copy), and sketch a proof of the correspondence between evaluation contexts and configuration contexts of the form \( \langle \Gamma, [x] \rangle, S \). The proofs of the more complex laws (e.g. (var-\( \beta \)), (var-abs), (var-subst), and (in-out-float)) have a similar structure to that for (value-\( \beta \)), except they require more use of open uniform computation.
A.5.1 Proof: (value-\beta)

Recall (value-\beta):

\[
\text{let } \{ x = V, \tilde{y} = \tilde{V}[x] \} \text{ in } C[x] \\
\implies \text{let } \{ x = V, \tilde{y} = \tilde{V}[x] \} \text{ in } C'[x].
\]

Let \( W \equiv x'V \) throughout. It suffices to show

\[
\forall \Gamma, S. (\Gamma[x]\{ x = V \}, C[x], S[x], \psi^n) \iff (\Gamma[W]\{ x = V \}, C[W], S[W], \psi^n)
\]

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

Suppose \( (\Gamma[x]\{ x = V \}, C[x], S[x], \psi^n) \). We proceed by induction on \( n \). By open uniform computation, \( (\Gamma, C, S) \) reduces in \( k \geq 0 \) steps to one of

1. \( (\Delta, V, \epsilon) \), 2. \( (\Delta, \{.]\}, \Upsilon) \), 3. \( (\Delta, x, \Upsilon) \).

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

\[
(\Gamma[x]\{ x = V \}, C[x], S[x], \psi^n) \rightarrow^k (\Delta[x]\{ x = V \}, x, \Upsilon[x]) \\
\rightarrow^2 (\Delta[x]\{ x = V \}, V, \Upsilon[x]).
\]

and by extension and the definition of \( W \),

\[
(\Gamma[W]\{ x = V \}, C[W], S[W]) \\
\rightarrow^k (\Delta[W]\{ x = V \}, W, \Upsilon[W]) \\
\rightarrow^2 (\Delta[W]\{ x = V \}, V, \Upsilon[W]).
\]

Since \( (\Delta[x]\{ x = V \}, V, \Upsilon[x], \psi^{n-(k+2)}) \), by the inductive hypothesis we have \( (\Delta[W]\{ x = V \}, V, \Upsilon[W], \psi^{n-(k+2)}) \), and the result follows.

In case (3), we have \( (\Delta[x]\{ x = V \}, V, \Upsilon[x], \psi^{n-k-2}) \), as above. Furthermore, by extension, (Lookup) and (Update), we have

\[
(\Gamma[W]\{ x = V \}, C[W], S[W]) \\
\rightarrow^k (\Delta[W]\{ x = V \}, x, \Upsilon[W]) \\
\rightarrow^2 (\Delta[W]\{ x = V \}, V, \Upsilon[W]).
\]

From the inductive hypothesis, we have \( (\Delta[W]\{ x = V \}, V, \Upsilon[W], \psi^{n-k-3}) \), and the result follows.

A.5.2 Proof: (value-copy)

Recall (value-copy):

\[
\text{let } \{ \tilde{x} = \tilde{V}[\tilde{y}], \tilde{y} = \tilde{V}[\tilde{x}] \} \text{ in } N \\
\implies \text{let } \{ \tilde{x} = \tilde{V}[\tilde{y}], \tilde{z} = \tilde{M}[\tilde{y}] \} \text{ in } N[\tilde{y}], \\
\]

where \( \sigma = [\tilde{z}]_{\tilde{y}} \) and \( \theta = \tilde{V}[\tilde{x}] \).

It suffices to show that for all \( \Gamma, S, \) and \( n \),

\[
(\Gamma[\tilde{x} = \tilde{V}[\tilde{y}], \tilde{y} = \tilde{V}[\tilde{x}], N, S], \psi^n) \\
\iff (\Gamma[\tilde{x} = \tilde{V}[\tilde{y}], N[\tilde{y}], S], \psi^n).
\]

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.

Consider the (holeless) open configuration context \( (\Gamma, N, S) \), in which the \( \bar{x} \) and \( \bar{y} \) may appear free. By open uniform computation, this reduces in \( k \geq 0 \) steps to one of:

1. \( (\Delta, W, \epsilon) \), 2. \( (\Delta, x_i, T) \), 3. \( (\Delta, y_i, T) \).

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

\[
(\Gamma[\tilde{x} = \tilde{V}[\tilde{y}], \tilde{y} = \tilde{V}[\tilde{x}], N, S]) \\
\rightarrow^k (\Delta[\tilde{x} = \tilde{V}[\tilde{y}], \tilde{y} = \tilde{V}[\tilde{x}], x_i, T) \\
\rightarrow^2 (\Delta[\tilde{x} = \tilde{V}[\tilde{y}], \tilde{y} = \tilde{V}[\tilde{x}], V[\tilde{y}], T)
\]

and furthermore,

\[
(\Delta[\tilde{x} = \tilde{V}[\tilde{y}], \tilde{y} = \tilde{V}[\tilde{x}], V[\tilde{y}], T] \psi^{n-(k+2)}).
\]

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

\[
(\Gamma[\tilde{x} = \tilde{V}[\tilde{y}], N[\tilde{y}], S]) \\
\rightarrow^k (\Delta[\tilde{x} = \tilde{V}[\tilde{y}], x_i, T) \\
\rightarrow^2 (\Delta[\tilde{x} = \tilde{V}[\tilde{y}], \tilde{V}[\tilde{y}], \tilde{V}[\tilde{y}], T)
\]

By elementary properties of substitution,

\[
V_1[\tilde{y}, \tilde{y}][\tilde{y}]_y \equiv V_1[\tilde{y}, \tilde{y}][\tilde{y}]_y
\]

so the inductive hypothesis applies (with \( N \equiv V_1[\tilde{y}, \tilde{y}]_y \)), yielding the desired result.

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

\[
(\Gamma[\tilde{x} = \tilde{V}[\tilde{y}], \tilde{y} = \tilde{V}[\tilde{x}], N, S]) \\
\rightarrow^k (\Delta[\tilde{x} = \tilde{V}[\tilde{y}], \tilde{y} = \tilde{V}[\tilde{x}], y_i, T) \\
\rightarrow^2 (\Delta[\tilde{x} = \tilde{V}[\tilde{y}], \tilde{y} = \tilde{V}[\tilde{x}], V[\tilde{y}], T)
\]

and furthermore,

\[
(\Delta[\tilde{x} = \tilde{V}[\tilde{y}], \tilde{y} = \tilde{V}[\tilde{x}], V[\tilde{y}], T] \psi^{n-(k+2)}).
\]

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

\[
(\Gamma[\tilde{x} = \tilde{V}[\tilde{y}], N[\tilde{y}], S]) \\
\rightarrow^k (\Delta[\tilde{x} = \tilde{V}[\tilde{y}], y_i, T) \\
\equiv (\Delta[\tilde{x} = \tilde{V}[\tilde{y}], x_i, T) \\
\rightarrow^2 (\Delta[\tilde{x} = \tilde{V}[\tilde{y}], V[\tilde{y}], T).
\]

The inductive hypothesis applies (with \( N \equiv V_1[\tilde{y}]_y \)), yielding the desired result.
A.5.3 Proof Sketch: Lemma 4.1

Recall the statement of lemma 4.1: \( \Lambda_E \) is equal to \( \{ \text{trans}(\Gamma, [, ] , S) \mid \text{all } \Gamma \text{ and } S \} \). So we need to show that:

(i) \( \forall \Gamma, S. \exists E. \text{trans}(\Gamma, [, ] , S) \equiv E \), and

(ii) \( \forall E. \exists \Gamma, S. \text{trans}(\Gamma, [, ] , S) \equiv E \).

First note that \( \Lambda_A \) (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):

\[ [.]^\circ \equiv \epsilon \]
\[ (\lambda x)^\circ = \lambda^\circ [x] \]

(case \( \lambda \) of \( \alpha \))

\( (\cdot)^\circ \) takes \( \Lambda_A \) into the set of update-marker free stacks. Call its inverse \( (\cdot)^* \).

Then show that \( \{ \Gamma, \Lambda[\alpha], \alpha \} \rightarrow^* \{ \Gamma, \alpha, \alpha^\circ \} \) and that \( \text{trans}(\Gamma, \Lambda[\alpha], \alpha) \) is identical to \( \text{trans}(\Gamma, \alpha, \alpha^\circ) \) by induction on the structure of \( \alpha \).

To show (i), generalise the statement to show that for all \( \Gamma \) and \( S \) both \( \text{trans}(\Gamma, \Lambda, S) \) and \( \text{trans}(\Gamma[x_0 = \lambda_0[x_1], \ldots, x_n = \lambda_n], S) \) are evaluation contexts by induction on the number of update markers in \( S \).

To show (ii), proceed by case analysis on \( E \), and produce a \( \Gamma \) and \( S \) in each case. The difficult case is when

\[ E \equiv \text{let } \{ \tilde{y} = \tilde{M}, x_0 = \lambda_0[x_1], \ldots, x_n = \lambda_n \} \]

Here, let \( \Gamma \) be \( \{ \tilde{y} = \tilde{M} \} \) and let \( S \) be \( \lambda_0^\circ # x_0 \ldots \lambda_n^\circ # x_1 \lambda_0^\circ # x_0 \lambda_0^\circ \).

A.5.4 A lemma for (case-E) and (let-E)

The following lemma can be used to validate (case-E) and (let-E). \( \text{CV}(E) \) denotes the capture variables of \( E \).

**Lemma A.6** For all \( E \), there exist \( \Gamma, S \), such that \( \text{dom}(\Gamma, S) \subseteq \text{CV}(E) \) and \( \forall \Delta, T. \langle \Delta, E, T \rangle \rightarrow^* \langle \Delta, [\cdot], ST \rangle \).

**Proof**. By lemma 4.1, there exist \( \Gamma \) and \( S \) such that \( \text{trans}(\Gamma, [, ] , S) \equiv E \), so by translation \( \{ \emptyset, E, \epsilon \} \rightarrow \langle \Gamma, [, ] , S \rangle \), and thus by extension, provided \( \text{dom}(\Gamma, S) \subseteq \text{CV}(E) \), \( \langle \Delta, E, T \rangle \rightarrow^* \langle \Delta, [\cdot], ST \rangle \).

A.6 Proof: the Unwinding Lemma

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

**Lemma A.7** For all \( M, \Gamma, S, V \) and \( n \),

\[ \langle \Gamma[x \downarrow^k V], M, S \rangle \psi^n \implies \langle \Gamma[x \downarrow^k V], M \sigma, S \sigma \rangle \psi^n \]

where \( \sigma = [x_{k+1}^n], M \sigma, S \sigma \psi^n \)

Recall the statement of the Unwinding lemma:

For all \( \Gamma, S, \) and \( n \),

\[ \langle \Gamma, \text{let } \{ f \equiv V \} \text{ in } M, S \rangle \psi^n \implies \exists m. \langle \Gamma, \text{let } \{ f \downarrow^m \} \text{ in } M[\text{f/\{f\}}, S \rangle \psi^n. \]

It suffices to prove that for all \( \Gamma, S, \) and \( n \) such that \( \{x_i\}_{i=0}^n \notin \text{FV}(V, S) \).

\[ \langle \Gamma[x = V], M, S \rangle \psi^n \implies \langle \Gamma \sigma[x \downarrow^k V], M \sigma, S \sigma \rangle \psi^n \]

where \( \sigma = [x_{k+1}^n] \) (i.e. \( m = n \)). Suppose \( \langle \Gamma[x = V], M, S \rangle \psi^k \). We proceed by induction on \( n \). By open uniform computation, \( \langle \Gamma, M, S \rangle \) reduces in \( k \geq 0 \) steps to one of

\[ (1) \langle \Delta, W, e \rangle, \quad (2) \langle \Delta, x, T \rangle. \]

(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 \( k \) steps to \( \langle \Delta \sigma[x = V], W \sigma, e \rangle \) and we are done, since \( k = n \). In case (2), by extension, \( \langle \text{Lookup}, \rangle \), and \( \langle \text{Update}, \rangle \),

\[ \langle \Gamma \sigma[x = V], M \sigma, S \sigma \rangle \rightarrow^k \langle \Delta \sigma[x = V], x_n, T \sigma \rangle \rightarrow^2 \langle \Delta \sigma[x = V], V \sigma, T \sigma \rangle. \]

Similarly, \( \langle \Gamma[x = V], M, S \rangle \) reduces in \( k + 2 \) steps to \( \langle \Delta[x = V], V, T \rangle \). By the inductive hypothesis, we know that \( \langle \Delta \sigma'[x \downarrow^k V], V \sigma', T \sigma' \rangle \psi^{k'} \) where \( \sigma' = [x_{k+1}^n] \) and \( k' = n - k - 2 \). By repeated application of lemma A.7, we have that \( \langle \Delta \sigma[x \downarrow^k V], V \sigma, T \sigma \rangle \psi^{k'} \) and hence \( \langle \Gamma \sigma[x \downarrow^k V], M \sigma, S \sigma \rangle \psi^n \) as required.

A.7 Proof: the Improvement Theorem

This lemma is used to prove lemma A.9.

**Lemma A.8** For all \( \Gamma, S, \) and \( n \),

\[ \langle \Gamma[x \equiv V], M, S \rangle \psi^n \iff \langle \Gamma \sigma[x \equiv \tilde{V}], \tilde{y} \equiv \tilde{V} \sigma, M \sigma, S \sigma \rangle \psi^n, \]

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 the improvement theorem, we will need the following lemma, which is stated without proof. It follows in a straightforward fashion from lemma A.8. (It may seem to follow from the context lemma, but the \( x \) may appear free in \( \Gamma \) in the conclusion, so it does not.)
Lemma A.9 If \( \{ \bar{x} = \bar{V} \} \) in \( M \) \( \supseteq \) \( \{ \bar{x} = \bar{V} \} \) in \( N \) then for all \( \Gamma \) and \( S \),
\[
(\Gamma[\bar{x} = \bar{V}], M, S)\psi^n \Rightarrow (\Gamma[\bar{x} = \bar{V}], N, S)\psi^S^n.
\]
We prove the improvement theorem generalised to mutually-recursive definitions:

The following proof rule is sound:

\[
\forall j \in J. \forall \{ \bar{f}_i = \bar{V}_i \}_{i \in I} \text{ in } V_j \supseteq \forall \{ \bar{f}_i = \bar{V}_i \}_{i \in I} \text{ in } W_j
\]

let \( \{ \bar{f}_i = \bar{V}_i \}_{i \in I} \) in \( \forall \{ \bar{f}_i = \bar{W}_i \}_{i \in I} \) in \( N \)

By the context lemma it suffices to show that for all \( \Gamma, S, \) and \( n \),
\[
(\Gamma[\bar{f} = \bar{V}], N, S)\psi^n \Rightarrow (\Gamma[\bar{f} = \bar{W}], N, S)\psi^n.
\]
Assume the premise and suppose that \( (\Gamma[\bar{f} = \bar{V}], N, S)\psi^n \). We proceed by induction on \( n \). By open uniform computation, \( (\Gamma, N, S) \) reduces in \( k \geq 0 \) steps to one of

1. \( (\Delta, V, \epsilon) \)
2. \( (\Delta, f, T) \)

In case (1), we have by extension that \( (\Gamma[\bar{f} = \bar{V}], N, S) \) reduces in \( k \) steps to \( (\Delta[\bar{f} = \bar{V}], V, \epsilon) \) and \( k = n \), so we are done. In case (2),

\[
(\Gamma[\bar{f} = \bar{V}], N, S) \rightarrow^k (\Delta[\bar{f} = \bar{V}], f, T)
\]

\[
\rightarrow^2 (\Delta[\bar{f} = \bar{V}], V, T) \quad (A.6)
\]

and

\[
(\Gamma[\bar{f} = \bar{W}], N, S) \rightarrow^k (\Delta[\bar{f} = \bar{W}], f, T)
\]

\[
\rightarrow^2 (\Delta[\bar{f} = \bar{W}], W, T) \quad (A.7)
\]

so
\[
(\Delta[\bar{f} = \bar{V}], V, T)\psi^{n-(k+2)} \quad (A.6)
\]

\[
\Rightarrow (\Delta[\bar{f} = \bar{V}], W, T)\psi^S^{n-(k+2)} \quad \text{ (ass., lem. A.9) }
\]

\[
\Rightarrow (\Delta[\bar{f} = \bar{W}], W, T)\psi^S^{n-(k+2)} \quad \text{ (I.H.) }
\]

\[
\Rightarrow (\Gamma[\bar{f} = \bar{W}], N, S)\psi^S^n. \quad (A.7)
\]

A.8 Proof: Improvement Induction

Recall the statement of improvement induction:

For any \( M, N \) and substitution \( \sigma \), the following proof rule is sound:

\[
\frac{\Gamma \vdash \Delta \quad \Sigma(\bar{x}) \sigma \equiv \Sigma(\bar{x}) \quad \bar{x} \not\in \text{fv}(\Delta)}{\Gamma \vdash \Delta[\bar{x} = \bar{V}] \quad \Sigma(\bar{x}) \sigma \equiv \Sigma(\bar{x}) \quad \bar{x} \not\in \text{fv}(\Delta)}\]

We generalise \( \Sigma(\bar{x}) \sigma \equiv \Sigma(\bar{x}) \) to \( \Sigma(\bar{x}) \sigma \equiv \Sigma(\bar{x}) \). It suffices to show under assumption of the premise, that, for all \( \Sigma \) such that \( \Sigma[\bar{x}]M \) and \( \Sigma[\bar{x}]N \) are closed, \( \Sigma[\bar{x}]M)\psi^n \Rightarrow \Sigma[\bar{x}]N)\psi^S^n \).

Suppose \( \Sigma[\bar{x}]M)\psi^n \). We proceed by induction on \( n \). By open uniform computation, \( \Sigma \) reduces in \( k \geq 0 \) to one of

1. \( (\Delta, \forall, \epsilon) \)
2. \( (\Delta, \xi \cdot \bar{y}, \forall) \)