A Foundation for Space-Safe Transformations of Call-by-Need Programs

Jörgen Gustavsson and David Sands

Chalmers*

Abstract

We introduce a space-improvement relation on programs which guarantees that whenever M is improved by N, replacement of M by N in a program can never lead to asymptotically worse space (heap or stack) behaviour, for a particular model of garbage collection. This study takes place in the context of a call-by-need programming language. For languages implemented using call-by-need, e.g, Haskell, space behaviour is notoriously difficult to predict and analyse, and even innocent-looking equivalences like $x+y = y+x$ can change the asymptotic space requirements of some programs. Despite this, we establish a fairly rich collection of improvement laws, with the help of a context lemma for a finer-grained improvement relation. We briefly consider an application of the theory; we prove that inlining of affine-linear bindings (as introduced by a certain class of "used-once" type-systems) is work- and space-safe. We also show that certain weaker type systems for usage do not provide sufficient conditions for space-safe inlining.

1 Introduction

The space-usage of lazy functional programs is perhaps the most thorny problem facing programmers using languages such as Haskell. Almost all programmers unable to predict or control the space behaviour of their lazy programs. Even the most advanced programmers, who are able to visualise the space use of their programs, complain that the "state-of-the-art" compilers *introduce* space-leaks into programs that they believe ought to be space-efficient.

In recent years a successful line of research into profiling tools for lazy functional languages [RW93, RR96] has greatly improved a programmer's chances of locating sources of space leaks. But apart from a few high-level operational semantics which claim to model space behaviour, to the best of our knowledge there have been no formal/theoretical/semantics-based approaches to reasoning about space behaviour of programs.

Rather than tackling the problem of determining the absolute space behaviour of a program, in this paper we study notions of relative space efficiency. We pose the question: when it is space-safe to replace one program fragment by another? To this end we introduce a space-*improvement* relation on terms, which guarantees that whenever M is improved by N, replacement of M by N in a program can never lead to asymptotically worse space (heap or stack) behaviour, for a particular model of computation and garbage collection.

The fact that we only aim to prevent *asymptotic* worsening might seem rather weak. One reason is that we (wish to) work with high-level semantic models of space behaviour, so it is not meaningful for us to make stronger claims. Another reason is that asymptotic changes in space behaviour are not at all unusual. (We consider such an example below.)

Why is the space behaviour of lazy functional programs difficult to predict? One reason is of course that all memory management is automatic, coupled with the fact that the heap allocation rate of functional programs is very high; just about everything lives in the heap. A second reason is that the non-strict evaluation order that is required by the language specification means that computation-order bears no obvious relation to textual structure of code. The third, and perhaps most subtle reason is that all realistic implementations of lazy languages use a call-by-need. Call-by-need optimises call-by-name by ensuring that when evaluating a given function application, arguments are evaluated at most once. The effect of sharing is to reduce – often dramatically – the time required to execute a program. But the effect of this additional sharing on the space behaviour is to prolong the lifetime of data, and this is often at the cost of space.

As an illustration of some of these problems, consider one of the most innocent of the extensional equivalences that functional programming languages enjoy: $x + y = y + x$. Now consider the following Haskell program:

$$\begin{array}{l}\mathsf{let}\ xs = [1..n] \\ \quad\ x = head\ xs \\ \quad\ y = last\ xs \\ \mathsf{in}\ x + y\end{array}$$

This program runs in $\mathcal{O}(n)$ time and in constant space. First x is evaluated[1] to obtain 1, then y is evaluated, which involves constructing and traversing the entire list $[1..n]$. Fortunately, the cocktail of lazy evaluation, tail recursion and garbage collection guarantees that as this list is constructed

*Chalmers University of Technology and Göteborg University, Sweden. {gustavss,dave}@cs.chalmers.se

Draft of 2 July; submitted for review to HOOTS'99

[1] Evaluation order is intentionally left unspecified in the Haskell language definition, but at the time of writing the main implementations coincidentally evaluate the arguments to such primitive functions from left to right.

and traversed it can also be garbage collected, and thus requiring only constant space.

Now consider what happens when $x + y$ is replaced by $y + x$. In this case the time complexity is the same, but now the *space* required is $\mathcal{O}(n)$. This is because when y builds and traverses the list $[1..n]$, the elements cannot be garbage-collected because the whole list is still referenced by the variable x.

So we can conclude that replacing $x + y$ by $y + x$ can give an asymptotic change in space behaviour – i.e., there is no constant which bounds the potential worsening in space when this law is applied in an arbitrary context. So our theory of improvement will not relate this particular pair of terms.

But expressions that fall outside our improvement theory are easy to find (see e.g., [PJ87] for more tricky examples). Are there any improvements which *are* guaranteed to hold in absolutely any program context? In this article we show that there are indeed many valid space-improvement laws, and thus we lay the foundations for a theory of space behaviour of call-by-need programs.

The remainder of the article is organised as follows. **Section 2** describes related work. **Section 3** gives the syntax and operational semantics of our language. **Section 4** defines what we mean by the space-use of programs, in terms of a definition of garbage collection for abstract-machine configurations. We informally argue the ways in which this definition agrees with lower-level models, and mention a number of subtle choices and variations in actual implementation methods. **Section 5** defines the main improvement relation, *weak improvement*, and presents the basic laws and properties of this relation. **Section 6** describes a finer-grained improvement relation, *strong improvement*, which is used to establish the weak improvement laws via a *context lemma*. **Section 7** considers an application of the theory; we prove that inlining of affine-linear bindings (as introduced by a certain class of "used-once" type-systems) is work- and space-safe. **Section 8** concludes.

2 Related Work

Improvement theory was originally developed in the call-by-name setting [San95, San91, San96] for the purpose of reasoning about running-times of programs. Moran and Sands [MS99] developed a call-by-need time-improvement theory, together with a number of induction principles, and essentially all the laws that we prove here are also time-improvements.

A number of operational semantics have been proposed which are intended to model the space requirements of call-by-need programs. Launchbury suggested adding a garbage collection rule to his natural semantics [Lau93]; instead, our notion of computation and memory usage is based on Sestoft's mark-1 abstract machine[Ses97], which makes stack-usage explicit (and thereby shortcuts some space-leaks and anomalies in Launchbury's proposal). Rose [Ros96] uses a graph reduction model based on explicit substitutions in which the correct modelling of space is emphasised, and in [BLR96] a sketch of the space-safety of aspects of the STG-machine implementation with respect to the model is given.

Morrisett and Harper [MH98] use a similar style of abstract machine description to that used here in order to investigate the semantics of memory management in an ML-like language (see also [MFH95]). They give abstract specifications of garbage collection, and prove the correctness of a particular type-based collection scheme.

3 Operational Semantics

Our language is an untyped lambda calculus with recursive lets, structured data, case expressions, integers (ranged over by n and m) with addition and a zero test. We work with a restricted syntax in which arguments to functions (including constructors) are always variables:

$$\begin{aligned} L, M, N ::= {} & x \mid \lambda x.M \mid M\,x \mid c\,\vec{x} \\ & \mid n \mid M + N \mid \mathsf{add}_n\,M \mid \mathsf{iszero}\,M \\ & \mid \mathsf{let}\ \{\vec{x} = \vec{M}\}\ \mathsf{in}\ N \\ & \mid \mathsf{case}\ M\ \mathsf{of}\ \{c_i\,\vec{x}_i \to N_i\} \end{aligned}$$

The syntactic restriction is now rather standard, following its use in core language of the Glasgow Haskell compiler, *e.g.*, [PJPS96, PJS98], and in [Lau93, Ses97].

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 $(\lambda x.M \mid c\,\vec{x} \mid n)$. 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

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

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 $\{c_i\,\vec{x}_i \to N_i\}$.

The functions add_n are included for convenience in the definition of the abstract machine, and represent an intermediate step in the addition of n to a term.

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 [Ses97]. Transitions are over configurations consisting of a *heap*, containing bindings, the expression currently being evaluated, and a *stack*. We write $\langle\, \Gamma,\ M,\ S \,\rangle$ for the abstract machine configuration with heap Γ, expression M, and stack S. A heap is a set of bindings; we denote the empty heap by $\emptyset$, and the addition of a group of fresh bindings $\vec{x} = \vec{M}$ to a heap Γ by juxtaposition: $\Gamma\{\vec{x} = \vec{M}\}$. The stack written $b : S$ will denote the 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).

Stack elements are either:

- a *reduction context*, or
- an *update marker* $\#x$, indicating that the result of the current computation should be bound to the variable x in the heap.

The reduction contexts on the stack are shallow contexts containing a single hole in a "reduction" position - i.e. in a position where the current computation is being performed. They are defined as:

$$\mathbb{R} ::= [\cdot]\, x \mid \mathsf{case}\ [\cdot]\ \mathsf{of}\ \{c_i\, \vec{x}_i \to N_i\} \mid [\cdot] + M \mid \mathsf{add}_n[\cdot] \mid \mathsf{iszero}\,[\cdot]$$

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 $P \natural Q$ 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 1; we implicitly restrict the definition to well-formed closed configurations.

The first group of rules are the standard call-by-need rules. 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. 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.

The basic computation rules are captured by the (*Push*) and (*Reduce*) rules schemas. The rule (*Push*) allows us to get to the heart of the evaluation by "unwinding" a shallow reduction context. When the term to be evaluated is a value and there is a reduction context on the stack, the (*Reduce*) rule is applied.

4 Space Use and Garbage Collection

A desired property of our model of space-use is that it is true to actual implementations. Unfortunately, different abstract machines and garbage collection strategies differ in their asymptotic space behaviour. Consider for example an application of the function

$$f = \lambda x.\mathsf{let}\ y = f\ y\ \mathsf{in}\ y$$

using some of the main Haskell implementations.[2] This runs in constant space under HUGS'98 and GHC 4.01, but runs out of stack in hbc 0.9999.5a, and in some older versions of GHC. Given the different space behaviours of different implementations there is no hope that we can construct a theory which applies to all implementations. Although we will choose a particular model of space use we believe that most of the results and techniques developed in this paper can be adopted to any reasonable model. We will return to this subject shortly and discuss some of the subtle ways in which implementations differ.

[2] www.haskell.org/implementations.html

4.1 Measuring space

In principle the heap and the stack can share the same memory, and thus a program transformation which trades heap for stack, or vice versa, should be perfectly fine. However, in practice most implementations allocate separate memory for the heap and the stack, and we would therefore like to reject such transformations. If a program transformation improves the maximum required stack and heap uses respectively, it can never worsen the maximum required total space used by more than a factor of two – but the converse is not necessarily true. Therefore we have decided to measure the heap and the stack separately.

We measure the heap space occupied by a configuration by counting the number of bindings in the heap and the number of update markers on the stack. We count update markers on the stack as also occupying heap space, since in a typical implementation an update marker refers to a so-called "blackhole closure" in the heap – a placeholder where the update eventually will take place. We will count every binding as occupying one unit of space. In practice the size of a binding varies since a binding is typically represented by a tag or a code pointer plus an environment with one entry for every free variable. However, the right hand side of every binding is always a (possibly renamed) subexpression of the original program (a property sometimes called *semi-compositionality*[3]), so counting a binding as one unit gives a measure which is within a constant factor (depending only on the program size) of the actual space used. We measure stack space by simply counting the number of elements on the stack, and so an update marker will be viewed as occupying both heap and stack space. In practice every element on the stack does not occupy the same amount of space, but again, semicompositionality of the abstract machine assures that our measure is within a program-size-dependent constant factor. We will write $|\langle\, \Gamma,\ M,\ S\,\rangle|$ for the pair (h, s) where h and s is the amount of heap and stack respectively occupied by the the configuration.

4.2 Garbage collection

To reason about space use we must model garbage collection, which allows us to decrease the amount of space used by a configuration during a computation. Garbage collection is simply the removal of any number of bindings and update markers from the heap and the stack respectively, *providing that the configuration remains closed.*

Definition 4.1 (GC) *Garbage collection can be applied to a closed configuration* $\langle\, \Gamma,\ M,\ S\,\rangle$ *to obtain* $\langle\, \Gamma',\ M,\ S'\,\rangle$, *written* $\langle\, \Gamma,\ M,\ S\,\rangle \gg \langle\, \Gamma',\ M,\ S'\,\rangle$ *if and only if* $\langle\, \Gamma',\ M,\ S'\,\rangle$ *is closed, and can be obtained from* $\langle\, \Gamma,\ M,\ S\,\rangle$ *by removing zero or more bindings and update markers from the heap and the stack respectively.*

This is an accessibility-based definition as found in e.g., the gc-reduction rule of [MH98]. The removal of update-markers from the stack is not surprising given that they are viewed as the binding occurrences of the variables in question.

4.3 Evaluation in fixed space

We are now ready to define what it means for a computation to be possible in certain fixed amount of space.

[3] A term due to Neil D. Jones

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

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

$$\langle\, \Gamma,\ \mathsf{let}\ \{\vec{x} = \vec{M}\}\ \mathsf{in}\ N,\ S\,\rangle \to \langle\, \Gamma\{\vec{x} = \vec{M}\},\ N,\ S\,\rangle \quad \vec{x} \not\downarrow \mathrm{dom}(\Gamma, S) \qquad (\mathit{Letrec})$$

$$\langle\, \Gamma,\ \mathbb{R}[M],\ S\,\rangle \to \langle\, \Gamma,\ M,\ \mathbb{R} : S\,\rangle \qquad (\mathit{Push})$$

$$\langle\, \Gamma,\ V,\ S\rangle\langle\, \Gamma,\ V,\ \mathbb{R} : S\,\rangle \to \langle\, \Gamma,\ M,\ S\,\rangle \quad \text{if } \mathbb{R}[V] \rightsquigarrow M \qquad (\mathit{Reduce})$$

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

$$\mathsf{case}\ c_j\ \vec{y}\ \mathsf{of}\ \{c_i\ \vec{x}_i \to M_i\} \rightsquigarrow M_j[\vec{y}/\vec{x}_j]$$

$$m + N \rightsquigarrow \mathsf{add}_m N$$

$$\mathsf{add}_m n \rightsquigarrow \ulcorner m + n \urcorner$$

$$\mathsf{iszero}\ m \rightsquigarrow \begin{cases} \mathsf{true} & \text{if } m = 0 \\ \mathsf{false} & \text{otherwise} \end{cases}$$

Figure 1: Abstract machine semantics

Definition 4.2 (Convergence in fixed space)

$$\langle\, \Gamma,\ M,\ S\,\rangle \twoheadrightarrow^{(h,s)} \langle\, \Delta,\ N,\ T\,\rangle$$
$$\stackrel{\text{def}}{=} \exists \Phi.\langle\, \Gamma,\ M,\ S\,\rangle \to \Phi \gg \langle\, \Delta,\ N,\ T\,\rangle$$
$$\textit{and } |\langle\, \Delta,\ N,\ T\,\rangle| \le (h, s)$$

$$\twoheadrightarrow\!\!\!\!\twoheadrightarrow^{(h,s)} \stackrel{\text{def}}{=} \textit{the refl. and trans. closure of } \twoheadrightarrow^{(h,s)}$$

$$\langle\, \Gamma,\ M,\ S\rangle\Downarrow^{(h,s)} \stackrel{\text{def}}{=} \exists \Delta, V.\langle\, \Gamma,\ M,\ S\,\rangle \twoheadrightarrow\!\!\!\!\twoheadrightarrow^{(h,s)} \langle\, \Delta,\ V,\ \epsilon\,\rangle$$

$$M\Downarrow^{(h,s)} \stackrel{\text{def}}{=} \langle\, \emptyset,\ M,\ \epsilon\rangle\Downarrow^{(h,s)}$$

4.4 Some subtleties

Different implementations vary in their space behaviour in a number of rather subtle ways. We will discuss some of those points below and how they relate to our particular space model. The reader who is less interested in the details of memory management for lazy languages may safely skip this section.

Environment trimming Our abstract machine is based on substitution of variables for variables, but lower level abstract machines are usually based on environments [Ses97]. To avoid space leaks in environment-based machines it is crucial to remove redundant bindings from environments on the stack. This is sometimes called *environment trimming* [Ses97] or stack stubbing [PJ92]. Some implementations do not properly trim environments and programs like

$$f\ x\ y = \mathsf{case}\ g\ x\ \mathsf{of}$$
$$\ldots \to \ldots y \ldots$$

can lead to space leaks when compiled with the nhc compiler, since a reference to x is kept on the stack during the evaluation of $g\,x$ [Röj95]. Our definition of space is consistent with an environment machine which *does* perform environment trimming.

Blackholing In our abstract machine, the lookup rule removes the binding from the heap while it is being evaluated. This corresponds to so called "black holing" in real implementations where the closure is overwritten with a special "black hole closure" without free variables [PJ92]. In some early implementations the closure was instead left untouched in the heap [RW93, Jon92]. This has the effect that the garbage collector can not reclaim space that the free variables of the closure hangs on to.

Garbage collection of update markers In our model we allow for the garbage collection of update markers which allows an application of f

$$f = \lambda x.\mathsf{let}\ y = f\ y\ \mathsf{in}\ y$$

to run in constant space – as it does in HUGS'98 and GHC 4.01, but not in hbc 0.9999.5a, or in some older versions of GHC.

Avoiding value copying and shortcutting indirections When running

$$\mathsf{let}\ x = 1 + 2, y = id\ x\ \mathsf{in}\ y + M$$

in our abstract machine we will end up with both x and y bound to 3 in the heap. Some implementations would instead bind x to 3 and create an indirection $y = x$ from y to x (or vice versa). If this is combined with a garbage collector which can shortcut indirections (by in this case replacing all occurrences of y with x and removing $y = x$) then space can be saved. This can only reduce the total space used by a constant factor, but it can have a quite dramatic effect in practice [RW93]. However since our space model is only adequate up to constant factors anyway, this is not a serious drawback of our space model. For implementations that create indirections in this way it is important that the garbage collector can shortcut indirections. Otherwise not much would be gained, and in our example, the space for x cannot be reclaimed before $y = x$ is reclaimed, thus possibly increasing the space used. Since our abstract machine do not create indirections we have not included shortcutting of indirections in our garbage collector. This has the effect

that programs which contains indirections in the source code (that is, subterms of the form $\mathsf{let}\ y = x\ \mathsf{in}\ M$) may use up more space than necessary, but otherwise the model is well-behaved.

Shortcutting update marker chains Sometimes two or more update markers, say $\#x$ and $\#y$, end up on top of each other on the stack. Then both x and y will eventually be updated with the same value or, in implementations which introduce indirections, one will be indirected to the other. Some implementations preclude this situation by never pushing update markers on top of each other: if $\#x$ is already on the stack, an indirection $y = x$ is created instead of pushing $\#y$. When this is combined with garbage collection of indirections, the effect is similar to the combined effect of garbage collection of update markers, avoiding value copying and garbage collecting indirections. As far as we know this trick has not been documented, but it is used in the GHC compiler[4].

Lazy patterns Lazy patterns in let bindings like

$$\mathsf{let}\ (x, y) = M\ \mathsf{in}\ N$$

are an important feature of real lazy languages such as Haskell. They might be encoded in our language in the following manner

$$\mathsf{let}\ p = M, x = \mathsf{fst}\, p, y = \mathsf{snd}\, p\ \mathsf{in}\ N$$

This encoding seems to be used in, for example HUGS'98, but can lead to space leaks (unless special consideration is taken in the garbage collector which essentially amounts to letting the garbage collector do a restricted form of computation [Wad87]). Instead, as suggested by Sparud [Spa93] one can extend the core language to cope with lazy patterns directly. We have left out lazy patterns to keep things simple but there should be no problem with adding them.

5 Weak Space Improvement

In the previous section we defined a notion of space which we believe is realistic in the sense that an actual implementation (using our reasonably aggressive garbage collection) will require space within a constant factor of our abstract measure, where the constant depends on the size of the program to be executed.

In this section we define *space improvement within a constant factor* – what we will simply refer to as *Weak Improvement* – which says that if M is improved by N, replacing M by N in any program context will never lead to more than a constant factor worsening in space behaviour, where the constant factor is independent of the context.

The starting point for an operational theory is usually an approximation and an equivalence defined in terms of *program contexts*. Program contexts are usually 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 such that holes may not occur in argument positions of an application or a constructor, for if this were the case, then filling a hole (with a non variable) would violate the syntax. 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. For the technical development we work with the second order syntax for contexts as in e.g., [San98a][MS99], although we elide these technicalities in the present abstract.

Definition 5.1 (Weak Improvement) *We say that M is* weakly improved by *N, written $M \gtrapprox N$, if there exists a linear function $f \in \mathbb{N} \to \mathbb{N}$ such that for all $\mathbb{C}$ such that $\mathbb{C}[M]$ and $\mathbb{C}[N]$ are closed,*

$$\mathbb{C}[M]{\Downarrow}^{(h,s)} \implies \mathbb{C}[N]{\Downarrow}^{(f(h),f(s))}.$$

So $M \gtrapprox N$ means that N never takes up more than a constant factor more space than M (but it might still use non-constant factor less space). We write $M \lessapprox N$ to mean that $M \gtrapprox N$ and $N \gtrapprox M$.

Proposition 5.1 (Precongruence) *$\gtrapprox$ is a* precongruence *– i.e., it is a transitive and reflexive relation which is preserved by contexts.*

Proof. The proof of all but transitivity is immediate. Transitivity follows from the fact that the composition of any two linear functions is linear. □

The improvement relations in this article also imply possibly improved termination properties, although this choice is not a particularly significant, and all laws, excepting those involving explicitly nonterminating terms, are also operational equivalences.

5.1 Weak Improvement Laws

Here we summarise a collection of laws for weak improvement. The laws are established using a strong version of improvement, and a *context lemma* that is described in the next section. The first property is fundamental, and highlights the significance of free variables in this theory:

Theorem 5.2 (Free Variable Property) *If $M \gtrapprox N$ then $\mathsf{FV}(M) \supseteq \mathsf{FV}(N)$.*

Proof. Suppose that $M \gtrapprox N$. Then there exits a linear function f which bounds the extra space required to compute with N instead of M. Assume, towards a contradiction, that there exists a variable x such that $x \in \mathsf{FV}(N)$ but $x \notin \mathsf{FV}(M)$. Without loss of generality we can assume that $\mathsf{FV}(N) = \{x\}$ and $\mathsf{FV}(M) = \emptyset$ (since by congruence of $\gtrapprox$ we can wrap a context around M and N which ensures this property). Now consider the context $\mathbb{C}$:

$$\begin{array}{lll}
\mathsf{let} & \mathit{traverse} = \lambda xs.\mathsf{case}\ xs\ \mathsf{of} & \\
 & \qquad\qquad \mathsf{nil} \to 1 & \\
 & \qquad\qquad h : t \to \mathit{traverse}\, t & \\
 & \mathit{count} = \lambda n.\ \mathsf{case}\ \mathsf{iszero}\ n\ \mathsf{of} & \\
 & \qquad\qquad \mathsf{true} \to \mathsf{nil} & \\
 & \qquad\qquad \mathsf{false} \to \mathsf{let} & a = n - 1 \\
 & & \quad r = \mathit{count}\, a \\
 & & \mathsf{in} \quad n : r \\
 & x = \mathit{count}\, \mathbf{k} & \\
 & z = [\cdot] & \\
\mathsf{in} & \mathit{traverse}\, x + (\lambda y.1) z &
\end{array}$$

[4] Simon Peyton Jones, Personal communication, June 1999.

where : is an infix cons constructor. It can be seen (we omit a formal proof, which would be somewhat tedious) that $\mathbb{C}[M]$ evaluates with some constant space, independent of $\mathbf{k}$. This is because the list *count* $\mathbf{k}$ can be garbage collected as it is traversed. However $\mathbb{C}[N]$ requires space proportional to $\mathbf{k}$, since there is a (dead code) reference to x which prevents any of the list from being collected until it has been completely constructed. Since we can make $\mathbf{k}$ arbitrarily large we cannot have $M \gtrapprox N$. □

The sketch proof above relies on unbounded integers. A similar example can be constructed using just a finite set of constructors and a logarithmic-space encoding of $\mathbf{k}$.

In figure 2 we collect some weak improvement laws. Like any other contextual program ordering, it is not a recursively enumerable, so any such collection is inevitably somewhat ad hoc.

In presenting laws, we will follow two conventions. The first is the standard free-variable convention [Bar81] that all bound variables in the statement of a law are distinct, and that they are disjoint from the free variables. The second is that in any instance of a law, the free variable property is respected. When we state a cost-equivalence, it should be taken as a pair of improvement laws. For example, from the cost-equivalence law

$$(\lambda x.M)\, y \lessapprox\!\gtrapprox M[y/x]$$

we can conclude that $(\lambda x.x)\, y \lessapprox\!\gtrapprox y$ and $(\lambda x.3)\, y \gtrapprox 3$, but not that $3 \gtrapprox (\lambda x.3)\, y$.

5.2 On divergent terms

There is no bottom element in the space-ordering relation. By Theorem 5.2, it follows that divergent terms containing different numbers of free variables are not cost equivalent – simply because when placed in a program context, their free variables can affect the amount of garbage. Thus the more free variables a divergent term contains, the lower in the improvement ordering it sits.

Proposition 5.3 *Let Ω_1 and Ω_2 denote an arbitrary terms which are operationally equivalent (with respect to observing termination behaviour) to the divergent term* $\mathsf{let}\ x = x\ \mathsf{in}\ x$. *Then $\Omega_1 \gtrapprox \Omega_2$ if and only if $\mathsf{FV}(\Omega_1) \supseteq \mathsf{FV}(\Omega_2)$.*

Let Ω_X denote an arbitrary divergent term with $\mathsf{FV}(\Omega_X) = X$ (such terms, by the above proposition, will all be cost equivalent).

6 Strong Improvement

The weak improvement laws are established with the use of a finer-grained notion of improvement which we call strong improvement. This section describes the properties of strong improvement.

Definition 6.1 (Strong improvement) *We say that M is* strongly improved by N, *written $M \gtrapprox N$, if for all $\mathbb{C}$ such that $\mathbb{C}[M]$ and $\mathbb{C}[N]$ are closed,*

$$\mathbb{C}[M]{\Downarrow}^{(h,s)} \implies \mathbb{C}[N]{\Downarrow}^{(h,s)}.$$

We write $M \lessapprox\!\gtrapprox N$ to mean that $M \gtrapprox N$ and $N \gtrapprox M$.

For strong improvement we have established a *context lemma* [Mil77]: to prove that M is strongly improved by N, one only needs to compare their behaviour with respect to a much smaller set of contexts, namely the context which immediately need to evaluate their holes.

Lemma 6.1 (Context Lemma) *For all terms M and N such that $\mathsf{FV}(M) \supseteq \mathsf{FV}(N)$, if for all Γ and S,*

$$\langle\, \Gamma,\ M,\ S \,\rangle{\Downarrow}^{(h,s)} \implies \langle\, \Gamma,\ N,\ S \,\rangle{\Downarrow}^{(h,s)}$$

then $M \gtrapprox N$.

The configuration contexts of the form $\langle\, \Gamma,\ [\cdot],\ S \,\rangle$ correspond to the notion of *evaluation contexts* described in [MS99].

6.1 Spikes and ballast

To prove, for example, (restricted) beta-reduction $(\lambda x.M)\, y \gtrapprox M[y/x]$ we will show the stronger property: $(\lambda x.M)\, y \gtrapprox M[y/x]$. The context lemma makes this property very easy to establish. The converse direction also holds within a constant factor (under the assumption that y occurs free in $M[y/x]$). The only difference when going from the right-hand side to the left is that the left hand side will momentarily use up one stack unit more than the right-hand side.

In order to express the latter property using the more precise improvement theory, we need some space analogue of the time-tick from [MS99]. In fact, we will use four kinds of "tick", two of which are minor language extensions.

The first two devices, the *stack spike* and the *heap spike*, are (like the time-tick) representable in the language. The stack spike is defined thus

$${}^{\curlyvee}M \stackrel{\text{def}}{=} \mathsf{case}\ \mathsf{true}\ \mathsf{of}\ \{\mathsf{true} \to M\}$$

It has the short-lived effect of increasing the stack usage by one unit, at the moment that M is about to be evaluated. The improvement above can now be made more precise by noting that:

$$(\lambda x.M)\, y \lessapprox {}^{\curlyvee}M[y/x] \qquad \text{if } y \in \mathsf{FV}(M[y/x]).$$

The *heap spike* is the heap analogue of the stack spike; it momentarily increases the size of the heap at the point in time when the term is ready to be evaluated.

$${}^{\curlywedge}M \stackrel{\text{def}}{=} \mathsf{let}\ x = M\ \mathsf{in}\ x \qquad x \notin \mathsf{FV}(M)$$

Now we come to two constructs which involve minor language extensions which affect the definition of space but do not otherwise change the definition of evaluation. The first is *stack ballast*, which corresponds to adding extra space to a stack-element. We now assume that every reduction context is labelled with a natural number, ${}^{n}\mathbb{R}$ – which we call stack ballast. A ballast-free reduction context will now be taken as shorthand for a reduction context with ballast of 0. The following modifications are made to the definitions:

- Ballast is transparent from the point of view of the computation rules, so $\mathbb{R}[V] \rightsquigarrow N \implies {}^{n}\mathbb{R}[V] \rightsquigarrow N$;
- the stack consumption of a reduction context with ballast is given by: $|{}^{n}\mathbb{R}| = n + 1$.

Thus ballast holds on to n additional units of stack space, for the length of time that the reduction context remains on the stack.

Dually, *heap ballast* adds some n units of heap consumption to a binding. Heap ballast remains attached to the binding for its lifetime, but is otherwise transparent from the point of view of computation. When a binding is subject of evaluation, that is when there is an update marker for the binding on the stack the heap ballast attaches to the update marker and adds to the heap consumption of the marker.

- The heap consumption of a binding with ballast: $|^n x = M| = n + 1$.
- The heap consumption of an update marker with ballast: $|\#^n x| = n + 1$.
- The stack consumption of an update marker with ballast: $|\#^n x| = 1$.

With the help of ballasts we can increase the size of the stack and heap spikes.

$$ {}^{n\curlyvee}M \stackrel{\text{def}}{=} {}^n\mathsf{case}\ \mathsf{true}\ \mathsf{of}\ \{\mathsf{true} \to M\} $$

$$ {}^{n\curlywedge}M \stackrel{\text{def}}{=} \mathsf{let}\ {}^n x = M\ \mathsf{in}\ x \qquad x \notin \mathsf{FV}(M) $$

Of course, spikes and ballast have no intrinsic interest for programmers – they are a bookkeeping mechanism which we use to syntactically account for different types of space usage. The crucial property of stack and heap ballast, and the respective spikes is that they do not change space behaviour by more than a constant factor:

Lemma 6.2 (Spike and Ballast Introduction)

1. $M \mathrel{\lessapprox\gtrapprox} {}^{n\curlyvee}M$
2. $M \mathrel{\lessapprox\gtrapprox} {}^{n\curlywedge}M$
3. ${}^n\mathbb{R}[M] \mathrel{\lessapprox\gtrapprox} {}^{n+m}\mathbb{R}[M]$
4. $\mathsf{let}\ \{{}^l x = L, {}^{\vec{m}}\vec{y} = \vec{M}\}\ \mathsf{in}\ N \mathrel{\lessapprox\gtrapprox} \mathsf{let}\ \{{}^{l+n} x = L, {}^{\vec{m}}\vec{y} = \vec{M}\}\ \mathsf{in}\ N$

The proof for ballast is straightforward, by showing that the largest ballast contained in a configuration can never increase during computation. The proof for heap and stack spikes is analogous to that for time ticks in [MS99].

Although ballast is a seemingly small language extension we suspect that it is not a conservative extension with respect to the space improvement theory. Thus we cannot, a priori, expect that every law that holds in the language without ballast can be established via the language with ballasts.

6.2 Strong Improvement Laws

In figure 3 we have collected a number of laws for strong improvement. The laws are related to the laws for weak improvement by careful addition spikes and ballast to each side. The laws for weak improvement thus follows immediately by an appeal to the spike and ballast introduction. Proving these laws amounts to establishing the premise of the context lemma. For this we use the notion of uniform computation from [San98a] which, with a very careful treatment of garbage, can be adapted to reasoning about space. The individual proofs are very much reminiscent of the proof of the corresponding laws for time improvement [MS99], (except that we cannot use an "open" version of uniform computation as used there).

6.3 Strong improvement and induction principles

In this paper we have used strong improvement merely as a vehicle for establishing properties of weak improvement. Although it remains as future work, we believe that we can establish some principles, such as improvement induction [San98b, MS99], for reasoning about recursive functions. We believe that these induction principles would then have premises involving strong improvement (rather than weak improvement). If so, the rôle of strong improvement would go beyond the present and would involve equational reasoning about terms with spikes and ballast. The fact that we have established a rich spike and ballast algebra (given in figure 4) which permits spikes to be introduced, moved and eliminated suggests that such equational reasoning is possible.

7 Work and Space-Safe Inlining

In this section we briefly consider an application of the theory to the problem determining when it is safe to inline a definition. Inlining is a standard compiler optimization, but one which is potentially dangerous in the context of a lazy language. The improvement relation guarantees in a certain sense that a local transformation is space-safe. As we have seen from the preceding sections, inlining of values is space safe, since from *weak-value-*β have that $\mathsf{let}\ x = V\ \mathsf{in}\ \mathbb{C}[x] \mathrel{\gtrapprox} \mathsf{let}\ x = V\ \mathsf{in}\ \mathbb{C}[V]$. Another form of inlining which is validated is $\mathsf{let}\ x = M\ \mathsf{in}\ \mathbb{R}[x] \mathrel{\gtrapprox} \mathbb{R}[M]$ (*weak-inline,weak-let-*$\mathbb{R}$). The same observations also hold for time-improvement [MS99]. But what if x is bound to a non-value or $\mathbb{C}$ is not an reduction context? To inline a non value in a non reduction context:

$$ \mathsf{let}\ \{x = M\}\ \mathsf{in}\ \mathbb{C}[x] \Rightarrow \mathsf{let}\ \{x = M\}\ \mathsf{in}\ \mathbb{C}[M] $$

can sometimes be very worthwhile [PJS98]. In more advanced transformations such as higher-order deforestation [Wad90, Mar95] it is crucial. However, as is well-known, the transformation risks duplicating computation, and this can lead to an asymptotically worse program – in both space and time. The time issue is discussed in [PJS98], where they illustrate that the naive "solution" of ensuring that there is only one syntactic occurrence of x does *not* guarantee work-safe inlining.

A number of researchers have sought to find criteria for when such transformations are work-safe, based on linear type systems and notions of "used at most once" [TWM95, Gus98, WJ99, Gus99]. Despite the fact that Turner et al discuss inlining of "used-one" bindings in some detail, as far as we are aware, it remains an open problem to actually prove that these criteria actually do guarantee work-safety. Another question (one which to our knowledge has not even been posed) is whether the "used at most once" criteria might also guarantee space safety.

In the remainder of this section we outline answers to these questions. One problem is that time and space safety do not go hand in hand. Sometimes inlining can lead to

Weak Improvement Laws

$$M \mathrel{\lessapprox\gtrapprox} N \qquad \text{if } M \rightsquigarrow N \tag{\textit{weak-red-eq}}$$

$$\mathsf{let}\ \{x = \mathbb{V}[x], \vec{y} = \vec{\mathbb{D}}[x]\}\ \mathsf{in}\ \mathbb{C}[x] \mathrel{\gtrapprox} \mathsf{let}\ \{x = \mathbb{V}[\mathbb{V}[x]], \vec{y} = \vec{\mathbb{D}}[\mathbb{V}[x]]\}\ \mathsf{in}\ \mathbb{C}[\mathbb{V}[x]] \tag{\textit{weak-value-}β}$$

$$\mathbb{R}[\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to N_i\}] \mathrel{\lessapprox\gtrapprox} \mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to \mathbb{R}[N_i]\} \tag{\textit{weak-case-}$\mathbb{R}$}$$

$$\mathbb{R}[\mathsf{let}\ \{\vec{x} = \vec{M}\}\ \mathsf{in}\ N] \mathrel{\lessapprox\gtrapprox} \mathsf{let}\ \{\vec{x} = \vec{M}\}\ \mathsf{in}\ \mathbb{R}[N] \tag{\textit{weak-let-}$\mathbb{R}$}$$

$$\mathsf{let}\ \{x = M\}\ \mathsf{in}\ x \mathrel{\lessapprox\gtrapprox} M, \quad \text{if } x \notin \mathsf{FV}(M) \tag{\textit{weak-inline}}$$

$$\mathsf{let}\ \{\vec{x} = \vec{M}\}\ \mathsf{in}\ N \mathrel{\lessapprox\gtrapprox} N, \quad \text{if } \vec{x} \not\downarrow \mathsf{FV}(N) \tag{\textit{weak-gc}}$$

$$\mathsf{let}\ \{\vec{x} = \vec{L}\}\ \mathsf{in}\ \mathsf{let}\ \{\vec{y} = \vec{M}\}\ \mathsf{in}\ N \mathrel{\lessapprox\gtrapprox} \mathsf{let}\ \{\vec{x} = \vec{L}, \vec{y} = \vec{M}\}\ \mathsf{in}\ N \tag{\textit{let-flatten}}$$

$$\mathsf{let}\ \{x = \mathsf{let}\ \{\vec{y} = \vec{L}, \vec{z} = \vec{M}\}\ \mathsf{in}\ N\}\ \mathsf{in}\ N' \mathrel{\gtrapprox} \mathsf{let}\ \{x = \mathsf{let}\ \{\vec{z} = \vec{M}\}\ \mathsf{in}\ N, \vec{y} = \vec{L}\}\ \mathsf{in}\ N' \tag{\textit{let-let-}1}$$

$$\mathsf{let}\ \{x = \mathsf{let}\ \{\vec{z} = \vec{M}\}\ \mathsf{in}\ N, \vec{y} = \vec{L}\}\ \mathsf{in}\ N' \mathrel{\gtrapprox} \mathsf{let}\ \{x = \mathsf{let}\ \{\vec{y} = \vec{L}, \vec{z} = \vec{M}\}\ \mathsf{in}\ N\}\ \mathsf{in}\ N' \quad \text{if } \vec{y} \subseteq \mathsf{FV}(N, \vec{M}) \tag{\textit{let-let-}2}$$

$$\Omega_X \mathrel{\gtrapprox} M \tag{Ω}$$

Figure 2: Weak Improvement Laws

Strong Improvement Laws

$${}^{n}\mathbb{R}[V] \mathrel{\lessapprox} {}^{n\curlyvee}N \qquad \text{if } {}^{n}\mathbb{R}[V] \rightsquigarrow N \tag{\textit{reduction-eq}}$$

$$\mathsf{let}\ \{{}^{m}x = \mathbb{V}[x], {}^{\vec{n}}\vec{y} = \vec{\mathbb{D}}[x]\}\ \mathsf{in}\ \mathbb{C}[x] \mathrel{\lessapprox} \mathsf{let}\ \{{}^{m}x = \mathbb{V}[\mathbb{V}[x]], {}^{\vec{n}}\vec{y} = \vec{\mathbb{D}}[\mathbb{V}[x]]\}\ \mathsf{in}\ \mathbb{C}[\mathbb{V}[x]] \tag{\textit{value-}β}$$

$${}^{m}\mathbb{R}[{}^{n}\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to N_i\}] \mathrel{\lessapprox} {}^{m+n+1}\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to {}^{m}\mathbb{R}[N_i]\} \tag{\textit{case-}$\mathbb{R}$}$$

$${}^{m}\mathbb{R}[\mathsf{let}\ \{{}^{\vec{n}}\vec{x} = \vec{M}\}\ \mathsf{in}\ N] \mathrel{\lessapprox} {}^{m\curlyvee}\mathsf{let}\ \{{}^{\vec{n}}\vec{x} = \vec{M}\}\ \mathsf{in}\ {}^{m}\mathbb{R}[N] \tag{\textit{let-}$\mathbb{R}$}$$

$$\mathsf{let}\ \{{}^{n}x = M\}\ \mathsf{in}\ x \mathrel{\lessapprox} {}^{n\curlywedge}M, \quad \text{if } x \notin \mathsf{FV}(M) \tag{\textit{inline}}$$

$$\mathsf{let}\ \{\vec{x} = \vec{M}\}\ \mathsf{in}\ N \mathrel{\lessapprox} N, \quad \text{if } \vec{x} \not\downarrow \mathsf{FV}(N) \tag{\textit{gc}}$$

$${}^{\Sigma(\vec{l}+1)\curlywedge}\mathsf{let}\ \{{}^{\vec{l}}\vec{x} = \vec{L}\}\ \mathsf{in}\ \mathsf{let}\ \{{}^{\vec{m}}\vec{y} = \vec{M}\}\ \mathsf{in}\ N \mathrel{\lessapprox} {}^{\Sigma(\vec{l}+1)\curlywedge}\mathsf{let}\ \{{}^{\vec{l}}\vec{x} = \vec{L}, {}^{\vec{m}}\vec{y} = \vec{M}\}\ \mathsf{in}\ N \tag{\textit{let-flatten}}$$

$$\mathsf{let}\ \{{}^{l+\Sigma(\vec{m}+1)}x = \mathsf{let}\ \{{}^{\vec{m}}\vec{y} = \vec{L}, {}^{\vec{n}}\vec{z} = \vec{M}\}\ \mathsf{in}\ N\}\ \mathsf{in}\ N' \mathrel{\gtrapprox} \mathsf{let}\ \{{}^{l}x = \mathsf{let}\ \{{}^{\vec{n}}\vec{z} = \vec{M}\}\ \mathsf{in}\ N, {}^{\vec{m}}\vec{y} = \vec{L}\}\ \mathsf{in}\ N' \tag{\textit{let-let-}1}$$

$$\mathsf{let}\ \{{}^{l}x = \mathsf{let}\ \{{}^{\vec{m}}\vec{y} = \vec{L}, {}^{\vec{n}}\vec{z} = \vec{M}\}\ \mathsf{in}\ N\}\ \mathsf{in}\ N' \lessapprox \mathsf{let}\ \{{}^{l}x = \mathsf{let}\ \{{}^{\vec{n}}\vec{z} = \vec{M}\}\ \mathsf{in}\ N, {}^{\vec{m}}\vec{y} = \vec{L}\}\ \mathsf{in}\ N' \tag{\textit{let-let-}2}$$
$$\text{if } \vec{y} \in \mathsf{FV}(\mathsf{let}\ \{{}^{\vec{n}}\vec{z} = \vec{M}\}\ \mathsf{in}\ N)$$

Figure 3: Strong improvement laws.

$${}^{m\curlywedge n\curlywedge}M \mathrel{\lessapprox} {}^{n\curlywedge m\curlywedge}M \tag{1}$$

$${}^{m\curlywedge n\curlywedge}M \mathrel{\lessapprox} {}^{n\curlywedge}M \quad \text{if } m \leq n \tag{2}$$

$${}^{m\curlywedge}\mathsf{let}\ \{{}^{\vec{n}}\vec{x} = \vec{M}\}\ \mathsf{in}\ N \mathrel{\lessapprox} \mathsf{let}\ \{{}^{n}x = M\}\ \mathsf{in}\ N \qquad \text{if } m+1 \leq \Sigma(\vec{n}+1) \text{ and } \vec{x} \in \mathsf{FV}(N) \tag{3}$$

$${}^{m+\Sigma(\vec{n}+1)\curlywedge}\mathsf{let}\ \{{}^{\vec{n}}\vec{x} = \vec{M}\}\ \mathsf{in}\ N \mathrel{\lessapprox} \mathsf{let}\ \{{}^{n}x = M\}\ \mathsf{in}\ {}^{m\curlywedge}N \quad \text{if } \vec{x} \in \mathsf{FV}(N) \tag{4}$$

$${}^{m\curlywedge}\mathbb{R}[M] \mathrel{\lessapprox} \mathbb{R}[{}^{m\curlywedge}M] \tag{5}$$

$${}^{m\curlyvee n\curlyvee}M \mathrel{\lessapprox} {}^{n\curlyvee m\curlyvee}M \tag{6}$$

$${}^{m\curlyvee n\curlyvee}M \mathrel{\lessapprox} {}^{n\curlyvee}M \quad \text{if } m \leq n \tag{7}$$

$${}^{m\curlyvee n}\mathbb{R}[M] \mathrel{\lessapprox} {}^{n}\mathbb{R}[M] \quad \text{if } m \leq n \tag{8}$$

$${}^{m+n+1\curlyvee n}\mathbb{R}[M] \mathrel{\lessapprox} {}^{n}\mathbb{R}[{}^{m\curlyvee}M] \tag{9}$$

$${}^{m\curlyvee}\mathsf{let}\ \{{}^{\vec{n}}\vec{x} = \vec{M}\}\ \mathsf{in}\ N \mathrel{\lessapprox} \mathsf{let}\ \{{}^{\vec{n}}\vec{x} = \vec{M}\}\ \mathsf{in}\ {}^{m\curlyvee}N \quad \text{if } \vec{x} \in \mathsf{FV}(N) \tag{10}$$

$${}^{m\curlywedge n\curlyvee}M \mathrel{\lessapprox} {}^{n\curlyvee m\curlywedge}M \tag{11}$$

Figure 4: Spike and ballast algebra.

asymptotically worse space behaviour *even when it is work-safe*. For example,

$$\begin{array}{ll} \mathsf{let} & x = \mathit{count}\,\mathbf{k} \\ & y = \mathit{head}\,x \\ \mathsf{in} & y + \mathit{traverse}\,x + (\lambda z.1)\,y \\ \Rightarrow & \\ \mathsf{let} & x = \mathit{count}\,\mathbf{k} \\ & y = \mathit{head}\,x \\ \mathsf{in} & \mathit{head}\,x + \mathit{traverse}\,x + (\lambda z.1)\,y \end{array}$$

where *count* is the function from Section 5 which produces the list of integers counting down from its argument to zero. The inlining above is work-safe but not space safe: the left hand side can run in constant space but the right hand side requires heap space proportional to $\mathbf{k}$. This example is enough to show that the "used at most once" criteria alone is not enough to guarantee space safety (although it does, as expected, guarantee work-safety).

In the remainder of this section we will strengthen the use-once criteria so that it is enough to guarantee both work and space safety, and outline our approach to establishing these results.

The program analysis by Gustavsson [Gus98, Gus99] has already been proven to satisfy the stronger criteria and we believe that the analyses by Turner et al [TWM95] and Wansbrough and Peyton-Jones [WJ99] do so as well. However the "used at most once" analyses by Sestoft [Ses91], Marlow [Mar93] and Mogensen [Mog97] do not satisfy the additional criteria, and we believe that as a result their analyses do not provide conditions for space-safe inlining.

7.1 Affine-linear bindings

Our approach is to extend the language with affine-linear "use-once" bindings, equipped with a direct operational interpretation. Such a binding will be written $x \stackrel{\bullet}{=} M$ and we will write $\hat{M}$ for the term obtained by removing the linear annotations from bindings in M. Linear bindings are transparent to all computation rules except *Lookup*. We now have two lookup rules, one for ordinary bindings, and one for linear bindings:

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

$$\langle\, \Gamma\{x \stackrel{\bullet}{=} M\},\ x,\ S \,\rangle \to \langle\, \Gamma,\ M,\ S \,\rangle \quad \text{if } x \notin \mathsf{FV}(\Gamma, M, S) \qquad (\mathit{Lookup}\text{-}\bullet)$$

The rule for linear bindings looks up the binding without pushing an update marker. Without the side condition $x \notin \mathsf{FV}(\Gamma, M, S)$ this could lead to an open configuration, which in a lower-level implementation would correspond to the creation of a dangling pointer – something which could crash an unwary garbage collector. The side condition prevents such a situation, and instead the computation gets stuck. Note then that with this semantics the computation may get stuck due to a linear binding even though the binding is not used more than once. For example,

$$\mathsf{let}\ \{x \stackrel{\bullet}{=} 1 + 2\}\ \mathsf{in}\ x + (\lambda y.1)\,x$$

gets stuck since when x is going to be used there is a remaining (semantically dead) occurrence of x in $(\lambda y.1)\,x$. If a term do not get stuck due to a linear binding then its time and space behaviour is closely coupled to the term obtained by removing the linear annotations.

Proposition 7.1 *If $M\Downarrow$ then,*

$$\hat{M}\Downarrow^{(h,s)} \text{ in } t \text{ steps} \Longrightarrow M\Downarrow^{(h,s)} \text{ in} \leq t \text{ steps}$$
$$M\Downarrow^{(h,s)} \text{ in } t \text{ steps} \Longrightarrow \hat{M}\Downarrow^{(h,s)} \text{ in } t \text{ steps}$$

The proof is straightforward since the computations are lockstep, and whenever the computation of M applies a linear-lookup step, the fact that M does not become stuck implies that in the corresponding lookup step in $\hat{M}$, the update marker can be immediately garbage collected.

7.2 Inlining linear bindings

This crucial strengthening of the "used at most once" criteria will allow us to establish space safety. We have the following strong improvements (where $\gtrsim_{\mathrm{work}}$ is the improvement relation for time as defined in [MS99]).

Theorem 7.2

$$\begin{array}{l} \mathsf{let}\ \{{}^m x \stackrel{\bullet}{=} M, {}^{\vec{n}}\vec{y} = \vec{\mathbb{D}}[x]\}\ \mathsf{in}\ \mathbb{C}[x] \\ \quad \gtrsim_{work} \mathsf{let}\ \{{}^m x \stackrel{\bullet}{=} M, {}^{\vec{n}}\vec{y} = \vec{\mathbb{D}}[M]\}\ \mathsf{in}\ \mathbb{C}[M] \end{array}$$

$$\begin{array}{l} \mathsf{let}\ \{{}^m x \stackrel{\bullet}{=} M, {}^{\vec{n}}\vec{y} = \vec{\mathbb{D}}[x]\}\ \mathsf{in}\ \mathbb{C}[x] \\ \quad \gtrsim \mathsf{let}\ \{{}^m x \stackrel{\bullet}{=} M, {}^{\vec{n}}\vec{y} = \vec{\mathbb{D}}[M]\}\ \mathsf{in}\ \mathbb{C}[M] \end{array}$$

The reason that we include the time-improvement law here is that once we add linear bindings to the language, even the time-improvement theory must be free-variable aware, and thus the proofs for space and time become very closely related. The theorem is proved in a manner similar to the *value*-β, and does not present any particular difficulties.

It is easy to show that adding linear bindings to the language is not a conservative extension and thus the space improvement laws from section 5 and 6 do not, a priori, hold in the extended language. However, in anticipation of this application, we have established all the laws in the language with both linear bindings and ballast. This allows us to reason about the space safety of transformations entirely in the language with linear bindings. For the same reason the time improvement laws of [MS99] do not, a priori, hold. We have not yet determined which of the time-improvement laws carry over, but we believe that those which are space improvements will be time improvements in the extended language. The intuition behind this is again that both space and (our form of) linear bindings are sensitive to free variables.

7.3 Work and Space-Safety of Linear Type Systems

It is perhaps not immediately apparent how the untyped notion of linear binding and Theorem 7.2 can be used to argue the work and space safety of linear type systems. Linear type systems are global program analyses; they can take the context in which a term occurs into account. Not surprisingly the established results for these type systems involve the whole program.

Proposition 7.3 *If P is a program (a closed term with no linear bindings) and P' is obtained from P by replacing bindings with linear bindings whenever one of the linear type systems of [TWM95, Gus98, Gus99, WJ99] claims the variable has linear type, then*

$$P\Downarrow \implies P'\Downarrow$$

The proof of this claim is essentially the subject reduction property of the respective type systems, which implies that well-typed programs cannot become stuck due to the configuration becoming open. [The latter point is only proved explicitly in [Gus98, Gus99]. In [TWM95] and [WJ99] the result is established for the weaker notion of "used at most once" but we believe it is straightforward to strengthen their results.]

The property that we wish to prove is a similarly global property, rather than a context-insensitive improvement relation:

Theorem 7.4 *If P is a program (a closed term with no linear bindings), such that*

$$P\Downarrow^{(h,s)} \textit{in } t \textit{ steps},$$

and Q is obtained from P by inlining some of the bindings which are linear according to one of the linear type systems, then

$$Q\Downarrow^{(h,s)} \textit{in} \leq t \textit{ steps},$$

PROOF. Suppose that $P\Downarrow^{(h,s)}$in t steps, and that P' is the result of replacing all bindings which have linear type (according to one of the type systems) with actual linear bindings. Suppose further that Q' is the result of inlining some linear bindings in P', and that Q is the result of removing all linear annotations from Q'. From Proposition 7.3 we know that $P'\Downarrow$ so by proposition 7.1

$$P'\Downarrow^{(h,s)} \text{ in} \leq \text{t steps}.$$

Now since Q' is obtained from P' by inlining linear bindings, from Theorem 7.2 and the definitions of improvement, it follows that

$$Q'\Downarrow^{(h,s)}\text{in} \leq t \text{ steps}.$$

Finally, since $\hat{Q'} = Q$, by proposition 7.1 we have $Q\Downarrow^{(h,s)}\text{in} \leq t$ steps as required. □

8 Conclusions and Future Work

We have presented a surprisingly[5] rich operational theory for the space use of call-by-need programs, based on a space improvement ordering on programs. The theory allows one to argue that transforming a program fragment M into N is space safe in the sense that replacing M by N in any program can never lead to asymptotically worse space (heap or stack) behaviour.

As a first application of the theory, we have proved that inlining of affine-linear bindings (as introduced by a certain class of "used-once" type-systems) is work- and space-safe.

A key problem which remains to be solved is to establish some principles for reasoning about recursive functions. We have briefly considered whether we can establish the improvement theorem or improvement induction and we have good reasons to believe that it should be possible.

Another item on our work list is to add lazy patterns and strict lets to our language since these features seem to be crucial when defining space efficient versions of some functions.

[5] At least, suprising to us!

An important piece of future work is to try to apply the space improvement laws to larger examples. This will surely reveal the need for more laws, for example laws concerning strictness and evaluation order.

Another interesting direction for future work would be to consider the time and space safety of a larger-scale program transformation, such as deforestation [Wad90]. This should be possible thanks to work and space safe inlining.

References

[Bar81] H. Barendregt. *The Lambda Calculus*. North Holland, 1981.

[BLR96] Z.-E.-A. Benaissa, P. Lescanne, and K. H. Rose. 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*, pages 393–407. Springer-Verlag, 1996.

[GP98] A. D. Gordon and A. M. Pitts, editors. *Higher Order Operational Techniques in Semantics*. Publications of the Newton Institute. Cambridge University Press, 1998.

[Gus98] J. Gustavsson. A type based sharing analysis for update avoidance and optimisation. In *Proc. ICFP'98, the 3rd ACM SIGPLAN International Conference on Functional Programming*, pages 39–50, September 1998.

[Gus99] Jörgen Gustavsson. A Type Based Sharing Analysis for Update Avoidance and Optimisation. Licentiate thesis, May 1999.

[Jon92] Richard Jones. Tail recursion without space leaks. *Journal of Functional Programming*, 2(1):73–79, January 1992.

[Lau93] J. Launchbury. A natural semantics for lazy evaluation. In *Proc. POPL'93, the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 144–154. ACM Press, January 1993.

[Mar93] S. Marlow. Update avoidance analysis by abstract interpretation. In *Proc. 1993 Glasgow Functional Programming Workshop*, Workshops in Computing. Springer-Verlag, August 1993.

[Mar95] Simon Marlow. *Deforestation for Higher-Order Functional Programs*. PhD thesis, University of Glasgow, 1995.

[MFH95] Greg Morrisett, Matthias Felleisen, and Robert Harper. Abstract Models of Memory Management. In *Proc. Int'l Conf. on Functional Programming Languages and Computer Architecture (FPCA'95)*, pages 66–77. ACM Press, June 1995.

[MH98] Greg Morriset and Robert Harper. Semantics of memory management for polymorphic languages. In Gordon and Pitts [GP98], pages 175–226.

[Mil77] R. Milner. Fully abstract models of the typed λ-calculus. *Theoretical Computer Science*, 4:1–22, 1977.

[Mog97] T. Mogensen. Types for 0, 1 or many uses. In *Proc. Workshop on Implementation of Functional Languages (Draft)*, 1997.

[MS99] Andrew Moran and David Sands. 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*, pages 43–56. ACM Press, January 1999.

[PJ87] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[PJ92] S. L. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2), April 1992.

[PJPS96] S. Peyton Jones, W. Partain, and A. Santos. Let-floating: moving bindings to give faster programs. In *Proc. ICFP'96, the 1st ACM SIGPLAN International Conference on Functional Programming*, pages 1–12. ACM Press, May 1996.

[PJS98] S. Peyton Jones and A. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1–3):3–47, 1998.

[Röj95] Niklas Röjemo. *Garbage collection, and memory efficiency, in lazy functional languages*. PhD thesis, Chalmers Tekniska Högskola, 1995.

[Ros96] K. H. Rose. *Operational Reduction Models for Functional Programming Languages*. PhD thesis, DIKU, University of Copenhagen, Denmark, February 1996. available as DIKU report 96/1.

[RR96] Colin Runciman and Niklas Röjemo. New Dimensions in Heap Profiling. *Journal of Functional Programming*, 6(4):587–620, 1996.

[RW93] Colin Runciman and David Wakeling. Heap Profiling of Lazy Functional Programs. *Journal of Functional Programming*, 3(2):217–245, April 1993.

[San91] D. Sands. Operational theories of improvement in functional languages (extended abstract). In *Proc. 1991 Glasgow Functional Programming Workshop*, Workshops in Computing Series, pages 298–311. Springer-Verlag, August 1991.

[San95] D. Sands. A naïve time analysis and its theory of cost equivalence. *Journal of Logic and Computation*, 5(4):495–541, 1995.

[San96] D. Sands. Total correctness by local improvement in the transformation of functional program. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(2):175–234, March 1996.

[San98a] D. Sands. Computing with contexts: A simple approach. In A. D. Gordon, A. M. Pitts, and C. L. Talcott, editors, *Proc. HOOTS II, the 2nd Workshop on Higher Order Operational Techniques in Semantics*, volume 10 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers B.V., 1998. at http://www.elsevier.nl/cas/tree/store/tcs/free/noncas/pc/menu.htm.

[San98b] D. Sands. Improvement theory and its applications. In Gordon and Pitts [GP98], pages 275–306.

[Ses91] P. Sestoft. *Analysis and Efficient Implementation of Functional Programs*. PhD thesis, DIKU, University of Copenhagen, Denmark, October 1991.

[Ses97] P. Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, May 1997.

[Spa93] Jan Sparud. Fixing Some Space Leaks without a Garbage Collector. In *Proc. 6th Int'l Conf. on Functional Programming Languages and Computer Architecture (FPCA'93)*, pages 117–122. ACM Press, June 1993.

[TWM95] D. N. Turner, P. Wadler, and C. Mossin. Once upon a type. In *Proc. FPCA'95, ACM Conference on Functional Programming Languages and Computer Architecture*, pages 1–11. ACM Press, June 1995.

[Wad87] P. Wadler. Fixing Some Space Leaks with a Garbage Collector. *Software Practice and Experience*, September 1987.

[Wad90] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

[WJ99] Keith Wansbrough and Simon Peyton Jones. Once upon a polymorphic type. In *Proc. POPL'99, the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, January 1999.