Possibilities and Limitations of Call-by-Need Space Improvement *

Jörgen Gustavsson
www.cs.chalmers.se/~gustavss

David Sands
www.cs.chalmers.se/~dave

ABSTRACT

Innocent-looking program transformations can easily change the space complexity of lazy functional programs. The theory of *space improvement* seeks to characterise those local program transformations which are guaranteed never to worsen asymptotic space complexity of any program. Previous work by the authors introduced the space improvement relation and showed that a number of simple local transformation laws are indeed space improvements. This paper seeks an answer to the following questions: is the improvement relation inhabited by interesting program transformations, and, if so, how might they be established? We show that the asymptotic space improvement relation is semantically badly behaved, but that the theory of *strong space improvement* possesses a fixed-point induction theorem which permits the derivation of improvement properties for recursive definitions. With the help of this tool we explore the landscape of space improvement by considering a range of classical program transformations.

1. INTRODUCTION

Consider the following equivalence for a pure functional language: $x+y = y+x$. How does this affect the space complexity of a program? Of course, it depends on the program – and the language. In a lazy functional language the transformation is not *space safe*; there are programs for which this innocent-looking transformation will change their space complexity. Now consider the following family of Haskell programs, indexed by some integer n:

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

If addition is evaluated from left-to-right then this program runs in constant space. First x is evaluated to obtain 1, then y is evaluated, which involves constructing and traversing the entire list $[1..n]$. Fortunately, the combination of lazy evaluation, tail recursion and garbage collection guarantees that as this list is constructed and traversed it can also be garbage collected, and thus the computation requires only constant space. But if $x+y$ is replaced by $y+x$ 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 live via the occurrence of xs in the body of 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. The example is taken from our previous work [10], and serves to

- illustrate the subtleties of space behaviours of lazy functional programs[1], and to
- motivate the study of the class of program transformations which *are* space-safe.

To this end, we introduced 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.

Space improvement guarantees, by construction, that it never worsens the space complexity of a program. But given the previous example, it is not immediately clear that there are *any* interesting transformations which are space improvements. In the previous work we showed that there are indeed transformation laws which are improvements. [2] For example, the *beta-var* transformation between $(\lambda x.M)\,y$ and $M[y/x]$ is shown to be a space improvement.

But the previous work did not provide any principles for establishing properties of recursive functions (other than a general context lemma), and did not yield any improvement examples beyond those obtainable by composing simple laws. This paper seeks an answer to the following questions: is the improvement relation inhabited by interesting program transformations, and, if so, how might they be established? For example, is the associativity property of list

*An extended version of this article is available from www.cs.chalmers.se/~gustavss. Authors address: Department of Computing Science, Chalmers Univertity of Technology and Göteborg University.

[1] Under call-by-value there is no problem: both programs have $\mathcal{O}(n)$ space complexity – although one can construct similar examples for call-by-value.

[2] We also showed how space improvement could be used indirectly to verify the space-safety of inlining transformations which make use of certain single-usage type systems, but this application is somewhat orthogonal to the concerns of the present paper, since it deals with a global program transformation.

ICFP'01, September 3-5, 2001, Florence, Italy.

concatenation a space improvement in either direction? Are typical tail recursion optimisations space safe? Although the asymptotic space improvement relation, *weak improvement*, is semantically badly behaved, the theory of *strong space improvement* possesses a fixed-point induction theorem which permits the derivation of improvement properties for recursive definitions. With the help of this tool we explore the landscape of space improvement by considering a range of classical program transformations, and uncovering a number of fundamental limitations to what can be achieved by local improvement.

Overview The remainder of the article is organised as follows. **Section 2** gives the syntax and operational semantics of our language. **Section 3** defines what we mean by the space-use of programs, in terms of a definition of garbage collection for abstract-machine configurations. **Section 4** defines the main improvement relation, *weak improvement*, and presents the basic laws and properties of this relation. **Section 5** describes a finer-grained improvement relation, *strong improvement*, and establishes a fixed-point induction principle. **Section 6** applies the theory to investigate a range of transformations. **Section 7** describes related work and concludes.

2. OPERATIONAL SEMANTICS

Our language is an untyped lambda calculus with recursive lets, structured data, case expressions, bounded 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:

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

The syntactic restriction is now rather standard, following its use in core language of the Glasgow Haskell compiler, *e.g.*, [22, 23], and in [14, 28]. In examples we will sometimes use unrestricted application MN as syntactic sugar for $\mathsf{let}\ \{x = N\}\ \mathsf{in}\ M\,x$ where x is a fresh variable. Similarly for constructor expressions.

All constructors have a fixed arity, and are assumed to be saturated. By $c\,\vec{x}$ we mean $c\,x_1 \cdots x_n$. Throughout, x, y, z etc., 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, $\mathsf{case}\ M\ \mathsf{of}\ \{c_i\,\vec{x}_i \to N_i\}$ is a shorthand 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 $\{c_i\,\vec{x}_i \to N_i\}$.

Our integers are bounded (i.e., for an integer n, $MININT \leq n \leq MAXINT$) so that they can be represented in constant space. For simplicity, no exception occurs at overflow. Instead the result wraps as in e.g., C. 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).

2.1 The Abstract Machine

The semantics presented in this section is essentially Sestoft's "mark 1" abstract machine for laziness [28]. 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.

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:

$$\begin{array}{rcl} R & ::= & [\cdot]\,x \mid \mathsf{case}\ [\cdot]\ \mathsf{of}\ \{c_i\,\vec{x}_i \to N_i\} \mid \mathsf{seq}\,[\cdot]\,M \\ & & [\cdot] + M \mid \mathsf{add}_n[\cdot] \mid \mathsf{iszero}\,[\cdot] \end{array}$$

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 $\mathrm{dom}(\Gamma, S)$. 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 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.

3. SPACE USE AND GARBAGE COLLECTION

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

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

$$\langle\, \Gamma,\, \mathsf{let}\ \Gamma'\ \mathsf{in}\ N,\, S\,\rangle \to \langle\, \Gamma\Gamma',\, N,\, S\,\rangle \qquad (\mathit{Letrec})$$

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

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

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

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

$$\mathsf{seq}\, V\, M \rightsquigarrow M$$

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

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

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

Figure 1: Abstract machine semantics

A desired property of our model of space-use is that it is asymptoticly correct with respect to actual implementations. Unfortunately, different abstract machines and garbage collection strategies differ in their asymptotic space behaviour. 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 adapted to any reasonable model. In [9] we discuss some of the subtle ways in which different abstract machines and implementations described in the literature differ from our model and each other. Bakewell and Runciman [2] focus on techniques for comparing different evaluators.

Another point of dispute is whether to distinguish between heap and stack space. Many implementations allocate separate memory for the heap and the stack, but in principle the stack and the heap can share the same memory. So, should a transformation which trades heap for stack, or vice versa, be rejected? And do such transformation show up "in practice"? We focus mainly on a theory which keeps stack and heap usage separate. However, we will see examples of transformations which usefully trade stack for heap.

3.1 Measuring space

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 a (possibly renamed) subexpression of the original program, (a property of the semantics sometimes called *semi-compositionality*) so counting it as occupying one unit of space gives a measure which is within a constant factor (depending only on the program size) of the actual space used. Integers are an exception to this claim, but recall that our integers are bounded so they can also be represented in a constant amount of space.

We measure stack space by simply counting the number of elements on the stack, 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. The size of a configuration, written $|\langle\, \Gamma,\, M,\, S\,\rangle|$ is a pair (h, s) where h and s is the amount of heap and stack respectively occupied by the the configuration.

3.2 Garbage collection

We cannot reason about space usage without modelling garbage collection. During a computation, garbage collection allows us to decrease the amount of space used by a configuration. It is modelled simply by the removal of any number of bindings and update markers from the heap and the stack respectively, *providing that the configuration remains closed.*

DEFINITION 3.1 (GC). *Garbage collection can be applied to a closed configuration* $\langle\, \Gamma,\, M,\, S\,\rangle$ *to obtain* $\langle\, \Gamma',\, M,\, S'\,\rangle$, *written* $\langle\, \Gamma,\, M,\, S\,\rangle \geqslant \langle\, \Gamma',\, M,\, S'\,\rangle$ *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 [20]. 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.

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

DEFINITION 3.2 (CONVERGENCE IN FIXED SPACE).

1. $\Sigma \to^{(h,s)} \Sigma' \stackrel{\text{def}}{=} \Sigma \to \Sigma'$ *and* $|\Sigma| \leq (h, s)$.

2. $\twoheadrightarrow^{(h,s)} \stackrel{\text{def}}{=}$ *the reflexive and transitive closure of the relational composition of* $\to^{(h,s)}$ *and* $\geqslant$.

3. $\langle\, \Gamma,\, M,\, S\,\rangle \Downarrow_{(h,s)} \stackrel{\text{def}}{=} \exists \Delta, V.$
   $\langle\, \Gamma,\, M,\, S\,\rangle \twoheadrightarrow^{(h,s)} \langle\, \Delta,\, V,\, \epsilon\,\rangle$ *and* $|\langle\, \Delta,\, V,\, \epsilon\,\rangle| \leq (h, s)$.

4. $M \Downarrow_{(h,s)} \stackrel{\text{def}}{=} \langle\, \emptyset,\, M,\, \epsilon\,\rangle \Downarrow_{(h,s)}$.

We read $M \Downarrow_{(h,s)}$ as M can converge within (h, s) space, i.e., the maximum heap, and stack is less than or equal to h and s respectively.

4. WEAK 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. We will write $\mathbb{C}[M]$ for filling the holes in $\mathbb{C}$ with M and we will write $\mathsf{CV}(\mathbb{C})$ for the variables which may be captured by filling the holes in $\mathbb{C}$.

DEFINITION 4.1 (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 $\mathbb{C}$, σ such that $\mathbb{C}[M\sigma]$ and $\mathbb{C}[N\sigma]$ are closed,*

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

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 $N \gtrapprox M$. In [10] we established a number of properties and laws for weak improvements. We recite two of them here.

PROPOSITION 4.1 (PRECONGRUENCE[10]). *$\gtrapprox$ is a* precongruence *– i.e., it is a transitive and reflexive relation which is preserved by contexts and substitutions.*

The following property is fundamental, and highlights the significance of free variables in this theory:

THEOREM 4.2 (FREE VARIABLE PROPERTY[10]). *$M \gtrapprox N \implies \mathsf{FV}(M) \supseteq \mathsf{FV}(N)$.*

Free variables are significant because they can have an effect on the space usage of a program even when they are semantically "dead code".

4.1 Limitations of Weak Improvement

A standard result for any operational theory is a *context lemma* [16]. A context lemma in this case would establish that to prove that M is weakly 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.

Despite our efforts, in [10] we were not able to prove the context lemma. The reason is that the context lemma, as we envisage it, does not hold for weak improvement:

THEOREM 4.3 (FAILURE OF THE CONTEXT LEMMA). *There exist terms M and N with $\mathsf{FV}(M) \supseteq \mathsf{FV}(N)$ and a linear function f such that for every Γ, S and σ,*

$$\langle \Gamma,\, M\sigma,\, S\rangle\Downarrow_{(h,s)} \implies \langle \Gamma,\, N\sigma,\, S\rangle\Downarrow_{(f(h),f(s))},$$

but where $M \not\gtrapprox N$.

The proof can be found in [9].

4.2 Fixed Point Approximation

It is typical in semantics to characterise recursion in terms of the "finite approximations" of recursive definitions. This approach is built in to the Scott-style denotational semantics approach where recursion is modelled by a least fixed point construction. The essence of this approach can be expressed in a purely operational setting. See e.g. [29, 15].

The natural formulation of the least fixed-point property also fails to hold for weak improvement (a precise formulation can be found in [9]). Intuitively the reason for this failure is that once we fix an unwinding we may be able to find a constant factor that bounds the space difference, but we can't find a single constant factor that works for *all* unwindings. Fortunately it does hold for a stronger notion of improvement introduced in the next section.

5. STRONG IMPROVEMENT

The failure of the context lemma and the fixed-point approximation property give a very concrete motivation for studying a stronger relation, *strong improvement*:

DEFINITION 5.1 (STRONG IMPROVEMENT). *M is* strongly improved by N, *written $M \gtrsim N$, if for all $\mathbb{C}$, σ such that $\mathbb{C}[M\sigma]$ and $\mathbb{C}[N\sigma]$ are closed,*

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

We write $M \lesssim N$ to mean that $M \gtrsim N$ and $N \gtrsim M$.

Although the definition of strong improvement is somewhat arbitrary – since it deals with constant factors for a high-level abstract machine – it provides a practical means to establish weak improvement laws, since whenever $M \gtrsim N$ then clearly $M \gtrapprox N$. In this section we present some of the basic properties of strong improvement, and our key technical result: a fixed-point approximation theorem for establishing improvement properties of recursive definitions.

We begin with some technical developments which are necessary to support reasoning about strong improvement. For strong improvement we have also established a *context lemma* [16]: 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 5.1 (CONTEXT LEMMA [10]). *For all M and N such that $\mathsf{FV}(M) \supseteq \mathsf{FV}(N)$, if for all Γ, S and σ, $\langle \Gamma,\, M\sigma,\, S\rangle\Downarrow_{(h,s)} \implies \langle \Gamma,\, N\sigma,\, S\rangle\Downarrow_{(h,s)}$ then $M \gtrsim N$.*

With help of the context lemma we have established a set of basic laws of strong improvement. The laws were presented in [10] and we will not reproduce them here.

5.1 The Space Gadgets

The *space gadgets* are syntactic means to represent and control the space properties of terms. They play a crucial role in strong improvement calculations. We describe each gadget in turn.

Dummy References The use of dummy references allows one to make assertions about, and to modify the liveness properties of variables. To this end we introduce the following notational extension, terms of the form ^{X}M where X is a multiset of variables. The construct is representable in the language and is defined thus

$$^{\{\vec{x}\}}M \stackrel{\text{def}}{=} \mathsf{let}\ \{\vec{y} = \vec{x}\}\ \mathsf{in}\ M \qquad \text{where } \vec{y} \text{ are fresh.}$$

Hence ^{X}M behaves as M but in addition holds on to the variables in X until the evaluation of M starts. If X would range over a set, rather than a multiset, then the notation would not be well defined with respect to substitution.

Dummy references can express certain liveness properties. For example, if $\mathbb{C}[M] \trianglerighteq \mathbb{C}[^{\{y\}}M]$ then we know that y is still live at the occurrence of M. Among other things we will use dummy references to control the life time of *dummy bindings*, i.e., bindings which play no rôle in the term but to take up space. To add dummy bindings is harmless in the weak theory as long as their life time is coupled to another binding.

LEMMA 5.2 (DUMMY BINDING INTRODUCTION).
$\mathsf{let}\ \{x = M\}\ \mathsf{in}\ N \mathrel{\lhd\!\rhd\!\!\!\approx} \mathsf{let}\ \{z = \Omega, x = ^{\{z\}}M\}\ \mathsf{in}\ N, z$ *fresh*

Spikes Spikes are amortisation device which allow us to represent a very short-lived space usage – a spike in the space-usage profile. Spikes come in two varieties, *heap* spikes and *stack* spikes.

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. To see how stack spikes are used, consider how one might prove the (restricted) beta-reduction cost equivalence $(\lambda x.M)\,y \mathrel{\lhd\!\rhd\!\!\!\approx} M[y/x]$. To do this we use strong improvement. The context lemma makes it easy to establish that $(\lambda x.M)\,y \trianglerighteq M[y/x]$. 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. To prove that $M[y/x] \trianglerighteq (\lambda x.M)\,y$ we use the context lemma to show that

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

All that is left is to establish that spike introduction is harmless in the weak theory:

LEMMA 5.3 (SPIKE INTRODUCTION [10]). $M \mathrel{\lhd\!\rhd\!\!\!\approx} ^{\curlyvee}M$.

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.

$$^{\curlywedge}M \stackrel{\text{def}}{=} \mathsf{let}\ x = \Omega\ \mathsf{in}\ ^{\{x\}}M \qquad \text{where } x \text{ is fresh}$$

Heap spikes are also harmless in the weak theory, i.e., $M \mathrel{\lhd\!\rhd\!\!\!\approx} ^{\curlywedge}M$.

Weights The most complex gadgets are the weights[3]. Weights are more involved because they cannot be defined in terms of existing language constructs, but must be added as a collection of term-annotations with a specially defined space-semantics.

In our definition of space use we count every entity on the stack or on the heap as occupying exactly one unit of space, a choice justified by our desire to ultimately reason about asymptotic behaviour. But it turns out to be crucial to be able to selectively choose exactly how much space each entity shall account for – i.e., what the *weight* of the entity should be. Consider, for example, the following weak equivalence law for reduction contexts:

$$R[\mathsf{case}\ M\ \mathsf{of}\ \{pat_i{\to}N_i\}] \mathrel{\lhd\!\rhd\!\!\!\approx} \mathsf{case}\ M\ \mathsf{of}\ \{pat_i{\to}R[N_i]\}$$

It is not a strong space equivalence since the left hand side takes up more space: while M is being evaluated, both R and the case-alternatives take up stack space (2 units of space). In the right hand side, while M is being evaluated there is just a single set of case alternatives (1 unit of stack space). We can compensate for this, and simplify our calculations, if we count the case in the right hand side as occupying two units of stack, which we denote by the following weight annotation:

$$R[\mathsf{case}\ M\ \mathsf{of}\ \{pat_i{\to}N_i\}] \mathrel{\lhd\!\rhd\!\!\!\approx} {}^{2}\mathsf{case}\ M\ \mathsf{of}\ \{pat_i{\to}R[N_i]\}$$

This is not the only form of weight, but before we consider further examples we will sketch the semantics of weights.

We will annotate every entity on the heap and the stack with a weight $w \geq 0$. Binding occurrences of variables, including update markers (which are considered to take up both heap and stack space, see Section 3.1) are annotated with two weights, one for the heap and one for the stack. The space consumption of each entity is given by the following:

$$|^{w}_{v}x = M| = (v, 0) \qquad |^{w}R| = (0, w) \qquad |\#^{w}_{v}x| = (v, w)$$

So the upper weight of the binder is the stack weight, incurred when the update marker is on the stack; the lower weight is the heap weight – the size of the binding on the heap.

Note that weights may be zero so we can specify that an entity shouldn't be counted for at all. An entity without a weight annotation will now be taken as shorthand for a weight of 1. The weight on bindings and stack elements originate from annotation in the program. Our annotated term language is

$$\begin{aligned} L, M, N ::= {} & x \mid \lambda x.M \mid {}^{w}(M\,x) \mid c\,\vec{x} \mid {}^{w}(\mathsf{seq}\ M\ N) \\ & \mid n \mid {}^{w_0}(M + {}^{w_1}N) \mid {}^{w}(\mathsf{add}_n M) \mid {}^{w}(\mathsf{iszero}\ M) \\ & \mid \mathsf{let}\ \{^{n_i}_{m_i}x_i = M_i\}_{i\in I}\ \mathsf{in}\ N \\ & \mid {}^{w}(\mathsf{case}\ M\ \mathsf{of}\ \{c_i\,\vec{x}_i{\to}N_i\}). \end{aligned}$$

Weights on binding occurrences of variables are permanent. The only rule which eliminates weights (garbage collection excepted) is

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

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

Of course, weights have no intrinsic interest for programmers – they are a bookkeeping mechanism which we use to syntactically account for certain forms of space usage. As with spikes, a crucial property of weights is that they increase space use in the strong theory but do not change space behaviour by more than a constant factor:

[3] A generalisation of the ballasts from [10].

LEMMA 5.4. *For $v, w > 0$,*

1. $R[M] \lessapprox\gtrapprox {}^{w}R[M]$.
2. $(M + N) \lessapprox\gtrapprox {}^{w}(M +^{v} N)$.
3. $\mathsf{let}\ \Gamma\{x = M\}\ \mathsf{in}\ N \lessapprox\gtrapprox \mathsf{let}\ \Gamma\{{}^{w}_{v}x = M\}\ \mathsf{in}\ N$.

Zero weights or "balloons" play a special role, and must be handled with care. A zero weight permits costs to be hidden. This is very useful in strong improvement calculations since it cuts down significantly on the "noise" of weight bookkeeping. However, adding zero-weights is potentially unsound, since we might end up hiding an asymptotic amount of space usage. In other words, we cannot arbitrarily introduce zero weights in the weak improvement theory (c.f. Lemma 5.4). There are two ways in which we can justify zero-weight introduction. The first is if an entity is short-lived so that it can't affect the asymptotic space behaviour. We will heavily use two instances of this: that the update marker weight of a value binding can be safely ignored and that the weight of a stack frame associated with an application of a known function can be ignored. This is because its lifetime on the stack is only one computation step.

LEMMA 5.5 (BALLOON INTRODUCTION).

1. $\mathsf{let}\ \{x = V\}\ \mathsf{in}\ N \lessapprox\gtrapprox \mathsf{let}\ \{{}^{0}x = V\}\ \mathsf{in}\ N$
2. $(\lambda x.M)\, y \lessapprox\gtrapprox {}^{0}((\lambda x.M)\, y)$

We will use zero-weights on applications often so we introduce an abbreviation and write $M \cdot x$ for ${}^{0}(M\, x)$. The second way that we introduce zero weights is via a "top-level" assumption. It is safe to introduce zero weights to bindings which will not be allocated multiply. Unfortunately this is not a property that holds in all contexts, but is still reasonable. For example, functions from a standard library are typically allocated just once – i.e. they are top level definitions. If a function is defined at top-level then setting heap-weight to zero can have at most a constant factor effect:

LEMMA 5.6. *For every Γ there exist k such that for every M, if* $\mathsf{let}\ \{\Delta\}\ \mathsf{in}\ M{\Downarrow_{(h,s)}}$ *then* $\mathsf{let}\ \{\Gamma\}\ \mathsf{in}\ M{\Downarrow_{(h+k,s)}}$, *where Δ is the result of setting all heap weights on bound variables in Γ to zero.*

Finally, we note that with the help of weights we can increase the size of the stack and heap spikes:

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

$${}^{n\curlywedge}M \stackrel{\text{def}}{=} \mathsf{let}\ {}_{n}x = \Omega\ \mathsf{in}\ {}^{\{x\}}M \qquad \text{where } x \text{ is fresh}$$

Now that we have our space gadgets we will use them both in the next section to develop our main technical result, a fixed-point induction principle, and also to apply it to concrete examples in the following section.

5.2 Fixed-Point Induction

In this section we introduce the least fixed-point property for strong improvement, which will provide the principal tool for reasoning about the relative space behaviour of recursive functions, a simple form of fixed-point induction.

We start at the bottom. A consequence of Theorem 4.2 is that there is no bottom element in the space-ordering relation, since 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 live data, and hence the space. The more free variables a divergent term contains the more space it can retain, and hence the lower in the improvement ordering it sits. This is significant when we define the notion of a chain of finite unwindings of a recursive definition. Usually the first approximation in such a chain is the bottom element but here we need to start from a divergent term with the right amount of free variables.

We are now ready to define precisely the finite unwindings of a recursive definition.

DEFINITION 5.2 (FINITE UNWINDINGS). *Let $\mathbb{V}$ be a value context with at least one occurrence of the hole. We define* $\mathsf{let}\ \{{}^{w}_{v}f = \mathbb{V}[f]\}\ \mathsf{in}\ \mathbb{C}[f^{n}]$ *inductively by the following clauses:*

$$\mathsf{let}\ \{{}^{w}_{v}f = \mathbb{V}[f]\}\ \mathsf{in}\ \mathbb{C}[f^{0}] \stackrel{\text{def}}{=} \mathsf{let}\ \{{}^{w}_{v}f = \mathbb{V}[f]\}\ \mathsf{in}\ \mathbb{C}[{}^{\{f\}}\Omega]$$

$$\mathsf{let}\ \{{}^{w}_{v}f = \mathbb{V}[f]\}\ \mathsf{in}\ \mathbb{C}[f^{n+1}] \stackrel{\text{def}}{=} \mathsf{let}\ \{{}^{w}_{v}f = \mathbb{V}[f]\}\ \mathsf{in}\ \mathbb{C}[{}^{w\curlyvee}\mathbb{V}[f^{n}]]$$

Using the results from [10] it is easy to show that the approximations form an improvement chain: For all $0 \leq i < j$

$$\mathsf{let}\ \{{}^{w}_{v}f = \mathbb{V}[f]\}\ \mathsf{in}\ \mathbb{C}[f^{i}] \gtrsim \mathsf{let}\ \{{}^{w}_{v}f = \mathbb{V}[f]\}\ \mathsf{in}\ \mathbb{C}[f^{j}]$$

and that $\mathsf{let}\ \{{}^{w}_{v}f = \mathbb{V}[f]\}\ \mathsf{in}\ \mathbb{C}[f]$ is an upper bound of the chain - i.e., for all i,

$$\mathsf{let}\ \{{}^{w}_{v}f = \mathbb{V}[f]\}\ \mathsf{in}\ \mathbb{C}[f^{i}] \gtrsim \mathsf{let}\ \{{}^{w}_{v}f = \mathbb{V}[f]\}\ \mathsf{in}\ \mathbb{C}[f].$$

The crucial property of strong improvement is that the relation is continuous with respect to unwinding of recursion. The definition of f is the *least* upper bound of this chain.

THEOREM 5.7 (SYNTACTIC CONTINUITY).
$\mathsf{let}\ \{{}^{w}_{v}f = \mathbb{V}[f]\}\ \mathsf{in}\ \mathbb{C}[f] \gtrsim M \iff$
$\forall n.\, \mathsf{let}\ \{{}^{w}_{v}f = \mathbb{V}[f]\}\ \mathsf{in}\ \mathbb{C}[f^{n}] \gtrsim M$

The theorem forms the basis of the fixed-point induction technique which we spell out at the end of the section. A proof can be found in [9].

5.3 Derivations in Context

We will often express properties which are relative to a fixed set of function definitions. It is cumbersome to carry such definitions in explicit let-terms, so we adopt a useful notation for derivations in context:

DEFINITION 5.3. *We write $\Gamma \vdash M \gtrsim N$ as an abbreviation for the following property: For all Γ', $\mathbb{C}$ and σ, if*

- $\mathrm{dom}\,\Gamma' \cap \mathrm{dom}\,\Gamma = \emptyset$,
- $\mathsf{CV}(\mathbb{C}) \cap (\mathrm{dom}\,\Gamma \cup \mathsf{FV}(\Gamma)) = \emptyset$ *and*
- $\mathrm{dom}\,\sigma \cap (\mathrm{dom}\,\Gamma \cup \mathsf{FV}(\Gamma)) = \emptyset$,

then $\mathsf{let}\ \{\Gamma\Gamma'\}\ \mathsf{in}\ \mathbb{C}[M\sigma] \gtrsim \mathsf{let}\ \{\Gamma\Gamma'\}\ \mathsf{in}\ \mathbb{C}[N\sigma]$.

We will write a derivation

$$\Gamma \vdash M_0 \gtrsim M_1 \gtrsim M_2 \gtrsim \ldots$$

to mean $\Gamma \vdash M_0 \gtrsim M_1$ and $\Gamma \vdash M_1 \gtrsim M_2$ and so on. These contextual judgements satisfy a number of simple properties which facilitate their use.

PROPOSITION 5.8. *The following proof rules are sound:*

- $\dfrac{M \gtrsim N}{\Gamma \vdash M \gtrsim N}$ - $\dfrac{\Gamma \vdash M \gtrsim N \qquad \Gamma \vdash N \gtrsim L}{\Gamma \vdash M \gtrsim L}$
- $\dfrac{\Gamma \vdash M \gtrsim N \qquad \mathrm{dom}\,\Gamma' \cap \mathrm{dom}\,\Gamma = \emptyset}{\Gamma\Gamma' \vdash M \gtrsim N}$
- $\dfrac{\Gamma \vdash M \gtrsim N \qquad \mathsf{CV}(\mathbb{C}) \cap (\mathrm{dom}\,\Gamma \cup \mathsf{FV}(\Gamma)) = \emptyset}{\Gamma \vdash \mathbb{C}[M] \gtrsim \mathbb{C}[N]}$
- $\dfrac{\Gamma \vdash M \gtrsim N \qquad \mathrm{dom}\,\sigma \cap (\mathrm{dom}\,\Gamma \cup \mathsf{FV}(\Gamma)) = \emptyset}{\Gamma \vdash M\sigma \gtrsim N\sigma}$

We also extend our notation for finite unwindings in the obvious way and write $\Gamma \vdash \mathbb{C}[f^n]$ for the n'th unwinding of f (where f is bound in Γ).

With the above notation and properties we have the following simple corollary of syntactic continuity, expressed in an informal natural-deduction style.

COROLLARY 5.9 (FIXED POINT INDUCTION). *The following proof rule is sound:*

$$\frac{\Gamma \vdash \mathbb{C}[f^0] \gtrsim M \qquad \forall n \left(\begin{array}{c} \Gamma \vdash \mathbb{C}[f^n] \gtrsim M \\ \vdots \\ \Gamma \vdash \mathbb{C}[f^{n+1}] \gtrsim M \end{array} \right)}{\Gamma \vdash \mathbb{C}[f] \gtrsim M}$$

That is to say, if we can establish $\Gamma \vdash \mathbb{C}[f^0] \gtrsim M$ *and that under the assumption that* $\Gamma \vdash \mathbb{C}[f^n] \gtrsim M$ *for some arbitrary* n *we can show* $\Gamma \vdash \mathbb{C}[f^{n+1}] \gtrsim M$*, then it holds that* $\Gamma \vdash \mathbb{C}[f] \gtrsim M$.

6. POSSIBILITIES AND LIMITATIONS

Armed with a means to establish improvement properties for recursive functions, in the rest of this paper we will investigate the possibilities and limitations of space improvement.

The requirement is that transformed programs should improve on the space behaviour in all contexts. Are there any interesting transformations which are space improvements? In this section we present examples of some standard program transformations, and show how space improvement can be established using the tools from the previous sections. The results are not all positive; we will also show that there are many transformations that are not space improvements.

Case Study 1: Cyclic Structures

We will start with a very simple and intuitive space improvement which serves, above all else, to illustrate the use of the fixed-point induction method. We will show that the cyclic data structure $xs = x : xs$ improves on the non-cyclic structure that is generated by *repeat* x where *repeat* is defined as

$$repeat = \lambda x.\mathsf{let}\ \{ys = repeat\ x\}\ \mathsf{in}\ x : ys.$$

Using fixed-point induction we will prove a strong improvement property from which the desired weak improvement follows directly.

PROPOSITION 6.1.
$\Gamma \vdash \mathsf{let}\ \{xs = repeat\ x\}\ \mathsf{in}\ M \gtrsim \mathsf{let}\ \{xs = x : xs\}\ \mathsf{in}\ M$
where Γ *contains the definition of repeat.*

PROOF. We proceed by fixed-point induction over the definition of *repeat*. The base case is trivial and has been omitted. The following derivation shows the inductive step, where we have elided steps which only manipulate spikes.

$$\begin{array}{ll} & \Gamma \vdash \mathsf{let}\ \{xs = repeat^{n+1}\ x\}\ \mathsf{in}\ M \\ \equiv & \mathsf{let}\ \{xs = {}^{\curlyvee}(\lambda x.\mathsf{let}\ \{ys = repeat^n\ x\}\ \mathsf{in}\ x : ys)\ x\}\ \mathsf{in}\ M \\ \gtrsim & \mathsf{let}\ \{xs = \mathsf{let}\ \{ys = repeat^n\ x\}\ \mathsf{in}\ x : ys\}\ \mathsf{in}\ M \\ \gtrsim & \mathsf{let}\ \{xs = \mathsf{let}\ \{ys = x : ys\}\ \mathsf{in}\ x : ys\}\ \mathsf{in}\ M \\ \gtrsim & \mathsf{let}\ \{xs = x : xs\}\ \mathsf{in}\ M \end{array}$$

□

Case Study 2: Intermediate Data Structures

Our next example concerns intermediate data structures produced by a definition of the Haskell prelude function *any*.[4] The function takes two arguments: a predicate p and a list xs and tests whether any of the elements of the list fulfils the predicate. The function can be defined in a direct recursive style:

$$\begin{array}{l} any\ p\ xs = \mathsf{case}\ xs\ \mathsf{of} \\ \qquad\qquad \mathsf{nil} \to \mathsf{false} \\ \qquad\qquad y : ys \to p\ y\ ||\ any\ p\ ys \end{array}$$

where $||$ is the infix logical or operator. However in the Haskell report [13] *any* is defined in an elegant combinator style:

$$any'\ p = or\ \circ\ map\ p$$

where *or* is defined as

$$or = foldr\ (||)\ \mathsf{false}$$

Apart from the stylistic differences, there is a key operational difference between the two definitions. The latter, when applied to p and xs, builds a list $map\ p\ xs$. Interestingly, several discussions on the Haskell mailing list where concerned about the efficiency of the latter definition. In particular, that the construction of the list would lead to a space leak proportional to the length of the list. The replies on the mailing list were of two kinds. The first kind emphasised that the definition in the Haskell report should be seen as a specification (a reference implementation) of only the extensional behaviour of *any*. A particular code distribution would be free to provide the presumably more efficient definition of *any*. A clever compiler might even automatically derive it using *deforestation* [32]. The second kind of reply appealed to the folklore of call-by-need: the list is only an intermediate data structures and the two definitions have the same asymptotic space behaviour. The following result confirms the folklore.

PROPOSITION 6.2. $\Gamma \vdash any\ p\ xs \mathrel{\Leftrightarrow\!\!\!\!\!\approx} any'\ p\ xs$
where Γ *contains the definitions of any and any' (and the definitions of the other functions they rely on).*

The relevance of the result is twofold. Firstly, the definition in the Haskell report is at most a constant factor worse than the direct recursive definition so it serves perfectly well as a reference implementation with respect to space use. Secondly, a compiler which replaces the latter definition with

[4]The example and its space properties were discussed on the Haskell mailing list in January 2001 (www.haskell.org).

the former doesn't risk to introduce a space leak in some weird case. This might seem obvious at first thought but having worked with space improvement for a while we have learnt to not jump to such conclusions.

Let us sketch the proof of Proposition 6.2. As you would expect the proof is via a strong improvement. However the proof is considerably more involved than our previous example because we cannot show the strong improvement

$$\Gamma \vdash any\, p\, xs \lessapprox any'\, p\, xs$$

because *any* and *any'* use different amounts of space – although the difference is within a constant factor. The solution is to introduce alternative definitions of *any* and *any'* which we will call any_a and any'_a respectively such that

$$\Gamma \vdash any\, p\, xs \lessapprox\!\!\approx any_a\, p\, xs \lessapprox any'_a\, p\, xs \lessapprox\!\!\approx any'\, p\, xs.$$

To come up with the definitions of any_a and any'_a is non trivial and requires some creativity and/or hard work. Our experience has led us to the following methodology: We modify the original definitions in a way such that

- the modified definitions are weakly equivalent to the original definitions, and
- we can show it just using the laws of weak improvement without the need of fixed-point induction.

The modifications are of two kinds:

- First, wherever it can be justified, we put in zero weights on short-lived structures such as arguments to known functions. This reduces the "noise" from the computations and is sometimes necessary to make the definitions strongly equivalent. It also vastly simplifies the proof of the strong improvement since it eliminates lots of spikes that would otherwise clutter the derivations.
- Second, more difficult step: whenever the two original definitions have a different space behaviour modulo "noise", we level them up by adding spikes, dummy bindings and extra weights. However when we do this we have to be careful to not increase space use by more than a constant factor.

Let us return to our example. In the first step we add zero weight on all applications of known functions. We make these modifications also to *foldr*, *map* and *or* which are called by *any'*. The second step is to add dummy space use to *any* to make it take up as much space as *any'*. Recall the definition of *any*, here spelled out without syntactic sugar:

$$\begin{array}{l} any = \lambda p.\lambda xs.\mathsf{case}\ xs\ \mathsf{of} \\ \qquad\qquad \mathsf{nil} \to \mathsf{false} \\ \qquad\qquad y : ys \to \mathsf{let}\ a = p\, y \\ \qquad\qquad\qquad\qquad\qquad\ b = any\, p\, ys \\ \qquad\qquad\qquad\qquad \mathsf{in}\ (||)\, a\, b \end{array}$$

We modify the definition as follows.

$$\begin{array}{l} any_a = \lambda p.\lambda xs.{}^{\curlywedge 2}\mathsf{case}\ xs\ \mathsf{of} \\ \qquad\qquad \mathsf{nil} \to {}^{\curlyvee}\mathsf{false} \\ \qquad\qquad y : ys \to {}^{\curlyvee}\mathsf{let}\ z = \Omega \\ \qquad\qquad\qquad\qquad\qquad\ a = p \cdot y \\ \qquad\qquad\qquad\qquad\qquad\ b = {}^{\{z\}}(any_a \cdot p \cdot ys) \\ \qquad\qquad\qquad\qquad \mathsf{in}\ (||) \cdot a \cdot b \end{array}$$

There are two interesting modifications. The first one is the extra weight on the case expression, which compensates for the extra stack space used by *any'*; for *any'* to scrutinise the head of its input *xs* it calls *or* with the argument *map p xs* and *or* passes the argument to *foldr*. Then, *foldr* pushes a stack-frame and forces the computation of its input *map p xs*. In turn, *map* pushes a stack-frame and forces the computation of *xs*. Thus two stack-frames have been pushed onto the stack. The extra weight on the case in any_a mimics this behaviour. From Lemma 5.4 we know that the extra weight can make any_a use up at most a constant factor more stack than *any*.

The other interesting modification is the dummy binding of *z* in the cons-branch of any_a. The dummy binding lives until *b* is evaluated or until *b* becomes garbage. We get this effect because of the dummy reference to *z* in the right hand side of the definition of *b*. The dummy binding is there to mimic the space used up by the list *map p xs* which *any'* constructs. It is worth noting that although the list is an intermediate data structure it is not necessarily short-lived. It will stay in memory during the evaluation of *p y* which can be arbitrarily long and which may even call *any'* itself. But the extra structure can not change the asymptotic space behaviour because there are other structures in the heap which are at least as long lived. This is not easy to see from the definition of *any'* but in the definition of any_a we can see that the dummy binding that mimics the structure cannot live longer than the binding *b* (Lemma 5.2). With the appropriate definitions of any_a and any'_a it is straightforward to show that

$$\Gamma \vdash any\, p\, xs \lessapprox\!\!\approx any_a\, p\, xs \lessapprox any'_a\, p\, xs \lessapprox\!\!\approx any'\, p\, xs.$$

The complete derivation of $\Gamma \vdash any_a\, p\, xs \lessapprox any'_a\, p\, xs$ can be found in [9]. The plethora of spikes, dummy references, dummy bindings and weights that are necessary in this kind of derivation make the process of constructing derivations extremely error prone. We found it necessary to develop a simple tool to formally check derivations, and the steps of the derivation in this case study have been verified in this way.

Case Study 3: Trading Stack for Heap

This case study is about the associativity of append, $(+\!\!+)$. It is interesting because it is an example of a transformation that can increase heap usage with more than a constant factor so it falls outside of $\lessapprox\!\!\approx$. However the transformation can only lead to a constant factor difference in the *total amount* of space used. The reason is that in all cases where the amount of heap increases, a corresponding amount of stack space is used already.

To make this claim precise we define a relaxed version of $\gtrapprox\!\!\approx$, which allows stack space to be traded for heap space:

DEFINITION 6.1 (STACK WEAK IMPROVEMENT). *We say that M* is stack weakly improved by *N, written $M \blacktriangleright\!\!\approx N$, if there exists a linear function $f \in \mathbb{N} \to \mathbb{N}$ such that for all $\mathbb{C}$, σ such that $\mathbb{C}[M\sigma]$ and $\mathbb{C}[N\sigma]$ are closed,*

$$\mathbb{C}[M\sigma]\Downarrow_{(h,s)} \implies \mathbb{C}[N\sigma]\Downarrow_{(h',s')}$$

for some h' and s' such that $s' \leq f(s)$ and $h' + s' \leq f(h+s)$.

We can now state an improvement property of append:

PROPOSITION 6.3. $(xs +\!\!+ ys) +\!\!+ zs \blacktriangleright\!\!\approx xs +\!\!+ (ys +\!\!+ zs)$.

Note that the relaxed relation is only required in one direction. But it is the direction that one most often would like to use when applying this equivalence – it can lead to an asymptotic speedup in some contexts. We will see such an example later.

Now let us outline the proof of Proposition 6.3. We will follow the methodology from the previous example and come up with modified versions of append for which we can establish a strong improvement. We will need four different versions, one for each occurrence of append, which we call $+\!\!+_a, \ldots, +\!\!+_d$. The strong improvement part of the proposition turns out to be valuable in its own right (see case study 4) so we spell it out here.

LEMMA 6.4.

$$\Gamma \vdash \mathsf{let}\ \{ps = (+\!\!+_a) \cdot xs \cdot ys\}\ \mathsf{in}\ (+\!\!+_b) \cdot ps \cdot zs \lessapprox \mathsf{let}\ \{qs = (+\!\!+_d) \cdot ys \cdot zs\}\ \mathsf{in}\ (+\!\!+_c) \cdot xs \cdot qs$$

We have stated the lemma without syntactic sugar. We have found that this is often the first step towards an intuition about space use. Indeed, it is now explicit that the terms allocate space in the heap before they call the append function. How long lived are these bindings? Clearly, the binding for ps in the left hand side of the improvement is very short lived: $+\!\!+_b$ immediately evaluates its first argument and then there is no remaining references to ps. However in the right hand side the binding for qs may live for a long time. To compensate for this and make the strong improvement hold we have added a dummy allocation in the definition of $+\!\!+_a$:

$$\begin{array}{l}(+\!\!+_a) = \lambda as.\lambda bs.\mathsf{let}\ z = \Omega \\ \qquad\qquad \mathsf{in\ case}\ as\ \mathsf{of} \\ \qquad\qquad\quad \mathsf{nil} \to {}^{\{z\}\wedge} bs \\ \qquad\qquad\quad c : cs \to {}^{\{z\}} \mathsf{let}\ ds = (+\!\!+_a) \cdot cs \cdot bs \\ \qquad\qquad\qquad\qquad \mathsf{in}\ c : ds\end{array}$$

The dummy binding is allocated just before the case expression is executed, and lives until just after a branch has been selected. Thus the lifetime of the binding matches the lifetime of the stackframe pushed for the case expression. The binding exactly compensates for the different heap behaviours of the original functions. There is also a difference in stack usage between $(xs +\!\!+ ys) +\!\!+ zs$ and $xs +\!\!+ (ys +\!\!+ zs)$. This difference is of a similar nature to the difference between *any* and *any'* from our previous case study. We need to put an extra weight on the case in $+\!\!+_c$:

$$\begin{array}{l}(+\!\!+_c) = \lambda as.\lambda bs.{}^{2}\mathsf{case}\ as\ \mathsf{of} \\ \qquad\qquad\quad \mathsf{nil} \to bs \\ \qquad\qquad\quad c : cs \to \mathsf{let}\ ds = (+\!\!+_c) \cdot cs \cdot bs \\ \qquad\qquad\qquad\qquad \mathsf{in}\ c : ds\end{array}$$

For $+\!\!+_b$ and $+\!\!+_d$ the modifications are minor and only involve zero weights on short lived stack elements. With these definitions at hand it is not difficult to show Lemma 6.4 although the derivations are lengthy.

It is easy to see that the modifications in $+\!\!+_c$ and $+\!\!+_d$ are within a constant factor of the original definition of append (Lemma 5.4 etc.), so we have

$$\Gamma \vdash \mathsf{let}\ \{qs = (+\!\!+_d) \cdot ys \cdot zs\}\ \mathsf{in}\ (+\!\!+_c) \cdot xs \cdot qs \lessapprox xs +\!\!+ (ys +\!\!+ zs).$$

To show Proposition 6.3 it remains to show that

$$\Gamma \vdash (xs +\!\!+ ys) +\!\!+ zs \gtrapprox \mathsf{let}\ \{ps = (+\!\!+_a) \cdot xs \cdot ys\}\ \mathsf{in}\ (+\!\!+_b) \cdot as \cdot zs.$$

The difficulty lies in the dummy binding in the definition of $+\!\!+_a$. Recall that the lifetime of the dummy binding precisely matches the lifetime of the stack frame pushed by the case. Such a binding can at most double the *total amount* of space use – hence it is within a constant factor as stated by this lemma.

LEMMA 6.5.

$$\mathsf{case}\ M\ \mathsf{of}\ \{pat_i \to N_i\} \lessapprox \mathsf{let}\ \{z = \Omega\}\ \mathsf{in\ case}\ M\ \mathsf{of}\ \{pat_i \to^{\{z\}} N_i\}, \quad z\ \textit{fresh},$$

This completes the proof sketch of Proposition 6.3.

In the beginning of this section we made another claim which partly motivated the introduction of a new relation, namely that the transformation can lead to an asymptotic increase in heap usage. The following family of contexts, indexed by k shows that $\Gamma \vdash (xs +\!\!+ ys) +\!\!+ zs \not\gtrapprox xs +\!\!+ (ys +\!\!+ zs)$ by exhibiting a difference in heap behaviour which grows with k.

$$\begin{array}{l}\mathsf{let}\ g\,k\,ys\,zs = \mathsf{if}\ k = 0 \\ \qquad\qquad\qquad\quad \mathsf{then\ nil} \\ \qquad\qquad\qquad\quad \mathsf{else\ let}\ \{xs = g\,(k-1)\,ys\,zs\}\ \mathsf{in}\ [\cdot] \\ \mathsf{in}\ g\,k\,\mathsf{nil}\,\mathsf{nil}\end{array}$$

Case Study 4: Tail Recursion

This case study is about tail recursion – a transformation very much aimed at improvement in space behaviour. But tail recursive transformations may also improve time complexity and this case study is about such an example. Consider the naive definition of a function that reverses a list:

$$\begin{array}{l}reverse\ xs = \mathsf{case}\ xs\ \mathsf{of} \\ \qquad\qquad\quad \mathsf{nil} \to \mathsf{nil} \\ \qquad\qquad\quad y : ys \to reverse\ ys +\!\!+ [y]\end{array}$$

The function uses up stack proportional to the length of the list and it also suffers from a quadratic time complexity due to the repeated applications of append. The cure is well-known: transform the function to a tail recursive accumulating parameter definition:

$$\begin{array}{l}reverse'\ xs = rev\ []\ xs \\ rev\ as\ xs = \mathsf{case}\ xs\ \mathsf{of} \\ \qquad\qquad\quad \mathsf{nil} \to \mathsf{nil} \\ \qquad\qquad\quad y : ys \to rev\ (y : as)\ ys\end{array}$$

The tail recursive $reverse'$ has a linear time complexity and the following result confirms our hopes about its space use.

PROPOSITION 6.6. $\Gamma \vdash reverse\ xs \gtrapprox reverse'\ xs$

We will not go into any details about the proof of this proposition but comment on one aspect of the proof. In a proof of contextual equivalence of the two definitions it is helpful to fall back on a result about the associativity of append. Proposition 6.3 provides such a result of weak improvement but it is useless for our proof of Proposition 6.6 because our proof relies on strong improvement. Instead we use the strong improvement in Lemma 6.4. It complicates

matters because Lemma 6.4 refers to four different "gadget-versions" $+\!\!+_a \ldots +\!\!+_d$ of append. This illustrates a general problem: when working with strong improvement we cannot rely on weak improvement results.

Case Study 5: Strict Accumulating Parameters

This case study is about an example where a tail recursion transformation alone does not solve the problem but where we also need a transformation step guided by strictness information.

Consider the naive definition of *sum*.

$$\begin{aligned} sum\ xs = {} & \mathsf{case}\ xs\ \mathsf{of} \\ & \mathsf{nil} \to 0 \\ & y : ys \to y + sum ys \end{aligned}$$

The definition suffers from the same problem as the naive definition of *reverse* – it requires stack proportional to the length of the input list. At first it may appear that a plain tail recursion transformation would do the job:

$$\begin{aligned} sum' xs = {} & asum\ 0\ xs \\ asum\ a\ xs = {} & \mathsf{case}\ xs\ \mathsf{of} \\ & \mathsf{nil} \to a \\ & y : ys \to \mathsf{let}\ a' = a + y \\ & \qquad \mathsf{in}\ asum\ a'\ ys \end{aligned}$$

But sum' still uses stack proportional to the length of its argument: Because of lazy evaluation, the evaluation of $a + y$, in the recursive call of *asum*, is delayed until required. As a result a chain of closures representing the sum builds up in the heap and when the computation is forced it takes up stack proportional to the length of the input list. The next transformation step hinges on the fact that *asum* is strict in the accumulating parameter and forces the accumulator to be computed in each step of the recursion:

$$\begin{aligned} sum'' xs = {} & asum'\ 0\ xs \\ asum'\ a\ xs = {} & \mathsf{case}\ xs\ \mathsf{of} \\ & \mathsf{nil} \to a \\ & y : ys \to \mathsf{let}\ a' = a + y \\ & \qquad \mathsf{in}\ \mathsf{seq}\ a'\ (asum'\ a'\ ys) \end{aligned}$$

This is the kind of transformation that a complier with a strictness analyser typically performs. But strictness transformations in general are dangerous from the point of view of space use because they may change evaluation order. Consider, for example the strict function $\lambda y.\lambda x.x + y$. A compiler with strictness analysis might well change the order of the evaluation of the arguments, and from the example in the introduction it should be clear why this is not a space improvement.

Indeed, it happens in this case also: *asum* will traverse the entire spine of its input before evaluating any of its elements, but $asum'$ will evaluate the elements as it traverses the list. The following family of contexts (indexed by k) explores the difference in evaluation order to show that $\Gamma \vdash sum'\ xs \not\lessapprox sum''\ xs$:

$$\begin{aligned} & \mathsf{let}\ f\ a = \mathsf{nil} \\ & \quad ys = fromto\ 1\ k \\ & \quad xs = (traverse\ ys) : (f\ ys) \\ & \mathsf{in}\ [\cdot] \end{aligned}$$

where *traverse* is a function that traverses a list and returns 0.

It seems that any transformation which changes the evaluation order of arguments or free variables (or their substructures) can never be a space improvement. At this point it seems that all is lost. However, it is still possible to use strictness transformations as a part of a transformation if it is combined with *another* transformation step which inverts the change made by the strictness phase. This is exactly what happens in this case study! The transformation from *sum* to sum' that introduced the accumulating parameter also changes the evaluation order: *sum* evaluates the elements of its input as it traverses the list but sum' traverses the entire spine of the list first. As a result this individual transformation step is not space safe either, i.e., $\Gamma \vdash sum\ xs \not\gtrapprox sum'\ xs$, which can be shown by a family of contexts similar in spirit to the one above. But taken together the transformations as a whole do not change evaluation order and moreover can be shown to be space safe:

PROPOSITION 6.7. $\Gamma \vdash sum\ xs \gtrapprox sum''\ xs$

The proof is along the lines of the previous proofs where we add gadgets to *sum* to obtain:

$$\begin{aligned} sum_a\ xs = {} & \mathsf{let}\ z = \Omega \\ & \mathsf{in}\ \mathsf{case}\ xs\ \mathsf{of} \\ & \quad \mathsf{nil} \to {}^{\{z\}}0 \\ & \quad y : ys \to \mathsf{let}\ w = \Omega \\ & \qquad \mathsf{in}\ {}^{3 \curlyvee \{z\} 3}(y + {}^{\{w\}} sum_a \cdot ys) \end{aligned}$$

The calculation steps in the proof, (omitted in order to make the paper space-safe), have also been formally verified. It is worth noting that we found it very useful in the course of the proof to employ explicit constructs for boxing and unboxing of integers in the language. This allows the proof and the required basic laws to be more fine-grained. The usefulness of these language constructs when performing program transformation is also noted by Peyton Jones [12].

Case Study 6: Tupling

Tupling is the name of a set of program transformations that bring together computations over the same input [21, 6]. Tupling transformations can dramatically reduce the amount of space and time required. Consider for example the naive function to compute the average value of the elements of a list:

$$average\ xs = sum\ xs / length\ xs.$$

The function requires linear space even if *sum* and *length* are space-efficient tail recursive functions. The reason is that (assuming / evaluates from left to right) while *sum* traverses (the lazily produced) input list, the call to *length* holds on to a reference to the start of the list so the entire list will be live. Another example which suffers from the same problem is the naive definition of the function *split* which splits a list of characters into two lists, one containing the first line, and one containing what remains after the first (if any) newline character:

$$split\ xs = (beforeNewline\ xs\ ,\ afterNewline\ xs)$$

where *beforeNewline* and *afterNewline* are defined in the obvious way. A solution to the space problems could be to tuple the computations, i.e., to simultaneously compute the first line and the remainder by a single traversal of the input

list. Such a function can be defined as follows.

$$\begin{array}{l} split'\ xs = \mathsf{case}\ xs\ \mathsf{of} \\ \qquad \mathsf{nil} \to (\mathsf{nil}, \mathsf{nil}) \\ \qquad y : ys \to \mathsf{if}\ y = newline \\ \qquad\qquad \mathsf{then}\ (\mathsf{nil}, ys) \\ \qquad\qquad \mathsf{else\ let}\ p = split'\ ys \\ \qquad\qquad\quad \mathsf{in}\ (y : (fst\ p), snd\ p) \end{array}$$

Note that *split′*, in contrast to *split*, is strict. However, this definition doesn't solve the problem. The reason is the use of the projections $fst\ p$ and $snd\ p$. Due to lazy evaluation, the projections are not evaluated until needed and therefore hold on to the reference to p, which in turn holds on to both the results of the recursive call. As a result, we have combined not only the computations but also the lifetimes of the two results.

Intriguingly, this problem appears to be linked to the intensional expressiveness of the language. Hughes has argued that it is impossible to define *split* in a space efficient way using a particular lazy evaluator [11]. He proposed a solution involving combinators for explicit parallelism and synchronisation. With these language primitives the original definition of *split* can be made efficient by having just the right degree of parallelism. Another proposal, due to Wadler [31], is to solve the problem by extending the garbage collector. Whenever the garbage collector encounters a term of the form $fst\ p$ where p is bound to an evaluated pair, it may perform the reduction of the projection. A more recent proposal is due to Sparud [30]. He proposes to treat *pattern bindings* in let expressions specially. A pattern binding in a let expression takes the form

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

Prior to Sparud's proposal, these kind of bindings were thought of as mere syntactic sugar and a compiler (e.g. [1]) would typically translate it into the following

$$\mathsf{let}\ \{p = M, x_1 = \Pi_{c_1}\ p, \ldots, x_n = \Pi_{c_n}\ p\}\ \mathsf{in}\ N$$

which reintroduces the "dangerous" projections.

Sparud's proposal was to have pattern bindings as a first class construct which the evaluator treats in a space efficient manner. We have adopted Sparud's proposal because we think it is the most natural and because it leads to a reasonably well behaved space theory. Implementing Wadler's proposal in our model of garbage collection would destroy many of the nice properties of our theory. For example, beta-expansion would no longer be space safe, because it may result in the elimination of a "garbage collector redex".

We have formalised Sparuds proposal as an extension to our language. The details can be found in [9]. With pattern bindings at hand we can rewrite *split′* as follows.

$$\begin{array}{l} split''\ xs = \mathsf{case}\ xs\ \mathsf{of} \\ \qquad \mathsf{nil} \to (\mathsf{nil}, \mathsf{nil}) \\ \qquad y : ys \to \mathsf{if}\ y = newline \\ \qquad\qquad \mathsf{then}\ (\mathsf{nil}, ys) \\ \qquad\qquad \mathsf{else\ let}\ (ps, qs) = split''\ ys \\ \qquad\qquad\quad \mathsf{in}\ (y : ps, qs) \end{array}$$

So, what is the relation between the different versions of *split*? Let us start with the relation between *split′* and *split″* where we have that $\Gamma \vdash split'\ xs \gtrapprox split''\ xs$. It follows directly from the following lemma.

LEMMA 6.8.

$$\mathsf{let}\ \{{}_2p = M\}\ \mathsf{in}\ \mathbb{C}[fst\ p][snd\ p] \gtrapprox \mathsf{let}\ \{(x,y) = M\}\ \mathsf{in}\ \mathbb{C}[x][y] \quad \textit{if } p \notin \mathsf{FV}(M, \mathbb{C})$$

The lemma says that it is always space safe to use pattern bindings instead of projections. So what about *split* and *split″*? Convinced that

$$\Gamma \vdash \mathsf{let}\ \{(x,y) = split\ xs\}\ \mathsf{in}\ M \gtrapprox \mathsf{let}\ \{(x,y) = split''\ xs\}\ \mathsf{in}\ M$$

we spent considerable effort trying to prove it only to realise that it is not the case. The family of contexts that distinguishes the two terms is somewhat involved so we found it better to present the intuition about why $split''\ xs$ in some contexts may use more space than $split\ xs$.

Consider a context where the second component of the pair is used before the first, i.e., a program which processes the second line of its input before the first. In that case the tupling has the effect that the spine of the list representing the first line of input is constructed *before* it is needed (in our definition of *split″* this allocation is hidden in the syntactic sugar). This in itself does not lead to a non constant factor worsening if the spine of the input list may be garbage collected. But what if it can't? Consider a program which processes its second line of input repeatedly and selects the line from the input by repeatedly applying *split″* to the input. Suppose also that it keeps references to the different copies of the first line that is constructed. Such a context, however unlikely in practice, would show that $\Gamma \vdash split\ xs \not\gtrapprox split''\ xs$.

This has lead us to the general observation that tupling of computations which need to allocate space in order to produce their output are unlikely to be space improvements, although we have not been able to make this statement more precise.

Another observation, at this point maybe not surprising, is that tupling transformations which change the order in which inputs (or the substructures thereof) are traversed are unlikely to be space improvements. The tupling of the sum and the length of a list is an example of this. In a context where the length of the list is needed before the sum, the untupled definition would traverse the spine of the list before any of the elements, but the tupled definition would force the computation of the elements as it traverses the list. These two observations have made us rather pessimistic about showing that tupled functions improve on their untupled counterparts. However, in contexts which are guaranteed to require the result of the tupled computation in a specific order the situation may be different. For example, we believe that for *average′* defined using a tupled computation of the sum and the length we would have $\Gamma \vdash average\ xs \gtrapprox average'\ xs$ because the functions (due to the evaluation order of /) require the sum before the length.

7. RELATED WORK AND CONCLUSIONS

Improvement theory was first developed in the call-by-name setting [26, 25, 27] for the purpose of reasoning about running-times of programs. Moran and Sands [19] developed a call-by-need time-improvement theory, together with a variety of induction principles. This present work, and its predecessor [10] are the only attempts (of which we are aware)

which formalise space safety properties of local (non-whole-program) transformations. More details of this work can be found in the first author's PhD thesis [9].

Other related work includes the development of "space-aware" operational models for call-by-need languages [28, 24, 4, 3], studies of space-safety properties of global transformations [17, 18] and of the relative efficiency of different abstract machines [5, 7, 2, 18]. Minamide [18] suggests an alternative to our definition of improvement based on additive constant factors. Its properties are not studied for any particular language, although we suspect that it would fail to satisfy the syntactic continuity property, so would not serve as an alternative to strong improvement.

Areas for further work include the introduction of context information to the theory in order to represent constraints on the whole-program context which can be used to help establish space improvements.

8. REFERENCES

[1] L. Augustsson. *Compiling Lazy Functional Languages, Part II.* PhD thesis, Department of Computer Science, Chalmers University of Technology, November 1987.

[2] A. Bakewell and C. Runciman. A model for comparing the space usage of lazy evaluators. In *Proceedings of PPDP'00*, September 2000.

[3] A. Bakewell and C. Runciman. A space semantics for core haskell. In *Proceedings of the Haskell Workshop*, September 2000.

[4] 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*, volume 1140 of *LNCS*, pages 393–407. Springer-Verlag, 1996.

[5] G. E. Blelloch and J. Greiner. A provably time and space efficient implementation of nesl. In *Proc. ICFP'96*, pages 213–225, 1996.

[6] W.-N. Chin. Towards an automated tupling strategy. In *Proceedings of PEPM'93*, pages 119–132, Copenhagen, Denmark, 1993.

[7] W. D. Clinger. Proper tail recursion and space efficiency. In *Proc. PLDI'98*, 1998.

[8] A. D. Gordon and A. M. Pitts, editors. *Higher Order Operational Techniques in Semantics.* Cambridge University Press, 1998.

[9] J. Gustavsson. *Space-Safe Transformations and Usage Analysis for Call-by-Need Languages.* PhD thesis, Department of Computer Science, Chalmers University of Technology and Göteborg University, May 2001.

[10] J. Gustavsson and D. Sands. A foundation for space-safe transformations of call-by-need programs. In *Proceedings of HOOTS III*, volume 26 of *ENTCS*. Elsevier, 1999.

[11] R. J. M. Hughes. *The Design and Implementation of Programming Languages.* PhD thesis, Programming Research Group, Oxford University, July 1983.

[12] S. P. Jones. Compiling haskell by program transformation: a report from the trenches. In *Proceedings of ESOP'96*, April 1996.

[13] S. P. Jones and J. Hughes. Haskell 98: A non-strict, purely functional language. Available at www.haskell.org.

[14] J. Launchbury. A natural semantics for lazy evaluation. In *Proc. POPL'93*, pages 144–154. ACM Press, Jan. 1993.

[15] I. A. Mason, S. F. Smith, and C. L. Talcott. From operational semantics to domain theory. *Information and Computation*, 128(1):26–47, 10 July 1996.

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

[17] Y. Minamide. Space-profiling semantics of the call-by-value lambda calculus and the cps transformation. In *Proceedings of HOOTS III*, volume 26 of *ENTCS*. Elsevier, 1999.

[18] Y. Minamide. A new criterion for safe program transformations. In *Proceedings of HOOTS IV*, volume 41 of *ENTCS*. Elsevier, 2000.

[19] A. Moran and D. Sands. Improvement in a lazy context: An operational theory for call-by-need. In *Proc. POPL'99*, pages 43–56. ACM Press, Jan. 1999.

[20] G. Morrisett and R. Harper. Semantics of memory management for polymorphic languages. In Gordon and Pitts [8], pages 175–226.

[21] A. Pettorossi. Transformation of programs and use of tupling strategy. In *Proceedings Informatica 77, Bled, Yugoslavia.*, pages 1–6, 1977.

[22] S. Peyton Jones, W. Partain, and A. Santos. Let-floating: moving bindings to give faster programs. In *Proc. ICFP'96*, pages 1–12. ACM Press, May 1996.

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

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

[25] 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, Aug. 1991.

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

[27] D. Sands. Total correctness by local improvement in the transformation of functional program. *ACM TOPLAS*, 18(2):175–234, Mar. 1996.

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

[29] S. F. Smith. From operational to denotational semantics. In *Proceedings of MFPS'92*, LNCS, pages 54–76. Springer Verlag, 1992.

[30] J. Sparud. Fixing Some Space Leaks without a Garbage Collector. In *Proceedings of FPCA'93*, pages 117–122. ACM Press, June 1993.

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

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