Electronic Notes in Theoretical Computer Science 41 No. 3 (2000)
URL: http://www.elsevier.nl/locate/entcs/volume41.html pages 20–34

A New Criterion for Safe Program Transformations

Yasuhiko Minamide

Institute of Information Sciences and Electronics
University of Tsukuba
and
PRESTO
Japan Science & Technology Corporation

Email: minamide@score.is.tsukuba.ac.jp

Abstract

Previous studies on safety of program transformations with respect to performance considered two criteria: preserving performance within a constant factor and preserving complexity. However, as the requirement of program transformations used in compilers the former seems too restrictive and the latter seems too loose. We propose a new safety criterion: a program transformation preserves performance within a factor proportional to the size of a source program. This criterion seems natural since several compilation methods have effects on performance proportional to the size of a program. Based on this criterion we have shown that two semantics formalizing the size of stack space are equivalent. We also discuss the connection between this criterion and the properties of local program transformations rewriting parts of a program.

1 Introduction

Recent compilers utilize advanced program transformations to obtain high-performance executable code. For these advanced program transformations, it is not so straightforward to guarantee that they are safe with respect to performance. In fact, some program transformations have been shown to improve the performance of most programs, while degrading the performance of some programs severely [9,12].

To remedy this situation, several papers have discussed the safety of program transformations based on semantics formalizing the performance of programs [7,6,11,2,8]. In those studies, two safety criteria for whole-program transformations were discussed. However, these criteria do not seem appropriate to impose on program transformations used in compilers, for reasons we

discuss below. In this paper we focus on the space requirement of programs in call-by-value functional languages, but the general framework can be adopted for other languages and performance metrics.

The first of the above two criteria ensures that a program transformation preserves space requirement within a constant factor. If a program transformation satisfies this property, we say that the program transformation is space efficient. Many program transformations seem to satisfy this criterion, but there are some useful transformations that do not satisfy this criterion. Furthermore, to show this criterion is satisfied we must formalize the details of semantics formalizing space requirements. In the study of the CPS transformation, it was necessary to revise the space profiling semantics of a call-by-value functional language by Blelloch and Greiner [3], to show that this transformation satisfies this criterion [8].

The second criterion is space safety: a program transformation is space safe if it does not raise the complexity of programs. Clearly, program transformations used in a compiler must satisfy this criterion. However, we think that this criterion is too loose. This criterion does not impose any restriction for programs without inputs. However, showing space safety is simpler than showing space efficiency, in the sense that it is possible to adopt a simpler profiling semantics that ignores details such as sizes of closures and stack frames.

In this paper we propose a new criterion that falls between the above two criteria. The new criterion is that a program transformation preserves the space requirement within a factor proportional to the size of a source program. Most useful transformations used in a compiler seem to satisfy this criterion. This criterion seems natural because several compilation methods have effects on performance proportional to the size of a program. Furthermore, we can show this criterion based on semantics ignoring some details as we show space safety.

Based on the new criterion we have shown that two semantics of a simple call-by-value functional language profiling stack space are equivalent. One models evaluation by an interpreter and the other models execution based on compilation. They are not equivalent in the sense of space efficiency, but they are equivalent in the sense of our new criterion. We also show that A-normalization preserves stack space modeled by the second semantics. This backs the claim that the second semantics models stack space required for execution based on compilation.

The criterion we propose is a property of a whole-program transformation. On the other hand, some transformations used in compilers are based on local program transformations. We therefore also study the connection between the properties of local transformations and the properties of global transformations. We will show that some restricted class of local transformations induces whole-program transformations satisfying our new criterion.

This paper is organized as follows. We begin by reviewing the two safety criteria discussed in previous studies and discussing why they are not suitable as the criterion we impose on the transformations used in a compiler. In Section 3 we introduce our new safety criterion and the equivalence of semantics on a programming language induced by the criterion. In Section 4, based on this new safety criterion, we show that two operational semantics of a call-by-value functional language are equivalent. In Section 5 we discuss the connection between the properties of local transformations and our new criterion. Finally, we give our conclusions and directions for future work.

2 Safety criteria of program transformations

We review the safety criteria of program transformations with respect to performance discussed in previous studies. To formalize safety criteria we must first develop semantics to formalize performance of programs. We call such semantics profiling semantics. For a simple programming language, profiling semantics can be given as a partial function $eval(M)$:

$$eval(M) = (v, n)$$

This function gives the computation result v and a non-negative integer n, which represents such performance values of programs as execution time or space required for execution.

In this paper, to simplify our discussion, we focus on the space requirement of a program and ignore the computation result. For this purpose we define $space(M)$ as below:

$$space(M) = n \textit{ iff } eval(M) = (v, n) \textit{ for some } v$$

This function is only defined if the evaluation of M terminates. We call this semantics for specifying space required for execution of a program *space semantics*.

Let us now review two space safety criteria of program transformations discussed in previous work [1,3,8]. In this paper, we consider a program transformation as a binary relation between programs in a source language and a target language. Let us consider a program transformation $\rightsquigarrow$ between a source language and a target language that have space semantics $space(M)$ and $space'(M')$ respectively: $M \rightsquigarrow M'$ means that M is translated to M' by the program transformation.

The first criterion ensures that a program transformation preserves the space required for execution of a program within a constant factor.

Definition 2.1 We say that a program transformation $\rightsquigarrow$ is space efficient if constants k_1 and k_2 exist such that:

$$space(M) = n \implies space'(M') \leq k_1 n + k_2$$

for any programs M and M' with $M \leadsto M'$.

We admit constants k_1 and k_2 because they seem dependent on the details of definition of space semantics and not essential. Moreover, there are several transformations in which $k_1 > 1$ is required to show this property. This property was first discussed by Blelloch and Greiner for an implementation of NESL [3]. Minamide also showed that the CPS transformation is space efficient [8].

The second criterion is called space safety: a transformation is *space safe* if it does not increase the space complexity of programs [1]. To formalize this idea we must consider programs with an input; thus we can consider a programming language with input commands and the semantics $space(M, I)$, which formalizes the space required for execution of the program M for the input I.

Definition 2.2 We say that a program transformation $\leadsto$ is space safe if for any programs M and M' such that $M \leadsto M'$, constants k_1 and k_2 exist such that for any input I the following holds:

$$space(M, I) = n \implies space'(M', I) \leq k_1 n + k_2$$

The key difference from space efficiency is that the constants k_1 and k_2 are program-dependent. Space efficiency usually implies space safety because space efficiency provides the constants k_1 and k_2 to show space safety without depending on programs.

Although many program transformations used in compilers seem space efficient, some useful transformations are not space efficient, but only space safe. Furthermore, to show that a program transformation is space efficient we must consider too many details of the operational semantics of the source language. In the study of the CPS transformation it was necessary to revise the semantics proposed by Blelloch and Greiner [3] to show that the CPS transformation was space efficient [8].

Code motion is a typical example of a transformation that is space safe, but not space efficient. Consider the following expression where `M` is a pure expression that contains the variable `n` but does not contain a function application:

```
fun loop 0 = ()
  | loop n = let val x = M  in  loop (n - 1) end
```

The value of `M` is loop-invariant, thus we want to hoist the binding of `x` as follows:

```
val x = M
fun loop 0 = ()
  | loop n = loop (n - 1)
```

This usually improves the performance of the program. However, if the func-

tion `loop` is used only as `loop 0` or is not actually used, this transformation results in extra computation of `M`. The extra time and space to evaluate `M` is not uniformly bounded by a constant, but depends on `M`.

On the other hand, space safety seems too weak as a requirement of transformations used in compilers. Space safety is trivial for programs without inputs. Even for programs with inputs, it is impossible to estimate the performance of a program since the constants k_1 and k_2 are program-dependent. Thus in this paper we propose a new criterion that falls between space efficiency and space safety.

3 A new criterion

In this section, we propose a new safety criterion for program transformations. First, to simplify our discussion, we compare two space semantics of a programming language.

Let us consider two space semantics $space_1(M)$ and $space_2(M)$ of a programming language. As a natural extension of space efficiency, we can consider the following property: there exists a polynomial f such that

$$space_1(M) = n \implies space_2(M) \leq f(n)$$

However, this property is not suitable as a criterion that $space_2(M)$ is safe with respect to $space_1(M)$. By extending the language and the semantics with inputs, we have the following property:

$$space_1(M, I) = n \implies space_2(M, I) \leq f(n)$$

Even if this holds, it might happen that for an input of size n $space_1$ requires n space, but $space_2$ requires n^2 space, because $f(x) = x^2$. Thus, this extended property does not imply safety and is therefore not suitable as a criterion that $space_2$ is safe with respect to $space_1$.

When we consider various semantics of a language, the difference in space usage often depends on the size of a program. Thus, it is natural to consider the following relation of semantics.

Definition 3.1 We say that the semantics $space_1$ is weakly simulated by $space_2$ if

$$space_1(M) = n \implies space_2(M) \leq f_1(|M|)n + f_2(|M|)$$

where $f_1(x)$ and $f_2(x)$ are polynomials with positive coefficients and $|M|$ is the size of M.

Hereafter in this paper we simply say "a polynomial" for "a polynomial with positive coefficients."

This relation induces equivalence of semantics as follows.

Definition 3.2 We say that semantics $space_1$ is weakly space equivalent to $space_2$ if $space_1$ is weakly simulated by $space_2$ and $space_2$ is weakly simulated by $space_1$.

Most reasonably defined semantics of a language seem weakly space equivalent. Moreover, it is possible to define simpler semantics weakly space equivalent to the semantics considered in previous studies.

Example 3.3 Let $space_1$ be a semantics of a functional language that accounts for the sizes of closures: the size of a closure with n free variables is $n+1$. Let $space_2$ be a semantics where the sizes of closures are ignored: the sizes of closures are always 1. Then $space_1$ and $space_2$ are weakly equivalent, since the sizes of the closures constructed during evaluation of a program are bounded by the size of the program.

Example 3.4 Let $space_1$ and $space_2$ be a semantics that accounts for the size of each stack frame and a semantics that ignores the size of each stack frame, respectively. Then $space_1$ and $space_2$ are weakly equivalent, since the sizes of the stack frames constructed during evaluation of a program are bounded by the size of the program.

Although these examples are rather straightforward, they show that we can adopt a simple space semantics when we consider weak simulation. We will also show that two space semantics profiling stack space are equivalent in Section 4. The proof of this equivalence requires detailed analysis of the space semantics.

Now we extend weak simulation as a safety criterion for program transformations. Consider a program transformation $\leadsto$ between a source language and a target language that have space semantics $space(M)$ and $space'(M')$, respectively, as before.

Definition 3.5 We say that a program transformation $\leadsto$ is weakly space efficient if polynomials $f_1(x)$ and $f_2(x)$ exist such that:

$$space(M) = n \implies space'(M') \leq f_1(|M|)n + f_2(|M|)$$

for any programs M and M' with $M \leadsto M'$.

This criterion clearly falls between space efficiency and space safety. It admits that a program transformation degrades performance within a factor dependent on the size of a program. This gives us much more freedom to design program transformations used in compilers than is possible with space efficiency.

To construct a compiler that is a weakly efficient transformation as a whole, the composition of transformations must be weakly space efficient since actual compilers consist of many phases. To make the composition weakly space efficient we should restrict program transformations so that the expansion of the size of a program is limited by some polynomial.

Definition 3.6 We say that a program transformation $M \rightsquigarrow M'$ is polynomial size safe if $|M'| \leq f(|M|)$ for all programs M where $f(x)$ is a polynomial of x.

This is a natural restriction because actual compilers already avoid exponential blowup of code size. We can now construct a weakly space efficient compiler by composing weakly space efficient and polynomial size safe transformations.

Theorem 3.7 *Let $\rightsquigarrow_1$ and $\rightsquigarrow_2$ be program transformations from L_1 to L_2 and from L_2 to L_3 respectively. If $\rightsquigarrow_1$ is weakly space efficient and polynomial size safe, and $\rightsquigarrow_2$ is weakly space efficient, then their composition $\rightsquigarrow_1 \circ \rightsquigarrow_2$ is a weakly space efficient transformation from L_1 to L_3.*

Proof. Let $M \rightsquigarrow_1 \circ \rightsquigarrow_2 N$. Then P exists such that $M \rightsquigarrow_1 P$ and $P \rightsquigarrow_2 N$. By weak space efficiency we have:

$$space_2(P) \leq f_1^1(|M|)space_1(M) + f_2^1(|M|)$$
$$space_3(N) \leq f_1^2(|P|)space_2(P) + f_2^2(|P|)$$

By polynomial size safety we have $|P| \leq g(|M|)$. Then:

$$space_3(N) \leq f_1^2(g(|M|))(f_1^1(|M|)space_1(M) + f_2^1(|M|)) + f_2^2(g(|M|))$$

Here, $f_1^2(g(x))f_1^1(x)$ and $f_1^2(g(x))f_2^1(x) + f_2^2(g(x))$ are clearly polynomials of x. □

This proof clarifies why we adopted a polynomial instead of a linear function in the definition of weakly efficient transformation. Even if two transformations are bounded by linear functions of the size of a program, their composition is not necessarily bounded by some linear function.

4 Weak equivalence on stack space

In this section we consider two space semantics profiling stack space for a simple call-by-value functional language. One semantics models evaluation by an interpreter and the other models evaluation based on compilation. Both semantics properly model tail calls. We show that although they allocate different numbers of stack frames during evaluation, the semantics are weakly space equivalent. Furthermore, it is shown that A-normalization preserves stack space for the second semantics.

4.1 Equivalence of two semantics profiling stack space

We consider the following untyped call-by-value λ-calculus with a constant c:

$$M ::= x \mid c \mid \lambda x.M \mid M_1 M_2$$

$$E \vdash c \downarrow_1 c \quad E \vdash x \downarrow_1 E(x) \quad E \vdash \lambda x.M \downarrow_1 \langle \mathsf{cl}\, E, x, M \rangle$$

$$\frac{E \vdash M_1 \downarrow_l \langle \mathsf{cl}\, E', x, M \rangle \quad E \vdash M_2 \downarrow_m v_2 \quad E'[v_2/x] \vdash M \downarrow_n v}{E \vdash M_1 M_2 \downarrow_{\mathsf{max}(l+1,m+1,n)} v}$$

Fig. 1. Operational semantics profiling stack size (interpreter-based)

$$E \vdash_2 c \downarrow_0 c \quad E \vdash_2 x \downarrow_0 E(x) \quad E \vdash_2 \lambda x.M \downarrow_0 \langle \mathsf{cl}\, E, x, M \rangle$$

$$\frac{E \vdash_2^n M_1 \downarrow_l \langle \mathsf{cl}\, E', x, M \rangle \quad E \vdash_2^n M_2 \downarrow_m v_2 \quad E'[v_2/x] \vdash_2^t M \downarrow_n v}{E \vdash_2^n M_1 M_2 \downarrow_{\mathsf{max}(l,m,n+1)} v}$$

$$\frac{E \vdash_2^n M_1 \downarrow_l \langle \mathsf{cl}\, E', x, M \rangle \quad E \vdash_2^n M_2 \downarrow_m v_2 \quad E'[v_2/x] \vdash_2^t M \downarrow_n v}{E \vdash_2^t M_1 M_2 \downarrow_{\mathsf{max}(l,m,n)} v}$$

Fig. 2. Operational semantics profiling stack size (compiler-based)

We define two space semantics $space_1^\lambda(M)$ and $space_2^\lambda(M)$ by deductive systems. For the definition of the deductive systems, we first define values: a value v is either a constant c or a closure $\langle \mathsf{cl}\, E, x, M \rangle$ consisting of an environment E mapping variables to values, a variable and an expression.

$$v ::= c \mid \langle \mathsf{cl}\, E, x, M \rangle$$

The space semantics $space_1^\lambda(M)$ models evaluation by an interpreter and is defined by the deductive system given in Figure 1.

$$space_1^\lambda(M) = n \text{ iff } \emptyset \vdash M \downarrow_n v$$

The space semantics $space_2^\lambda(M)$ models stack space required for execution based on compilation and is defined by the deductive system given in Figure 2.

$$space_2^\lambda(M) = n \text{ iff } \emptyset \vdash_2^t M \downarrow_n v$$

The deductive system is defined mutually inductively by the following two judgments: $E \vdash_2^t M \downarrow_n v$ models execution at tail call positions and $E \vdash_2^n M \downarrow_n v$ models execution at non-tail call positions. The application at a tail call position does not allocate a new stack frame. In the figure, we write $E \vdash_2 M \downarrow_n v$ for $E \vdash_2^t M \downarrow_n v$ and $E \vdash_2^n M \downarrow_n v$.

We have shown that the two semantics $space_1^\lambda(M)$ and $space_2^\lambda(M)$ are weakly equivalent.

Theorem 4.1 *If* $\emptyset \vdash M \downarrow_i v$ *and* $\emptyset \vdash_2^t M \downarrow_{i'} v$, *then* $i' + 1 \leq i \leq |M| \cdot (i' + 1)$.

To prove this theorem we must generalize the claim so that non-empty environments can be treated. To treat non-empty environments we define the size of a value and an environment as follows:

$$\begin{aligned} |c|_v &= 0 \\ |\langle \mathsf{cl}\, E, \lambda x.M \rangle|_v &= \mathsf{max}(|E|_v, |M|_v) \\ |E|_v = e\ \mathsf{max}\{|E(x)|_v \mid x \in Dom(E)\} \end{aligned}$$

Then the theorem is generalized to the following lemma. This lemma is proved by induction on the derivation of evaluation.

Lemma 4.2 *Let* K *be a constant such that* $|M| \leq K$ *and* $|E|_v \leq K$.

(i) *If* $E \vdash M \downarrow_i v$ *and* $E \vdash_2^t M \downarrow_{i'} v$, *then* $i' + 1 \leq i \leq K \cdot (i' + 1)$.

(ii) *If* $E \vdash M \downarrow_i v$ *and* $E \vdash_2^n M \downarrow_{i'} v$, *then* $i' \leq i \leq K \cdot i' + |M|$.

4.2 Preservation of stack space by A-normalization

In this section we show that A-normalization preserves stack space given by $space_2^\lambda(M)$ and thus $space_2^\lambda(M)$ actually models execution based on compilation. This also shows that A-normalization is weakly space efficient with respect to $space_1^\lambda(M)$.

We define the syntax of the language of A-normal forms as follows:

$$\begin{aligned} &\textit{Values} && V ::= x \mid \lambda x.M \\ &\textit{Expressions} && M ::= V \mid V_1 V_2 \mid \texttt{let}\ x = V_1 V_2\ \texttt{in}\ M \end{aligned}$$

The application $V_1 V_2$ represents tail calls and the application in $\texttt{let}\ x = V_1 V_2\ \texttt{in}\ M$ represents non-tail calls.

The semantics of this language is naturally given by the $C_a EK$ Machine defined in Figure 3 [5]. In this operational semantics continuation clearly corresponds to stack. The size of continuation is naturally defined as follows:

$$\begin{aligned} size(\mathsf{stop}) &= 0 \\ size(\langle \mathsf{ar}\, x, M, E, K \rangle) &= size(K) + 1 \end{aligned}$$

In this definition we ignore the size of each frame because it is bounded by the size of the program and we discuss weak space efficiency in this paper. To discuss space efficiency it is natural to count the number of free variables of M.

The stack space of state $\langle M, E, K \rangle$ is defined by $size(K)$. Then we define the space semantics of this language as follows: $space_A(M) = n$ if $\langle M, \emptyset, \mathsf{stop} \rangle \mapsto^* \langle V, E'', \mathsf{stop} \rangle$ and n is the maximum size of the states in the transition.

$$State\ \ S = \langle M, E, K\rangle$$
$$Continuation\ \ K = \mathsf{stop} \mid \langle \mathsf{ar}\ x, M, E, K\rangle$$

Transition Rules:

$$\begin{aligned}
\langle v, E, \langle \mathsf{ar}\ x, M, E', K'\rangle\rangle &\mapsto \langle M, E'[\gamma(v,E)/x], K'\rangle \\
\langle \texttt{let}\ \ x = V_1V_2\ \texttt{in}\ \ M, E, K\rangle &\mapsto \langle M', E'[V_2/x], \langle \mathsf{ar}\ x, M, E, K'\rangle\rangle \\
&\quad \textit{where}\ \gamma(V_1, E) = \langle \mathsf{cl}\ x, M', E'\rangle \\
\langle V_1V_2, E, K\rangle &\mapsto \langle M', E'[V_2/x], K\rangle \\
&\quad \textit{where}\ \gamma(V_1, E) = \langle \mathsf{cl}\ x, M', E'\rangle
\end{aligned}$$

$$\gamma(V, E) = \begin{cases} E(x) & \text{if } V \equiv x \\ \langle \mathsf{cl}\ x, M, E\rangle & \text{if } V \equiv \lambda x.M \end{cases}$$

Fig. 3. The C_aEK Abstract Machine

A-normalization can be defined as one pass translation [5,4]. In the following definition, we use a two-level lambda calculus where $\overline{\lambda}$ and $\overline{@}$ are meta-level abstraction and application. $||M||_A\kappa$ translates expressions at non-tail call positions and $||M||'_A$ translates expressions at tail call positions. The entire program is translated by $||M||'_A$.

$$\begin{aligned}
|x|_A &= x \\
|\lambda x.M|_A &= \lambda x.||M||'_A
\end{aligned}$$

$$\begin{aligned}
||V||_A\kappa &= k\overline{@}|V|_A \\
||M_1M_2||_A\kappa &= ||M_1||_A(\overline{\lambda}x_1.||M_2||_A(\overline{\lambda}x_2.\texttt{let}\ \ z = x_1x_2\ \texttt{in}\ \ \kappa\overline{@}z)) \\
||v||'_A &= |v|_A \\
||M_1M_2||'_A &= ||M_1||_A(\overline{\lambda}x_1.||M_2||_A(\overline{\lambda}x_2.x_1x_2))
\end{aligned}$$

Then it is shown that the stack space required for execution is preserved by A-normalization.

Theorem 4.3 *If* $space_2^{\lambda}(M) = n$, *then* $space_A(||M||'_A) = n$.

This theorem shows that $space_2^{\lambda}(M)$ models the stack space required for execution based on compilation. Furthermore, since $space_2^{\lambda}(M)$ is weakly equivalent to $space_1^{\lambda}(M)$, it is enough to consider $space_1^{\lambda}(M)$ even when we

consider weak efficiency of program transformations with respect to execution based on compilation.

5 Local transformations

In this section we discuss the connection between our new criterion and the properties of local program transformations. We show that some class of local transformations induces weakly space efficient transformations.

Based on the classification of local transformations by Gustavsson and Sands [7] we define two classes of local transformations.

Definition 5.1 Let R be a relation on terms of a programming language.

(i) We say that R is a strong improvement relation if we have:

$$space(\mathbb{C}[M]) = n \implies space(\mathbb{C}[N]) \leq n$$

for all $(M, N) \in R$ and all context $\mathbb{C}[\cdot]$ producing a whole program for M and N.

(ii) We say that R is a weak improvement relation if there exists some linear function f such that the following holds for all $(M, N) \in R$ and all context $\mathbb{C}[\cdot]$ producing a whole program for M and N.

$$space(\mathbb{C}[M]) = n \implies space(\mathbb{C}[N]) \leq f(n)$$

We should remark that there is one subtle difference in our definition of weak improvement from that of Gustavsson and Sands. They defined a single weak improvement relation as follows. $M \underset{\approx}{\rhd} N$ if some linear function f exists such that for all contexts $\mathbb{C}[\cdot]$ the following holds:

$$space(\mathbb{C}[M]) = n \implies space(\mathbb{C}[N]) \leq f(n)$$

The relation $\underset{\approx}{\rhd}$ is the union of all the weak improvement relations. However, this relation $\underset{\approx}{\rhd}$ itself is not a weak improvement relation in our sense.

To discuss the connection between these properties and the properties of global transformations, we first define the induced global transformation $M \rightsquigarrow_R N$ as follows: $M \rightsquigarrow_R N$ if some $\mathbb{C}[\cdot]$, M' and N' exist such that $M = \mathbb{C}[M']$, $N = \mathbb{C}[N']$, and $(M', N') \in R$. Then we immediately obtain the following theorem.

Theorem 5.2 *If R is a weak or strong improvement relation, $\rightsquigarrow_R$ is space efficient.*

On the other hand, the relation $\underset{\approx}{\rhd}$ does not induce a space efficient transformation. This is because there is no single linear function f such that $space(\mathbb{C}[N]) \leq f(space(\mathbb{C}[M]))$ for all $M \underset{\approx}{\rhd} N$.

The theorem above is still not enough to use a local transformation in a compiler. In a compiler we usually apply local transformations n times in one phase of a compiler where n is proportional to the size of a program. Even for such composition, a strong improvement relation induces a space efficient transformation.

Theorem 5.3 *If R is a strong improvement relation, $\rightsquigarrow_R^*$ is space efficient.*

On the other hand, a weak improvement relation does not necessarily induce a weakly space efficient transformation. Consider the following sequence of transformations where R is a weak improvement relation with $space(\mathbb{C}[N]) \leq kspace(\mathbb{C}[M])$ for all $(M, N) \in R$.

$$M_0 \rightsquigarrow_R M_1 \rightsquigarrow_R M_2 \rightsquigarrow_R M_3 \rightsquigarrow_R \ldots \rightsquigarrow_R M_n$$

The space requirement of M_n can be calculated as follows:

$$space(M_2) \leq kspace(M_1)$$

$$space(M_3) \leq kspace(M_2) \leq k^2 space(M_1)$$

$$space(M_n) \leq k^n space(M_1)$$

Then it is clear that there is no single linear function f such that:

$$space(N) \leq f(space(M))$$

for $M \rightsquigarrow_R^* N$. Even if we restrict the number of repetitions to $|M_0|$, k^n is not a polynomial of $|M_0|$. Thus, it is not even weakly space efficient.

We therefore must consider stricter conditions on local transformations. In the following definition a local transformation is permitted to add only a constant amount of extra space.

Definition 5.4 We say that R is a semi-strong improvement relation if some constant k exists such that:

$$space(\mathbb{C}[M]) = n \implies space(\mathbb{C}[N]) \leq n + k$$

for all $(M, N) \in R$ and all context $\mathbb{C}[\cdot]$ producing a whole program for M and N.

It can be shown that this class of local transformations induces weakly space efficient transformations if the number of applications of the transformation is limited by the size of a source program. We write $M \mapsto_R N$ if $M \rightsquigarrow_R^n N$ where $n \leq |M|$.

Theorem 5.5 *If R is a semi-strong improvement relation, $M \mapsto_R N$ is weakly efficient.*

Although this theorem relates a semi-strong improvement relation to weakly efficient transformations, semi-strong improvement relations seem too restrictive. There are many useful transformations R that are not semi-strong improvement relations, but $\mapsto_R$ seem weakly space efficient. The following transformation is an example.

$$\lambda x.\texttt{let } y = M \texttt{ in } N \Rightarrow \texttt{let } y = M \texttt{ in } \lambda x.N$$

We have not shown formally that this transformation is weakly space efficient. For such proof we think that we require further study on the connections between global transformations and local transformations.

6 Discussion and future work

We have shown weak efficiency only for stack space for two semantics of a simple functional language. It will not be very difficult to deal with execution time or heap space. For example, the proof that the CPS transformation is space efficient [8] can easily be modified to show that the CPS transformation is weakly space efficient with respect to a simpler space semantics of the source language that ignores the sizes of closures and stack frames.

We have shown no examples of local program transformations that are weak improvement relations or semi-strong improvement relations. We are planning to show that various optimizations formalized as local program transformations have these kinds of properties. In this area, Gustavsson and Sands have developed a theory of space improvement relations for call-by-need programming languages and have shown that several local transformations are weak improvements [7]. Their work will be also applicable to call-by-value languages.

We think that the framework we have developed in this paper requires further refinement. For example, although intuitively clear, it is not proved that space safety, weak space efficiency and space efficiency ensure that the space complexity of programs is preserved. Bakewell and Runciman discussed these kinds of issues more formally in their study on the comparison of space usage of lazy evaluators [2]. They modeled lazy evaluators by graph rewriting systems. This kind of uniform formalization of semantics may help develop a theory of safe program transformations.

There is an implementation strategy of ML that is not space efficient, but is space safe. That is the implementation strategy that uses types as parameters at runtime [10,13]. This is because the extra work and space necessary for type parameters cannot be bounded by any constant. Furthermore, this implementation strategy is not even weakly space efficient, because the types appearing in the typing derivation of a program may have a size exponential to the size of the program. However, if we take the sum of the size of a program and the maximum size of types appearing in typing derivation of the program as the size of the program, this implementation strategy can be

considered weakly space efficient. By choosing the definition of the size of a program in this way we can control the class of transformations that can be used in compilers of the language.

Acknowledgement

This work is partially supported by the Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Encouragement of Young Scientists of Japan No. 11780216, 1999. We would like to thank anonymous reviewers for their many helpful comments and suggestions.

References

[1] Appel, A. W., "Compiling with Continuation," Cambridge University Press, 1992.

[2] Bakewell, A. and C. Runciman, *A model for comparing the space usage of lazy evaluators*, in: *2nd International Conference on Principles and Practice of Declarative Programming (PPDP 2000)*, 2000.

[3] Blelloch, G. E. and J. Greiner, *A provably time and space efficient implementation of NESL*, in: *Proc. of ACM SIGPLAN International Conference on Functional Programming*, 1996, pp. 213–225.

[4] Danvy, O. and A. Filinski, *Representing control: a study of the CPS transformation*, Mathematical Structures in Computer Science **2** (1992), pp. 361 – 391.

[5] Flanagan, C., A. Sabry, B. F. Duba and M. Felleisen, *The essence of compiling with continuations*, in: *Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1993, pp. 237–247.

[6] Greiner, J. and G. E. Blelloch, *A provably time-efficient parallel implementation of full speculation*, in: *Proc. of ACM Symposium on Principles of Programming Languages*, 1996, pp. 309 – 321.

[7] Gustavsson, J. and D. Sands, *A foundation for space-safe transformations of call-by-need programs*, in: *Proc. of the Third International Workshop on Higher Order Operational Techniques in Semantics (HOOTS99)*, ENTCS **26**, 1999.

[8] Minamide, Y., *A space-profiling semantics of call-by-value lambda calculus and the CPS transformation*, in: *Proc. of the Third International Workshop on Higher Order Operational Techniques in Semantics (HOOTS99)*, ENTCS **26**, 1999.

[9] Minamide, Y. and J. Garrigue, *On the runtime complexity of type-directed unboxing*, in: *Proc. of ACM SIGPLAN International Conference on Functional Programming*, 1998, pp. 1–12.

[10] Ohori, A. and N. Yoshida, *Type inference with rank 1 polymorphism for type-directed compilation of ML*, in: *Proc. of ACM SIGPLAN International Conference on Functional Programming*, 1999, pp. 160– 171.

[11] Santos, A. L., "Compilation by Transformation in Non-strict Functional Languages," Ph.D. thesis, Department of Computing Science, University of Glasgow (1995).

[12] Shao, Z., *Flexible representation analysis*, in: *Proc. of ACM SIGPLAN International Conference on Functional Programming*, 1997, pp. 85 – 98.

[13] Tarditi, D., G. Morrisett, P. Cheng, C. Stone, R. Harper and P. Lee, *TIL: A type-directed optimizing compiler for ML*, in: *Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1996, pp. 181–192.