Runtime Behavior of Conversion Interpretation of Subtyping

Yasuhiko Minamide

Institute of Information Sciences and Electronics
University of Tsukuba
and
PRESTO, JST
minamide@is.tsukuba.ac.jp

Abstract. A programming language with subtyping can be translated into a language without subtyping by inserting conversion functions. Previous studies of this interpretation showed only the extensional correctness of the translation. We study runtime behavior of translated programs and show that this translation preserves execution time and stack space within a factor determined by the types in a program. Both the proofs on execution time and stack space are based on the method of logical relations where relations are extended with the factor of slowdown or increase of stack space.

1 Introduction

A programming language with subtyping can be translated into a language without subtyping by inserting conversion functions. Previous studies of this interpretation showed only the extensional correctness of the translation [3, 13]. In this paper, we study runtime behavior of the conversion interpretation of subtyping in call-by-value evaluation. We show that this translation preserves execution time and stack space within a factor determined by the types in a program, if subtyping relation is a partial order.

The translation of conversion interpretation changes the runtime behavior of programs in several respects. It inserts conversion functions and may increase the total number of closures allocated during execution. It translates tail-calls into non-tail-calls and, therefore, it may increase stack space usage. Although the translation causes these changes of runtime behavior, execution time and stack space usage are preserved asymptotically. This contrasts with type-directed unboxing of Leroy where both time and space complexity are not preserved [11].

Type systems with subtyping can be used to express information obtained by various program analyses such as control flow analyses [7]. One strategy of utilizing types obtained by program analysis is to adopt conversion interpretation of the subtyping. For example, it is possible to choose an optimized representation of values based on types and to insert conversion functions as type-directed unboxing of polymorphic languages [8]. In order to adopt this compilation method

we need to show that the conversion interpretation is safe with respect to performance. The results in this paper ensure safety with respect to execution time and stack space.

Both the safety proofs on execution time and stack space are based on the method of logical relations. The method of logical relations has been used for correctness proofs of many type-directed program transformations [8, 14, 12] and was extended to prove time safety of unboxing by Minamide and Garrigue [11]. One motivation of this work is to show that the method of logical relations can be extended to prove safety with respect to stack space. The structure of the proof we obtained for stack space is almost the same as that for the execution time. This is because the operational semantics profiling stack space can be formalized in the same manner as the semantics profiling execution time. We believe this is the first proof concerning stack space based on the method of logical relations.

We believe that the conversion interpretation is also safe with respect to heap space. However, it seems that it is difficult to extend the method of logical relations for heap space. We would like to study safety with respect to heap space in future work.

This paper is organized as follows. We start with a review of conversion interpretation and runtime behavior of translated programs. In Section 3 we define the language we will use in the rest of the paper and formally introduce conversion interpretation. We prove that conversion interpretation preserves stack space and execution time in Section 4 and Section 5. Finally we review related work and presents the conclusions.

2 Review of Conversion Interpretation

We review conversion interpretation of subtyping and intuitively explain that the interpretation preserves stack space and execution time if the subtyping relation is a partial order. Since the subtyping relation is transitive and reflexive, the subtyping relation is a partial order if it contains no equivalent types.

The conversion interpretation is a translation from a language with subtyping into a simply typed language without subtyping. The idea is to insert a conversion function (or coercion) where the subsumption rule is used. If the following subsumption rule is used in the typing derivation,

$$\frac{\Gamma \vdash M : \tau \quad \tau \leq \sigma}{\Gamma \vdash M : \sigma}$$

the conversion function $\mathsf{coerce}_{\tau \leq \sigma}$ of type $\tau \to \sigma$ is inserted and we obtain the following term.

$$\mathsf{coerce}_{\tau \leq \sigma}(M)$$

Coercion $\mathsf{coerce}_{\tau \leq \sigma}$ is inductively defined on structure of types. If there are two base types bigint and int for integers where int is a subtype of bigint, we need to

have a coercion primitive int2bigint of type int $\rightarrow$ bigint. A conversion function on function types is constructed as follows.

$$\lambda f.\lambda x.\mathsf{coerce}_{\tau_2 \leq \sigma_2}(f(\mathsf{coerce}_{\sigma_1 \leq \tau_1}(x)))$$

This is a coercion from $\tau_1 \rightarrow \tau_2$ to $\sigma_1 \rightarrow \sigma_2$.

We show that this interpretation of subtyping is safe with respect to execution time and stack space if the subtyping relation is a partial order. Intuitively, this holds because only a finite number of coercions can be applied to any value. If a subtyping relation is not a partial order, i.e., there exist two types τ and σ such that $\tau \leq \sigma$ and $\sigma \leq \tau$, we can easily construct counter examples for both execution time and stack space. A counter example for execution time is the following translation of term M of type τ,

$$\mathsf{coerce}_{\sigma \leq \tau}(\mathsf{coerce}_{\tau \leq \sigma}(\ldots(\mathsf{coerce}_{\sigma \leq \tau}(\mathsf{coerce}_{\tau \leq \sigma}(M)))))$$

where $\tau \leq \sigma$ and $\sigma \leq \tau$. The execution time to evaluate the coercions in the translation depends on the number of the coercions and cannot be bounded by a constant. It may be possible to avoid this silly translation, but it will be difficult to avoid this problem in general if we have equivalent types.

The conversion interpretation translates tail-call applications into non-tail-call applications. Let us consider the following translation of application $x\,y$.

$$\mathsf{coerce}_{\tau \leq \sigma}(x\,y)$$

Even if $x\,y$ is originally at a tail-call position, after translation it is not at a tail-call position. Therefore, it is not straightforward to show the conversion interpretation preserves stack space asymptotically. In fact, if we have equivalent types, we can demonstrate a counter example. Let us consider the following program where types `A` and `B` are equivalent.

```
fun f (0, x : A) = x                (* f: int * A -> A *)
  | f (n, x : A) = g (n-1, x)
and g (n, x : A) = f (n, x) : B     (* g: int * A -> B *)
```

We have a type annotation `f (n, x) : B` in the body of `g` and thus `g` has type `A -> B`. This program contains only tail-calls, and thus requires only constant stack space. By inserting conversion functions we obtain the following program:

```
fun f (0, x : A) = x
  | f (n, x : A) = B2A (g (n-1, x))
and g (n, x : A) = A2B (f (n, x))
```

where `A2B` and `B2A` are coercions between `A` and `B`. For this program, evaluation of `f n` requires stack space proportional to `n` since both the applications of `f` and `g` are not tail-calls.

In order to preserve time and stack space complexity, it is essential that the subtyping relation is a partial order. This ensures that there is no infinite subtyping chain of types if we consider only structural subtyping. Thus only a finite number of conversions can be applied to any value if the subtyping relation is a partial order.

3 Language and Conversion Interpretation

In this section we introduce a call-by-value functional language with subtyping and its conversion interpretation. We consider a call-by-value functional language with the following syntax.

$$V ::= x \mid \overline{i} \mid \underline{i} \mid \lambda x.M \mid \texttt{fix}^n\ x.\lambda y.M$$
$$M ::= V \mid M\ M \mid \texttt{let}\ x = M\ \texttt{in}\ M$$

There are two families of integers: $\overline{i}$ and $\underline{i}$ are integer values of types bigint and int respectively. The language includes bounded recursive functions where $\texttt{fix}^n\ x.\lambda y.M$ is expanded at most n times [4]. Any closed program with usual recursive functions can be simulated by bounded recursive functions.

For this language we consider a simple type system extended with subtyping. The types of the language are defined as follows.

$$\tau ::= \mathsf{bigint} \mid \mathsf{int} \mid \tau \to \tau$$

We consider two base types bigint and int where int is a subtype of bigint. A metavariable σ is also used to denote a type. The subtyping relation $\tau_1 \leq \tau_2$ is given by the following three rules.

$$\tau \leq \tau \qquad \mathsf{int} \leq \mathsf{bigint} \qquad \frac{\sigma_1 \leq \tau_1 \quad \tau_2 \leq \sigma_2}{\tau_1 \to \tau_2 \leq \sigma_1 \to \sigma_2}$$

The rule for transitivity is not included here because it can be derived from the other rules for this subtyping relation. We write $\tau < \sigma$ if $\tau \leq \sigma$ and $\tau \not\equiv \sigma$. It is clear that the subtyping relation is a partial order. The typing judgment has the following form:

$$\Gamma \vdash M{:}\tau$$

where Γ is a type assignment of the form $x_1{:}\tau_1, \ldots, x_n{:}\tau_n$. The rules of the type system are defined as follows.

$$\Gamma \vdash \overline{i}{:}\mathsf{bigint} \qquad \Gamma \vdash \underline{i}{:}\mathsf{int}$$

$$\frac{x{:}\tau \in \Gamma}{\Gamma \vdash x{:}\tau} \qquad \frac{\Gamma \vdash M_1{:}\tau_1 \to \tau_2 \quad \Gamma \vdash M_2{:}\tau_1}{\Gamma \vdash M_1 M_2{:}\tau_2}$$

$$\frac{\Gamma, x{:}\tau_1 \vdash M{:}\tau_2}{\Gamma \vdash \lambda x.M{:}\tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash M{:}\sigma \quad \sigma \leq \tau}{\Gamma \vdash M{:}\tau}$$

$$\frac{\Gamma, y{:}\tau_1 \to \tau_2, x{:}\tau_1 \vdash M{:}\tau_2}{\Gamma \vdash \texttt{fix}^n\ y.\lambda x.M{:}\tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash M_1{:}\tau_1 \quad \Gamma, x{:}\tau_1 \vdash M_2{:}\tau}{\Gamma \vdash \texttt{let}\ x = M_1\ \texttt{in}\ M_2{:}\tau}$$

Note that `let`-expressions do not introduce polymorphic types. They are used to simplify definition of coercions.

We consider a standard natural semantics for this language. A judgment has the following form: $M \Downarrow V$. The rules are given as follows.

$$V \Downarrow V \qquad \frac{M_1 \Downarrow V_1 \quad M_2[V_1/x] \Downarrow V}{\texttt{let}\ x = M_1\ \texttt{in}\ M_2 \Downarrow V}$$

$$\frac{M_1 \Downarrow \lambda x.M \quad M_2 \Downarrow V_2 \quad M[V_2/x] \Downarrow V}{M_1 M_2 \Downarrow V}$$

$$\frac{M_1 \Downarrow \texttt{fix}^{k+1}\, y.\lambda x.M \quad M_2 \Downarrow V_2 \quad M[\texttt{fix}^{k}\, y.\lambda x.M/y][V_2/x] \Downarrow V}{M_1 M_2 \Downarrow V}$$

When the recursive function $\texttt{fix}^{k+1} y.\lambda x.M$ is applied, the bound of the recursive function is decremented.

To formalize the conversion interpretation we need to introduce a target language without subtyping that includes a coercion primitive. We consider the following target language. The only extension is the coercion primitive $\mathsf{int2bigint}$ from int into bigint.

$$W ::= x \mid \underline{i} \mid \overline{i} \mid \lambda x.N \mid \texttt{fix}^n\, x.\lambda y.N$$

$$N ::= W \mid N\ N \mid \texttt{let}\ x = N\ \texttt{in}\ N \mid \mathsf{int2bigint}(N)$$

The operational semantics and type system of the language are almost the same as those of the source language. The rule of subsumption is excluded from the type system. The typing rule and evaluation of coercion are defined as follows.

$$\frac{N \Downarrow \underline{i}}{\mathsf{int2bigint}(N) \Downarrow \overline{i}} \qquad \frac{\Gamma \vdash N{:}\mathsf{int}}{\Gamma \vdash \mathsf{int2bigint}(N){:}\mathsf{bigint}}$$

The conversion interpretation is defined inductively on structure of the typing derivation of a program: the translation $\mathcal{C}[\![\Gamma \vdash M{:}\tau]\!]$ below gives a term of the target language.

$$\begin{aligned}
\mathcal{C}[\![\Gamma \vdash x{:}\tau]\!] &= x \\
\mathcal{C}[\![\Gamma \vdash \lambda x.M{:}\tau_1 \to \tau_2]\!] &= \lambda x.\mathcal{C}[\![\Gamma, x{:}\tau_1 \vdash M{:}\tau_2]\!] \\
\mathcal{C}[\![\Gamma \vdash \texttt{fix}^n\, y.\lambda x.M{:}\tau_1 \to \tau_2]\!] &= \texttt{fix}^n\, y.\lambda x.\mathcal{C}[\![\Gamma, y{:}\tau_1 \to \tau_2, x{:}\tau_1 \vdash M{:}\tau_2]\!] \\
\mathcal{C}[\![\Gamma \vdash M_1 M_2{:}\tau_2]\!] &= \mathcal{C}[\![\Gamma \vdash M_1{:}\tau_1 \to \tau_2]\!]\mathcal{C}[\![\Gamma \vdash M_2{:}\tau_1]\!] \\
\mathcal{C}[\![\Gamma \vdash M{:}\tau]\!] &= \mathsf{coerce}_{\sigma \le \tau}(\mathcal{C}[\![\Gamma \vdash M{:}\sigma]\!]) \\
\mathcal{C}[\![\Gamma \vdash \texttt{let}\ x = M_1\ \texttt{in}\ M_2{:}\tau_2]\!] &= \texttt{let}\ x = \mathcal{C}[\![\Gamma \vdash M_1{:}\tau_1]\!]\ \texttt{in}\ \mathcal{C}[\![\Gamma \vdash M_2{:}\tau_2]\!]
\end{aligned}$$

The coercion used in the translation is defined inductively on structure of derivation of subtyping as follows [1].

$$\begin{aligned}
\mathsf{coerce}_{\tau\le\tau}(M) &= M\\
\mathsf{coerce}_{\mathsf{int}\le\mathsf{bigint}}(M) &= \mathsf{int2bigint}(M)\\
\mathsf{coerce}_{\tau_1\to\tau_2\le\sigma_1\to\sigma_2}(M) &= \mathtt{let}\ x = M\ \mathtt{in}\ \lambda y.\mathsf{coerce}_{\tau_2\le\sigma_2}(x(\mathsf{coerce}_{\sigma_1\le\tau_1}(y)))
\end{aligned}$$

Note that $\mathsf{coerce}_{\tau\le\tau}(M)$ must be not only extensionally equivalent to M, but also intensionally equivalent to M. If we adopt $(\lambda x.x)\,M$ for $\mathsf{coerce}_{\tau\le\tau}(M)$, we have the same problem when we have equivalent types, and thus execution time and stack space are not preserved.

We define two measures, $\lfloor\tau\rfloor$ and $\lceil\tau\rceil$, of types as follows.

$$\begin{aligned}
\lfloor\mathsf{int}\rfloor &= 0 & \lceil\mathsf{int}\rceil &= 1\\
\lfloor\mathsf{bigint}\rfloor &= 1 & \lceil\mathsf{bigint}\rceil &= 0\\
\lfloor\tau_1\to\tau_2\rfloor &= \lceil\tau_1\rceil + \lfloor\tau_2\rfloor & \lceil\tau_1\to\tau_2\rceil &= \lfloor\tau_1\rfloor + \lceil\tau_2\rceil
\end{aligned}$$

It is clear that $\sigma < \tau$ implies $\lfloor\sigma\rfloor < \lfloor\tau\rfloor$ and $\lceil\sigma\rceil > \lceil\tau\rceil$. Since $\lfloor\tau\rfloor$ and $\lceil\tau\rceil$ are non-negative integers, we also obtain the following properties.

$$\tau_n < \ldots < \tau_1 < \tau_0 \quad\Rightarrow\quad \lfloor\tau_n\rfloor < \ldots < \lfloor\tau_1\rfloor < \lfloor\tau_0\rfloor \quad\Rightarrow\quad n \le \lfloor\tau_0\rfloor$$

$$\tau_0 < \tau_1 < \ldots < \tau_n \quad\Rightarrow\quad \lceil\tau_0\rceil > \lceil\tau_1\rceil > \ldots > \lceil\tau_n\rceil \quad\Rightarrow\quad n \le \lceil\tau_0\rceil$$

From the property we can estimate the maximum number of conversions applied a value of τ_0. In the following program, we know that $n \le \lceil\tau_0\rceil$ by the property.

$$\mathsf{coerce}_{\tau_{n-1}\le\tau_n}(\ldots(\mathsf{coerce}_{\tau_0\le\tau_1}(V))$$

Intuitively, this is the property that ensures that conversion interpretation preserves execution time and stack within a factor determined by types in a program.

4 Preservation of Stack Space

We show that coercion interpretation of subtyping preserves stack space within a factor determined by types occurring in a program. Strictly speaking, the factor is determined by the types occurring in the typing derivation used in translation of a program. We prove this property by the method of logical relations.

First we extend the operational semantics to profile stack space usage. The extended judgment has the form $M \Downarrow^n V$ where n models the size of stack space required to evaluate M to V. The following are the extended rules.

$$V \Downarrow^1 V \qquad \frac{M_1 \Downarrow^m V_1 \quad M_2[V_1/x] \Downarrow^n V}{\mathtt{let}\ x = M_1\ \mathtt{in}\ M_2 \Downarrow^{\max(m+1,n)} V}$$

[1] We assume that $\tau_1 \to \tau_2 \le \tau_1 \to \tau_2$ is not derived from $\tau_1 \le \tau_1$ and $\tau_2 \le \tau_2$, but from the axiom.

$$\frac{M_1 \Downarrow^l \lambda x.M \quad M_2 \Downarrow^m V_2 \quad M[V_2/x] \Downarrow^n V}{M_1 M_2 \Downarrow^{\mathsf{max}(l+1,m+1,n)} V}$$

$$\frac{M_1 \Downarrow^l \mathtt{fix}^{k+1}\, y.\lambda x.M \quad M_2 \Downarrow^m V_2 \quad M[\mathtt{fix}^k\, y.\lambda x.M/y][V_2/x] \Downarrow^n V}{M_1 M_2 \Downarrow^{\mathsf{max}(l+1,m+1,n)} V}$$

This semantics is considered to model evaluation by an interpreter: $M \Downarrow^n V$ means that a standard interpreter requires n stack frames to evaluate M to V. In the rule of application, evaluation of M_1 and M_2 are considered as non-tail-calls and evaluation of the body of the function is considered as a tail-call. This is the reason that the number of stack frames used to evaluate the application is $\mathsf{max}(l+1, m+1, n)$.

This semantics and the correspondence to a semantics modeling evaluation after compilation is discussed in [10]: the ratio to the stack space used by compiled executable code is bounded by the size of a program.

By the rule for values, a value V is evaluated to itself with 1 stack frame. Instead, you can choose $V \Downarrow^0 V$ as the rule for values. This choice does not matter much because the difference caused by the choice is always only 1 stack frame. We have chosen our rule to simplify our proofs.

We write $e \Downarrow^n$ if $e \Downarrow^n v$ for some v and $e \Downarrow^{\leq n}$ if $e \Downarrow^m$ for some $m \leq n$.

The main result of this section is that the conversion interpretation preserves stack space within a factor determined by the sizes of types appearing in a program.

Theorem 1. *Let $\mathcal{C}[\![\emptyset \vdash M{:}\tau]\!] = N$ and let C be an integer such that $C > \lfloor \sigma \rfloor + 3$ for all σ appearing in the derivation of $\emptyset \vdash M{:}\tau$. If $M \Downarrow^n V$ then $N \Downarrow^{\leq Cn} W$ for some W.*

Let us consider the following translation where the type of $\lambda x.\underline{1}$ is obtained by subsumption for $\mathsf{int} \to \mathsf{int} \leq \mathsf{int} \to \mathsf{bigint}$.

$$\mathcal{C}[\![(\lambda x.\underline{1})\underline{2}]\!] = (\mathtt{let}\ \ y = \lambda x.\underline{1}\ \mathtt{in}\ \ \lambda z.\mathsf{int2bigint}(y\ z))\underline{2}$$

The source program is evaluated with 2 stack frames.

$$(\lambda x.\underline{1})\underline{2} \Downarrow^2 \underline{1}$$

On the other hand, the translation is evaluated with 4 stack frames.

$$\frac{\lambda x.\underline{1} \Downarrow^1 \lambda x.\underline{1} \qquad \dfrac{V \Downarrow^1 V \quad \underline{2} \Downarrow^1 \underline{2} \quad \dfrac{(\lambda x.\underline{1})\ \underline{2} \Downarrow^2 \underline{1}}{\mathsf{int2bigint}((\lambda x.\underline{1})\ \underline{2}) \Downarrow^3 \overline{1}}}{(\lambda z.\mathsf{int2bigint}((\lambda x.\underline{1})\ \underline{z}))\ \underline{2} \Downarrow^3 \overline{1}}}{(\mathtt{let}\ \ y = \lambda x.\underline{1}\ \mathtt{in}\ \ \lambda z.\mathsf{int2bigint}(y\ z))\underline{2} \Downarrow^3 \overline{1}}$$

where $V \equiv \lambda z.\mathsf{int2bigint}((\lambda x.\underline{1})\ \underline{z})$. In this case, the factor of increase is $3/2$.

We prove the main theorem by the method of logical relations. Before defining the logical relations we define the auxiliary relation $V_1 V_2 \downarrow^n V$ defined as follows.

$$\frac{M[V_2/x] \Downarrow^n V}{(\lambda x.M)V_2 \downarrow^n V} \qquad \frac{M[V_2/x][\mathtt{fix}^k\, y.\lambda x.M/y] \Downarrow^n V}{(\mathtt{fix}^{k+1}\, y.\lambda x.M)V_2 \downarrow^n V}$$

By using this relation we can combine the two rules for the evaluation of the application into the following rule.

$$\frac{M_1 \Downarrow^l V_1 \quad M_2 \Downarrow^m V_2 \quad V_1 V_2 \downarrow^n V}{M_1 M_2 \Downarrow^{\max(l+1,m+1,n)} V}$$

This reformulation simplifies the definition of the logical relations and our proof. We define logical relations $V \approx^C_\tau W$ indexed by a type τ and a positive integer C as follows.

$$\begin{array}{ll} \underline{i} \approx^C_{\mathsf{int}} \underline{i} & \\ V \approx^C_{\mathsf{bigint}} \overline{i} & V = \underline{i} \text{ or } V = \overline{i} \\ V \approx^C_{\tau_1 \to \tau_2} W & \begin{cases} \text{for all } V_1 \approx^C_{\tau_1} W_1, \text{ if } V V_1 \downarrow^{n+1} V_2 \\ \text{ then } W W_1 \downarrow^{\le Cn + \lfloor \tau_2 \rfloor + 3} W_2 \text{ and } V_2 \approx^C_{\tau_2} W_2 \end{cases} \end{array}$$

We implicitly assume that V and W have type τ for $V \approx^C_\tau W$. The parameter C corresponds to the factor of increase of stack space usage. Note that the increase of stack space usage depends on only the range type τ_2 of a function type $\tau_1 \to \tau_2$. This is explained by checking the following translation of a function f of type $\tau_1 \to \tau_2$ [2].

$$\mathsf{coerce}_{\tau_1 \to \tau_2 \le \sigma_1 \to \sigma_2}(f) \equiv \lambda y.\mathsf{coerce}_{\tau_2 \le \sigma_2}(f\ \mathsf{coerce}_{\sigma_1 \le \tau_1}(y))$$

In this translation, only the coercion $\mathsf{coerce}_{\tau_2 \le \sigma_2}$ causes increase of stack space usage.

We first show that a conversion from τ to σ behaves well with respect to the logical relations.

Lemma 1. *If $\tau < \sigma$ and $V \approx^C_\tau W$ then $\mathsf{coerce}_{\tau \le \sigma}(W) \Downarrow^2 W'$ and $V \approx^C_\sigma W'$ for some W'.*

Proof. By induction on derivation of $\tau < \sigma$.

Case: $\mathsf{int} \le \mathsf{bigint}$. By the definition of $V \approx^C_{\mathsf{int}} W$, both V and W must be $\underline{i}$ for some i. Since $\mathsf{coerce}_{\mathsf{int} \le \mathsf{bigint}}(\underline{i}) = \mathsf{int2bigint}(\underline{i})$, we have $\mathsf{int2bigint}(\underline{i}) \Downarrow^2 \overline{i}$ and $\underline{i} \approx^C_{\mathsf{bigint}} \overline{i}$.

Case: $\tau \equiv \tau_1 \to \tau_2$ and $\sigma \equiv \sigma_1 \to \sigma_2$ where $\sigma_1 \le \tau_1$ and $\tau_2 \le \sigma_2$. There are two subcases: $\tau_2 < \sigma_2$ and $\tau_2 \equiv \sigma_2$. We show the former case here. The proof the latter case is similar.

$$\mathsf{coerce}_{\tau \le \sigma}(W) \Downarrow^2 \lambda y.\mathsf{coerce}_{\tau_2 \le \sigma_2}(W(\mathsf{coerce}_{\sigma_1 \le \tau_1}(y)))$$

Let $V_0 \approx^C_{\sigma_1} W_0$ and $V V_0 \downarrow^{m+1} V_2$. By induction hypothesis,

$$\mathsf{coerce}_{\sigma_1 \le \tau_1}(W_0) \Downarrow^{\le 2} W_1$$

and $V_0 \approx^C_{\tau_1} W_1$ for some W_1. By definition of the logical relations

$$W W_1 \downarrow^{\le Cm + \lfloor \tau_2 \rfloor + 3} W_3$$

[2] Strictly speaking, it is `let` $x = f$ `in` $\lambda y.\mathsf{coerce}_{\tau_2 \le \sigma_2}(x\ \mathsf{coerce}_{\sigma_1 \le \tau_1}(y))$.

and $V_2 \approx^C_{\tau_2} W_3$ for some W_3. Then we obtain the following evaluation.

$$\frac{W \Downarrow^1 W \quad \mathsf{coerce}_{\sigma_1 \le \tau_1}(W_0) \Downarrow^{\le 2} W_1 \quad W W_1 \downarrow^{\le Cm + \lfloor \tau_2 \rfloor + 3} W_3}{W(\mathsf{coerce}_{\sigma_1 \le \tau_1}(W_0)) \Downarrow^{\le \mathsf{max}(2,3,Cm+\lfloor \tau_2 \rfloor + 3)} W_3}$$

where $\mathsf{max}(2, 3, Cm + \lfloor \tau_2 \rfloor + 3) = Cm + \lfloor \tau_2 \rfloor + 3$.
By induction hypothesis,

$$\mathsf{coerce}_{\tau_2 \le \sigma_2}(W_3) \Downarrow^2 W_2$$

and $V_2 \approx^C_{\sigma_2} W_2$ for some W_2. Then

$$\mathsf{coerce}_{\tau_2 \le \sigma_2}(W(\mathsf{coerce}_{\sigma_1 \le \tau_1}(W_0))) \Downarrow^{\le Cm + \lfloor \tau_2 \rfloor + 4} W_2$$

where $Cm + \lfloor \tau_2 \rfloor + 4 \le Cm + \lfloor \sigma_2 \rfloor + 3$

□

The next lemma indicates that we can choose a constant C such that the evaluation of a source program and its translation are related by C. For ρ and ρ' two environments with the same domain, $\rho \approx^C_\Gamma \rho'$ means that they are pointwise related. The main theorem is obtained by restricting this lemma to $\Gamma = \emptyset$ and taking C such that $C > \lfloor \sigma \rfloor + 3$ for all σ appearing in the typing derivation.

Lemma 2. *Let C be an integer such that $C > \lfloor \sigma \rfloor$ for all σ appearing in the derivation of $\Gamma \vdash M{:}\tau$. Let $\mathcal{C}[\![\Gamma \vdash M{:}\tau]\!] = N$ and $\rho \approx^C_\Gamma \rho'$.*
If $\rho(M) \Downarrow^{n+1} V$ then $\rho'(N) \Downarrow^{\le Cn + \lfloor \tau \rfloor + 3} W$ and $V \approx^C_\tau W$ for some W.

Proof. By lexicographic induction on derivation of $\Gamma \vdash M{:}\tau$ and the sum of bounds of recursive functions in M.

Case: $\Gamma \vdash M{:}\sigma$ is derived from $\Gamma \vdash M{:}\tau$ and $\tau \le \sigma$. We assume $\tau < \sigma$. The case of $\tau \equiv \sigma$ is trivial. By definition, N must be $\mathsf{coerce}_{\tau \le \sigma}(N_0)$ for some N_0 and $\mathcal{C}[\![\Gamma \vdash M : \tau]\!] = N_0$. By induction hypothesis,

$$\rho'(N_0) \Downarrow^{\le Cn + \lfloor \tau \rfloor + 3} W_0$$

and $V \approx^C_\tau W_0$ for some W_0. By Lemma 1,

$$\rho'(\mathsf{coerce}_{\tau \le \sigma}(N_0)) \Downarrow^{\le Cn + \lfloor \tau \rfloor + 3 + 1} W$$

and $V \approx^C_\sigma W$ for some W. The proof of this case completes since $Cn + \lfloor \tau \rfloor + 3 + 1 \le Cn + \lfloor \sigma \rfloor + 3$.

Case: $\Gamma \vdash M_1 M_2{:}\tau_2$ is derived from $\Gamma \vdash M_1{:}\tau_1 \to \tau_2$ and $\Gamma \vdash M_2{:}\tau_1$. By the definition of the translation $N \equiv N_1 N_2$ for some N_1 and N_2.
$\rho(M_1 M_2) \Downarrow^{k+1} V$ is derived from $\rho(M_1) \Downarrow^l V_1$ and $\rho(M_2) \Downarrow^m V_2$ and $V_1 V_2 \downarrow^n V$ where $l \le k$, $m \le k$ and $n \le k + 1$. By induction hypothesis for M_1,

$$\rho'(N_1) \Downarrow^{C(l-1) + \lfloor \tau_1 \to \tau_2 \rfloor + 3} W_1$$

and $V_1 \approx^C_{\tau_1 \to \tau_2} W_1$ for some W_1. Then we have $\rho'(N_1) \Downarrow^{\leq Cl+2} W_1$ because $\lfloor \tau_1 \to \tau_2 \rfloor + 1 \leq C$. By induction hypothesis for M_2,

$$\rho'(N_2) \Downarrow^{C(m-1)+\lfloor \tau_1 \rfloor + 3} W_2$$

and $V_2 \approx^C_{\tau_1} W_2$ for some W_2. Then we also have $\rho'(N_2) \Downarrow^{\leq Cm+2} W_2$ because $\lfloor \tau_1 \rfloor + 1 \leq C$. By definition of the logical relations

$$W_1 W_2 \downarrow^{\leq C(n-1)+\lfloor \tau_2 \rfloor + 3} W$$

and $V \approx^C_{\tau_2} W$ for some W. We have the following inequality.

$$\mathsf{max}(Cl + 2 + 1, Cm + 2 + 1, C(n-1) + \lfloor \tau_2 \rfloor + 3) \leq Ck + \lfloor \tau_2 \rfloor + 3$$

Hence,

$$\rho'(N_1 N_2) \Downarrow^{\leq Ck+\lfloor \tau_2 \rfloor + 3} W$$

Case: $\Gamma \vdash \mathtt{fix}^{a+1}\ y.\lambda x.M{:}\tau_1 \to \tau_2$ is derived from $\Gamma, y{:}\tau_1 \to \tau_2, x{:}\tau_1 \vdash M{:}\tau_2$.
By definition, $\mathcal{C}[\![\Gamma, y{:}\tau_1 \to \tau_2, x{:}\tau_1 \vdash M{:}\tau_1]\!] = N$ for some N.
We have the following evaluation.

$$\rho(\mathtt{fix}^{a+1}\ y.\lambda x.M) \Downarrow^1 \rho(\mathtt{fix}^{a+1}\ y.\lambda x.M)$$

$$\rho'(\mathtt{fix}^{a+1}\ y.\lambda x.N) \Downarrow^1 \rho'(\mathtt{fix}^{a+1}\ y.\lambda x.N)$$

Let $V \approx^C_{\tau_1} W$ and $\rho(M)[V/x][\rho(\mathtt{fix}^a\ y.\lambda x.M)/y] \Downarrow^{n+1} V'$.
By induction hypothesis,

$$\rho(\mathtt{fix}^a\ y.\lambda x.M) \approx^C_{\tau_1 \to \tau_2} \rho'(\mathtt{fix}^a\ y.\lambda x.N)$$

Let $\rho_0 = \rho[V/x][\rho(\mathtt{fix}^a\ y.\lambda x.M)/y]$ and $\rho'_0 = \rho'[W/x][\rho'(\mathtt{fix}^a\ y.\lambda x.N)/y]$.
We have $\rho_0 \approx^C_{\Gamma, y:\tau_1 \to \tau_2, x:\tau_1} \rho'_0$. By induction hypothesis,

$$\rho'_0(N) \Downarrow^{Cn+\lfloor \tau_2 \rfloor + 3} W'$$

and $V' \approx^C_{\tau_2} W'$. Hence, $\rho(\mathtt{fix}^{a+1}\ y.\lambda x.M) \approx^C_{\tau_1 \to \tau_2} \rho'(\mathtt{fix}^{a+1}\ y.\lambda x.N)$.

□

5 Preservation of Execution Time

We introduce the operational semantics profiling execution time and outline the proof that the coercion interpretation of subtyping is also safe with respect to execution time. The operational semantics for execution time is a simple extension of the standard semantics as that for stack space. As the previous section, we first extend judgment of operational semantics to the following form:

$$M \Downarrow^n V$$

where n represents execution time to evaluate M to V. For the rule of application we use an auxiliary relation: $V_1 V_2 \downarrow^n V$ as before. The rules are given as follows.

$$V \Downarrow^1 V \qquad \frac{M_1 \Downarrow^m V_1 \quad M_2[V_1/x] \Downarrow^n V}{\texttt{let } x = M_1 \texttt{ in } M_2 \Downarrow^{m+n+1} V}$$

$$\frac{M_1 \Downarrow^l V_1 \quad M_2 \Downarrow^m V_2 \quad V_1 V_2 \downarrow^n V}{M_1 M_2 \Downarrow^{l+m+n+1} V}$$

$$\frac{M[V_2/x] \Downarrow^n V}{(\lambda x.M) V_2 \downarrow^n V} \qquad \frac{M[V_2/x][\texttt{fix}^k\, y.\lambda x.M/y] \Downarrow^n V}{(\texttt{fix}^{k+1}\, y.\lambda x.M) V_2 \downarrow^n V}$$

All the rules are a straightforward extension of the standard rules.

Then it is shown that the conversion interpretation preserves execution time within a factor determined by the types appearing in a program.

Theorem 2. *Let $\mathcal{C}[\![\emptyset \vdash M : \tau]\!] = N$ and let C be an integer such that $C > 7\lfloor\sigma\rfloor$ for all σ appearing in the derivation of $\emptyset \vdash M : \tau$. If $M \Downarrow^n V$ then $N \Downarrow^{\leq Cn} W$ for some W.*

The factor of slowdown $7\lfloor\sigma\rfloor$ is bigger than the factor of increase of stack space $\lfloor\sigma\rfloor$. To prove this theorem, w use the method of logical relations which are indexed by a slowdown factor as well as a type. The relations $V \approx^C_\tau W$ are defined as follows.

$$\underline{i} \approx^C_{\mathsf{int}} \underline{i}$$

$$V \approx^C_{\mathsf{bigint}} \overline{i} \qquad V = \underline{i} \text{ or } V = \overline{i}$$

$$V \approx^C_{\tau_1 \to \tau_2} W \qquad \begin{cases} \text{for all } V_1 \approx^C_{\tau_1} W_1, \text{ if } V V_1 \downarrow^{n+1} V_2 \\ \text{ then } W W_1 \downarrow^{Cn+7\lfloor\tau_1\to\tau_2\rfloor+1} W_2 \text{ and } V_2 \approx^C_{\tau_2} W_2 \end{cases}$$

The important difference from the relations for stack space is that slowdown of the applications depends on the domain type τ_1 as well as the range type τ_2 of a function type $\tau_1 \to \tau_2$.

With this definition, the main theorem is proved in the same manner as the proof for stack space. It is shown that a conversion function behaves well with respect to the logical relations as before. Then the generalization of the main theorem is proved by induction on the derivation of the conversion interpretation of a program.

6 Conclusions and Related Work

We have shown that conversion interpretation of subtyping preserves execution time and stack space within a factor determined by the types in a program if the subtyping relation is a partial order. Type-directed unboxing of Leroy is a translation similar to conversion interpretation of subtyping, but it does not preserve execution time and stack space. This is because conversions of equivalent types appear in type-directed unboxing.

We have considered only a very simple type system which does not include product types and recursive types. We believe the results in this paper can be easily extended for product types. However, our results cannot be extended for recursive types. If we consider recursive types, cost of one application of coercion cannot be bounded by a constant as Leroy discussed in his work on type-directed unboxing for polymorphic languages [8]. Thus the conversion interpretation does not preserve execution time nor stack space usage in the presence of subtyping on recursive types.

We have shown that the conversion interpretation is safe with respect to time and stack space by the method of logical relations. We believe the conversion interpretation is also safe with respect to heap space, but it will be difficult to adopt the same method for heap space. We have no idea how to formalize logical relations for heap space because the semantics profiling heap space is much more complicated than those for time and stack space.

In the rest of this section I review other proof methods to show safety of program transformations with respect to performance.

David Sands studied time safety of transformations for call-by-name languages [16, 15]. In his study he extended applicative bisimulation and its context lemma to account execution time. Applicative bisimulation with the context lemma greatly simplifies safety proofs of many program transformations. As with the method of logical relations, it will be difficult to extend this method if we consider heap space or various extensions of languages.

Another approach is to analyze states of evaluation more directly, where proofs are often based on induction on length of evaluation. Blelloch and Greiner showed that an implementation of NESL based on an abstract machine preserves execution time and space within a constant factor based on this approach [2]. Minamide showed that the CPS transformation preserves space within a constant factor [9]. Gustavsson and Sands developed a theory of a space improvement relation for a call-by-need programming language [5, 6]. They clarified their proofs by considering evaluation of programs with holes based on a context calculus [17]. Bakewell and Runciman proposed an operational model for lazy functional programming languages based on graph rewriting [1]. As a proof method for the model they considered an extension of bisimulation.

Acknowledgments

This work is partially supported by Japan Society for the Promotion of Science, Grant-in-Aid for Encouragement of Young Scientists of Japan, No. 13780193, 2001.

References

[1] A. Bakewell and C. Runciman. A model for comparing the space usage of lazy evaluators. In *Proceedings of the 2nd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 151–162, 2000.

[2] G. E. Blelloch and J. Greiner. A provably time and space efficient implementation of NESL. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 213–225, 1996.

[3] V. Breazu-Tannen, C. A. Gunter, and A. Scedrov. Computing with coercions. In *Proceedings of the 1990 ACM Conference on LISP and Functional programming*, pages 44–60, 1990.

[4] C. A. Gunter. *Semantics of Programming Languages*, chapter 4. The MIT Press, 1992.

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

[6] J. Gustavsson and D. Sands. Possibilities and limitations of call-by-need space improvement. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, pages 265–276, 2001.

[7] N. Heintze. Control-flow analysis and type systems. In *Proceedings of the 1995 International Static Analysis Symposium*, volume 983 of *LNCS*, pages 189–206, 1995.

[8] X. Leroy. Unboxed objects and polymorphic typing. In *the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 177–188, 1992.

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

[10] Y. Minamide. A new criterion for safe program transformations. In *Proceedings of the Forth International Workshop on Higher Order Operational Techniques in Semantics (HOOTS)*, volume 41(3) of *ENTCS*, Montreal, 2000.

[11] Y. Minamide and J. Garrigue. On the runtime complexity of type-directed unboxing. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional programming*, pages 1–12, 1998.

[12] Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *Proceeding of the ACM Symposium on Principles of Programming Languages*, pages 271–283, 1996.

[13] J. C. Mitchell. *Foundations for Programming Languages*, chapter 10. The MIT Press, 1996.

[14] A. Ohori. A polymorphic record calculus and its compilation. *ACM Transaction on Programming Languages and Systems*, 17(6):844–895, 1995.

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

[16] D. Sands. Proving the correctness of recursion-based automatic program transformations. *Theoretical Computer Science*, 167(1&2):193–233, 1996.

[17] D. Sands. Computing with contexts: A simple approach. In *Proceedings of the Second Workshop on Higher-Order Operational Techniques in Semantics (HOOTS II)*, volume 10 of *ENTCS*, 1998.