Selective Tail Call Elimination

Yasuhiko Minamide

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

Abstract. Tail calls are expected not to consume stack space in most functional languages. However, there is no support for tail calls in some environments. Even in such environments, proper tail calls can be implemented with a technique called a trampoline. To reduce the overhead of trampolining while preserving stack space asymptotically we propose selective tail call elimination based on an effect system. The effect system infers the number of successive tail calls generated by the execution of an expression, and trampolines are introduced only when they are necessary.

1 Introduction

Tail calls are expected not to consume stack space in most functional languages. Implementation of proper tail calls requires some support from a target environment, but some environments including C and Java Virtual Machine (JVM) [10] do not provide such support. Even in such environments, proper tail calls can be implemented with a technique called a *trampoline*. However, the trampoline technique is based on a non-standard calling convention and is considered to introduce too much overhead. Thus, most compilers for such environments do not adopt trampolining and abandon proper tail calls [2, 16, 3].

To solve this problem we selectively introduce trampolines based on an effect system. Effect systems were originally proposed to infer side-effects of a program by Gifford and Lucassen [8, 11], and an extension was applied to estimate the execution time of a program [6, 14]. Our effect system infers the number of successive tail calls that can be generated by the execution of an expression. Based on effects, functions are divided into two kinds: those that lead to a finite number of successive tail calls and those that may lead to an infinite number of successive tail calls. Then, it is necessary to adopt trampolines only for the latter kind. In this manner we can reduce the overhead of trampolining while preserving stack space asymptotically.

Our effect system includes the rule of subtyping, and some applications may call both kinds of functions. To support subtyping on functions and enable selective elimination of tail calls, we introduce a transformation that translates a function into a record containing two functions supporting different calling conventions. We prove that the increase of stack space usage caused by this transformation is bounded by a factor determined by the effect system.

We have incorporated selective tail call elimination into the MLj compiler [2], which complies Standard ML into Java bytecodes. Results for benchmark programs show that our effect system is strong enough to indicate that most functions are safe without tail call elimination. We show that there is little degradation of performance for most benchmark programs. We also measure impact of tail call elimination on stack space usage. It shows that general tail call elimination sometimes greatly reduces stack space usage.

This paper is organized as follows. In Section 2, we review the trampolining transformation and clarify its problem. In Section 3, we introduce our selective tail call elimination and prove its soundness. In Section 4, the effect system is extended with a wider class of effects. In Section 5 we outline effect inference for our effect system. Section 6 shows some examples where our effect system shows that a function is unsafe without tail call elimination. In Section 7 we describe our implementation and discuss the results of our experiments. Finally, we review related work and present our conclusions.

2 Tail call elimination with trampolines

In functional languages, loops are often expressed with tail calls and thus they are expected not to consume stack space. Let us consider the following program. The application `sum (x-1, a+x)` in this program is called a tail call because there is nothing to do in the function `sum` after the application.

```
let fun sum (x, a) = if x = 0 then a else sum (x-1, a+x)
in
    sum (y, 0)
end
```

If the tail call is properly implemented, the program above requires only constant stack space. On the other hand, if it is not properly implemented, it requires stack space proportional to `y`. Loops are often expressed in this way in functional languages and thus it is important to implement proper tail calls.

However, it is not straightforward to implement proper tail calls in environments without direct support of tail calls, such as in C and JVM. In such environments it is possible to implement proper tail calls with a non-standard calling convention called a trampoline [1]. We will explain the trampoline technique as a source-to-source transformation. The example above is transformed into the following program.

```
datatype 'a ret = Thunk (unit -> 'a ret) | Val 'a

fun loop (Val x) = x
  | loop (Thunk f) = loop (f ())
```

```
let fun sum (x, a) =
        if x=0 then Val a else Thunk (fn () => sum (x-1, a+x))
in
    loop (sum (y, 0))
end
```

The tail call in the program is translated into the creation of a closure. Then the closure is called from the loop [1]. This technique has been used in a Standard ML to C compiler [17] and is also useful to implement various features of programming languages [7]. However, it is clear that this technique introduces a lot of overhead. Thus, most compilers into C and JVM do not adopt trampolining and abandon proper tail calls.

3 Selective tail call elimination

To reduce the overhead introduced by implementation of proper tail calls by techniques such as trampolining, we propose selective elimination of tail calls that preserves the asymptotic complexity of the stack space.

The basic idea is that if the number of successive tail calls generated by a function call is bounded by some constant, it is not necessary to adopt a trampoline for the function. Let us consider the following program.

```
let fun f x = x
    fun g x = f x
in
    g 0
end
```

There are two tail calls in this program: `f x` and `g 0`. However, it is not necessary to introduce a trampoline for this program. The execution of the function `f` leads to no tail call and thus `f x` generates only one successive tail call. The function `g` calls `f` at a tail-call position and thus the function call `g 0` generates two successive tail calls. Since the number of successive tail calls is bounded in this program, it is safe to execute this program without tail call elimination.

On the other hand, in the following program the number of successive tail calls that the application `h y` leads to cannot be bounded. Thus it is necessary to introduce a trampoline for the program.

```
let fun h x = if x > 0 then h (x - 1) else 0
in
    h y
end
```

If we can statically analyze the number of successive tail calls generated by execution of each function, we can avoid introducing trampolines for functions

[1] The function `loop` should be implemented so as not to consume stack space.

satisfying some safety condition and selectively eliminate tail calls with trampolines. In the next section we introduce an effect system to eliminate tail calls selectively.

3.1 Effect system

We introduce sized effects to check the number of successive tail calls generated by the execution of an expression. The following are the effects we consider.

$$\rho ::= \omega \mid 0 \mid 1 \mid \ \ldots$$

We consider an order relation $\rho \leq \rho'$ between effects: the usual order relation between natural numbers and $i \leq \omega$ for any natural number i. For a finite effect i, the effect i^+ is defined as $i^+ = i + 1$. Then we consider the following types where a function type is annotated with an effect.

$$\tau ::= \mathsf{nat} \mid \tau \rightarrow^{\rho} \tau$$

We include `nat` as a base type for natural numbers and we also use other base types in examples. The functions we discussed above have the following types.

$\texttt{f : int} \rightarrow^{1} \texttt{int}$
$\texttt{g : int} \rightarrow^{2} \texttt{int}$
$\texttt{h : int} \rightarrow^{\omega} \texttt{int}$

Application of `f` at a tail-call position leads to one tail call. Since the application of `g` leads to a subsequent application of `f`, the type of `g` is annotated with 2. On the other hand, the number of successive tail calls generated by `h` cannot be bounded. Thus its type is annotated with ω.

We consider the subtyping relation defined by the following rules.

$$\tau \leq \tau \qquad \frac{\tau_1 \leq \tau_1' \quad \tau_2' \leq \tau_2 \quad \rho' \leq \rho}{\tau_1' \rightarrow^{\rho'} \tau_2' \leq \tau_1 \rightarrow^{\rho} \tau_2}$$

We formalize our effect system for the following language, where abstractions and applications are annotated with effects.

$$\begin{aligned} V &::= x \mid 0 \mid \mathsf{suc}(V) \mid \mathsf{fix}\, x.\lambda^{\rho} y.M \\ M &::= V \mid @^{\rho} M M \mid \mathsf{case}\, V\, \mathsf{of}\, 0 \Rightarrow M,\, \mathsf{suc}(x) \Rightarrow M \end{aligned}$$

The values 0 and $\mathsf{suc}(V)$ are values of natural numbers. We will discuss how to annotate the language without effect annotations in Section 5. Judgments of the effect system have the following forms:

$$E \vdash V : \tau$$

$$E \vdash M : \tau \,!\, \rho$$

where ρ represents the maximum number of successive tail calls generated by evaluation of M. The rules of the effect system are defined in Figure 1. These rules are standard in effect systems. The rules of abstraction and application are explained as follows.

$$\frac{x : \tau \in E}{E \vdash x : \tau} \qquad E \vdash 0 : \mathsf{nat} \qquad \frac{E \vdash V : \mathsf{nat}}{E \vdash \mathsf{suc}(V) : \mathsf{nat}} \qquad \frac{E \vdash V : \tau}{E \vdash V : \tau\,!\,0}$$

$$\frac{E, x : \tau_1 \rightarrow^{i^+} \tau_2, y : \tau_1 \vdash M : \tau_2\,!\,i}{E \vdash \mathsf{fix}\, x.\lambda^{i^+} y.M : \tau_1 \rightarrow^{i^+} \tau_2} \qquad \frac{E, x : \tau_1 \rightarrow^{\omega} \tau_2, y : \tau_1 \vdash M : \tau_2\,!\,\omega}{E \vdash \mathsf{fix}\, x.\lambda^{\omega} y.M : \tau_1 \rightarrow^{\omega} \tau_2}$$

$$\frac{E \vdash M_1 : \tau_1 \rightarrow^{\rho} \tau_2\,!\,\rho_1 \quad E \vdash M_2 : \tau_1\,!\,\rho_2}{E \vdash @^{\rho} M_1 M_2 : \tau_2\,!\,\rho} \qquad \frac{E \vdash M : \tau'\,!\,\rho' \quad \rho' \leq \rho \quad \tau' \leq \tau}{E \vdash M : \tau\,!\,\rho}$$

$$\frac{E \vdash V : \mathsf{nat} \quad E \vdash M_1 : \tau\,!\,\rho \quad E, x : \mathsf{nat} \vdash M_2 : \tau\,!\,\rho}{E \vdash \mathsf{case}\, V \,\mathsf{of}\, 0 \Rightarrow M_1,\, \mathsf{suc}(x) \Rightarrow M_2 : \tau\,!\,\rho}$$

Fig. 1. Type system

- If the body of a function leads to i successive tail calls, application of the function at a tail-call position leads to i^+ successive tail calls.
- If the body of a function has effect ω, the function has effect ω. That means successive tail calls generated by application of the function cannot be bounded.
- The effects of M_1 and M_2 are ignored in the rule of application because they correspond to evaluation at non-tail-call positions. Thus, the effect of the application $@^{\rho} M_1 M_2$ is determined only by the effect annotation of the function type.

To discuss the soundness of selective tail elimination we introduce an operational semantics that profiles stack space, and models evaluation with proper implementation of tail calls. We define a big-step operational semantics with the following judgments:

$$\vdash_T M \downarrow^i V$$

$$\vdash_N M \downarrow^i V$$

with the meanings that M is evaluated to V with i stack frames at a tail-call position or a non-tail-call position, respectively. A whole program is considered to be evaluated at a tail-call position. The rules are given in Figure 2 where $\vdash_\alpha M \downarrow^n V$ means the rule holds both for $\alpha = N$ and $\alpha = T$. At a tail-call position, the evaluation of the body of a function requires no new stack frame. Thus, the stack space required for evaluation of the application is $\mathsf{max}(l, m, n)$. On the other hand, at a non-tail-call position, it requires a new stack frame: the stack space is $\mathsf{max}(l, m, n + 1)$. This semantics models stack space usage when a program is executed after compilation. Correspondence to a semantics that models execution based on an interpreter is discussed in [12].

With respect to the operational semantics, the soundness of the type system in the usual sense is proved. However, the following lemma says nothing about effects inferred by the effect system.

$$\vdash_T V \downarrow^0 V \qquad\qquad \vdash_N V \downarrow^0 V$$

$$\frac{\vdash_N M_1 \downarrow^l \mathsf{fix}\, x.\lambda^{\rho'} y.M_0 \quad \vdash_N M_2 \downarrow^m V_2 \quad \vdash_T M_0[\mathsf{fix}\, x.\lambda^{\rho'} y.M_0/x][V_2/y] \downarrow^n V}{\vdash_T @^\rho M_1 M_2 \downarrow^{\mathsf{max}(l,m,n)} V}$$

$$\frac{\vdash_N M_1 \downarrow^l \mathsf{fix}\, x.\lambda^{\rho'} y.M_0 \quad \vdash_N M_2 \downarrow^m V_2 \quad \vdash_T M_0[\mathsf{fix}\, x.\lambda^{\rho'} y.M_0/x][V_2/y] \downarrow^n V}{\vdash_N @^\rho M_1 M_2 \downarrow^{\mathsf{max}(l,m,n+1)} V}$$

$$\frac{\vdash_\alpha M_1 \downarrow^n V}{\vdash_\alpha \mathsf{case}\, 0\, \mathsf{of}\, 0 \Rightarrow M_1,\, \mathsf{suc}(x) \Rightarrow M_2 \downarrow^n V}$$

$$\frac{\vdash_\alpha M_2[V_0/x] \downarrow^n V}{\vdash_\alpha \mathsf{case}\, \mathsf{suc}(V_0)\, \mathsf{of}\, 0 \Rightarrow M_1,\, \mathsf{suc}(x) \Rightarrow M_2 \downarrow^n V}$$

Fig. 2. Operational semantics

Lemma 1 (Soundness).

1. *If $\emptyset \vdash M : \tau!\rho$ and $\vdash_N M \downarrow^m V$ then $\emptyset \vdash V : \tau$.*
2. *If $\emptyset \vdash M : \tau!\rho$ and $\vdash_T M \downarrow^m V$ then $\emptyset \vdash V : \tau$.*

3.2 Transformation

We introduce a program transformation that selectively eliminates tail calls based on effect annotations. The idea is to eliminate tail calls of the form $@^\omega MM$ with trampolining and to adopt the standard calling convention for $@^i MM$ [2].

However, implementation based on this idea is not so simple because $\tau_1 \rightarrow^i \tau_2$ can be considered as $\tau_1 \rightarrow^\omega \tau_2$ by subtyping. Let us consider the following program.

```
let fun f x = x
    fun g x = if ... then x else g (x - 1)
in
    (if ... then f else g) 0
end
```

The functions `f` and `g` have types $\mathsf{int} \rightarrow^1 \mathsf{int}$ and $\mathsf{int} \rightarrow^\omega \mathsf{int}$, respectively. It is not possible to determine the kinds of functions that are called by the application in the body of the `let`-expression. Thus, it is not straightforward to compile the application in the body.

To solve this problem, we represents $\tau_1 \rightarrow^i \tau_2$ as a record that contains two functions: one for a trampoline and one for the standard calling convention. Then subtyping on function types is translated into record (object) subtyping.

[2] We assume that the standard calling convention does not support tail call elimination.

For the target language of the transformation we consider the following types.

$$\sigma ::= \mathsf{nat} \mid \sigma \to \sigma \mid \sigma \to^t \sigma \mid \{\mathsf{fun} : \sigma, \mathsf{tfun} : \sigma\} \mid \{\mathsf{tfun} : \sigma\}$$

There are two kinds of function types: $\sigma \to \sigma$ uses the standard calling convention without proper tail calls, and $\sigma \to^t \sigma$ uses the non-standard calling convention with tail call elimination. There is no subtyping relation between $\sigma_1 \to \sigma_2$ and $\sigma_1 \to^t \sigma_2$. Two kinds of record types, $\{\mathsf{tfun} : \sigma\}$ and $\{\mathsf{fun} : \sigma, \mathsf{tfun} : \sigma\}$, are included to translate function types and we consider the following subtyping relation between them.

$$\{\mathsf{fun} : \sigma_1, \mathsf{tfun} : \sigma_2\} \leq \{\mathsf{tfun} : \sigma_2\}$$

Then our transformation translates function types into record types so that the subtyping relation is preserved. The translation of types $|\tau|$ is defined as follows:

$$|\tau_1 \to^\omega \tau_2| = \{\mathsf{tfun} : |\tau_1| \to^t |\tau_2|\}$$

$$|\tau_1 \to^i \tau_2| = \{\mathsf{fun} : |\tau_1| \to |\tau_2|, \mathsf{tfun} : |\tau_1| \to^t |\tau_2|\}$$

We therefore have the following translation of subtyping.

$$|\tau_1 \to^i \tau_2| \leq |\tau_1 \to^\omega \tau_2|$$

This translation of subtyping is natural for compilation to Java bytecodes because JVM has subtyping on objects through inheritance of classes.

The syntax of the target language is defined as follows. It includes two kinds of abstraction and application, and syntax for records and field selection.

$$\begin{aligned} N &::= W \mid @NN \mid @^t NN \mid N.\mathsf{fun} \mid N.\mathsf{tfun} \mid \mathsf{case}\, W \,\mathsf{of}\, 0 \Rightarrow N,\ \mathsf{suc}(x) \Rightarrow N \\ W &::= x \mid 0 \mid \mathsf{suc}(W) \mid \lambda x.N \mid \lambda^t x.N \mid \mathsf{fix}\, x.\{\mathsf{fun} = W, \mathsf{tfun} = W\} \mid \\ &\quad\ \mathsf{fix}\, x.\{\mathsf{tfun} = W\} \end{aligned}$$

The fields of a record expression are restricted to values: this restriction is sufficient for our transformation. The type system of the target language is standard and does not include effects. The definition of the type system is shown in Appendix A.

We define an operational semantics of the target language in the same manner as that of the source language: we define a big-step operational semantics with following judgments.

$$\vdash_T N \downarrow^i W$$
$$\vdash_N N \downarrow^i W$$

The main rules are defined as follows. It should be noted that tail calls are not properly implemented for application $@N_1N_2$ and thus the evaluation of the body of the function requires a new stack frame: the stack space required is not

$\mathsf{max}(l,m,n)$, but $\mathsf{max}(l,m,n+1)$.

$$\frac{\vdash_N N_1 \downarrow^l \lambda^t x.N \quad \vdash_N N_2 \downarrow^m W_2 \quad \vdash_T N[W_2/x] \downarrow^n W}{\vdash_T @^t N_1 N_2 \downarrow^{\mathsf{max}(l,m,n)} W}$$
$$\frac{\vdash_N N_1 \downarrow^l \lambda x.N \quad \vdash_N N_2 \downarrow^m W_2 \quad \vdash_T N[W_2/x] \downarrow^n W}{\vdash_T @ N_1 N_2 \downarrow^{\mathsf{max}(l,m,n+1)} W}$$

The other rules are shown in Appendix B.

The transformation of selective tail call elimination is defined as follows:

$$\begin{aligned}
[\![x]\!] &= x\\
[\![0]\!] &= 0\\
[\![\mathsf{suc}(V)]\!] &= \mathsf{suc}([\![V]\!])\\
[\![\mathsf{fix}\, x.\lambda^\omega y.M]\!] &= \mathsf{fix}\, x.\{\mathsf{tfun} = \lambda^t y.[\![M]\!]\}\\
[\![\mathsf{fix}\, x.\lambda^i y.M]\!] &= \mathsf{fix}\, x.\{\mathsf{fun} = \lambda y.[\![M]\!], \mathsf{tfun} = \lambda^t y.[\![M]\!]\}\\
[\![@^i M_1 M_2]\!] &= @([\![M_1]\!].\mathsf{fun})\, [\![M_2]\!]\\
[\![@^\omega M_1 M_2]\!] &= @^t([\![M_1]\!].\mathsf{tfun})\, [\![M_2]\!]\\
[\![\mathsf{case}\, V\, \mathsf{of}\, 0 \Rightarrow M_1,\, \mathsf{suc}(x) \Rightarrow M_2]\!] &= \mathsf{case}\, [\![V]\!]\, \mathsf{of}\, 0 \Rightarrow [\![M_1]\!],\, \mathsf{suc}(x) \Rightarrow [\![M_2]\!]
\end{aligned}$$

We extend the translation of types to type environments as $|E|(x) = |E(x)|$. Then the type correctness of this transformation is formulated as the following lemma and proved by induction on the derivation of $E \vdash M : \tau!\rho$.

Lemma 2 (Type Soundness). *If $E \vdash M : \tau!\rho$ then $|E| \vdash [\![M]\!] : |\tau|$.*

To formalize the soundness of the transformation we introduce the following notation: $\vdash_T M \downarrow^{\leq k} V$ if $\vdash_T M \downarrow^{k'} V$ for some $k' \leq k$. The factor of increase of stack space usage by selective tail call elimination is determined by the maximum of the effect annotations in M, denoted by $\mathsf{max}(M)$.

Theorem 1 (Soundness). *Let $C = \mathsf{max}(M) + 1$.*

1. *If $\emptyset \vdash M : \tau\,!\,i$ and $\vdash_T M \downarrow^k V$ then $\vdash_T [\![M]\!] \downarrow^{\leq Ck+i} [\![V]\!]$.*
2. *If $\emptyset \vdash M : \tau\,!\,\omega$ and $\vdash_T M \downarrow^k V$ then $\vdash_T [\![M]\!] \downarrow^{\leq Ck+C-1} [\![V]\!]$.*
3. *If $\emptyset \vdash M : \tau\,!\,\rho$ and $\vdash_N M \downarrow^k V$ then $\vdash_N [\![M]\!] \downarrow^{\leq Ck} [\![V]\!]$.*

This theorem ensures that the stack space usage of a program is preserved asymptotically.

For example, $@^\omega(\texttt{fix}\, f.\lambda^\omega x.@^1(\texttt{fix}\, g.\lambda^1 y.y)x)0$ and its translation are evaluated as follows:

$$\vdash_N @^\omega(\texttt{fix}\, f.\lambda^\omega x.@^1(\texttt{fix}\, g.\lambda^1 y.y)x)0 \downarrow^1 0$$

$$\vdash_N @^t(\texttt{fix}\, f.\{\mathsf{tfun} = \lambda^t x.@(\texttt{fix}\, g.\{\mathsf{fun} = \lambda y.y, \mathsf{tfun} = \lambda^t y.y\}.\mathsf{fun})x\}.\mathsf{tfun})0 \downarrow^2 0$$

This example corresponds to the worst case: $k = 1$ and $C = 2$. The proof of the theorem appears in Appendix C.

4 Extension of the effect system

Th effect system we have presented has one unnatural limitation: ω must always be assigned to a function which calls a function with effect ω at tail call position, even if the function is safe without tail call elimination. In this section, we extend our effect system to overcome this limitation by considering a wider class of effects.

We first show an example where the limitation of our effect system appears. Let us consider the following program.

```
fun f x = f x
fun g x = f x
fun h (0,x) = g x
  | h (n,x) = h (n-1,x)
```

The function `g` is safe without tail call elimination: the stack space usage is increased by 1 even if it is implemented with the standard calling convention. However, in our effect system the function is assigned the effect ω to because it calls the function `f` of the effect ω at a tail call position.

We solve this limitation by extending the effects in our type system into the following form.

$$\rho ::= \omega \cdot i + j$$

where i and j are natural numbers. The intuition is that the function with effect $\omega \cdot i + j$ such that $j > 0$ is safe without tail call elimination. We identifies $\omega \cdot i + 0$ and $\omega \cdot 0 + j$ with $\omega \cdot i$ and j, respectively. The effect ρ^+ and the subeffect relation $\rho \leq \rho'$ are defined as follows:

$$(\omega \cdot i + j)^+ = \omega \cdot i + (j+1)$$

$$\omega \cdot i + j \leq \omega \cdot i' + j' \quad \text{iff} \quad i < i', \text{ or } i = i' \text{ and } j \leq j'$$

The typing rules of abstraction in the effect system are extended as follows:

$$\frac{E, x : \tau_1 \rightarrow^{\rho^+} \tau_2, y : \tau_1 \vdash M : \tau_2 \,!\, \rho}{E \vdash \mathsf{fix}\, x.\lambda^{\rho^+} y.M : \tau_1 \rightarrow^{\rho^+} \tau_2} \qquad \frac{E, x : \tau_1 \rightarrow^{\omega \cdot i} \tau_2, y : \tau_1 \vdash M : \tau_2 \,!\, \omega \cdot i}{E \vdash \mathsf{fix}\, x.\lambda^{\omega \cdot i} y.M : \tau_1 \rightarrow^{\omega \cdot i} \tau_2}$$

Then we can assign the following types and thus `g` can be safely implemented with the standard calling convention.

```
f : int →^ω int
g : int →^(ω+1) int
h : int × int →^(ω·2) int
```

We also need to modify the transformation to implement selective tail call elimination. Since a function with effect $\omega \cdot i$ can be considered to have effect

$\omega \cdot i + 1$ in this system, a function with effect $\omega \cdot i$ must support both calling conventions. The transformation is modified as follows:

$$\begin{aligned}
[\![\mathsf{fix}\, x.\lambda^{\rho} y.M]\!] &= \mathsf{fix}\, x.\{\mathsf{fun} = \lambda y.[\![M]\!], \mathsf{tfun} = \lambda^{t} y.[\![M]\!]\} \\
[\![@^{\omega \cdot i + j} M_1 M_2]\!] &= @([\![M_1]\!].\mathsf{fun})\, [\![M_2]\!] \\
[\![@^{\omega \cdot i} M_1 M_2]\!] &= @^{t}([\![M_1]\!].\mathsf{tfun})\, [\![M_2]\!]
\end{aligned}$$

where $j > 0$. The soundness theorem is extended in the following form. We write $\mathsf{max}^{i}(M)$ for the maximum j of $\omega \cdot i + j$ appearing in M. The proof of the theorem is similar to that of Theorem 1.

Theorem 2 (Soundness). *Let* $C = \Sigma_{i=0}^{\infty} \mathsf{max}^{i}(M) + 1$ *and* $D(j) = \Sigma_{i=0}^{j-1} \mathsf{max}^{i}(M)$.

1. *If* $\emptyset \vdash M : \tau \,!\, (\omega \cdot i + j)$ *and* $\vdash_T M \downarrow^{k} V$ *then* $\vdash_T [\![M]\!] \downarrow^{\leq Ck + D(i) + j} [\![V]\!]$.
2. *If* $\emptyset \vdash M : \tau \,!\, \rho$ *and* $\vdash_N M \downarrow^{k} V$ *then* $\vdash_N [\![M]\!] \downarrow^{\leq Ck} [\![V]\!]$.

5 Effect inference

We show how to infer effects in this section. Th effect inference can be formalized as a type system with constraints, where a constraint generated by the type system is solved with a simple graph-based algorithm. We assume that types are already inferred with the standard type inference and consider the following explicitly-typed language for effect inference.

$$\begin{aligned}
\tau &::= \mathsf{nat} \mid \tau \rightarrow^{\alpha} \tau \\
V &::= 0 \mid \mathsf{suc}(V) \mid x \mid \mathtt{fix}\, x : \tau.\lambda y.M \\
M &::= V \mid @^{\alpha} M M \mid \mathsf{case}\, V\, \mathsf{of}\, 0 \Rightarrow M, \mathsf{suc}(x) \Rightarrow M
\end{aligned}$$

where α denotes an effect variable. An application is annotated with an effect variable. The effect annotation of a lambda abstraction can be determined from the type annotation of the `fix`-expression. We assume effect variables appearing in a program are distinct.

Judgments of the effect system have the following forms: $E; C \vdash V : \tau$ and $E; C \vdash M : \tau!\alpha$ where C is a set of subeffect relations: $\alpha < \alpha'$ and $\alpha \leq \alpha'$. Intuitively, a constraint $\alpha < \alpha'$ holds if $\alpha \leq \alpha'$ or $\alpha = \alpha' = \omega \cdot i$ for some i. The main rules of the effect system are given as follows:

$$\frac{E; C \vdash V : \tau \quad \alpha \text{ is fresh}}{E; C \vdash V : \tau!\alpha} \qquad \frac{E, x : \tau_1 \rightarrow^{\alpha} \tau_2, y : \tau_1; C \vdash M : \tau_2 \,!\, \alpha'}{E; C \cup \{\alpha' < \alpha\} \vdash \mathtt{fix}\, x : \tau_1 \rightarrow^{\alpha} \tau_2.\lambda y.M : \tau_1 \rightarrow^{\alpha} \tau_2}$$

$$\frac{E; C_1 \vdash M_1 : \tau_1 \rightarrow^{\alpha'} \tau_2 \,!\, \alpha_1 \quad E; C_2 \vdash M_2 : \tau_1' \,!\, \alpha_2}{E; C_1 \cup C_2 \cup \{\alpha' \leq \alpha\} \cup C_{\leq}(\tau_1', \tau_1) \vdash @^{\alpha} M_1 M_2 : \tau_2 \,!\, \alpha}$$

where $C_{\leq}(\tau_1', \tau_1)$ is the constraint to obtain the subtyping $\tau_1' \leq \tau_1$.

$$\begin{aligned}
C_{\leq}(\mathsf{nat}, \mathsf{nat}) &= \emptyset \\
C_{\leq}(\tau_1 \rightarrow^{\alpha} \tau_2, \tau_1' \rightarrow^{\alpha'} \tau_2') &= C_{\leq}(\tau_1', \tau_1) \cup C_{\leq}(\tau_2, \tau_2') \cup \{\alpha \leq \alpha'\}
\end{aligned}$$

The constraint obtained by the rules above can be solved in the following manner. We consider the graph of the relations $\alpha < \alpha'$ and $\alpha \leq \alpha'$, and compute the strongly connected components of the graph. If a strongly connected component contains a relation of the form $\alpha < \alpha'$, the effect of the form $\omega \cdot i$ must be assigned to the effect variables appearing in the component. It is clear that an effect $\omega \cdot i + j\,(j > 0)$ can be assigned to an effect variable not belonging to such components.

6 Examples

There are several situations where an effect of the form $\omega \cdot i$ is assigned to a function. Although some of them are actually unsafe without tail call elimination, our effect system sometimes assigns $\omega \cdot i$ to functions safe without tail call elimination. In this section we show some examples of both the situations.

In our effect system, $\omega \cdot i$ must be assigned to tail recursive functions in general. However, tail recursive calls in a single recursive function can be implemented as a loop and thus trampolining can be avoided for such tail calls. Our effect system can be extended to be consistent with this implementation and then such functions are not assigned $\omega \cdot i$ to. Then there are two common examples where recursive functions are unsafe without tail call elimination: mutually tail recursive functions and higher order recursive functions.

In the following example, the functions `f` and `g` contains mutually recursive tail calls and thus must be assigned ω to.

```
fun f 0 = 0
  | f n = g (n-1)
and g n = f (n-1)
```

However, it is possible to implement the tail calls as a loop if only one of `f` and `g` are used from the other part of a program or the functions are copied into two definitions.

The following is an example with a higher order function.

```
fun h 0 y = y
  | h x y = h (x-1) (x+y)
```

The function `h` has type `int` $\rightarrow$ `int` $\rightarrow$ `int`. The tail recursive call of `h (x-1) (x+y)` cannot be implemented as a loop. However, if the function is uncurried, the tail call in the function can be implemented as a loop and thus the function can be safely implemented without tail call elimination.

As the third example, we show a program that is safe without tail call elimination, but is assigned ω to with our effect system.

```
let fun f (g:int -> int) = g 0
in
    f (fn x => f (fn y => y))
end
```

You can check this program is safe without tail call elimination. Let us infer the type of `f`. By assuming that `f` has type (`int` $\to^{\alpha_1}$ `int`) $\to^{\alpha_2}$ `int`, the constraints $\alpha_1 < \alpha_2$ and $\alpha_2 < \alpha_1$ must be satisfied where the first constraint is obtained from the definition of `f` and the second constraint is obtained from the body of the `let`-expression. Then the effects α_1 and α_2 must be $\omega \cdot i$ for some i. This weakness of our effect system appears in the benchmark "logic" we will discuss in the next section: many functions similar to the function above appear in the program.

7 Implementation and measurements

We have incorporated our selective tail call elimination into the MLj compiler [2], which translates Standard ML into Java bytecodes. The MLj compiler uses an intermediate language based on monads to represent effects such as IO operations, exception, and non-termination. We extended the effects of the intermediate language with our effect. The effects for selective tail call elimination are inferred in a late stage of compilation and the transformation is merged into the code generation phase of the compiler.

It should be noted that a recursive tail call in a single recursive function can be implemented without trampolining by translating it into a loop in JVM. MLj implements a recursive tail call in this way and our selective tail call elimination is extended to be consistent with this.

The translation of a function presented in Section 3.2 and 4 has one problem: the body of $\lambda^i x.M$ is copied into two functions and thus the translation may increase the code size. This problem can be solved by the following translation.

$$\lambda^i y.M = \texttt{let}\ \ y = \lambda x.[\![M]\!]\ \texttt{in}\ \ \{\mathsf{fun} = y, \mathsf{tfun} = \lambda^t x.@yx\}$$

However, we do not adopt this translation because it makes it difficult to compare stack space usage. The worst case increase of code size observed for the benchmark programs we will discuss later is about 35 % [3].

We measured the effectiveness of our selective tail call elimination for the most benchmark programs obtained from the following URL [4].

`ftp://ftp.research.bell-labs.com/dist/smlnj/benchmarks/`

Table 1 summarizes the results of our effect analysis. The column total shows the number of the functions generated for each benchmark program. The columns (A), (B) and (C) are the numbers of functions the analysis assigns an effect of the form $\omega \cdot i$: (A), (B) and (C) are the numbers for selective tail call elimination without extension, with extension and with extension and an extra phase of optimization, respectively. The column (D) shows $\Sigma_{i=0}^{\infty}\mathsf{max}(P)$ for each program P, which determines the theoretical upper bound of increase of stack space usage.

[3] The pair representation is not used for the known function that are safe without tail call elimination.

[4] We excluded two programs count-graphs and mlyacc that are difficult to compile with MLj because of the limitation of MLj.

	total	(A)	(B)	(C)	(D)		total	(A)	(B)	(C)	(D)
barnes-hut	25	0	0	0	2	mandelbrot	6	0	0	0	1
boyer	24	0	0	0	2	nucleic	38	0	0	0	3
fft	28	0	0	0	3	ratio-regions	51	0	0	0	2
knuth-bendix	83	5	5	5	3	ray	82	2	2	0	3
lexgen	110	13	11	2	3	simple	175	0	0	0	4
life	30	1	1	0	2	tsp	20	0	0	0	2
logic	58	51	36	36	2	vliw	463	19	18	16	4

Table 1. Results of effect inference

	MLj	TCE	STCE
knuth-bendix	3895	3373	3485
lexgen	1259	94	94
life	298	49	49
logic	3814	236	260

Table 2. Maximum stack size: (in number of frames)

- Eight programs out of 13 are shown safe without tail call elimination. Even for the other programs except for the program logic, the ratio of the function of effect $\omega \cdot i$ is small.
- The most functions of the program logic have effect $\omega \cdot i$ by the reason we described in Section 6. The extension reduces the number, but more than half of the functions still have $\omega \cdot i$.
- Since the maximum of the numbers in column (D) is 4, the theoretical upper bound of stack space increase for selective tail call elimination compared to tail call elimination is 5.
- The effectiveness of our selective tail call elimination depends on other phases of compilation. An extra phase of optimization including uncurrying decreased the number of functions of effect $\omega \cdot i$ for three programs.

Table 2 shows the maximum stack size during execution measured by the number of frames. The table shows the results for the benchmark programs where tail call elimination has some impact on the results. For all the other program, the numbers are between 33 and 103. The results supports that tail call elimination is desirable: stack sizes are greatly reduced for several programs. Selective tail call elimination may increase stack size compared to tail call elimination. However, the increase is relatively small, compared to the theoretical upper bound.

Table 3 shows execution times. The columns TCE and STCE are the results for tail call elimination and selective tail call elimination, respectively. The numbers in the parenthesis are the ratios to those of MLj. Even for TCE, all the

	Interpreted-mode			HotSpot Client VM		
	MLj	TCE	STCE	MLj	TCE	STCE
barnes-hut	5.77	6.30(109.2)	5.83(101.0)	1.22	1.21(99.2)	1.23(100.8)
boyer	1.41	1.65(117.0)	1.41(100.0)	0.81	0.58(71.6)	0.82(101.2)
fft	1.71	2.12(124.0)	1.71(100.0)	0.57	0.60(105.3)	0.64(112.3)
knuth-bendix	11.09	10.80(97.4)	8.19(73.9)	2.00	2.12(106.0)	1.61(80.5)
lexgen	3.50	3.40(97.1)	3.49(99.7)	0.61	0.75(123.0)	0.63(103.3)
life	1.24	1.24(100.0)	1.15(92.7)	0.36	0.37(102.8)	0.35(97.2)
logic	17.40	18.90(108.6)	16.49(94.8)	4.51	2.79(61.9)	2.70(59.9)
mandelbrot	4.18	6.45(154.3)	4.22(101.0)	0.52	1.49(286.5)	0.55(105.8)
nucleic	0.68	0.73(107.4)	0.67(98.5)	0.34	0.44(129.4)	0.34(100.0)
ratio-regions	212.93	216.38(101.6)	209.11(98.2)	33.62	42.34(125.9)	33.67(100.1)
ray	5.77	6.20(107.5)	5.68(98.4)	1.88	1.13(60.1)	2.12(112.8)
simple	6.61	7.35(111.2)	6.38(96.5)	1.49	1.63(109.4)	1.54(103.4)
tsp	4.78	5.13(107.3)	4.75(99.4)	0.88	0.96(109.1)	0.84(95.5)
vliw	6.26	7.32(116.9)	6.42(102.6)	1.34	2.37(176.9)	1.46(109.0)

Table 3. Execution time (in seconds)

non-tail-calls are implemented with the standard calling convention based on the pair representation of functions. Measurements were done using Sun JDK 1.4.0, Java HotSpot Client VM on a Linux PC. We measured execution time on the interpreted-mode with the `-Xint` option, and on the mode where HotSpot compilation is enabled because it is sometimes difficult to interpret results on the HotSpot VM. Each benchmark was run five times and we chose the fastest run.

- TCE sometimes degrades the performance a lot. The worst case overhead is 54.3 % and 186.5 % for the interpreted-mode and the HotSpot VM, respectively. Compared to TCE, STCE causes little overhead: the worst case overhead is 2.8 % and 12.8 %, respectively.
- For benchmark programs where stack size is reduced by tail call elimination, execution times are sometimes reduced for both TCE and STCE: knuth-bendix and logic. This can be explained as a reduction of garbage collection (GC) time. For example, the GC times for logic are 2.43, 0.49 and 0.49 for MLj, TCE and STCE, respectively. The same phenomenon was observed by Schinz and Odersky in their tail call elimination for JVM [15].
- There are uncommon results on the programs boyer and ray: the big improvement of execution time over MLj and STCE is observed for TCE. We checked the profiling data of executions and found that better decisions on compilation are made by the HotSpot VM for TCE and the programs compiled by MLj and STCE spent more time on interpreted methods.

8 Related work

Dornic, Jouvelot and Gifford proposed an effect system to estimate execution time of a program [6], and their work was extended by Reistad and Gifford [14] with sized types [9]. By adapting the effect system extended with sized types we may obtain more information about the stack usage of a program. However, our simple effect system gives enough information for selective tail call elimination.

Implementation of proper tail calls and space safety are discussed by Clinger [5]. He considered that an implementation is safe with respect to space if it does not increase the asymptotic space complexity of programs. Our selective tail call elimination satisfies the criterion on stack space, but the factor of increase of stack space depends on the program and is determined by the effect system.

Schinz and Odersky proposed tail call elimination for the Java virtual machine [15] that preserves complexity on stack space. Their method is dynamic: it keeps track of the number of successive tail calls and execution is returned to a trampoline if some predefined limit is exceeded. We think that it is possible to reduce the overhead of their method with selective elimination of tail calls.

We have translated a function in the source language into a record with two functions supporting different calling conventions. Similar translation was used to support multiple calling conventions in type-directed unboxing by Minamide and Garrigue [13], and the vectorized functions of Chakravarty and Keller [4].

9 Conclusion and future work

We have presented an effect system and a program transformation to eliminate tail calls selectively. The transformation translates a function into a record with two functions supporting different calling conventions. The transformation preserves stack space asymptotically.

Our effect system will be useful even for target environments that directly supports tail calls. Various program transformations sometimes translate tail calls into non-tail calls. With our effect system, it is possible to check if such translation is safe for each tail call.

We incorporated our effect system into the MLj compiler and measured the proportion of functions that are unsafe without tail call elimination. The results indicated that selective tail call elimination is very effective for most programs and most functions can be implemented with the standard calling convention. However, there is a limitation that our effect system is monovariant. This limitation may be solved if we extend our effect system with effect polymorphism or intersection types.

By selective tail call elimination, the asymptotic complexity of stack space is preserved. The factor of the increase is determined by effect analysis and depends on a program. However, it is also possible to guarantee the factor of stack space increase by translating applications with annotations greater than the predefined factor as applications with annotation ω.

References

1. H. Baker. Cons should not cons its arguments, part II: Cheney on the M.T.A. *SIGPLAN Notices*, 30(9):17–20, 1995.
2. N. Benton, A. Kennedy, and G. Russell. Compiling Standard ML to Java bytecodes. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, pages 129–140, 1998.
3. P. Bothner. Kawa - compiling dynamic languages to the Java VM. In *Proceedings of the 1998 Usenix conference*, 1998.
4. M. M. T. Chakravarty and G. Keller. More types for nested data parallel programming. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, pages 94–105, 1999.
5. W. D. Clinger. Proper tail recursion and space efficiency. In *Proceedings of the ACM SIGPLAN '98 conference on Programming language design and implementation*, pages 174–185. ACM Press, 1998.
6. V. Dornic, P. Jouvelot, and D. K. Gifford. Polymorphic time systems for estimating program complexity. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(1):33–45, 1992.
7. S. D. Ganz, D. P. Friedman, and M. Wand. Trampolined style. In *Proceedings of the 4th ACM SIGPLAN International Conference on Functional Programming (ICFP '99)*, pages 18–22, 1999.
8. D. K. Gifford and J. M. Lucassen. Integrating functional and imperative programming. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 28–38, 1986.
9. J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 410–423, 1996.
10. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1999.
11. J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the Fifteenth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 47–57, 1988.
12. 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.
13. 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.
14. B. Reistad and D. K. Gifford. Static dependent costs for estimating execution time. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 65–78, 1994.
15. M. Schinz and M. Odersky. Tail call elimination on the Java virtual machine. In *Proceedings of the First International Workshop on Multi-Language Infrastructure and Interoperability (BABEL)*, volume 59(1) of *ENTCS*, 2001.
16. B. Serpette and M. Serrano. Compiling Scheme to JVM bytecode: a performance study. In *Proceedings of the seventh ACM SIGPLAN International Conference on Functional Programming*, pages 259–270, 2002.
17. D. Tarditi, A. Acharya, and P. Lee. No assembly required: Compiling standard ML to C. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(2):161–177, 1992.

A The type system of the target language

The type system of the target language is defined as a deductive system with judgments of the form $E \vdash N : \sigma$. We do not show the rules for 0, $\mathsf{suc}(W)$, and $\mathtt{case}$-expressions that are the same as those for the source language.

$$\frac{x : \sigma \in E}{E \vdash x : \sigma} \qquad \frac{E \vdash N : \sigma' \quad \sigma' \leq \sigma}{E \vdash N : \sigma}$$

$$\frac{E, x : \sigma_1 \vdash N : \sigma_2}{E \vdash \lambda x.N : \sigma_1 \to \sigma_2} \qquad \frac{E, x : \sigma_1 \vdash N : \sigma_2}{E \vdash \lambda^t x.N : \sigma_1 \to^t \sigma_2}$$

$$\frac{E \vdash N_1 : \sigma_1 \to \sigma_2 \quad E \vdash N_2 : \sigma_1}{E \vdash @N_1 N_2 : \sigma_2} \qquad \frac{E \vdash N_1 : \sigma_1 \to^t \sigma_2 \quad E \vdash N_2 : \sigma_1}{E \vdash @^t N_1 N_2 : \sigma_2}$$

$$\frac{E, x : \{\mathsf{fun} : \sigma_1, \mathsf{tfun} : \sigma_2\} \vdash W_1 : \sigma_1 \quad E, x : \{\mathsf{fun} : \sigma_1, \mathsf{tfun} : \sigma_2\} \vdash W_2 : \sigma_2}{E \vdash \mathsf{fix}\, x.\{\mathsf{fun} = W_1, \mathsf{tfun} = W_2\} : \{\mathsf{fun} : \sigma_1, \mathsf{tfun} : \sigma_2\}}$$

$$\frac{E, x : \{\mathsf{tfun} : \sigma\} \vdash W : \sigma}{E \vdash \mathsf{fix}\, x.\{\mathsf{tfun} = W\} : \{\mathsf{tfun} : \sigma\}}$$

$$\frac{E \vdash N : \{\mathsf{fun} : \sigma_1, \mathsf{tfun} : \sigma_2\}}{E \vdash N.\mathsf{fun} : \sigma_1} \quad \frac{E \vdash N : \{\mathsf{tfun} : \sigma\}}{E \vdash N.\mathsf{tfun} : \sigma}$$

B The operational semantics of the target language

The following are the rules of the operational semantics of the target language. We write $\vdash_\alpha N \downarrow^n W$ if the rule holds for both $\vdash_N N \downarrow^n W$ and $\vdash_T N \downarrow^n W$.

$$\frac{\vdash_N N_1 \downarrow^l \lambda x.N \quad \vdash_N N_2 \downarrow^m W_2 \quad \vdash_\alpha N[W_2/x] \downarrow^n W}{\vdash_\alpha @N_1 N_2 \downarrow^{\mathsf{max}(l,m,n+1)} W}$$

$$\vdash_\alpha W \downarrow^0 W \quad \frac{\vdash_N N \downarrow^n \mathsf{fix}\, x.\{\mathsf{tfun} = W\}}{\vdash_\alpha N.\mathsf{tfun} \downarrow^n W[\mathsf{fix}\, x.\{\mathsf{tfun} = W\}/x]}$$

$$\frac{\vdash_N N \downarrow^n \mathsf{fix}\, x.\{\mathsf{fun} = W_1, \mathsf{tfun} = W_2\}}{\vdash_\alpha N.\mathsf{tfun} \downarrow^n W_2[\mathsf{fix}\, x.\{\mathsf{fun} = W_1, \mathsf{tfun} = W_2\}/x]}$$

$$\frac{\vdash_N N \downarrow^n \mathsf{fix}\, x.\{\mathsf{fun} = W_1, \mathsf{tfun} = W_2\}}{\vdash_\alpha N.\mathsf{fun} \downarrow^n W_1[\mathsf{fix}\, x.\{\mathsf{fun} = W_1, \mathsf{tfun} = W_2\}/x]}$$

C Proof of Soundness

The following lemma is crucial to establish soundness of transformation.

Lemma 3. *$[\![M[V/x]]\!] \equiv [\![M]\!][[\![V]\!]/x]$.*

The following lemma is also used to prove the soundness. It is shown by case analysis.

Lemma 4. *If $\emptyset \vdash \texttt{fix}\, x.\lambda^{\rho'}.M : \tau_1 \to^{\rho} \tau_2$, then $x : \tau_1 \to^{\rho'} \tau_2, y : \tau_1 \vdash M : \tau_2 ! \rho''$ such that $\rho'' \leq \rho$.*

We prove the main theorem in the following form to simplify case-analysis.

Lemma 5. *Let $C = \mathsf{max}(M) + 1$.*

1. *If $\emptyset \vdash M : \tau\,!\,\rho$ and $\vdash_T M \downarrow^k V$ then $\vdash_T [\![M]\!] \downarrow^{\leq Ck + D(\rho)} [\![V]\!]$.*
2. *If $\emptyset \vdash M : \tau\,!\,\rho$ and $\vdash_N M \downarrow^k V$ then $\vdash_N [\![M]\!] \downarrow^{\leq Ck} [\![V]\!]$.*

where $D(\rho)$ is a function such that $D(i) = i$ and $D(\omega) = \mathsf{max}(M)$.

Proof. By mutual induction on the derivations of $\vdash_T M \downarrow^k V$ and $\vdash_N M \downarrow^k V$. We show the only cases for application.

Proof of Property 1.

Case: $\vdash_T @^i M_1 M_2 \downarrow^k V$ is derived from $\vdash_N M_1 \downarrow^l V_1$ and $\vdash_N M_2 \downarrow^m V_2$ and $\vdash_T M[V_1/x][V_2/y] \downarrow^n V$ where $V_1 \equiv \texttt{fix}\, x.\lambda^{j^+} y.M$ and $k = \mathsf{max}(l, m, n)$. From the definition of the type system, $j < i$. From $\emptyset \vdash @^i M_1 M_2 : \tau!\rho$, $i \leq \rho$. We also have $x : \tau' \to^{j^+} \tau, y : \tau' \vdash M : \tau!j$.
By the induction hypothesis, $\vdash_N [\![M_1]\!] \downarrow^{\leq Cl} [\![V_1]\!]$ and $\vdash_N [\![M_2]\!] \downarrow^{\leq Cm} [\![V_2]\!]$. From $\emptyset \vdash M[V_1/x][V_2/y] : \tau!j$, by the induction hypothesis,

$$\vdash_T [\![M]\!][[\![V_1]\!]/x][[\![V_2]\!]/y] \downarrow^{\leq Cn+j} [\![V]\!]$$

This case is proved by the following derivation.

$$\frac{\vdash_N [\![M_1]\!].\mathsf{fun} \downarrow^{\leq Cl} \lambda y.[\![M]\!][[\![V_1]\!]/x] \quad \vdash_N [\![M_2]\!] \downarrow^{\leq Cm} [\![V_2]\!] \quad \vdash_T [\![M]\!][[\![V_1]\!]/x][[\![V_2]\!]/y] \downarrow^{\leq Cn+j} [\![V]\!]}{\vdash_T @([\![M_1]\!].\mathsf{fun})[\![M_2]\!] \downarrow^{\leq \mathsf{max}(Cl, Cm, Cn+j+1)} [\![V]\!]}$$

where $\mathsf{max}(Cl, Cm, Cn + j + 1) \leq Ck + D(\rho)$.

Case: $\vdash_T @^\omega M_1 M_2 \downarrow^k V$ is derived from $\vdash_N M_1 \downarrow^l V_1$ and $\vdash_N M_2 \downarrow^m V_2$ and $\vdash_T M[V_1/x][V_2/y] \downarrow^n V$ where $V_1 \equiv \texttt{fix}\, x.\lambda^{\rho'} y.M$ and $k = \mathsf{max}(l, m, n)$. From $\emptyset \vdash @^\omega M_1 M_2 : \tau!\rho$, ρ must be ω. By Lemma 4, $x : \tau' \to^{\rho'} \tau, y : \tau' \vdash M : \tau!\rho''$ where $\rho'' \leq \rho'$. By the induction hypothesis, $\vdash_N [\![M_1]\!] \downarrow^{\leq Cl} [\![V_1]\!]$ and $\vdash_N [\![M_2]\!] \downarrow^{\leq Cm} [\![V_2]\!]$. From $\emptyset \vdash M[V_1/x][V_2/y] : \tau!\rho''$, by the induction hypothesis

$$\vdash_T [\![M[V_1/x][V_2/y]]\!] \downarrow^{\leq Cn + D(\rho'')} [\![V]\!]$$

This case is proved by the following derivation.

$$\frac{\vdash_N [\![M_1]\!].\mathsf{tfun} \downarrow^{\leq Cl} \lambda^t y.[\![M]\!][[\![V_1]\!] \quad \vdash_N [\![M_2]\!] \downarrow^{\leq Cm} [\![V_2]\!] \\ \vdash_T [\![M]\!][[\![V_1]\!]/x][[\![V_2]\!]/y] \downarrow^{\leq Cn+D(\rho'')} [\![V]\!]}{\vdash_T @^t([\![M_1]\!].\mathsf{tfun})[\![M_2]\!] \downarrow^{\leq \mathsf{max}(Cl,Cm,Cn+D(\rho''))} [\![V]\!]}$$

where $\mathsf{max}(Cl, Cm, Cn + D(\rho'')) \leq Ck + C - 1 = Ck + D(\omega)$.

Proof of Property 2.

Case: $\vdash_N @^\omega M_1 M_2 \downarrow^k V$ is derived from $\vdash_N M_1 \downarrow^l V_1$ and $\vdash_N M_2 \downarrow^m V_2$ and $\vdash_N M[V_1/x][V_2/y] \downarrow^n V$ where $V_1 \equiv \mathtt{fix}\, x.\lambda^{\rho'} y.M$ and $k = \mathsf{max}(l, m, n + 1)$. From $\emptyset \vdash @^\omega M_1 M_2 : \tau!\rho$, ρ must be ω. By Lemma 4, we have $x : \tau' \rightarrow^{\rho'} \tau, y : \tau' \vdash M : \tau!\rho''$ where $\rho'' \leq \rho'$.
By the induction hypothesis, $\vdash_N [\![M_1]\!] \downarrow^{\leq Cl} [\![V_1]\!]$ and $\vdash_N [\![M_2]\!] \downarrow^{\leq Cm} [\![V_2]\!]$. From $\emptyset \vdash M[V_1/x][V_2/y] : \tau!\rho''$, by the induction hypothesis,

$$\vdash_T [\![M]\!][[\![V_1]\!]/x][[\![V_2]\!]/y] \downarrow^{\leq Cn+D(\rho'')} [\![V]\!]$$

This case is proved by the following derivation.

$$\frac{\vdash_N [\![M_1]\!].\mathsf{tfun} \downarrow^{\leq Cl} \lambda^t y.[\![M]\!][|V_1|/x] \quad \vdash_N [\![M_2]\!] \downarrow^{\leq Cm} [\![V_2]\!] \\ \vdash_T [\![M]\!][[\![V_1]\!]/x][[\![V_2]\!]/y] \downarrow^{\leq Cn+D(\rho'')} [\![V]\!]}{\vdash_N @^t([\![M_1]\!].\mathsf{tfun})[\![M_2]\!] \downarrow^{\leq \mathsf{max}(Cl,Cm,Cn+D(\rho'')+1)} [\![V]\!]}$$

where $\mathsf{max}(Cl, Cm, Cn + D(\rho'') + 1) \leq Ck$.

Case: $\vdash_N @^i M_1 M_2 \downarrow^k V$ is derived from $\vdash_N M_1 \downarrow^l V_1$ and $\vdash_N M_2 \downarrow^m V_2$ and $\vdash_N M[V_1/x][V_2/y] \downarrow^n V$ where $V_1 \equiv \lambda^{j^+} x.M$ and $k = \mathsf{max}(l, m, n + 1)$. Similar to the previous case. $\square$