A Type Theory for Krivine-Style Evaluation and Compilation*

Kwanghoon Choi and Atsushi Ohori**

School of Information Science,
Japan Advanced Institute of Science and Technology,
Tatsunokuchi, Ishikawa, Japan
{khchoi,ohori}@jaist.ac.jp

Abstract. This paper develops a type theory for Krivine-style evaluation and compilation. We first define a static type system for lambda terms where lambda abstraction is interpreted as a code to pop the "spine stack" and to continue execution. Higher-order feature is obtained by introducing a typing rule to convert a code to a closure. This is in contrast with the conventional type theory for the lambda calculus, where lambda abstraction always creates higher-order function. We then define a type system for Krivine-style low-level machine, and develops type-directed compilation from the term calculus to the Krivine-style machine. We establish that the compilation preserves both static and dynamic semantics. This type theoretical framework provides a proper basis to analyze various properties of compilation. To demonstrate the strength of our framework, we perform the above development for two versions of low-level machines, one of which statically determines the spine stack, and the other of which dynamically determines the spine stack using a run-time mark, and analyze their relative merit.

1 Introduction

The Krivine abstract machine [3] can be regarded as a system to transform a state of the form $(\mathcal{E}, \mathcal{S}, M)$ consisting of an environment $\mathcal{E}$, a stack $\mathcal{S}$, and a term M to be evaluated. An environment $\mathcal{E}$ represents the binding of the free variables in M as usual. The distinguishing feature of the Krivine machine lies in its usage of a stack $\mathcal{S}$, which maintains the application context (or *spine*) of the term M being evaluated. For example, for the term $(\lambda x.M)\ N_1\ N_2 \ldots N_n$, the machine causes the following transition.

$$(\mathcal{E}, \mathcal{S}, (\lambda x.M)\ N_1\ N_2 \ldots N_n) \stackrel{*}{\Longrightarrow} (\mathcal{E}, V_1 \cdot V_2 \cdot \ldots \cdot V_n \cdot \mathcal{S}, \lambda x.M)$$

* This work has been partially supported by Grant-in-aid for scientific research on basic research (B), no :15300006.

** The second author has also been supported in part by Grant-in-aid for scientific research on priority area "informatics" A01-08, no: 16016240.

W.-N. Chin (Ed.): APLAS 2004, LNCS 3302, pp. 213–228, 2004.

where each V_i is the value denoted by N_i under $\mathcal{E}$ and $V_1 \cdot V_2 \cdot ... \cdot V_n \cdot \mathcal{S}$ is the stack obtained by pushing $V_n, \ldots, V_1$ in this order. As seen from this example, the stack $\mathcal{S}$ in the state represents the arguments to the function denoted by the term $\lambda x.M$. The evaluation step for the lambda abstraction can then be performed simply by poping the stack and binding the variable to the poped value as seen below.

$$(\mathcal{E}, V_1 \cdot V_2 \cdot ... \cdot V_n \cdot \mathcal{S}, \lambda x.M) \Longrightarrow (\mathcal{E}\{x : V_1\}, V_2 \cdot ... \cdot V_n \cdot \mathcal{S}, M)$$

where $\mathcal{E}\{x : V_1\}$ is the environment obtained from $\mathcal{E}$ by extending it with $x : V_1$. When the stack is empty, the machine converts the current execution state to a closure and returns.

As observed by Leroy [6], this mechanism avoids unnecessary closure construction in evaluating nested application such as

$$(\lambda x_1. \cdots \lambda x_n.M)\ N_1 \cdots N_n$$

and yields potentially more efficient evaluation scheme for higher-order functional languages. The ZINC machine for a strict functional language exploits this mechanism. This mechanism is also closely related to an abstract machine with "spine" for a lazy functional language, where a spine denotes the evaluation context represented by a stack.

Despite the potential significance of Krivine-style evaluation strategy, however, there does not seem to exist type theoretical account for this evaluation mechanism. As a consequence, type information has not been well integrated in this evaluation mechanism. For example, if we statically know that a lambda term $\lambda x.\lambda y.M$ only flows into a context of two nested application of the form $[\_]\ M_1\ M_2$, then we expect that we should be able to compile the term into code that pops the spine stack without performing any runtime check of verifying that the current spine stack is not empty. We would like to establish a type theoretical framework to analyze the relationship between static structure of terms and Krivine-style operational semantics. Such a framework should be useful in verifying various desired properties such as type soundness and also in developing type-directed optimizing compiler for Krivine-style evaluation machine. This requires us to develop new type systems for both source lambda terms and for low-level code performing Krivine-style evaluation using a stack and environment.

As we have pointed out above, lambda abstraction in Krivine-style evaluation denotes an operation to pop the spine stack and to bind the lambda variable to the poped value. A closure is created at the time when no more evaluation is possible. This structure is not reflected in the conventional type theory of the lambda calculus, where lambda abstraction is interpreted as a constructor to introduce a function type $\tau \to \tau'$, whose denotation is a function closure. This suggests that there should be a new form of type discipline that provides static (abstract) interpretation of this dynamic behavior.

We first define a type system for lambda terms having the following features. First, the spine of a term to be evaluated is explicitly presented in typing judgment. Second, lambda abstraction is interpreted as a constructor to pop the

spine stack and bind the lambda variable. Third, closure creation is modeled by a separate structural rule (coercion).

For this typed calculus, we define a Krivine-style operational semantics and show that the type system is sound with respect to the operational semantics. The static type information made available by the type system leads us to develop a new form of Krivine-style abstract machine that does not require any explicit mark that indicates the end of the current spine stack at runtime, nor any runtime check of emptiness of the spine stack. We develop a type system for this new form of Krivine machine, and develop a type-directed compilation algorithm. We establish that the compilation algorithm preserves both static semantics (typing) and dynamic semantics by setting up a simulation relation between the evaluation relation for the lambda terms and the reduction system of our Krivine machine.

A feature that distinguishes our Krivine machine from existing Krivine-style machines is its ability to determine a spine stack statically. This is in contrast with ZINC machine [6], which determines the spine stack dynamically by explicitly inserting a mark indicating the end of each spine stack. To make precise comparison, we also develop a type system for ZINC machine, and type-directed compilation, and show that the compilation algorithm preserves both static semantics (typing) and dynamic semantics. This result also establishes that the two machines are operationally equivalent. Moreover, our type-directed compilation and semantic correctness for these two machines clarify the difference of the two machines and the role of dynamic check performed in ZINC machine.

The rest of the paper is organized as follows. Section 2 presents a Krivine-style type system for the lambda calculus, defines its operational semantics and establishes the type soundness. Section 3 defines a Krivine abstract machine that statically determines each spine stack as a typed system, and shows the type soundness. We then develop a type-directed compilation algorithm from the source calculus to this machine and establish that the compilation preserves both typing and behavior. Section 4 establishes the results for ZINC machine parallel to those of previous section, and compares the two machines. Section 5 discusses related works, and Section 6 concludes the paper with suggestions for further investigation.

Due to lack of space, we had to omit all proofs of theorems, and we could not cover interesting issues. The proofs can be found in our accompanying technical report [2]. We intend to present a more detailed account elsewhere in future.

2 A Typed Krivine-Style Calculus

We consider the following sets of lambda terms and types

$$\begin{aligned} M &::= c^b \mid x \mid \lambda x.M \mid M\ M \\ \tau &::= b \mid \Delta \to \tau \\ \Delta &::= [\tau_1, \ldots, \tau_n] \end{aligned}$$

$$\text{(var)}\quad \Gamma\{x:\tau\} \mid \emptyset \rhd x : \tau \qquad \text{(abs)}\quad \frac{\Gamma\{x:\tau_1\} \mid \Delta \rhd M : \tau_2}{\Gamma \mid \tau_1 \cdot \Delta \rhd \lambda x.M : \tau_2}$$

$$\text{(closure)}\quad \frac{\Gamma \mid \Delta \rhd \lambda x.M : \tau}{\Gamma \mid \emptyset \rhd \lambda x.M : \Delta \to \tau} \qquad \text{(install)}\quad \frac{\Gamma \mid \emptyset \rhd M : \Delta \to \tau}{\Gamma \mid \Delta \rhd M : \tau}$$

$$\text{(app)}\quad \frac{\Gamma \mid \emptyset \rhd M_2 : \tau_2 \quad \Gamma \mid \tau_2 \cdot \Delta \rhd M_1 : \tau_1}{\Gamma \mid \Delta \rhd M_1\ M_2 : \tau_1}$$

Fig. 1. The type system for Krivine-style term calculus

Δ is a list of types representing the types of the spine in which the term occurs. $\Delta \to \tau$ is a type of a function that consumes Δ and produces a value of type τ, and corresponds to n-ary function type in the conventional type system.

We use the following notations for lists. If X is a list then $v \cdot X$ is a list obtained by prepending v at the top of the list. The empty list is denoted by $\emptyset$. We identify a singleton list $v \cdot \emptyset$ with v and simply write v.

A term is typed relative to a *typing environment* Γ, which is a function from a finite set of variables to types, and a *stack type* Δ. We write $\Gamma\{x:\tau\}$ for the function Γ' s.t. $dom(\Gamma') = dom(\Gamma) \cup \{x\}$, $\Gamma'(x) = \tau$, and $\Gamma'(y) = \Gamma(y)$ if $y \neq x$.

A typing judgment is of the form

$$\Gamma \mid \Delta \rhd M : \tau$$

The set of typing rules is given in Figure 1. The intuitive meaning can be understood when one reads each rule backward. The rule (abs) indicates that $\lambda x.M$ pops the spine stack $\tau \cdot \Delta$ and binds x to the popped value. The rule (app) states that evaluation of $M_1 M_2$ proceeds by first evaluating M_2 in the empty spine stack, pushing the result on the spine stack and reducing M_1 under the stack. The rule (closure) converts the currently executing code into a closure. The rule (install) is its converse.

To see how this type system derives a typing, let us consider the following lambda term.

$$(\lambda f.(\lambda x.\lambda y.M)\ (f\ 1\ 2)\ (f\ 3))\ (\lambda w.\lambda z.N)$$

Figure 2 shows a typing derivation for this term. In this example, nested lambda abstraction and the top level nested applications in $(\lambda x.\lambda y.M)\ (f\ 1\ 2)\ (f\ 3)$ are typed without invoking the structural rules, so no closure is created for reducing this spine. On the other hand, the derivation of $(f\ 1\ 2)$ and $(\lambda w.\lambda z.N)$ involves structural rules. The extra structural rules are needed due to the implicit constraint of f occurring both in the contexts $([\,]\ 1\ 2)$ and $([\,]\ 3)$. Later we shall see that these structural rules are compiled to closure creation and closure installation.

It is easily shown that the typability of this system is the same as that of the simply typed lambda calculus.

Fact 1. *If $\Gamma \rhd M : \tau$ is derivable in the simple type system then $\Gamma \mid \emptyset \rhd M : \tau$ is derivable in our type system. Conversely, if $\Gamma \mid \Delta \rhd M : \tau$ is derivable in our type system, then $\Gamma \rhd M : \overline{\Delta \to \tau}$ is derivable in the simple type system, where $\overline{\Delta \to \tau}$ is the completely curried type.*

$$\mathcal{P} = \dfrac{\dfrac{\dfrac{\cdots}{\Gamma_f\{x : int, y : \{int\} \to int\} \mid \emptyset \rhd M : int}}{\Gamma_f\{x : int\} \mid \{int\} \to int \rhd \lambda y.M : int}}{\Gamma_f \mid int \cdot \{int\} \to int \rhd \lambda x.\lambda y.M : int} \qquad \mathcal{F} = \dfrac{\dfrac{\dfrac{\dfrac{\{w : int, z : int\} \mid \emptyset \rhd N : int}{\{w : int\} \mid \{int\} \rhd \lambda z.N : int}}{\{w : int\} \mid \emptyset \rhd \lambda z.N : \{int\} \to int}}{\emptyset \mid \{int\} \rhd \lambda w.\lambda z.N : \{int\} \to int}}{\emptyset \mid \emptyset \rhd \lambda w.\lambda z.N : \{int\} \to \{int\} \to int}$$

$$\dfrac{\dfrac{\dfrac{\mathcal{P} \quad \dfrac{\dfrac{\dfrac{\dfrac{\Gamma_f \mid \emptyset \rhd f : \{int\} \to \{int\} \to int}{\Gamma_f \mid int \rhd f : \{int\} \to int} \quad \Gamma_f \mid \emptyset \rhd 1 : int}{\Gamma_f \mid \emptyset \rhd f\ 1 : \{int\} \to int}}{\Gamma_f \mid int \rhd f\ 1 : int} \quad \Gamma_f \mid \emptyset \rhd 2 : int}{\Gamma_f \mid \emptyset \rhd f\ 1\ 2 : int}}{\Gamma_f \mid \{int\} \to int \rhd (\lambda x.\lambda y.M)(f\ 1\ 2) : int} \quad \dfrac{\dfrac{\Gamma_f \mid \emptyset \rhd f : \{int\} \to \{int\} \to int}{\Gamma_f \mid int \rhd f : \{int\} \to int} \quad \Gamma_f \mid \emptyset \rhd 3 : int}{\Gamma_f \mid \emptyset \rhd f\ 3 : \{int\} \to int}}{\dfrac{\Gamma_f \mid \emptyset \rhd (\lambda x.\lambda y.M)\ (f\ 1\ 2)\ (f\ 3) : int}{\emptyset \mid \{int\} \to \{int\} \to int \rhd \lambda f.(\lambda x.\lambda y.M)\ (f\ 1\ 2)\ (f\ 3) : int}} \quad \mathcal{F}}{\emptyset \mid \emptyset \rhd (\lambda f.(\lambda x.\lambda y.M)\ (f\ 1\ 2)\ (f\ 3))\ (\lambda w.\lambda z.N) : int}$$

where $\Gamma_f = \{f : \{int\} \to \{int\} \to int\}$

Fig. 2. An example typing derivation in the typed Krivine calculus

We first present a simple call-by-value operational semantics that reflects the feature of Krivine abstract machine, as in Figure 3. The semantics has a set of rules to determine an evaluation relation of the form

$$\mathcal{E}, \mathcal{S} \vdash M \Downarrow V$$

indicating the fact that M evaluates to a value V under an environment $\mathcal{E}$ and a Krivine stack $\mathcal{S}$. $\mathcal{E}$ is a mapping of a variable to a value, $\mathcal{S}$ is a sequence of values, and V is given by the following syntax.

$$V ::= c^b \mid cls(\mathcal{E}, M) \mid wrong$$

wrong represents runtime error.

The type system is sound with respect to the operational semantics. To show this, we first define typing relations on semantic objects

- $\models V : \tau$ (V has type τ)
  - $\models c^b : b$
  - $\models cls(\mathcal{E}, \lambda x.M) : \Delta \to \tau$ if there is some Γ s.t. $\models \mathcal{E} : \Gamma$, $\Gamma \mid \Delta \rhd \lambda x.M : \tau$
- $\models \mathcal{E} : \Gamma$ ($\mathcal{E}$ satisfies Γ)
  - if $dom(\mathcal{E}) = dom(\Gamma)$ and $\models \mathcal{E}(x) : \Gamma(x)$ for all $x \in dom(\mathcal{E})$.
- $\models \mathcal{S} : \Delta$ ($\mathcal{S}$ satisfies Δ)
  - if $|\mathcal{S}| = |\Delta|$ and $\models \mathcal{S}.i : \Delta.i$ for all $1 \leq i \leq |\mathcal{S}|$, where $\mathcal{S}.i$ and $\Delta.i$ denote the i-th elements of $\mathcal{S}$ and Δ respectively.

Fact 2. *If* $\Gamma \mid \Delta \rhd M : \Delta_1 \to \cdots \to \Delta_n \to \tau$, $\models \mathcal{E} : \Gamma$, $\models \mathcal{S}_1 : \Delta$, $S_2 = S_1^2 \cdot \ldots \cdot S_n^2$, $\models \mathcal{S}_i^2 : \Delta_i$, *and* $\mathcal{E}, \mathcal{S}_1 \cdot \mathcal{S}_2 \vdash M \Downarrow V$ *then* $\models V : \tau$.

We also define an alternative call-by-value operational semantics using extra continuation arguments roughly following the style of [4]. This refinement is needed to establish the semantic correctness of compilation in Section 3 and 4.

$$\frac{\mathcal{E}\{x:V\},\emptyset \vdash x \Downarrow V}{\mathcal{E},\emptyset \vdash \lambda x.M \Downarrow cls(\mathcal{E},\lambda x.M)} \qquad \frac{\mathcal{E}_0, V_0 \cdot \mathcal{S} \vdash \lambda y.M_0 \Downarrow V}{\mathcal{E}\{x:cls(\mathcal{E}_0,\lambda y.M_0)\}, V_0 \cdot \mathcal{S} \vdash x \Downarrow V} \qquad \frac{\mathcal{E}\{x:V_0\},\mathcal{S} \vdash M \Downarrow V}{\mathcal{E}, V_0 \cdot \mathcal{S} \vdash \lambda x.M \Downarrow V}$$

$$\frac{\mathcal{E},\emptyset \vdash M_2 \Downarrow V_2 \quad \mathcal{E}, V_2 \cdot \mathcal{S} \vdash M_1 \Downarrow V}{\mathcal{E},\mathcal{S} \vdash M_1\ M_2 \Downarrow V} \qquad \frac{\text{(if no other rule applies)}}{\mathcal{E},\mathcal{S} \vdash M \Downarrow wrong}$$

Fig. 3. A call-by-value semantics for the Krivine calculus

We let K range over *evaluation continuation* given by the following syntax.

$$K ::= \mathsf{retCont} \mid \mathsf{appCont}(\mathcal{E},\mathcal{S},M,K)$$

$\mathsf{retCont}$ is the empty context. $\mathsf{appCont}(\mathcal{E},\mathcal{S},M_1,K)$ represents the evaluation context of M_2 in M_1M_2 under $\mathcal{E},\mathcal{S},K$. The new evaluation relation with continuation is given in Figure 4. This is the same as the call-by-value operational semantics defined above, but it makes the evaluation order and the flow explicit.

With respect to this refined operational semantics, we establish the soundness of the type system. We introduce a typing relation on continuation of the form

$$\models K : \tau_1 \Rightarrow \tau_2 \quad \text{a continuation } K \text{ has type } \tau_1 \Rightarrow \tau_2$$

indicating that K accepts a value of type τ_1 and yields a value of type τ_2.

- $\models \mathsf{retCont} : \tau \Rightarrow \tau$ for any τ.
- $\models \mathsf{appCont}(\mathcal{E},\mathcal{S}_1 \cdot \mathcal{S}_2, M, K) : \tau_1 \Rightarrow \tau_2$ if there are some Γ,Δ and τ_3 such that $\models \mathcal{E} : \Gamma$, $\models \mathcal{S}_1 : \Delta$, $\mathcal{S}_2 = \mathcal{S}_1^2 \cdot ... \cdot \mathcal{S}_n^2$, $\models \mathcal{S}_i^2 : \Delta_i$, $\Gamma \mid \tau_1 \cdot \Delta \rhd M : \Delta_1 \rightarrow \cdots \rightarrow \Delta_n \rightarrow \tau_3$, and $\models K : \tau_3 \Rightarrow \tau_2$.

We now show the following.

Theorem 1. *If* $\Gamma \mid \Delta \rhd M : \Delta_1 \rightarrow \cdots \rightarrow \Delta_n \rightarrow \tau$, $\models \mathcal{E} : \Gamma$, $\models \mathcal{S}_1 : \Delta$, $\mathcal{S}_2 = \mathcal{S}_1^2 \cdot ... \cdot \mathcal{S}_n^2$, $\models \mathcal{S}_i^2 : \Delta_i$, $\models K : \tau \Rightarrow \tau_0$, *and* $\mathcal{E},\mathcal{S}_1 \cdot \mathcal{S}_2, K \vdash M \Downarrow V$ *then* $\models V : \tau_0$.

This result together with Fact 1 ensures that our type system can serve as an alternative type discipline to the conventional type system of the lambda calculus. In the next section, we shall develop a typed abstract machine and type directed compilation for this calculus.

3 A Krivine Machine and Compilation

This section defines a new Krivine abstract machine and its type system, develops a type-directed compilation, and establishes the correctness of compilation. Unlike existing variants of Krivine machines, there is no runtime "mark" that delimits the end of each spine, and sequence of spines of pending computation are placed in a single spine stack. The structure of the current spine is statically determined by the type system.

$$\text{(retCont)} \quad V : \mathsf{retCont} \Downarrow V$$

$$\text{(appCont)} \quad \frac{\mathcal{E}, V_1 \cdot \mathcal{S}, K \vdash M \Downarrow V_2}{V_1 : \mathsf{appCont}(\mathcal{E}, \mathcal{S}, M, K) \Downarrow V_2}$$

$$\text{(varRet)} \quad \frac{V_0 : K \Downarrow V}{\mathcal{E}\{x : V_0\}, \emptyset, K \vdash x \Downarrow V}$$

$$\text{(varCont)} \quad \frac{\mathcal{E}_0, V_0 \cdot \mathcal{S}, K \vdash \lambda y.M_0 \Downarrow V}{\mathcal{E}\{x : cls(\mathcal{E}_0, \lambda y.M_0)\}, V_0 \cdot \mathcal{S}, K \vdash x \Downarrow V}$$

$$\text{(absRet)} \quad \frac{cls(\mathcal{E}, \lambda x.M) : K \Downarrow V}{\mathcal{E}, \emptyset, K \vdash \lambda x.M \Downarrow V}$$

$$\text{(absCont)} \quad \frac{\mathcal{E}\{x : V_0\}, \mathcal{S}, K \vdash M \Downarrow V}{\mathcal{E}, V_0 \cdot \mathcal{S}, K \vdash \lambda x.M \Downarrow V}$$

$$\text{(apply)} \quad \frac{\mathcal{E}, \emptyset, \mathsf{appCont}(\mathcal{E}, \mathcal{S}, M_1, K) \vdash M_2 \Downarrow V}{\mathcal{E}, \mathcal{S}, K \vdash M_1\ M_2 \Downarrow V}$$

$$\text{(wrong)} \quad \frac{\text{(if no other rule applies)}}{\mathcal{E}, \mathcal{S}, K \vdash M \Downarrow wrong}$$

Fig. 4. A call-by-value semantics with continuation

3.1 Instruction, Machine State and Execution

The set of instructions (ranged over by I) and the set of values (ranged over by v) of our Krivine machine are given as follows.

$$I ::= \mathsf{Acc}(x) \mid \mathsf{Grab}(x) \mid \mathsf{Push} \mid \mathsf{Install} \mid \mathsf{MkCls}(C) \mid \mathsf{Return} \mid \mathsf{Const}(c) \mid \mathsf{Add}$$
$$v ::= c \mid cls(E, C) \mid wrong$$

A machine state is a 5-tuple (E, S, L, C, D) where an environment E is a function from a finite set of variables to values, a spine stack S is a list of values, a local stack L is a list of values, and a code C is a list of instructions, and a "dump" D represents suspended computation, whose syntax is as follows:

$$D ::= \emptyset \mid (E, L, C) \cdot D$$

A local stack is used for holding the return value and arguments to various primitive operation. As an example of primitive operation, we only include Add for integer addition. A spine stack S represents a series of spines, including the current spine for code C and those for the computation saved in D. As we shall comment more on this later, there is no explicit delimiter in S to separate each spine. The code is statically compiled to stop at the end of the current spine.

The behavior of this machine is defined by giving a state transition relation of the form

$$(E, S, L, C, D) \longrightarrow (E', S', L', C', D')$$

determined by the set of rules given in Figure 5. We write $(E, S, L, C, D) \Downarrow v$ if $(E, S, L, C, D) \xrightarrow{*} v$.

$$
\begin{aligned}
(E\{x:v\},\ S,\ L,\ \mathsf{Acc}(x)\cdot C,\ D) &\longrightarrow (E\{x:v\},\ S,\ v\cdot L,\ C,\ D)\\
(E,\ v\cdot S,\ L,\ \mathsf{Grab}(x)\cdot C,\ D) &\longrightarrow (E\{x:v\},\ S,\ L,\ C,\ D)\\
(E,\ S,\ v\cdot L,\ \mathsf{Push}\cdot C,\ D) &\longrightarrow (E,\ v\cdot S,\ L,\ C,\ D)\\
(E,\ S,\ v\cdot L,\ \mathsf{Return},\ (E_0,L_0,C_0)\cdot D) &\longrightarrow (E_0,\ S,\ v\cdot L_0,\ C_0,\ D)\\
(E,\ \emptyset,\ v\cdot L,\ \mathsf{Return},\ \emptyset) &\longrightarrow v\\
(E,\ S,\ L,\ \mathsf{MkCls}(C_0)\cdot C,\ D) &\longrightarrow (E,\ S,\ cls(E,C_0)\cdot L,\ C,\ D)\\
(E,\ S,\ cls(E_0,C_0)\cdot L,\ \mathsf{Install}\cdot C,\ D) &\longrightarrow (E_0,\ S,\ \emptyset,\ C_0,\ (E,L,C)\cdot D)\\
(E,\ S,\ L,\ C,\ D) &\longrightarrow \mathit{wrong}\ \text{(if no other rule applies)}
\end{aligned}
$$

Fig. 5. An operational semantics for Krivine machine

3.2 A Type System for Krivine Machine Code

We follow [9] and develop a type system for the Krivine machine code language by specifying a typing rule of the form for each instruction I

$$\text{(Rule-I)}\quad \frac{\Gamma' \mid \Delta' \mid \Pi' \rhd C : \tau}{\Gamma \mid \Delta \mid \Pi \rhd I\cdot C : \tau}$$

indicating the fact that I changes the computation state from the one represented by (Γ, Δ, Π) to the one represented by (Γ', Δ', Π'). τ indicates the type of the final result, which is determined by Return instruction. The set of typing rules is given in Figure 6.

We prove the soundness theorem for this type system with respect to the machine behavior. To do this, we define typing relations on each of machine components as below.

- *Value typing* :
  - $\models c : b$ (for some predefined base type)
  - $\models cls(E,C) : \Delta \to \tau$ if there is some Γ such that $\models E : \Gamma$, $\Gamma \mid \Delta \mid \emptyset \rhd C : \tau$
- *Environment Typing* :
  - $\models \emptyset : \emptyset$
  - $\models E\{x:v\} : \Gamma\{x:\tau\}$ if $\models v : \tau$ and $\models E : \Gamma$
- *Spine Stack Typing (similarly for local stack typing $\models L : \Pi$)* :
  - $\models \emptyset : \emptyset$
  - $\models v\cdot S : \tau\cdot\Delta$ if $\models v : \tau$ and $\models S : \Delta$
- *Dump typing* :
  - $\emptyset \models \emptyset : \tau \Rightarrow \tau$ for any τ
  - $S_1\cdot S_2 \models (E,L,C)\cdot D : \tau_1 \Rightarrow \tau_2$ if there are some $\Gamma, \Delta, \Pi, \tau_3$ such that $\models E : \Gamma$, $\models S_1 : \Delta$, $\models L : \Pi$, $\Gamma \mid \Delta \mid \tau_1\cdot\Pi \rhd C : \tau_3$, and $S_2 \models D : \tau_3 \Rightarrow \tau_2$

Using these definitions, we can establish the following.

Theorem 2. *If $\Gamma \mid \Delta \mid \Pi \rhd C : \tau$, $\models E : \Gamma$, $\models S_1 : \Delta$, $\models L : \Pi$, $S_2 \models D : \tau \Rightarrow \tau'$ and $(E, S_1\cdot S_2, L, C, D) \Downarrow v$ then $\models v : \tau'$.*

$$\text{(Return)} \quad \Gamma \mid \emptyset \mid \tau \rhd \mathsf{Return} : \tau$$

$$\text{(Acc)} \quad \frac{\Gamma\{x:\tau\} \mid \Delta \mid \tau \cdot \Pi \rhd C : \tau_0}{\Gamma\{x:\tau\} \mid \Delta \mid \Pi \rhd \mathsf{Acc}(x) \cdot C : \tau_0}$$

$$\text{(Grab)} \quad \frac{\Gamma\{x:\tau\} \mid \Delta \mid \Pi \rhd C : \tau_0}{\Gamma \mid \tau \cdot \Delta \mid \Pi \rhd \mathsf{Grab}(x) \cdot C : \tau_0}$$

$$\text{(Closure)} \quad \frac{\Gamma \mid \Delta_0 \mid \emptyset \rhd C_0 : \tau_0 \quad \Gamma \mid \Delta \mid \Delta_0 \to \tau_0 \cdot \Pi \rhd C : \tau}{\Gamma \mid \Delta \mid \Pi \rhd \mathsf{MkCls}(C_0) \cdot C : \tau}$$

$$\text{(Push)} \quad \frac{\Gamma \mid \tau \cdot \Delta \mid \Pi \rhd C : \tau_0}{\Gamma \mid \Delta \mid \tau \cdot \Pi \rhd \mathsf{Push} \cdot C : \tau_0}$$

$$\text{(Install)} \quad \frac{\Gamma \mid \Delta_2 \mid \tau \cdot \Pi \rhd C : \tau_0}{\Gamma \mid \Delta_1 \cdot \Delta_2 \mid \Delta_1 \to \tau \cdot \Pi \rhd \mathsf{Install} \cdot C : \tau_0}$$

Fig. 6. The type system for Krivine machine

3.3 Type-Directed Compilation

The type system of the source Krivine calculus statically determines the spine stack of each term. This information is essential for producing efficient code. To exploit this static information, we define a compilation algorithm as an algorithm that inductively translates a given typing derivation to Krivine code. We write

$$\Gamma \mid \Delta \rhd M \rightsquigarrow_k C$$

to indicate that a typing M under Γ and Δ is compiled to C. The algorithm is given in Figure 7.

Figure 8 shows the compiled code for the example typing derivation given in Figure 2. As shown in Figure 8 the derivation of $\lambda x.\lambda y.M$ does not involve any structural rule, and therefore the compilation algorithm generates two consecutive Grab instructions without creating any closure. On the other hand, the code for $\lambda w.\lambda z.N$ contains two closure generation instructions corresponding to (closure) rule in its typing derivation.

This algorithm preserves typing as shown in the following.

Theorem 3. *If $\Gamma \mid \Delta \rhd M : \tau$ and $\Gamma \mid \Delta \rhd M \rightsquigarrow_k C$ then for any C_0, Δ_0, Π_0, and τ_0, if $\Gamma \mid \Delta_0 \mid \tau \cdot \Pi_0 \rhd C_0 : \tau_0$ then $\Gamma \mid \Delta \cdot \Delta_0 \mid \Pi_0 \rhd C \cdot C_0 : \tau_0$.*

3.4 Correctness of Compilation

The combination of Theorem 2 and Theorem 3 establishes the soundness of the type system of the source language with respect to the operational semantics obtained by combining the compilation to the Krivine machine followed by executing the compiled code.

This only establishes a weak form of correctness. In this subsection, we establish a stronger property ensuring that the compiled code has the same operational behavior as that of the original source calculus. We achieve this by setting up a family of correspondence relations between the semantic objects in the source calculus and those of the target Krivine machine.

$$\text{(var)} \quad \Gamma\{x:\tau\} \mid \emptyset \rhd x \rightsquigarrow_k \mathsf{Acc}(x)$$

$$\text{(abs)} \quad \frac{\Gamma\{x:\tau\} \mid \Delta \rhd M \rightsquigarrow_k C}{\Gamma \mid \tau \cdot \Delta \rhd \lambda x.M \rightsquigarrow_k \mathsf{Grab}(x) \cdot C}$$

$$\text{(app)} \quad \frac{\Gamma \mid \tau \cdot \Delta \rhd M_1 \rightsquigarrow_k C_1 \quad \Gamma \mid \emptyset \rhd M_2 \rightsquigarrow_k C_2}{\Gamma \mid \Delta \rhd M_1\ M_2 \rightsquigarrow_k C_2 \cdot \mathsf{Push} \cdot C_1}$$

$$\text{(code)} \quad \frac{\Gamma \mid \emptyset \rhd M \rightsquigarrow_k C}{\Gamma \mid \Delta \rhd M \rightsquigarrow_k C \cdot \mathsf{Install}}$$

$$\text{(val)} \quad \frac{\Gamma \mid \Delta \rhd \lambda x.M \rightsquigarrow_k C}{\Gamma \mid \emptyset \rhd \lambda x.M \rightsquigarrow_k \mathsf{MkCls}(C \cdot \mathsf{Return})}$$

Fig. 7. Type-directed compilation for Krivine machine

- *Value correspondence* :
  - $\models c \sim c : b$ for any constant c of type b
  - $\models cls(\mathcal{E}, \lambda x.M) \sim cls(E, C_0 \cdot \mathsf{Return}) : \Delta \to \tau$ if there is some Γ such that $\models \mathcal{E} \sim E : \Gamma$, $\Gamma \mid \Delta \rhd \lambda x.M \rightsquigarrow_k C_0$.
- *Context correspondence* :
  - $\models \mathcal{E} \sim E : \Gamma$ if $\models \mathcal{E} : \Gamma$, $\models E : \Gamma$, and $\models \mathcal{E}(x) \sim E(x) : \Gamma(x)$ for all $x \in dom(\Gamma)$.
  - $\models \mathcal{S} \sim S : \Delta$ if $\models \mathcal{S} : \Delta$, $\models S : \Delta$, and $\models \mathcal{S}.i \sim S.i : \Delta.i$ for each $1 \le i \le |\mathcal{S}|$.
- *Continuation correspondence* :
  - $\emptyset \models \mathsf{retCont} \sim \emptyset : \tau \Rightarrow \tau$ for any τ.
  - $S_1 \cdot S_2 \models \mathsf{appCont}(\mathcal{E}, \mathcal{S}_1 \cdot \mathcal{S}_2, M, K) \sim (E, \emptyset, \mathsf{Push} \cdot C \cdot \mathsf{Install}_1 \cdot \ldots \cdot \mathsf{Install}_n \cdot \mathsf{Return}) \cdot D : \tau \Rightarrow \tau_0$ if there are some $\Gamma, \Delta, \Delta_i, \tau', \mathcal{S}_i^2$, and S_i^2 such that $\models \mathcal{E} \sim E : \Gamma$, $\models \mathcal{S}_1 \sim S_1 : \Delta$, $\Gamma \mid \tau \cdot \Delta \rhd M : \Delta_1 \to \cdots \to \Delta_n \to \tau'$, $\Gamma \mid \tau \cdot \Delta \rhd M \rightsquigarrow_k C$, $\mathcal{S}_2 = \mathcal{S}_1^2 \cdot \ldots \cdot \mathcal{S}_n^2$, $S_2 = S_1^2 \cdot \ldots \cdot S_n^2 \cdot S_{n+1}^2$, $\models \mathcal{S}_i \sim S_i : \Delta_i$, and $S_{n+1}^2 \models K \sim D : \tau' \Rightarrow \tau_0$.

We now establish the following semantic correctness theorem.

Theorem 4. *Suppose* $\Gamma \mid \Delta \rhd M : \Delta_1 \to \cdots \to \Delta_n \to \tau$, $\Gamma \mid \Delta \rhd M \rightsquigarrow_k C$, $\models \mathcal{E} \sim E : \Gamma$, $\models \mathcal{S}_1 \sim S_1 : \Delta$, $\mathcal{S}_2 = \mathcal{S}_1^2 \cdot \ldots \cdot \mathcal{S}_n^2$, $S_2 = S_1^2 \cdot \ldots \cdot S_n^2 \cdot S_{n+1}^2$, $\models \mathcal{S}_i \sim S_i : \Delta_i$, *and* $S_{n+1}^2 \models K \sim D : \tau \Rightarrow \tau_0$. *If* $\mathcal{E}, \mathcal{S}_1 \cdot \mathcal{S}_2, K \vdash M \Downarrow V$ *then* $(E, S_1 \cdot S_2, \emptyset, C \cdot \mathsf{Install}_1 \cdot \ldots \cdot \mathsf{Install}_n \cdot \mathsf{Return}, D) \Downarrow v$ *such that* $\models V \sim v : \tau_0$.

4 A Dynamically Typed Krivine Machine

This section considers Leroy's ZINC machine – an existing call-by-value Krivine abstract machine. Unlike in our Krivine machine we have just developed, each spine in the ZINC machine is explicitly separated by marks and each instruction grabbing an element in the current spine dynamically checks its availability. For this machine, we develop the results on compilation and its correctness parallel to those in the previous section and the relationship with the two machines.

(* $\lambda w.\lambda z.N$ *) $\mathsf{MkCls}(\mathsf{Grab}(w)\cdot\mathsf{MkCls}(\mathsf{Grab}(z)\cdot[\text{ code for } N \text{ }]\cdot\mathsf{Return})\cdot\mathsf{Return})\cdot\mathsf{Push}\cdot$
(* $\lambda f.\cdots(f\ 3)$ *) $\mathsf{Grab}(f)\cdot\mathsf{Const}(3)\cdot\mathsf{Push}\cdot\mathsf{Acc}(f)\cdot\mathsf{Install}\cdot\mathsf{Push}\cdot$
(* $f\ 1\ 2$ *) $\mathsf{Const}(2)\cdot\mathsf{Push}\cdot\mathsf{Const}(1)\cdot\mathsf{Push}\cdot\mathsf{Acc}(f)\cdot\mathsf{Install}\cdot\mathsf{Install}\cdot\mathsf{Push}\cdot$
(* $\lambda x.\lambda y.M$ *) $\mathsf{Grab}(x)\cdot\mathsf{Grab}(y)\cdot[\text{ code for } M \text{ }]\cdot$
(* *) Return

Fig. 8. Krivine machine code generated for the typing derivation of $(\lambda f.(\lambda x.\lambda y.M)\ (f\ 1\ 2)\ (f\ 3))\ (\lambda w.\lambda z.N)$ shown in Figure 2

4.1 Instruction, Machine State and Execution

To present type system of ZINC machine, we consider the following set of instructions (ranged over by I) and values (ranged over by v) given as follows.

$$I ::= \mathsf{Acc}(x) \mid \mathsf{Grab}(x) \mid \mathsf{Reduce}(C) \mid \mathsf{Push} \mid \mathsf{Return} \mid \mathsf{Const}(c) \mid \mathsf{Add}$$
$$v ::= c \mid cls(E, C) \mid wrong$$

Each state of the ZINC machine consists of an environment (E), a spine stack (S), a local stack (L), a code (C), and a dump stack (D). Different from the Krivine machine, a dump stack contains a spine stack, as defined below:

$$D ::= \emptyset \mid (E, S, L, C) \cdot D$$

S in the current machine state and in each dump entry corresponds to one spine i.e. an application context of the term being reduced. This structure enables the code to check dynamically in runtime if the current spine is empty. Due to this property, we regarded this form of Krivine machine "dynamically typed" with respect of the structure of spine. This is in contrast with our Krivine machine where sequence of spines are put into one stack without any delimiter.

We note that, in the Leroy's original presentation of ZINC machine [6], the machine state consists of an argument stack of the form $L_1S_1 \bullet L_2S_2 \bullet \cdots \bullet L_nS_n$ containing sequence of spines $S_1S_2\cdots S_n$ and local stacks $L_1L_2\cdots L_n$ separated by a special symbol called "mark" denoted here by $\bullet$, and a return stack of the form $(E_1, C_1)(E_2, C_2)\cdots(E_n, C_n)$. In this organization, the emptiness check of the current spine is performed by checking whether the top value of the argument stack is $\bullet$ or not. We can see that this original machine structure is isomorphic to the machine state defined above.

The behavior of ZINC machine is defined by a state transition relation of the form

$$(E, S, L, C, D) \longrightarrow (E', S', L', C', D')$$

determined by the set of rules given in Figure 9.

The machine starts evaluating a new application context after storing the current spine stack onto the dump stack and then making it empty by $\mathsf{Reduce}(C)$. The dynamic check precedes getting a value from the spine stack to see if some value is available by either $\mathsf{Grab}(x)$ or Return. The stored spine stack will be restored only after a probing instruction finds the current spine stack exhausted.

$$
\begin{aligned}
(E\{x:v\},\ S,\ L,\ \mathsf{Acc}(x)\cdot C,\ D) &\longrightarrow (E\{x:v\},\ S,\ v\cdot L,\ C,\ D)\\
(E,\ v\cdot S,\ L,\ \mathsf{Grab}(x)\cdot C,\ D) &\longrightarrow (E\{x:v\},\ S,\ L,\ C,\ D)\\
(E,\emptyset,L,\mathsf{Grab}(x)\cdot C,(E_0,S_0,L_0,C_0)\cdot D) &\longrightarrow (E_0,S_0,cls(E,\mathsf{Grab}(x)\cdot C)\cdot L_0,C_0,D)\\
(E,\ \emptyset,\ L,\ \mathsf{Grab}(x)\cdot C,\ \emptyset) &\longrightarrow cls(E,\mathsf{Grab}(x)\cdot C)\\
(E,\ S,\ v\cdot L,\ \mathsf{Push}\cdot C,\ D) &\longrightarrow (E,\ v\cdot S,\ L,\ C,\ D)\\
(E,\ v\cdot S,\ cls(E_0,C_0)\cdot L,\ \mathsf{Return},\ D) &\longrightarrow (E_0,\ v\cdot S,\ L,\ C_0,\ D)\\
(E,\emptyset,v\cdot L,\mathsf{Return},(E_0,S_0,L_0,C_0)\cdot D) &\longrightarrow (E_0,\ S_0,\ v\cdot L_0,\ C_0,\ D)\\
(E,\ \emptyset,\ v\cdot L,\ \mathsf{Return},\ \emptyset) &\longrightarrow v\\
(E,\ S,\ L,\ \mathsf{Reduce}(C_0)\cdot C,\ D) &\longrightarrow (E,\ \emptyset,\ \emptyset,\ C_0,\ (E,S,L,C)\cdot D)\\
(E,\ S,\ L,\ C,\ D) &\longrightarrow \mathit{wrong}\ \text{(if no other rule applies)}
\end{aligned}
$$

Fig. 9. An operational semantics for ZINC Machine

4.2 A Type System for ZINC Machine Code

We first define the typing relations for value, for environment, for local stack, for spine stack, and for dump stack as follows:

- *Value Typing* :
  - $\models c : b$ (for some predefined base type)
  - $\models cls(E,C) : \Delta \to \tau$ if there is some Γ such that $\models E : \Gamma$, $\Gamma \mid \Delta \mid \emptyset \rhd C : \tau$.
- *Environment Typing* :
  - $\models \emptyset : \emptyset$
  - $\models E\{x:v\} : \Gamma\{x:\tau\}$ if $\models v : \tau$ and $\models E : \Gamma$
- *Spine Stack Typing (similarly for local stack typing $\models L : \Pi$)* :
  - $\models \emptyset : \emptyset$
  - $\models v \cdot S : \tau \cdot \Delta$ if $\models v : \tau$ and $\models S : \Delta$
- *Dump typing* :
  - $\models \emptyset : \tau \Rightarrow \tau$ (for any type)
  - $\models (E,S,L,C)\cdot D : \tau \Rightarrow \tau_0$ if there are some Γ, Δ, Π, and τ' such that $\models E : \Gamma$, $\models S : \Delta$, $\models L : \Pi$, $\Gamma \mid \Delta \mid \tau \cdot \Pi \rhd C : \tau'$, $\models D : \tau' \Rightarrow \tau_0$.

The set of typing rules for the instructions is given in Figure 10. Note that there are two typing rules (Grab-abs) and (Grab-clo). This captures the dynamic behavior of $\mathsf{Grab}(x)$ depending on the spine stack. (Grab-abs) applies when a spine stack is not empty; otherwise (Grab-clo) applies. These two typing rules correspond to two distinct actions performed. When the spine is not empty, it actually grabs the top element from the spine and continues. Otherwise, it creates a closure.

Similarly to $\mathsf{Grab}(x)$, Return also performs two distinct actions – invoking the current closure when the spine is not empty, and returning to the caller otherwise; (Return-ins) and (Return-ret) apply respectively.

We prove the soundness theorem for this type system with respect to the ZINC machine.

Theorem 5. *If $\Gamma \mid \Delta \mid \Pi \rhd C : \tau$, $\models E : \Gamma$, $\models S : \Delta$, $\models L : \Pi$, $\models D : \tau \Rightarrow \tau_0$, and $(E,S,L,C,D) \Downarrow v$ then $\models v : \tau_0$.*

$$\text{(Return-ret)} \quad \Gamma \mid \emptyset \mid \tau \rhd \mathsf{Return} : \tau$$

$$\text{(Return-ins)} \quad \frac{\Gamma \mid \Delta' \mid \tau' \rhd \mathsf{Return} : \tau}{\Gamma \mid \Delta \cdot \Delta' \mid \Delta \to \tau' \rhd \mathsf{Return} : \tau}$$

$$\text{(Acc)} \quad \frac{\Gamma\{x : \tau\} \mid \Delta \mid \tau \cdot \Pi \rhd C : \tau'}{\Gamma\{x : \tau\} \mid \Delta \mid \Pi \rhd \mathsf{Acc}(x) \cdot C : \tau'}$$

$$\text{(Grab-abs)} \quad \frac{\Gamma\{x : \tau\} \mid \Delta \mid \emptyset \rhd C : \tau'}{\Gamma \mid \tau \cdot \Delta \mid \emptyset \rhd \mathsf{Grab}(x) \cdot C : \tau'}$$

$$\text{(Grab-clo)} \quad \frac{\Gamma \mid \Delta \mid \emptyset \rhd \mathsf{Grab}(x) \cdot C : \tau}{\Gamma \mid \emptyset \mid \emptyset \rhd \mathsf{Grab}(x) \cdot C : \Delta \to \tau}$$

$$\text{(Push)} \quad \frac{\Gamma \mid \tau \cdot \Delta \mid \Pi \rhd C : \tau'}{\Gamma \mid \Delta \mid \tau \cdot \Pi \rhd \mathsf{Push} \cdot C : \tau'}$$

$$\text{(Reduce)} \quad \frac{\Gamma \mid \emptyset \mid \emptyset \rhd C_0 : \tau_0 \quad \Gamma \mid \Delta \mid \tau_0 \cdot \Pi \rhd C : \tau}{\Gamma \mid \Delta \mid \Pi \rhd \mathsf{Reduce}(C_0) \cdot C : \tau}$$

Fig. 10. The type system for ZINC machine

4.3 Type-Preserving Compilation

The compilation for ZINC machine in Figure 11, opposed to that for Krivine machine, does not exploit static information about spine stacks as the machine totally depends on the dynamic situation of the spine stacks. As we shall comment more on this later in Section 4.5, the static information on the spine stacks could lead us to develop an optimization that enables ZINC machine to avoid some of dynamic tests on spine stack.

For consistency, we simply write the same compilation judgment

$$\Gamma \mid \Delta \rhd M \rightsquigarrow_z C$$

M is compiled to C under a typing M under Γ and Δ by the algorithm shown in Figure 11. An example of compiled code is shown in Figure 12.

This compilation algorithm preserves typing as shown in the following.

Theorem 6. *If* $\Gamma \mid \Delta \rhd M : \tau$ *and* $\Gamma \mid \Delta \rhd M \rightsquigarrow_z C$ *then* $\Gamma \mid \Delta \mid \emptyset \rhd C : \tau$.

4.4 Correctness of Compilation

Now we will show the correctness of compilation to establish a stronger property of semantic correctness of ZINC code with respect to the Krivine-style call-by-value operational semantics with continuation in Figure 4. Each correctness property of Krivine and ZINC machines allows us to confirm that the two machines are related indirectly through the behavior of source calculus.

We define correspondence between semantic objects of source calculus and those of ZINC machine by the following relations

- *Value correspondence* :
- $\models c \sim c : b$ for all constants of type b
- $\models cls(\mathcal{E}, \lambda x.M) \sim cls(E, C) : \Delta \to \tau$ if there is some Γ such that $\models \mathcal{E} \sim E : \Gamma$ and $\Gamma \mid \Delta \rhd \lambda x.M \rightsquigarrow_z C$

$$\text{(var)} \quad \Gamma\{x:\tau\} \mid \emptyset \rhd x \rightsquigarrow_z \mathsf{Acc}(x) \cdot \mathsf{Return}$$

$$\text{(abs)} \quad \frac{\Gamma\{x:\tau\} \mid \Delta \rhd M \rightsquigarrow_z C}{\Gamma \mid \tau \cdot \Delta \rhd \lambda x.M \rightsquigarrow_z \mathsf{Grab}(x) \cdot C}$$

$$\text{(app)} \quad \frac{\Gamma \mid \tau \cdot \Delta \rhd M_1 \rightsquigarrow_z C_1 \quad \Gamma \mid \emptyset \rhd M_2 \rightsquigarrow_z C_2}{\Gamma \mid \Delta \rhd M_1\ M_2 \rightsquigarrow_z \mathsf{Reduce}(C_2) \cdot \mathsf{Push} \cdot C_1}$$

$$\text{(code)} \quad \frac{\Gamma \mid \emptyset \rhd M \rightsquigarrow_z C}{\Gamma \mid \Delta \rhd M \rightsquigarrow_z C}$$

$$\text{(val)} \quad \frac{\Gamma \mid \Delta \rhd \lambda x.M \rightsquigarrow_z C}{\Gamma \mid \emptyset \rhd \lambda x.M \rightsquigarrow_z C}$$

Fig. 11. Type-preserving compilation for ZINC machine

- *Context correspondence* :
- $\models \mathcal{E} \sim E : \Gamma$ if $\models \mathcal{E} : \Gamma$, $\models E : \Gamma$, and $\models \mathcal{E}(x) \sim E(x) : \Gamma(x)$ for all $x \in dom(\Gamma)$.
- $\models \mathcal{S} \sim S : \Delta$ if $|\mathcal{S}| = |S|$ and $\models \mathcal{S}.i \sim S.i : \Delta.i$ for all $1 \leq i \leq |\mathcal{S}|$
- *Continuation correspondence* :
- $\models \mathsf{retCont} \sim \emptyset : \tau \Rightarrow \tau$ (for any type)
- $\models \mathsf{appCont}(\mathcal{E}, \mathcal{S}, M, K) \sim (E, S, \emptyset, \mathsf{Push} \cdot C) \cdot D : \tau \Rightarrow \tau_0$ if there are some Γ, Δ, Δ_i, τ', $\mathcal{S}_i$, and S_i such that $\models \mathcal{E} \sim E : \Gamma$, $\mathcal{S} = \mathcal{S}_0 \cdot \mathcal{S}_1 \cdot \ldots \cdot \mathcal{S}_n$, $S = S_0 \cdot S_1 \cdot \ldots \cdot S_n$, $\models \mathcal{S}_0 \sim S_0 : \Delta$, $\models \mathcal{S}_i \sim S_i : \Delta_i$, $\models K \sim D : \tau' \Rightarrow \tau_0$, $\Gamma \mid \tau \cdot \Delta \rhd M : \Delta_1 \to \cdots \to \Delta_n \to \tau'$, $\Gamma \mid \tau \cdot \Delta \rhd M \rightsquigarrow_z C$.

Now we can show the semantic correctness theorem as the following.

Theorem 7. *Suppose* $\Gamma \mid \Delta \rhd M : \Delta_1 \to \cdots \to \Delta_n \to \tau$, $\Gamma \mid \Delta \rhd M \rightsquigarrow_z C$, $\models \mathcal{E} \sim E : \Gamma$, $\mathcal{S} = \mathcal{S}_0 \cdot \mathcal{S}_1 \cdot \ldots \cdot \mathcal{S}_n$, $S = S_0 \cdot S_1 \cdot \ldots \cdot S_n$, $\models \mathcal{S}_0 \sim S_0 : \Delta$, $\models \mathcal{S}_i \sim S_i : \Delta_i$, *and* $\models K \sim D : \tau \Rightarrow \tau'$. *If* $\mathcal{E}, \mathcal{S}, K \vdash M \Downarrow V$ *then* $(E, S, \emptyset, C, D) \Downarrow v$ *s.t.* $\models V \sim v : \tau'$.

4.5 Comparison of Krivine and ZINC Abstract Machines

The Krivine and ZINC machine show the opposite behavior of each other in using marks. The Krivine machine determines the range of each application statically by exploiting compile-time information from our type theory. It can avoid the whole of dynamic check, but it sometimes creates unnecessary closures due to the approximate information on application contexts. The ZINC machine determines the range of each application dynamically by exploiting runtime information. It can avoid the whole of unnecessary closure creation by runtime check.

For $\lambda x.\lambda y.M$ in Figure 2, the corresponding Krivine code will safely grab each argument with no runtime check, as shown in Figure 8 because the function has type $\{int \cdot \{int\} \to int\} \to int$ from our type theory. However, the corresponding ZINC code will blindly check in runtime, as shown in Figure 12. This problem could be resolved by exploiting type information from our type theory.

For $\lambda w.\lambda z.N$ in Figure 2, the corresponding Krivine code will create a closure before it takes each argument because the function has type $\{int\} \to \{int\} \to int$ from our type theory. Even in the application context $f\ 1\ 2$ where two arguments

(* $\lambda w.\lambda z.N$ *) Reduce(Grab(w)·Grab(z)·[code for N])·
(* $\lambda f.\ \cdots\ (f\ 3)$ *) Grab(f)·Reduce(Const(3)·Push·Acc(f)·Return)·
(* $f\ 1\ 2$ *) Reduce(Const(2)·Push·Const(1)·Push·Acc(f)·Return)·
(* $\lambda x.\lambda y.M$ *) Grab(x)·Grab(y)·[code for M]

Fig. 12. ZINC machine code generated for the typing derivation of $(\lambda f.(\lambda x.\lambda y.M)\ (f\ 1\ 2)\ (f\ 3))\ (\lambda w.\lambda z.N)$ shown in Figure 2

are all available, the function has to be applied by two Install instructions, as shown in Figure 8, creating an unnecessary intermediate closure. The corresponding ZINC code will avoid this creation by runtime check.

5 Related Work

Although we do not aware of any work that directly addressed issues on type system and typed compilation for Krivine-style evaluation, there have been several researches worth being mentioned in terms of each facet of our type theory.

Hannan and Hick [5] proposed a type system as an analysis for uncurrying optimization identifying static application contexts by *uncurried function types*, which is very similar to spine stack types. Our type system, however, deals with nested application while their type system does not.

Existing approaches for typeful low-level machines, such as the typed assembly language [7] and the logical abstract machine [9, 8], have implemented higher-order functions with *dump stack*. As a natural consequence, they cannot properly deal with Krivine-style evaluation well. This phenomenon seems to be due to their choice of source calculus: *CPS* language and *A-normal form* language where both have no notion of spine stack. Our approach with the Krivine-style calculus as a source calculus perfectly fits for Krivine-style evaluation.

Choi and Han [1] proposed a type system for handling dynamic application contexts. Their type system captured dynamism through some non-standard type encoding, which made their type system complex and inflexible. Our type system for ZINC machine used nondeterministic typing rules to capture dynamism, and it remained simple and flexible.

6 Conclusions

We have investigated typing properties of Krivine-style evaluation from source lambda terms down to abstract machine codes. First, we have developed a typed term calculus that accounts for Krivine-style evaluation with "spine" stack, and have shown that the type system is sound with respect to Krivine-style operational semantics. The type system has the same expressive power as that of the conventional type system of the lambda calculus. It can therefore serve as an alternative type theoretical framework for analysis and type-based optimization of lambda terms such as uncurrying.

Second, we have designed type systems for two varieties of Krivine-style low level abstract machines – one with statically determined spine stacks and the other, similar to ZINC machine, with dynamically determined ones, and have established the type soundness property for both machines. For those Krivine abstract machines, we have developed type-directed compilation algorithms, and have shown that the algorithms preserve both typing and operational semantics. We have compared the relative merit of the two varieties of Krivine-style abstract machines within a single framework.

This is the first step toward a type theory for Krivine-style evaluation and compilation, and a number of issues remain to be investigated. First of all, we need to develop an optimal type inference algorithm that detects the longest evaluation context. Second, we need to develop a practical type-directed compilation algorithm that minimizes redundant closure creation. We are currently investigating the adaptation of the Krivine abstract machine in an ML compiler we have been developing at JAIST.

Acknowledgments

The first author would like to thank Kwangkeun Yi for his helpful comments on an earlier version of ZINC type system. The second author would like to thank René Vestergaard for insightful discussion on Krivine-style evaluation.

References

1. K. Choi and T. Han. A type system for the push-enter model. *Information Processing Letters*, 87:205–211, 2003.
2. K. Choi and A. Ohori. A type theory for Krivine-style evaluation and compilation. Technical report IS-RR-2004-014, JAIST, 2004.
3. P. Cregut. An abstract machine for the normalization of λ-terms. In *Proc. ACM Conference on LISP and Functional Programming*, pages 333–340, 1990.
4. B. Duba, R. Harper, and D. MacQueen. Typing first-class continuations in ML. In *Proc. ACM Symp. on Principles of Prog. Languages*, pages 163–73, 1991.
5. J. Hannan and P. Hicks. Higher-order uncurrying. In *Proc. ACM Symposium on Principles of Programming Languages*, 1998.
6. X. Leroy. The ZINC experiment: an economical implementation of the ML language. Technical Report 117, INRIA, 1992.
7. G. Morrisett, K. Crary, N. Glew, and D. Walker. Stack-based typed assembly language. In *Proc. Workshop on Types in Compilation, LNCS 1478*, 1998.
8. A. Ohori. A Curry-Howard isomorphism for compilation and program execution. In *Proc. Typed Lambda Calculi and Applications, LNCS 1581*, pages 258–179, 1999.
9. A. Ohori. The logical abstract machine: a Curry-Howard isomorphism for machine code. In *Proc. International Symp. on Functional and Logic Programming*, 1999.