PAPER

Typing ZINC Machine with Generalized Algebraic Data Types

Kwanghoon CHOI[†*a)], *Nonmember and* **Seog PARK**[†], *Member*

SUMMARY The Krivine-style evaluation mechanism is well-known in the implementation of higher-order functions, allowing to avoid some useless closure building. There have been a few type systems that can verify the safety of the mechanism. The incorporation of the proposed ideas into an existing compiler, however, would require significant changes in the type system of the compiler due to the use of some dedicated form of types and typing rules in the proposals. This limitation motivates us to propose an alternative light-weight Krivine typing mechanism that does not need to extend any existing type system significantly. This paper shows how GADTs (Generalized algebraic data types) can be used for typing a ZINC machine following the Krivine-style evaluation mechanism. This idea is new as far as we know. Some existing typed compilers like GHC (Glasgow Haskell compiler) already support GADTs; they can benefit from the Krivine-style evaluation mechanism in the operational semantics with no particular extension in their type systems for the safety. We show the GHC type checker allows to prove mechanically that ZINC instructions are well-typed, which highlights the effectiveness of GADTs.

key words: *type system, ZINC machine, generalized algebraic data type, functional language*

1. Introduction

The Krivine abstract machine [1] can be regarded as a system to transform a state of the form (E, S, M) consisting of an environment E, a stack S, and a term M to be evaluated [2]. The distinguishing feature of the Krivine machine is its usage of a stack S, which maintains the application context (or *spine*) of the term M being evaluated. For example, for the term $(\lambda x.M)\ M_1\ \cdots\ M_n$, the machine causes the following transition. $(E, S, (\lambda x.M)\ M_1 \cdots M_n) \Rightarrow^* (E, V_1 \cdots V_n{\cdot}S, \lambda x.M)$ where each V_i is the value denoted by M_i under E, and $V_1 \cdots\ V_n \cdot S$ is the stack obtained by pushing $V_n, \ldots,$ and V_1 in this order. As seen from this example, the stack 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 popping the stack and binding the variable to the popped value as this. $(E, V_1 \cdots V_n{\cdot}S, \lambda x.M) \Rightarrow (E\{x : V_1\}, V_2 \cdots V_n{\cdot}S, M)$

As observed by Leroy [3], this mechanism avoids unnecessary closure construction in evaluating the nested applications such as $(\lambda x_1.\cdots.\lambda x_n.M)\ M_1\ \cdots\ M_n$, and yields potentially more efficient evaluation scheme for higher-order functional languages. The ZINC machine exploits this mechanism for a strict functional language. This mechanism is also closely related to abstract machines with "spine" for a lazy functional language, where a spine denotes the evaluation context represented by a stack.

We are interested in the design of a typed intermediate language whose type system is capable of verifying a certain safety of an intermediate code. We do not assume any particular compilation strategies used to produce the intermediate code. The type systems presented in [2] and [4] permit to verify the safety of the Krivine-style evaluation mechanism. The type systems can guarantee an important correctness property of the mechanism as follows. In each state (E, S, M), the spine stack S always holds appropriate arguments to the term M under E. The spine stack S should be empty if M is an integer term while S can be non-empty if M is a functional term.

However, an existing compiler would require some substantial change in the type system to incorporate the proposed capability because the studies [2], [4] all introduced some dedicated form of types and typing rules. The type system in [4] incorporates a set of type equivalence rules between the special types and the rest. A lot of advanced types not dealt in [4] but in an existing compiler would require defining new type equivalence rules. The type system in [2] features an extra type environment to keep track of the shape of spine stacks. This is absent from any existing typed compilers. To integrate such a feature, one would need to extend all typing judgments in an existing typed compiler. These limitations motivate us to propose an alternative light-weight Krivine typing mechanism that does not require any significant extension in an existing type system.

GADTs (Generalized Algebraic Data Types [5]) are a simple extension of data types. In a generalized algebraic data type, each data constructor can have arbitrary parameter types in their result type possibly different from those in the data type itself. For example, a generalized algebraic data type *GList a* can be declared as this.

- $Nil\ ::\ \forall a.\ GList\ a$
- $Cons\ ::\ \forall a\ b.\ a \rightarrow GList\ b \rightarrow GList\ (a \rightarrow b)$

Here, $GList\ (Int \rightarrow Bool \rightarrow Int)$ does not denote a uniform list of the function type as the usual List type any more. It allows a heterogeneous list like $Cons\ V_1\ (Cons\ V_2\ Nil)$ where V_1 and V_2 are an integer and a boolean respectively. *Nil* is a list of type *List Int*, and so $Cons\ V_2\ Nil$ becomes a list of type $List\ (Bool \rightarrow Int)$. Note that lists of type $Int \rightarrow Bool \rightarrow Int$ cannot be longer than two while the list

Manuscript received October 29, 2010.
Manuscript revised January 21, 2011.

[†]The authors are with the Department of Computer Science and Engineering, Sogang University, Korea.
[*]Presently, with the Computer & Telecommunication Engineering Division, Yonsei University, Wonju, Korea.
a) E-mail: khchoi@acm.org

DOI: 10.1587/transinf.E94.D.1190

of type *List Int* is only *Nil*. These heterogeneous lists typed with *GList T* can be used to resemble the possible configuration of a spine stack allowed for a term of type T by having the list elements indexed by the argument types in the spine of the type T.

This paper shows that the notion of GADTs can assure of the correctness of the Krivine-style evaluation mechanism in ZINC machine due to the reasons:

- A GADT allows to express a list of heterogeneous values whose types are associated with a prefix of the argument types in the spine of a function type.
- A GADT offers non-uniform polymorphism by making usable only the data constructors with parameter types unifiable, which will be explained in Sect. 4.

The use of GADTs for the Krivine-style evaluation mechanism in a typed setting is new. This finding is important because the evaluation mechanism is well-known in compilation of higher-order functions as has long been discussed in [3], [6]–[8]. If a type-based compiler supports GADTs, our proposal can be a light-weight Krivine typing mechanism on it. Currently existing compilers such as ATS [9] and GHC [10], [11] do already supported GADTs. Therefore, their type checkers are able to verify automatically some correctness property of the evaluation mechanism with no particular extension to them.

Our methodology is to show that the type structure of a lambda calculus of System-F extended with GADTs can ensure the invariants associated with the spine stack under the Krivine-style evaluation mechanism by embedding the ZINC machine into the calculus. The machine language will form a combinator-based target language within the calculus where each machine instruction becomes a combinator, i.e. a lambda term with no free variables.

Our contribution is summarized as follows.

- We prove the soundness of the ZINC machine by showing that an embedding of the machine into System-F extended with GADTs is correct statically and dynamically.
- We verify mechanically that ZINC instructions are well-typed by showing that our Haskell implementation for the embedding is successfully type-checked in Glasgow Haskell compiler.

Section 2 reviews the previous works. Section 3 defines a ZINC machine and its simple type system as a basis of our study. After presenting a calculus of System-F extended with GADTs, Sect. 4 shows how every ZINC instruction can be simulated by a lambda term in the calculus statically and dynamically. Section 5 discusses a few potential extensions to this work. Section 6 concludes the paper. All the proofs for the theorems presented in this paper are available in the appendix.

2. Related Work

Typed assembly language [12] and the logical abstract machine [13] are the most notable approaches for typed low-level machines. However, they have implemented higher-order functions only with *dump stack*. This is due to their choice of source calculus: *CPS* language and *A-normal form* language where both have no notion of the spine stack. As a natural consequence, they cannot properly deal with the Krivine-style evaluation mechanism in a typed setting.

This limitation motivated us to propose two Krivine type systems allowing to make use of the Krivine-style evaluation mechanism also in a typed setting. First, Choi and Han [4] designed a type system for a variant of *CPS* conversion based on the Krivine-style evaluation mechanism. They introduced two data types together with a set of dedicated type equations, which could be seen as a special form of generalized algebraic data type. Second, the type system by Choi and Ohori [2] can directly express the shape of spine stacks in typing judgments. The two structural typing rules (Clo) and (Install) are able to capture where a closure is created and where it is opened to execute the code inside. Their type system can also provide a potential capability for optimization such as avoiding unnecessary dynamic argument checks, though this is not in the scope of this paper.

Xi [5] proposed the notion of GADTs to show some applications such as a representation of higher-order abstract syntax (HOAS). Guillemette and Monnier [14] used this for the verification of a type-preserving compiler.

Pottier and Gauthier [15] used GADTs to formulate a polymorphic typed *defunctionalization*, a global program transformation that turns a higher-order functional program into a first-order one. The transformation assigns unique tags to all functions in a program, and it employs an *apply* function identifying each function by its tag to do an appropriate function application. They used GADTs for typing the *apply* function, which requires typing some non-uniform structure due to the collection of all the functions. The use of GADTs for the purpose is similar to the idea of this paper, but they did not consider typing Krivine-style evaluation mechanism nor typing abstract machine instructions.

3. A Simply Typed ZINC Machine

This section defines a ZINC machine by an operational semantics. After defining a simple type system for the machine instruction set, we show the type soundness of the ZINC machine.

3.1 An Operational Semantics of ZINC Machine

The set of instructions (ranged over by I) in our ZINC machine is given by the following syntax.

$$I ::= \mathsf{Return} \mid \mathsf{Grab} \mid \mathsf{Push} \mid \mathsf{Access}(i) \mid \mathsf{Reduce}(\mathsf{C}) \mid \mathsf{Int}(i) \mid \mathsf{Add}$$

Values (v) are either a constant c or a closure $cls(E, C)$. The cons operator ($\cdot$) is used for construction of a non-empty sequence. The empty sequence is denoted by $\emptyset$. Environments, local stacks, and spine stacks are denoted by E, L and S, all of which are a sequence of values. Dump

$$\begin{array}{ll}
(E\{i:v\},\ L,\ \mathsf{Access}(i)\cdot C,\ S,\ D) & \longrightarrow (E\{i:v\},\ v\cdot L,\ C,\ S,\ D) \\
(E,\ \emptyset,\ \mathsf{Grab}\cdot C,\ v\cdot S,\ D) & \longrightarrow (v\cdot E,\ \emptyset,\ C,\ S,\ D) \\
(E,\ \emptyset, \mathsf{Grab}\cdot C, \emptyset, (E_0, L_0, C_0, S_0)\cdot D) & \longrightarrow (E_0,\ v\cdot L_0,\ C_0,\ S_0,\ D) \\
 & \quad \text{where } v = cls(E, \mathsf{Grab}\cdot C) \\
(E,\ \emptyset,\ \mathsf{Grab}\cdot C,\ \emptyset,\ \emptyset) & \longrightarrow cls(E, \mathsf{Grab}\cdot C) \\
(E,\ v\cdot L,\ \mathsf{Push}\cdot C,\ S,\ D) & \longrightarrow (E,\ L,\ C,\ v\cdot S,\ D) \\
(E,\ cls(E_0, C_0)\cdot L,\ \mathsf{Return},\ v\cdot S,\ D) & \longrightarrow (E_0,\ \emptyset,\ C_0,\ v\cdot S,\ D) \\
(E, v\cdot L, \mathsf{Return}, \emptyset, (E_0, L_0, C_0, S_0)\cdot D) & \longrightarrow (E_0,\ v\cdot L_0,\ C_0,\ S_0,\ D) \\
(E,\ v\cdot L,\ \mathsf{Return},\ \emptyset,\ \emptyset) & \longrightarrow v \\
(E,\ L,\ \mathsf{Reduce}(C_0)\cdot C,\ S,\ D) & \longrightarrow (E,\ \emptyset,\ C_0,\ \emptyset,\ (E, L, C, S)\cdot D) \\
(E,\ L,\ \mathsf{Int}(n)\cdot C,\ S,\ D) & \longrightarrow (E,\ n\cdot L,\ C,\ S,\ D) \\
(E,\ n_1\cdot n_2\cdot L,\ \mathsf{Add}\cdot C,\ S,\ D) & \longrightarrow (E,\ n_1+n_2\cdot L,\ C,\ S,\ D)
\end{array}$$

Fig. 1 An operational semantics for ZINC machine.

$$\begin{array}{rl}
\text{(Grab)} & \dfrac{\tau\cdot\Gamma\,|\,\emptyset \rhd C : \tau'}{\Gamma\,|\,\emptyset \rhd \mathsf{Grab}\cdot C : \tau\to\tau'} \\
\text{(Return)} & \Gamma\,|\,\tau\cdot\Pi \rhd \mathsf{Return} : \tau \\
\text{(Push)} & \dfrac{\Gamma\,|\,\Pi \rhd C : \tau\to\tau'}{\Gamma\,|\,\tau\cdot\Pi \rhd \mathsf{Push}\cdot C : \tau'} \\
\text{(Int)} & \dfrac{\Gamma\,|\,int\cdot\Pi \rhd C : \tau}{\Gamma\,|\,\Pi \rhd \mathsf{Int}(n)\cdot C : \tau} \\
\text{(Access)} & \dfrac{\Gamma\,|\,\tau\cdot\Pi \rhd C : \tau'}{\Gamma\{i=\tau\}\,|\,\Pi \rhd \mathsf{Access}(i)\cdot C : \tau'} \\
\text{(Add)} & \dfrac{\Gamma\,|\,int\cdot\Pi \rhd C : \tau}{\Gamma\,|\,int\cdot int\cdot\Pi \rhd \mathsf{Add}\cdot C : \tau} \\
\text{(Reduce)} & \dfrac{\Gamma\,|\,\emptyset \rhd C : \tau \quad \Gamma\,|\,\tau\cdot\Pi \rhd C' : \tau'}{\Gamma\,|\,\Pi \rhd \mathsf{Reduce}(C)\cdot C' : \tau'}
\end{array}$$

Fig. 2 Typing rules for instructions.

stacks are denoted by D, which is a sequence of quadruples (E, L, C, S). Codes are denoted by C which is a sequence of machine instructions. $E\{i : v\}$ indicates v as ith value in E.

A ZINC machine [2] is defined by a set of states and state transition rules. A machine state is of the form (E, L, C, S, D). Figure 1 describes an operational semantics for the ZINC machine.

We explain how the ZINC machine evaluates higher-order functions by presenting the simply typed lambda calculus and its standard compilation into the machine instructions. The syntax of lambda terms in the de Bruijn notation is as this.

$$M ::= i\,|\,\lambda.M\,|\,M\ M\,|\,c\,|\,M + M$$

The compilation of a lambda term into a sequence of ZINC machine instructions is defined as this.

$$\begin{array}{l}
[\![i]\!] = \mathsf{Access}(i)\cdot\mathsf{Return} \\
[\![n]\!] = \mathsf{Int}(n)\cdot\mathsf{Return} \\
[\![\lambda.M]\!] = \mathsf{Grab}\cdot[\![M]\!] \\
[\![M_1\ M_2]\!] = \mathsf{Reduce}\ [\![M_2]\!]\cdot\mathsf{Push}\cdot[\![M_1]\!] \\
[\![M_1 + M_2]\!] = \mathsf{Reduce}\ [\![M_1]\!]\ \cdot\mathsf{Reduce}\ [\![M_2]\!]\ \cdot\mathsf{Add}\cdot\mathsf{Return}
\end{array}$$

For example, $[\![(\lambda x.\lambda y.x)\ 1]\!]$, using the named variable notation for convenience, can be compiled into $\mathsf{Int(1)}\cdot\mathsf{Push}\cdot\mathsf{Grab}\cdot\mathsf{Grab}\cdot\mathsf{Access(1)}\cdot\mathsf{Return}$ where $\mathsf{Reduce}\ [\![1]\!]\ \cdot C$ is replaced with $\mathsf{Int(1)}\cdot C$. When we run the compiled code, Int(1) puts *1* on the local stack, and Push moves the integer to the spine stack. The first Grab will then find on the spine stack the integer to pop into the environment. However, the second Grab will see the empty spine stack since the execution started with it. At this moment, the instruction will create a closure with the code $\mathsf{Grab}\cdot\mathsf{Access(1)}\cdot\mathsf{Return}$ and the environment with the integer value in the *1*st entry. After that, the instruction will try to go back to the topmost saved context on the dump stack. Suppose M is the lambda term stated above. In $[\![(\lambda z.z\ 2)\ M]\!]$, the closure obtained from running $[\![M]\!]$ becomes bound to the *0*th entry of the environment associated with z. $[\![z\ 2]\!]$ can be compiled into $\mathsf{Int(2)}\cdot\mathsf{Push}\cdot\mathsf{Access(0)}\cdot\mathsf{Return}$ similarly. When we run $[\![z]\!]$, Access(0) will retrieve the closure onto the local stack. After Return finds the integer value 2 on the spine stack, it will unpack the closure to apply its code to the integer under its environment.

3.2 A Type System for the ZINC Machine

Types (ranged over by τ) are defined by either *int* or $\tau\to\tau$. Two typing contexts, environment type and local stack type (ranged over by Γ and Π), are defined as a sequence of types.

For notation we define $\Delta\to\tau$ as an abbreviation of $\tau_1\to\cdots\to\tau_n\to\tau$ when $\Delta = \tau_1\cdot\cdots\cdot\tau_n$. That is, we have $\Delta_1\to\Delta_2\to\tau \equiv \Delta_1\cdot\Delta_2\to\tau$ and $\tau\equiv\emptyset\to\tau$.

A typing judgment is of the form $\Gamma\,|\,\Pi \rhd C : \Delta\to\tau$. The typing judgment can be read as a code C is executed under an environment of type Γ and a local stack of type Π. The execution takes arguments specified by Δ, and it returns a value specified by τ. The topmost saved context on the dump stack will take the return value to continue.

It is important to understand that the code type $\Delta\to\tau$ in the typing judgment does not mean to specify the type of a value. Rather, it intends to specify the shape of a spine stack (by Δ) and an associated dump stack (by τ).

The set of typing rules is given in Fig. 2. By (Push), the potential spine stack grows from one specified by τ' to another by $\tau\to\tau'$ with the underlying dump stack unchanged. By (Grab), the spine stack shrinks in an opposite way. (Return) associates the shape of the return value with the shape of the spine and dump stack by having the return value type the same as the code type. For example, if the return value is of type *Int*, then the code type lets the spine stack empty; $\emptyset$ is the only Δ satisfying $Int\equiv\Delta\to Int$. If the return value type is $\tau_1\to\tau_2$, then we can find the code type equivalent to $\tau'_1\cdot\Delta'\to\tau'_2$ for some τ'_1, τ'_2, and Δ'. (Reduce) enforces the type of the value computed by the code to reduce to be the same as the type of what the subsequent code expects. (Int), (Add), and (Access) closely simulate the corresponding operational semantics.

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 in Fig. 3.

Using these definitions, we can establish the following theorem.

Theorem 1 (Soundness): If $\Gamma\,|\,\Pi \rhd C : \Delta\to\tau$, $\models E : \Gamma$, $\models L : \Pi$, $\models S : \Delta$, $\models D : \tau\Rightarrow\tau_0$, and $(E, L, C, S, D)\longrightarrow^* v$, then $\models v : \tau_0$.

Value Typing :

- $\models n : int$
- $\models cls(E, \mathsf{Grab} \cdot C) : \tau \to \tau'$ if there is some Γ such that $\models E : \Gamma$ and $\Gamma \,|\, \emptyset \rhd \mathsf{Grab} \cdot C : \tau \to \tau'$.

Environment Typing : (*Local stack typing* $\models L : \Pi$ *and Spine stack typing* $\models S : \Delta$ *are similarly defined*)

- $\models \emptyset : \emptyset$
- $\models v \cdot E : \tau \cdot \Gamma$ if $\models v : \tau$ and $\models E : \Gamma$

Dump Stack Typing :

- $\models \emptyset : \tau \Rightarrow \tau$ for any τ
- $\models (E, L, C, S) \cdot D : \tau \Rightarrow \tau_0$ if $\models E : \Gamma$, $\models L : \Pi$, $\models S : \Delta$, $\Gamma \,|\, \tau \cdot \Pi \rhd C : \Delta \to \tau'$, and $\models D : \tau' \Rightarrow \tau_0$

Fig. 3 Typing rules for values, environments, local stacks, spine stacks, and dump stacks.

The soundness theorem ensures that Return sees a closure on the local stack whenever it finds an argument on the spine stack. Otherwise the machine will go wrong due to the attempt to apply some integer to the argument.

4. Typing ZINC Machine Using Generalized Algebraic Data Types

We introduce a calculus of System-F with GADTs to represent ZINC instructions by lambda terms in the calculus. This will give an explicit type to each instruction, contrary to the previous type system in Fig. 2 that only relates each instruction with a simple type where the type of the instruction is implicit.

The notion of GADTs plays two important roles in typing the ZINC machine. First, it allows to give a type to the term representation of a spine stack, which forms a list of heterogeneous values, to assure that the spine stack is in an appropriate state. This cannot be achieved by usual algebraic data types. Second, it allows the non-uniform polymorphism of Return instruction by making usable only the data constructors with unifiable parameter types. Recall that *GList Int* in Sect. 1 is only inhabited by *Nil*.

4.1 A Calculus of System-F with GADTs

Generalized algebraic data types are a simple extension of ML data types [5], [15]. Suppose T is a data type constructor and c is one of data constructors associated with T. The type of c must be of the form $c : \forall\alpha_1, \ldots, \alpha_l.(\tau'_1, \ldots, \tau'_m) \to T(\alpha_1, \ldots, \alpha_l)$ under ML data types. Generalized algebraic data types, however, allow a type constructor T to be applied to a vector of arbitrary types $\tau_1, \ldots, \tau_n$ as $c : \forall\alpha_1, \ldots, \alpha_l.(\tau'_1, \ldots, \tau'_m) \to T(\tau_1, \ldots, \tau_n)$ where some of $\alpha_1, \ldots, \alpha_l$ might not even appear in $T(\tau_1, \ldots, \tau_n)$. We will show some examples of generalized algebraic data types in the next section.

Under the generalized algebraic data types, each pattern matching reveals extra static type information, or *guards*, to act as a constraint within the scope of the matched case. Suppose one matches a value of type $T(\sigma_1, \ldots, \sigma_n)$ against a pattern $c(\alpha_1, \ldots, \alpha_l, x_1, \ldots, x_m)$ of type $T(\tau_1, \ldots, \tau_n)$. The matched case yields a set of equations $\tau_i = \sigma_i$ for $1 \le i \le n$. This is a (unification) constraint C, which is a conjunction of type equations of the form $\tau = \sigma$. An *assignment* is a total mapping from type variables to ground types. An assignment satisfies an equation if and only if it maps both of its members to the same ground type; an assignment satisfies a conjunction of equations if and only if it satisfies all of its members. A constraint C *entails* a constraint C' (which we write $C \Vdash C'$) if and only if every assignment that satisfies C satisfies C' [5], [15].

$$
\begin{array}{rl}
\text{(G-Var)} & C \,|\, \Gamma\{x : \sigma\} \rhd x : \sigma \\
\text{(G-App)} & \dfrac{C \,|\, \Gamma \rhd e_2 : \sigma_2 \quad C \,|\, \Gamma \rhd e_1 : \sigma_2 \to \sigma_1}{C \,|\, \Gamma \rhd e_1\ e_2 : \sigma_1} \\
\text{(G-Abs)} & \dfrac{C \,|\, \Gamma\{x : \sigma_1\} \rhd e : \sigma_2}{C \,|\, \Gamma \rhd \lambda(x : \sigma_1).e : \sigma_2} \\
\text{(G-Int)} & C \,|\, \Gamma \rhd n : int \\
\text{(G-TApp)} & \dfrac{C \,|\, \Gamma \rhd e : \forall\alpha.\sigma}{C \,|\, \Gamma \rhd e\ \sigma' : \sigma[\sigma'/\alpha]} \\
\text{(G-Add)} & \dfrac{C \,|\, \Gamma \rhd e_1 : int \quad C \,|\, \Gamma \rhd e_2 : int}{C \,|\, \Gamma \rhd e_1 + e_2 : int} \\
\text{(G-TAbs)} & \dfrac{C \,|\, \Gamma \rhd e : \sigma \quad \alpha \notin ftv(C) \cup ftv(\Gamma)}{C \,|\, \Gamma \rhd \Lambda\alpha.e : \forall\alpha.\sigma} \\
\text{(G-Con)} & \dfrac{\begin{array}{c} c\ :\ \forall\alpha_1, ..., \alpha_l.(\sigma'_1, ..., \sigma'_m) \to T(\sigma''_1, ..., \sigma''_n) \\ C \,|\, \Gamma \rhd e_i : \sigma'_i\ \theta \\ \theta = [\sigma_1/\alpha_1, ..., \sigma_l/\alpha_l] \end{array}}{C \,|\, \Gamma \rhd c(\sigma_1, ..., \sigma_l, e_1, ..., e_m) : T(\sigma''_1, ..., \sigma''_n)\ \theta} \\
\text{(G-Case)} & \dfrac{C \,|\, \Gamma \rhd e : \sigma' \quad C \,|\, \Gamma \rhd alt_i : \sigma' \to \sigma}{C \,|\, \Gamma \rhd case\ e\ of\ \{alt_1, ..., alt_n\} : \sigma} \\
\text{(G-Alt)} & \dfrac{\begin{array}{c} c\ :\ \forall\alpha_1, ..., \alpha_l.(\sigma''_1, ..., \sigma''_m) \to T(\sigma'_1, ..., \sigma'_n) \\ \alpha_i \notin ftv(C) \cup ftv(\Gamma) \cup \bigcup_i ftv(\sigma_i) \cup ftv(\sigma) \\ \Gamma' = \Gamma \cup \{x_1 : \sigma''_1, ..., x_m : \sigma''_m\} \\ C \wedge \sigma'_1 = \sigma_1 \wedge ... \wedge \sigma'_n = \sigma_n \,|\, \Gamma' \rhd e : \sigma \end{array}}{C \,|\, \Gamma \rhd c(\alpha_1, ..., \alpha_l, x_1, ..., x_m) \to e : T(\sigma_1, ..., \sigma_n) \to \sigma} \\
\text{(G-TyEq)} & \dfrac{C \,|\, \Gamma \rhd e : \sigma' \quad C \Vdash \sigma' = \sigma}{C \,|\, \Gamma \rhd e : \sigma}
\end{array}
$$

Fig. 4 A calculus of System-F with GADTs.

The syntax of a calculus of System F extended with GADTs [5], [15] is given as:

$$
\begin{array}{rcl}
\sigma & ::= & \alpha \,|\, int \,|\, \sigma \to \sigma \,|\, \forall\alpha.\sigma \,|\, T(\sigma_1, \ldots, \sigma_l) \\
e & ::= & x \,|\, \lambda(x : \sigma).e \,|\, e\ e \,|\, \Lambda\alpha.e \,|\, e\ \sigma \,|\, n \,|\, e + e \\
 & | & c(\sigma_1, \ldots, \sigma_n, e_1, \ldots, e_m) \,|\, case\ e\ of\ \{alt_1, \ldots, alt_n\} \\
alt & ::= & c(\alpha_1, \ldots, \alpha_n, x_1, \ldots, x_m) \to e
\end{array}
$$

We use the notation $e[\sigma_1, \cdots, \sigma_n]$ to denote a multiple type application $e\ \sigma_1\ \cdots\ \sigma_n$. We regard a tuple type $(\sigma_1, \cdots, \sigma_n)$ as a special data type.

Figure 4 defines a set of typing rules for the extended System F where a typing judgment of the form $C \,|\, \Gamma \rhd e : \sigma$ is read as an expression e is of type σ under the type environment Γ when C has an assignment. Note that Γ is a mapping of variables to types.

4.2 A Typed Representation of ZINC Machine

The Representation of Machine Components: We represent values, spine stacks, and dump stacks by three gen-

Value type : $V(\sigma)$

- $Con : int \to V(int)$
- $Cls : \forall\alpha,\beta,\epsilon.\ (\sigma_{code},\epsilon) \to V(\alpha \to \beta)$
  - $\sigma_{code} = \forall\delta.\epsilon \to unit \to SpineStk(\alpha \to \beta,\delta) \to \delta \to unit$

Spine stack type : $SpineStk(\sigma,\sigma')$

- $Arg : \forall\alpha,\beta,\delta.\ (V(\alpha), SpineStk(\beta,\delta)) \to SpineStk(\alpha \to \beta,\delta)$
- $Nil : \forall\alpha,\beta,\delta.\ SpineStk(\alpha, DumpStk(\alpha,\beta,\delta))$

Dump stack type : $DumpStk(\sigma,\sigma',\sigma'')$

- $Ret : \forall\alpha,\beta,\epsilon,\iota,\delta.\ (\sigma_{code},\epsilon,\iota, SpineStk(\beta,\delta),\delta) \to DumpStk(\alpha,\beta,\delta)$
  - $\sigma_{code} = \epsilon \to (V(\alpha),\iota) \to SpineStk(\beta,\delta) \to \delta \to unit$
- $Nil : \forall\alpha.\ DumpStk(\alpha, unit, unit)$

Fig. 5 GADTs for values, spine stacks, and dump stacks.

eralized algebraic data types: $V(\sigma)$, $SpineStk(\sigma_1,\sigma_2)$, and $DumpStk(\sigma_1,\sigma_2,\sigma_3)$ as shown in Fig. 5.

The value type $V(\sigma)$ specifies the term representation for a value of type σ. For example, $Con(1)$ is of type $V(int)$. Suppose I_c is of type $\forall\delta.\epsilon \to unit \to SpineStk(int \to int,\delta) \to \delta \to unit$. Then a closure $Cls(I_c,())$ with the empty environment is of type $V(int \to int)$.

The declaration of Cls in the generalized algebraic data type hides the type variable ϵ from $V(\alpha \to \beta)$. GADTs thus express the existential type for closures, as described in [5], [15].

Note that we can distinguish constants or closures of type $V(\sigma)$ from each other by having an appropriate form of the type parameter σ, which is a feature of GADTs [5], [15]. A value of type $V(int)$ must be a constant while a value of type $V(\sigma \to \sigma')$ must be a closure. A value of type $V(\alpha)$ can be either a constant or a closure, depending on a type substitution for the type variable.

We represent environments and local stacks simply by the terms of nested pairs ended with the unit value. For example, an environment may look like $(V_1,(V_2,()))$ for some values V_1 and V_2.

The spine stack type $SpineStk(\Delta \to \sigma,\sigma_{dump})$ specifies the term representation for a spine stack that consists of a sequence of arguments specified by Δ. For example, $SpineStk(int \to int,\sigma_{dump})$ can inhabit both Nil and $Arg(Con(1),Nil)$ where Δ is $int \cdot \emptyset$.

A GADT allows to enable only the part of its data constructors. For example, we can enforce the form of spine stacks to be always Nil by assigning them a spine stack type $SpineStk(int,\sigma')$. The type of values in the form of $Arg(\cdots)$ cannot be unified with the spine stack type.

The spine stack type $SpineStk(\Delta \to \sigma,\sigma_{dump})$ also specifies the shape of dump stacks by σ_{dump}. We intend the spine stack type to specify a return value of type σ that the code in the topmost saved context on the dump stack will take to continue execution.

The dump stack type $DumpStk(\sigma,\Delta \to \sigma',\sigma_{dump})$ specifies the term representation for a dump stack, $Ret(e_I,e_E,e_L,e_S,e_D)$. Here, e_I is a code of type

$$\begin{aligned}
&Grab\ :\ \forall\alpha.\forall\beta.\forall\epsilon.\forall\delta.\ (\forall\delta.(V\ \alpha,\epsilon) \to unit \to SpineStk(\beta,\delta) \to \delta \to unit)\\
&\qquad\qquad \to \epsilon \to unit \to SpineStk(\alpha \to \beta,\ \delta) \to \delta \to unit\\
&Grab = \lambda I.\lambda E.\lambda L.\lambda S.\lambda D.\ case\ S\ of\\
&\quad Arg(V,S') \to I\ (V,E)\ L\ S'\ D\\
&\quad Nil \to case\ D\ of\\
&\qquad Ret(I_k,E_k,L_k,S_k,D_k) \to I_k\ E_k\ (Cls(Grab\ I,E),L_k)\ S_k\ D_k\\
&\qquad Nil \to Halt\ (Cls(Grab\ I,E))
\end{aligned}$$

$$\begin{aligned}
&Return\ :\ \forall\alpha.\forall\epsilon.\forall\iota.\forall\delta.\ \epsilon \to (V\ \alpha,\iota) \to SpineStk(\alpha,\delta) \to \delta \to unit\\
&Return = \lambda E.\lambda L.\lambda S.\lambda D.\ case\ L\ of\ (V,L') \to\ case\ S\ of\\
&\quad Arg(V',S') \to (\ case\ V\ of\ \ Cls(I_c,E_c) \to I_c\ E_c\ ()\ S\ \ D\ )\\
&\quad Nil \to case\ D\ of\\
&\qquad Ret(I_k,E_k,L_k,S_k,D_k) \to I_k\ E_k\ (V,L_k)\ S_k\ D_k\\
&\qquad Nil \to Halt\ V
\end{aligned}$$

$$\begin{aligned}
&Push\ :\ \forall\alpha.\forall\beta.\forall\epsilon.\forall\iota.\forall\delta.\ (\forall\delta.\epsilon \to \iota \to SpineStk(\alpha \to \beta,\delta) \to \delta \to unit)\\
&\qquad\qquad \to \epsilon \to (V\ \alpha,\iota) \to SpineStk(\beta,\delta) \to \delta \to unit\\
&Push = \lambda I.\lambda E.\lambda L.\lambda S.\lambda D.\ case\ L\ of\ (V,L') \to I\ E\ L'\ Arg(V,S)\ D
\end{aligned}$$

$$\begin{aligned}
&Reduce\ :\ \forall\alpha.\forall\beta.\forall\epsilon.\forall\iota.\forall\delta.\\
&\qquad\qquad (\forall\delta.\epsilon \to unit \to SpineStk(\alpha, DumpStk(\alpha,\beta,\delta))\\
&\qquad\qquad\qquad \to DumpStk(\alpha,\beta,\delta) \to unit)\\
&\qquad\qquad \to (\forall\delta.\epsilon \to (V\alpha,\iota) \to SpineStk(\beta,\delta) \to \delta \to unit)\\
&\qquad\qquad \to \epsilon \to \iota \to SpineStk(\beta,\delta) \to \delta \to unit\\
&Reduce = \lambda I.\lambda I_k.\lambda E.\lambda L.\lambda S.\lambda D.\ I\ E\ ()\ Nil\ Ret(I_k,E,L,S,D)
\end{aligned}$$

$$\begin{aligned}
&Access(i)\ :\ \forall\alpha_0.\cdots\forall\alpha_i.\forall\beta.\forall\epsilon.\forall\iota.\forall\delta.\\
&\qquad (\forall\delta.(\alpha_0,(\alpha_1,(\cdots,(\alpha_i,\epsilon)))) \to (\alpha_i,\iota) \to SpineStk(\beta,\delta) \to \delta \to unit)\\
&\qquad\quad \to (\alpha_0,(\alpha_1,(\cdots,(\alpha_i,\epsilon)))) \to \iota \to SpineStk(\beta,\delta) \to \delta \to unit\\
&Access(i) = \lambda I.\lambda E.\lambda L.\lambda S.\lambda D.\\
&\quad case\ E\ of\ (V_0,E_0) \to\ \cdots\ case\ E_{i-2}\ of\ (V_{i-1},E_{i-1}) \to\\
&\qquad case\ E_{i-1}\ of\ (V_i,E_i) \to I\ E\ (V_i,L)\ S\ \ D
\end{aligned}$$

$$\begin{aligned}
&Int\ :\ int \to \forall\beta.\forall\epsilon.\forall\iota.\forall\delta.(\forall\delta.\epsilon \to (V\ int,\iota) \to SpineStk(\beta,\delta) \to \delta \to unit)\\
&\qquad\qquad \to \epsilon \to \iota \to SpineStk(\beta,\delta) \to \delta \to unit\\
&Int = \lambda n.\lambda I.\lambda E.\lambda L.\lambda S.\lambda D.\ I\ E\ (Con(n),L)\ S\ \ D
\end{aligned}$$

$$\begin{aligned}
&Add\ :\ \forall\beta.\forall\epsilon.\forall\iota.\forall\delta.\ (\forall\delta.\epsilon \to (V\ int,\iota) \to SpineStk(\beta,\delta) \to \delta \to unit)\\
&\qquad\qquad \to \epsilon \to (V\ int,(V\ int,\iota)) \to SpineStk(\beta,\delta) \to \delta \to unit\\
&Add = \lambda I.\lambda E.\lambda L.\lambda S.\lambda D.\\
&\quad case\ L\ of\ (V_0,L_0) \to case\ L_0\ of\ (V_1,L_1) \to case\ V_0\ of\ Con(n_0) \to\\
&\quad case\ V_1\ of\ Con(n_1) \to I\ E\ (Con(n_0+n_1),L)\ S\ \ D
\end{aligned}$$

$$\begin{aligned}
&Halt\ :\ \forall\alpha.V\ \alpha \to unit\\
&Halt = \lambda V.\ ()
\end{aligned}$$

Fig. 6 Untyped lambda terms and their types for ZINC instructions.

$$\epsilon \to (V(\sigma),\iota) \to SpineStk(\Delta \to \sigma',\sigma_{dump}) \to \sigma_{dump}$$

that takes a return value of type $V(\sigma)$ under an environment e_E of type ϵ and a local stack e_L of type ι. The code continues to run consuming as many argument values from the spine stack e_S as the length of Δ, and it produces another value of type $V(\sigma')$. This value will be passed to the next saved context on the smaller dump stack e_D of type σ_{dump}.

The Representation of ZINC Instructions: We represent ZINC instructions by lambda terms, which is shown in Fig. 6[†]. We regard each ZINC instruction as a function of an environment, a local stack, a spine stack, and a dump stack. We obtained these lambda terms by combining the associated state transition rules in Fig. 1. For example, Grab

[†]For simplicity, Fig. 6 shows untyped lambda terms and their types for ZINC instructions. We use $f : \sigma$ to give a type to a term. The typed lambda terms are available in Fig. A· 1 of the Appendix.

has three state transition rules in the operational semantics. All the rules share the same form of environment and local stack, but each of them poses a different form for the spine stack and the dump stack. We, therefore, introduce one case analysis for the spine stack and the other for the dump stack, which leads to the lambda term *Grab* in Fig. 6.

Return can be made up similarly. For each of the other ZINC instructions, it is straightforward to define a lambda term by directly encoding the relevant state transition rule.

Recall that the typing rule for each ZINC instruction I in Fig. 2 looks like:

$$(\mathrm{I})\ \frac{\Gamma' \,|\, \Pi' \rhd C : \tau'}{\Gamma \,|\, \Pi \rhd I \cdot C : \tau}$$

which can be read as a transformation of a machine state by (Γ, Π, τ) into another by (Γ', Π', τ').

Suppose e is a lambda term obtained from the state transition rules for I in Fig. 1. Then the type of e will be

$$\forall\delta.\ (\forall\delta'.\, \sigma_{\Gamma'} \to \sigma_{\Pi'} \to \mathit{SpineStk}(\tau', \delta') \to \delta' \to \mathit{unit}) \to \\ \sigma_\Gamma \to \sigma_\Pi \to \mathit{SpineStk}(\tau, \delta) \to \delta \to \mathit{unit}$$

where σ_Γ, $\sigma_{\Gamma'}$, σ_Π, and $\sigma_{\Pi'}$ are the types of lambda terms for environments of type Γ and Γ' and local stacks of type Π and Π', respectively.

The terms *Push* and *Grab* manipulate a spine stack to make it grow and shrink, respectively. *Push* places a value V of type $V(\alpha)$ on top of the spine stack S of type $\mathit{SpineStk}(\beta, \delta)$, resulting in another spine stack $\mathit{Arg}(V, S)$ of type $\mathit{SpineStk}(\alpha \to \beta, \delta)$.

Grab performs the reverse by analyzing the spine stack S of type $\mathit{SpineStk}(\alpha \to \beta, \delta)$ to find either $\mathit{Arg}(V, S')$ or Nil. In case the spine stack is Nil (or $\mathit{Nil}(\alpha_1, \beta_1, \delta_1)$ with the complete type annotation that can be found in Fig. A· 1) giving rise to the type constraints $\{\alpha \to \beta = \alpha_1,\ \delta = \mathit{DumpStk}(\alpha_1, \beta_1, \delta_1)\}$ for some type variables α_1, β_1, and δ_1. Then we do a case analysis on the dump stack D of type δ to restore the topmost saved context, which could not be done without the constraint on δ.

Return is more complicated. The type of the instruction may be written as

$$\forall\delta.\epsilon \to (V\ \alpha, \iota) \to \mathit{SpineStk}(\alpha, \delta) \to \delta \to \mathit{unit}$$

The value type and the spine stack type have the same parameter type α. By this, the notion of GADTs enforces the invariant of the non-uniform behavior of the instruction. When α is substituted with *int*, only the case alternative with *Nil* is allowed. When α is substituted with a function type, both case alternatives are allowed.

When the return value is an integer, the instruction is supposed to pass it back to the topmost saved context. In this case, the type variable α in $V\ \alpha$ becomes substituted with *int*. This implies that the spine stack type becomes $\mathit{SpineStk}(\mathit{int}, \delta)$, which statically disallows to use the case alternative with $\mathit{Arg}(\cdots)$ in *Return* due to the parameter type not unifiable with *int*.

When the return value is a closure, both case alternatives are allowed for use. When the spine stack S specified by $\mathit{SpineStk}(\alpha, \delta)$ is $\mathit{Arg}(V', S')$ (or $\mathit{Arg}(\alpha_1, \beta_1, \delta_1, V', S')$), this yields the type constraints $\{\alpha = \alpha_1 \to \beta_1,\ \delta = \delta_1\}$. The return value V on the local stack is declared to be of type $V(\alpha)$. Under the type constraints, V is of type $V(\alpha_1 \to \beta_1)$, and is a closure denoted by $\mathit{Cls}(I_c, E_c)$ for some term variables I_c and E_c. Then we can safely apply the code I_c of the closure to the argument V', as in "$\mathit{case}\ V\ \mathit{of}\ \mathit{Cls}(I_c, E_c) \to I_c\ E_c\ ()\ S\ D$." Note that the V in the case expression can never be of the form $\mathit{Con}(\cdots)$, which will lead the machine to get stuck, because every case analysis on S against $\mathit{Arg}(V', S')$ yields the type constraint to make α be unifiable with a function type.

The term $(\mathit{Reduce}\ e_I\ e_{I_k})$ is intended to reduce e_I producing a value, with which we then continue to reduce e_{I_k}. For the reduction of e_I, we set up a local stack and a spine stack to be empty, saving the current context with e_{I_k} on the dump stack, as this: $(\mathit{Reduce}\ e_I\ e_{I_k})\ e_E\ e_L\ e_S\ e_D \to^* e_I\ e_E\ ()\ \mathit{Nil}\ \mathit{Ret}(e_{I_k}, e_E, e_L, e_S, e_D)$ where Nil is of type $\mathit{SpineStk}(\alpha, \mathit{DumpStk}(\alpha, \beta, \delta))$, and $\mathit{Ret}(e_{I_k}, e_E, e_L, e_S, e_D)$ is of type $\mathit{DumpStk}(\alpha, \beta, \delta)$ for some types α, β, and δ. The spine and dump stacks (e_S and e_D) in the saved context are of type $\mathit{SpineStk}(\beta, \delta)$ and δ, respectively.

The definitions of $\mathit{Access}(i)$, *Int*, and *Add* are straightforward. Note that we assume to have a separate definition of $\mathit{Access}(i)$ for each i. *Halt* is an auxiliary term to end the reduction in the calculus, as the ZINC machine stops when the spine and dump stacks are both empty.

Theorem 2 (Well-Typed Terms): All the lambda terms in Fig. A· 1 are well-typed in the extended System F.

In fact, we have implemented all the GADT declarations and the lambda terms in Figs. 5 and 6 using Haskell to ensure the correctness of the above theorem. We will discuss this later.

Using the definitions of Fig. A· 1, we can build a lambda term for a given ZINC code. The lambda term will be well-typed if the ZINC code is so, which will be shown in the next section.

4.3 A Typed Compilation of the ZINC Instructions

We first define type correspondence relations for values, environments, and local stacks. A value type correspondence is denoted by $\sigma \rightsquigarrow V(\sigma)$. An environment type correspondence is either $\emptyset \rightsquigarrow \mathit{unit}$ or $\tau \cdot \Gamma \rightsquigarrow (\sigma, \sigma')$ if $\tau \rightsquigarrow \sigma$ and $\Gamma \rightsquigarrow \sigma'$. A local stack type correspondence is similarly defined.

We write a compilation judgment as $\Gamma \,|\, \Pi \rhd C : \tau \rightsquigarrow e$ where C is compiled to e under an environment type Γ and a local stack type Π by the algorithm given in Fig. 7. The compilation algorithm preserves typing as follows.

Theorem 3 (Typed Compilation of the ZINC Instructions): Suppose we have $\Gamma \rightsquigarrow \sigma_e$, and $\Pi \rightsquigarrow \sigma_l$. If $\Gamma \,|\, \Pi \rhd C : \tau \rightsquigarrow e$ then $\emptyset \,|\, \emptyset \rhd e : \forall\delta.\sigma_e \to \sigma_l \to \mathit{SpineStk}(\tau, \delta) \to \delta \to \mathit{unit}$.

The theorem says that one can interpret a ZINC instruction as a lambda term whose arguments are the terms for the

$$\text{(C-Grab)}\quad \frac{\Gamma \leadsto \sigma_e \quad \tau \cdot \Gamma \,|\, \emptyset \rhd C : \tau' \leadsto e}{\Gamma \,|\, \emptyset \rhd \mathsf{Grab} \cdot C : \tau \to \tau' \leadsto \Lambda\delta.\mathit{Grab}\,[\tau, \tau', \sigma_e, \delta]\; e}$$

$$\text{(C-Reduce)}\quad \frac{\begin{array}{c}\Gamma \leadsto \sigma_e \;\; \Pi \leadsto \sigma_l \\ \Gamma \,|\, \emptyset \rhd C : \tau \leadsto e \;\; \Gamma \,|\, \tau \cdot \Pi \rhd C' : \tau' \leadsto e' \\ \mathit{TYS} = \tau, \tau', \sigma_e, \sigma_l, \delta\end{array}}{\Gamma \,|\, \Pi \rhd \mathsf{Reduce}(C) \cdot C' : \tau' \leadsto \Lambda\delta.\mathit{Reduce}\,[\mathit{TYS}]\; e\; e'}$$

$$\text{(C-Push)}\quad \frac{\Gamma \leadsto \sigma_e \;\; \Pi \leadsto \sigma_l \quad \Gamma \,|\, \Pi \rhd C : \tau \to \tau' \leadsto e}{\Gamma \,|\, \tau \cdot \Pi \rhd \mathsf{Push} \cdot C : \tau' \leadsto \Lambda\delta.\mathit{Push}\,[\tau,\ \tau',\ \sigma_e,\ \sigma_l,\ \delta]\; e}$$

$$\text{(C-Return)}\quad \frac{\Gamma \leadsto \sigma_e \;\; \Pi \leadsto \sigma_l}{\Gamma \,|\, \tau \cdot \Pi \rhd \mathsf{Return} : \tau \leadsto \Lambda\delta.\mathit{Return}\,[\tau, \sigma_e, \sigma_l, \delta]}$$

$$\text{(C-Access)}\quad \frac{\begin{array}{c}\tau_0 \leadsto \sigma_0 \;\ldots\; \tau_i \leadsto \sigma_i \;\; \Gamma' \leadsto \sigma_e \;\; \Pi \leadsto \sigma_l \\ \Gamma \,|\, \tau \cdot \Pi \rhd C : \tau' \leadsto e \\ \Gamma = \tau_0 \cdots \tau_i \cdot \Gamma' \quad \mathit{TYS} = \sigma_0, \ldots, \sigma_i, \tau', \sigma_e, \sigma_l, \delta\end{array}}{\Gamma\{i = \tau\} \,|\, \Pi \rhd \mathsf{Access}(i) \cdot C : \tau' \leadsto \Lambda\delta.\mathit{Access}(i)[\mathit{TYS}]\; e}$$

$$\text{(C-Int)}\quad \frac{\Gamma \leadsto \sigma_e \;\; \Pi \leadsto \sigma_l \quad \Gamma \,|\, \mathit{int} \cdot \Pi \rhd C : \tau \leadsto e}{\Gamma \,|\, \Pi \rhd \mathsf{Int}(n) \cdot C : \tau \leadsto \Lambda\delta.\mathit{Int}(n)\,[\tau, \sigma_e, \sigma_l, \delta]\; e}$$

$$\text{(C-Add)}\quad \frac{\Gamma \leadsto \sigma_e \;\; \Pi \leadsto \sigma_l \quad \Gamma \,|\, \mathit{int} \cdot \Pi \rhd C : \tau \leadsto e}{\Gamma \,|\, \mathit{int} \cdot \mathit{int} \cdot \Pi \rhd \mathsf{Add} \cdot C : \tau \leadsto \Lambda\delta.\mathit{Add}\,[\tau, \sigma_e, \sigma_l, \delta]\; e}$$

Fig. 7 Typed compilation of ZINC code.

$$\begin{array}{l}
\emptyset|\emptyset \rhd \mathsf{Int(1)} \cdot \mathsf{Push} \cdot \mathsf{Grab} \cdot \mathsf{Grab} \cdot \mathsf{Access(1)} \cdot \mathsf{Return}\ :\ \mathit{Int} \to \mathit{Int} \leadsto \\
\quad \Lambda\delta_1.\mathit{Int}\ 1\ [\mathit{Int} \to \mathit{Int}, \mathit{unit}, \mathit{unit}, \delta_1] \\
\quad (\Lambda\delta_2.\mathit{Push}\ [\mathit{Int}, \mathit{Int} \to \mathit{Int}, \mathit{unit}, \mathit{unit}, \delta_2] \\
\quad\; (\Lambda\delta_3.\mathit{Grab}\ [\mathit{Int}, \mathit{Int} \to \mathit{Int}, \mathit{unit}, \delta_3] \\
\quad\;\; (\Lambda\delta_4.\mathit{Grab}\ [\mathit{Int}, \mathit{Int}, (V(\mathit{Int}), \mathit{unit}), \delta_4] \\
\quad\;\;\; (\Lambda\delta_5.\mathit{Access}(1)\ [V(\mathit{Int}), V(\mathit{Int}), \mathit{Int}, \mathit{unit}, \mathit{unit}, \delta_5] \\
\quad\;\;\;\; (\Lambda\delta_6.\mathit{Return}\ [\mathit{Int}, (V(\mathit{Int}), (V(\mathit{Int}), \mathit{unit})), \mathit{unit}, \delta_6])))))
\end{array}$$

Fig. 8 A compilation example for the ZINC code $[\![(\lambda x.\lambda y.x)\ 1]\!]$.

four machine components. Note that the dump stack is polymorphically abstracted. This naturally reflects the fact that any saved context inside the dump stack does not affect the current computation of the ZINC machine.

Figure 8 shows a typed compilation example for the ZINC code discussed in Sect. 3.

4.4 Correctness of Compilation

We first define the correspondence between the machine components and the lambda term representations in the extended System-F calculus as in Fig. 9.

Under the correspondence relations, we now present the semantic correctness theorem as the following.

Theorem 4 (Semantic Correctness): Suppose

- $\Gamma \,|\, \Pi \rhd C : \Delta \to \tau \leadsto e$
- $\models E : \Gamma \leadsto e_E : \sigma_E$
- $\models L : \Pi \leadsto e_L : \sigma_L$
- $\tau, \sigma_D \models S : \Delta \leadsto e_S : \mathit{SpineStk}(\Delta \to \tau, \sigma_D)$
- $\models D : \tau \Rightarrow \tau_0 \leadsto e_D : \sigma_D$

If $(E, L, C, S, D) \longrightarrow^* v$ then $e[\sigma_D]\ e_E\ e_L\ e_S\ e_D \overset{*}{\longrightarrow}_\beta \mathit{Halt}[\tau](e_v)$ such that $\models v : \tau \leadsto e_v : V(\tau)$.

The theorem establishes the type soundness of the ZINC type system with respect to the operational semantics obtained by combining the compilation to the extended System-F followed by β reducing the compiled term.

Value correspondence :

- $\models n : \mathit{int} \leadsto \mathit{Con}(n) : V(\mathit{int})$
- $\models \mathit{cls}(E, C) : \tau \to \tau' \leadsto \mathit{Cls}[\tau, \tau', \sigma_E]\ (e_E, e_C) : V(\tau \to \tau')$
  - if there is some Γ such that $\models E : \Gamma \leadsto e_E : \sigma_E$ and $\Gamma \,|\, \emptyset \rhd C : \tau \to \tau' \leadsto e_C$

Environment correspondence : (*Local stack correspondence is similarly defined.*)

- $\models \emptyset : \emptyset \leadsto () : \mathit{unit}$
- $\models v \cdot E : \tau \cdot \Gamma \leadsto (e_v, e_E) : (\sigma_v, \sigma_E)$
  - if $\models v : \tau \leadsto e_v : \sigma_v$ and $\models E : \Gamma \leadsto e_E : \sigma_E$

Spine stack correspondence :

- $\tau, \sigma_D \models \emptyset : \emptyset \leadsto \mathit{Nil}(\tau, \tau', \sigma) : \mathit{SpineStk}(\tau, \sigma_D)$ where $\sigma_D = \mathit{DumpStk}(\tau, \tau', \sigma)$
- $\tau, \sigma \models v \cdot S : \tau_1 \cdot \Delta \leadsto \mathit{Arg}(\tau_1, \Delta \to \tau, \sigma, e_v, e_S) : \mathit{SpineStk}(\tau_1 \cdot \Delta \to \tau, \sigma)$
  - if $\models v : \tau_1 \leadsto e_v : V(\tau_1)$ and $\tau, \sigma \models S : \Delta \leadsto e_S : \mathit{SpineStk}(\Delta \to \tau, \sigma)$

Dump stack correspondence :

- $\models \emptyset : \tau \Rightarrow \tau \leadsto \mathit{Nil}(\tau) : \mathit{DumpStk}(\tau, \mathit{unit}, \mathit{unit})$
- $\models (E, L, C, S) \cdot D : \tau_1 \Rightarrow \tau_0 \leadsto \mathit{Ret}(\tau_1, \Delta \to \tau_2, \sigma_E, \sigma_L, \sigma, e_E, e_L, e_C[\sigma], e_S, e_D) : \mathit{DumpStk}(\tau_1, \Delta \to \tau_2, \sigma)$
  - if $\models E : \Gamma \leadsto e_E : \sigma_E$ and $\models L : \Pi \leadsto e_L : \sigma_L$ and $\Gamma \,|\, \tau_1 \cdot \Pi \rhd C : \Delta \to \tau_2 \leadsto e_C$ and $\tau_2, \sigma \models S : \Delta \leadsto e_S : \mathit{SpineStk}(\Delta \to \tau_2, \sigma)$ and $\models D : \tau_2 \Rightarrow \tau_0 \leadsto e_D : \sigma$.

Fig. 9 Correspondence relations.

5. Discussion

We have implemented all the GADTs and lambda terms in Figs. 5 and 6 using Haskell[†]. The GHC type checker can mechanically verify that the lambda terms are well-typed to assure ourselves of Theorem 2. We could proceed further to verify the other theorems automatically by following an approach of a type-preserving compiler making use of GADTs as dependent types to capture the invariant of programs [14]. We could adopt their typed abstract syntax tree using GADTs, from which we could implement our compilation into terms for ZINC instructions in Haskell.

A compilation from higher-order lambda terms to the restricted form of lambda terms in System-F extended with GADTs can be regarded as an alternative compilation based on the Krivine-style evaluation mechanism. ATS [9] and GHC [10] have already supported GADTs in their typed intermediate languages. They could adopt this compilation method to get the benefit of the Krivine-style evaluation mechanism, but with no change in their type systems.

The proposed Krivine typing mechanism with GADTs provides no optimization in avoiding unnecessary dynamic argument checks, which is a unique feature of [2]. The notion of GADTs could allow to specialize, say, the type of *Grab* to make unnecessary even some runtime

[†]The Haskell code is available as a supplementary material to this paper.

case analysis on the spine stack. We can specify a spine stack with an integer like $Arg(V, Nil)$ precisely by $SpineStk(int \to int, DumpStk(int, \beta, \delta))$ for some types β and δ. The first parameter type *int* of the dump stack type is obtained from removing one argument type in the spine of the function type $int \to int$. By such an appropriate specialization, we could prove that the case analysis on the corresponding spine stack is not necessary. We would also need to enhance our compilation method as in [2] to incorporate such a specialization.

We believe the notion of GADTs is applicable to all variants of Krivine-style evaluation mechanisms in [3], [6]–[8]. They all maintain the application context like a spine stack, and they all pose non-uniform polymorphic behavior as in Return. The notion of GADTs could express the two common features as for the ZINC machine shown previously, though each of the variants may declare GADTs differently due to their own representations.

6. Conclusion

We showed the soundness of the ZINC machine by proving that an embedding of the machine into a calculus of System-F extended with GADTs is correct statically and dynamically. The notion of GADTs can offer a light-weight type-based Krivine-style evaluation mechanism since it requires no change in the existing type systems. The use of GADTs for the evaluation mechanism in a typed setting is new. The result contrasts with the previous proposals that require some significant extension in existing type systems.

We also implemented all the lambda terms for ZINC instructions in the embedding using Haskell. The GHC type checker can verify mechanically that the lambda terms are all well-typed, which highlights the effectiveness of GADTs.

Acknowledgments

This work was supported by the second stage of the Brain Korea 21 Project in 2010.

References

[1] P. Cregut, "An abstract machine for the normalization of λ-terms," Proc. ACM Conference on LISP and Functional Programming, pp.333–340, 1990.
[2] K. Choi and A. Ohori, "A type theory for Krivine-style evaluation and compilation," 2nd Asian Symposium on Programming Languages and System, pp.213–228, 2004.
[3] X. Leroy, "The ZINC experiment: An economical implementation of the ML language," Technical Report 117, INRIA, 1992.
[4] K. Choi and T. Han, "A type system for the push-enter model," Inf. Process. Lett., vol.87, pp.205–211, 2003.
[5] H. Xi, C. Chen, and G. Chen, "Guarded recursive datatype constructors," Proc. ACM Symposium on Principles of Programming Languages, pp.224–235, 2003.
[6] R. Douence and P. Fradet, "A systematic study of functional language implementations," ACM Trans. Programming Languages and Systems, vol.20, no.2, pp.344–387, March 1998.
[7] S.P. Jones, "Implementing lazy functional languages on stock hardware: The spineless tagless G-machine," J. Functional Programming, vol.2, pp.127–202, 1992.
[8] M. Biernacka and O. Danvy, "A concrete framework for environment machines," ACM Trans. Computational Logic, vol.9, no.1, p.6, 2007.
[9] H. Xi, "The ATS language," http://www.ats-lang.org/, 2009.
[10] S.P. Jones, "The Glasgow Haskell compiler," http://www.haskell.org/ghc/, 2004.
[11] M. Sulzmann, M. Chakravarty, and S.P. Jones, "System-F with type equality coercions," Proc. ACM SIGPLAN International Workshop on Types in Language Design and Implementation, pp.53–66, 2007.
[12] G. Morrisett, D. Walker, K. Crary, and N. Glew, "From System F to typed assembly language," Proc. ACM Symposium on Principles of Programming Languages, 1998.
[13] A. Ohori, "The logical abstract machine: A Curry-Howard isomorphism for machine code," Proc. International Symposium on Functional and Logic Programming, 1999.
[14] L.J. Guillemette and S. Monnier, "A type-preserving compiler in Haskell," Proc. 13th ACM SIGPLAN International Conference on Functional Programming, pp.75–86, 2008.
[15] F. Pottier and N. Gauthier, "Polymorphic typed defunctionalization and concretization," Higher-Order and Symbolic Computation, vol.19, pp.125–162, March 2006.

Appendix: Proofs

Theorem 1 (Soundness): If $\Gamma \,|\, \Pi \rhd C : \Delta \to \tau$, $\models E : \Gamma$, $\models L : \Pi$, $\models S : \Delta$, $\models D : \tau \Rightarrow \tau_0$, and $(E, L, C, S, D) \Downarrow v$, then $\models v : \tau_0$.

Proof 1: The proof is by induction on the length of the execution steps.

$C = \mathsf{Return}$: By the rule (Return), $\Gamma \,|\, \tau' \cdot \Pi' \rhd \mathsf{Return} : \tau'$ where $\Pi = \tau' \cdot \Pi'$, $C = \mathsf{Return}$, and $\Delta \to \tau = \tau'$. By local stack typing, $L = v' \cdot L'$, $\models v' : \tau'$, and $\models L' : \Pi'$.

i) $S = v_{arg} \cdot S'$. By spine stack typing, $\models v_{arg} : \tau_{arg}$ and $\models S' : \Delta'$. Since $\Delta \to \tau = \tau_{arg} \to \Delta' \to \tau$, v' is a closure $cls(E', C')$ such that $\models E' : \Gamma'$ and $\Gamma' \,|\, \emptyset \rhd C' : \tau_{arg} \to \Delta' \to \tau$. We have $(E, L, C, S, D) = (E, cls(E', C') \cdot L', \mathsf{Return}, v_{arg} \cdot S', D) \to (E', \emptyset, C', v_{arg} \cdot S', D)$ by the operational semantics. Hence, the theorem is true by induction hypothesis.

ii) $S = \emptyset$ and $D = (E_0, L_0, C_0, S_0) \cdot D'$. By spine stack typing, $\Delta = \emptyset$, which implies $\tau = \tau'$ due to $\Delta \to \tau = \tau'$. By dump stack typing, $\models E_0 : \Gamma_0$, $\models L_0 : \Pi_0$, $\models S_0 : \Delta_0$, $\Gamma_0 \,|\, \tau \cdot \Pi_0 \rhd C_0 : \Delta_0 \to \tau''$, and $\models D' : \tau'' \Rightarrow \tau_0$ for some τ''. We have $(E, L, C, S, D) = (E, v' \cdot L', \mathsf{Return}, \emptyset, (E_0, L_0, C_0, S_0) \cdot D') \to (E_0, v' \cdot L_0, C_0, S_0, D')$ by the operational semantics. Since $\tau = \tau'$, $\models v' \cdot L_0 : \tau' \cdot \Pi_0$. Hence, the theorem is true by induction hypothesis.

iii) $S = \emptyset$ and $D = \emptyset$. By spine stack typing, $\Delta \to \tau = \emptyset \to \tau = \tau = \tau'$. By dump stack typing, $\tau = \tau_0$. We have $(E, L, C, S, D) = (E, v' \cdot L', \mathsf{Return}, \emptyset, \emptyset) \to v'$ by the operational semantics. Since $\models v' : \tau'$, $\models v' : \tau_0$.

$C = \mathsf{Grab} \cdot C'$: By the instruction typing, we have $\Pi = \emptyset$ and $\Delta \to \tau = \tau_1 \to \tau_2$. By local stack typing, L is also empty. By the rule (Grab), $\tau_1 \cdot \Gamma \,|\, \emptyset \rhd C' : \tau_2$.

i) $S = v_{arg} \cdot S'$. By spine stack typing, $\models v_{arg} : \tau_{arg}$ and $\models S' : \Delta'$ due to $\models v_{arg} \cdot S' : \tau_{arg} \cdot \Delta'$. Since $\Delta \to \tau = \tau_{arg} \cdot \Delta' \to \tau = \tau_{arg} \to \Delta' \to \tau$, we have $\tau_{arg} = \tau_1$

and $\Delta' \rightarrow \tau = \tau_2$. $(E, L, C, S, D) = (E, \emptyset, \mathsf{Grab} \cdot C', v_{arg} \cdot S', D) \rightarrow (v_{arg} \cdot E, \emptyset, C', S', D)$ by the operational semantics. By environment typing, $\models v_{arg} \cdot E : \tau_{arg} \cdot \Gamma$. Hence, the theorem is true by induction hypothesis.

ii) $S = \emptyset$ and $D = (E_0, L_0, C_0, S_0) \cdot D'$. Due to the instruction typing, $\Delta \rightarrow \tau = \tau_1 \rightarrow \tau_2$, which implies $\tau = \tau_1 \rightarrow \tau_2$. We have $\models cls(E, \mathsf{Grab} \cdot C') : \tau_1 \rightarrow \tau_2$ due to the environment typing and the instruction typing. $(E, L, C, S, D) = (E, \emptyset, \mathsf{Grab} \cdot C', \emptyset, (E_0, L_0, C_0, S_0) \cdot D') \rightarrow (E, cls(E, \mathsf{Grab} \cdot C') \cdot L_0, C_0, S_0, D')$ by the operational semantics. By dump stack typing, $\models E_0 : \Gamma_0$, $\models L_0 : \Pi_0$, $\models S_0 : \Delta_0$, $\Gamma | \tau \cdot \Pi_0 \rhd C_0 : \Delta_0 \rightarrow \tau''$, and $\models D_0 : \tau'' \Rightarrow \tau_0$. Since $\models cls(E, \mathsf{Grab} \cdot C') \cdot L_0 : \tau \cdot \Pi_0$, the theorem is true by induction hypothesis.

iii) $S = \emptyset$ and $D = \emptyset$.

$(E, L, C, S, D) = (E, \emptyset, \mathsf{Grab} \cdot C', \emptyset, \emptyset) \rightarrow cls(E, \mathsf{Grab} \cdot C)$ by the operational semantics. By the instruction typing, $\Gamma | \Pi \rhd \mathsf{Grab} \cdot C' : \tau_1 \rightarrow \tau_2$. Due to $\Delta = \emptyset$ and $\Delta \rightarrow \tau = \tau_1 \rightarrow \tau_2$, $\tau = \tau_1 \rightarrow \tau_2$. By dump stack typing, $\tau = \tau_0$. Since $\models cls(E, \mathsf{Grab}) \cdot C : \tau_1 \rightarrow \tau_2$, we can conclude $\models cls(E, \mathsf{Grab}) \cdot C : \tau_0$.

$C = \mathsf{Access(i)} \cdot C'$: By (Access), $\Gamma = \Gamma'\{i = \tau_i\}$, $\Gamma'\{i = \tau_i\} | \tau_i \cdot \Pi \rhd C' : \Delta \rightarrow \tau$. By environment typing, $E(i) = v_i$ such that $\models v_i : \tau_i$. By local stack typing, $\models v_i \cdot L : \tau_i \cdot \Pi$. $(E, L, \mathsf{Access(i)} \cdot C, S, D) = (E\{i = v_i\}, L, \mathsf{Access(i)} \cdot C, S, D) \rightarrow (E, v_i \cdot L, C, S, D)$ by the operational semantics. Hence, the theorem holds by induction hypothesis.

$C = \mathsf{Push} \cdot C'$: By (Push), $\Pi = \tau' \cdot \Pi'$ and $\Gamma | \Pi' \rhd C' : \tau' \rightarrow \Delta \rightarrow \tau$. By local stack typing, $\models v' : \tau'$ and $\models L' : \Pi'$. By spine stack typing, $\models v' \cdot S : \tau' \cdot \Delta$. We have $\tau' \rightarrow \Delta \rightarrow \tau = \tau' \cdot \Delta \rightarrow \tau$. $(E, v' \cdot L', \mathsf{Push} \cdot C', S, D) \rightarrow (E, L', C', v' \cdot S, D)$ by the operational semantics. Hence, the theorem holds by induction hypothesis.

$C = \mathsf{Reduce}(C_1) \cdot C_2$: By (Reduce), $\Gamma | \emptyset \rhd C_1 : \tau_1$ and $\Gamma | \tau_1 \cdot \Pi \rhd C_2 : \Delta \rightarrow \tau$. $(E, L, \mathsf{Reduce}(C_1) \cdot C_2, S, D) \rightarrow (E, \emptyset, C_1, \emptyset, (E, L, C_2, S) \cdot D)$ by the operational semantics. By dump stack typing, $\models (E, L, C_2, S) \cdot D : \tau_1 \Rightarrow \tau_0$. Since $\tau_1 = \emptyset \rightarrow \tau_1$ and $\models \emptyset : \emptyset$ for the empty spine stack, the theorem holds due to induction hypothesis.

The cases with $\mathsf{Int}(n)$ and Add can be similarly proved.

Theorem 2 (Well-Typed Terms): All the lambda terms in Fig. A· 1 are well-typed in the extended System F.

Proof 2: This proof is to derive an appropriate typing judgment for each term under the empty type constraint and the empty typing environment. The only interesting part is pattern matches that each imposes a new set of type constraints on the way the typing is derived.

Case Grab. It is straightforward to derive $\emptyset | \Gamma_0 \rhd$ *case* S *of* $\{Arg(\cdots) \rightarrow \cdots,\ Nil(\cdots) \rightarrow \cdots\} : unit$ for some Γ_0 such that $\Gamma_0(S) = SpineStk(\alpha \rightarrow \beta, \delta)$ and $\Gamma_0(D) = \delta$.

i) With $Arg(\alpha_1, \beta_1, \delta_1, V, S') \rightarrow \cdots$, the typing yields a set of type equations $\{\alpha_1 = \alpha, \beta_1 = \beta, \delta_1 = \delta\}$. This enables us to assign $V(\alpha)$ and $SpineStk(\beta, \delta)$ to V and S', respectively. Hence, the corresponding match expression can be proved to be typeable.

$$\begin{array}{l}
Grab = \Lambda\alpha.\Lambda\beta.\Lambda\epsilon.\Lambda\delta.\\
\quad \lambda(I : \forall\delta.(V(\alpha), \epsilon) \rightarrow unit \rightarrow SpineStk(\beta, \delta) \rightarrow \delta \rightarrow unit).\\
\quad \lambda(E : \epsilon).\ \lambda(L : unit).\ \lambda(S : SpineStk(\alpha \rightarrow \beta, \delta)).\ \lambda(D : \delta).\ case\ S\ of\\
\qquad Arg(\alpha_1, \beta_1, \delta_1, V, S') \rightarrow I[\delta]\ (V, E)\ L\ S'\ D\\
\qquad Nil(\alpha_1, \beta_1, \delta_1) \rightarrow case\ D\ of\\
\qquad\quad Ret(\alpha_2, \beta_2, \epsilon_2, \iota_2, \delta_2, I_k, E_k, L_k, S_k, D_k)\\
\qquad\qquad \rightarrow I_k\ E_k\ (Cls[\alpha, \beta, \epsilon](\Lambda\delta.Grab[\alpha, \beta, \epsilon, \delta]\ I, E), L_k)\ S_k\ D_k\\
\qquad\quad Nil(\alpha_2) \rightarrow Halt[\alpha \rightarrow \beta]\ (Cls[\alpha, \beta, \epsilon](\Lambda\delta.Grab[\alpha, \beta, \epsilon, \delta]\ I, E))\\
Return = \Lambda\alpha.\Lambda\epsilon.\Lambda\iota.\Lambda\delta.\ \lambda(E : \epsilon).\ \lambda(L : (V(\alpha), \iota)).\\
\quad \lambda(S : SpineStk(\alpha, \delta)).\ \lambda(D : \delta).\ case\ L\ of\ (V, L') \rightarrow\ case\ S\ of\\
\qquad Arg(\alpha_1, \beta_1, \delta_1, V', S') \rightarrow (case\ V\ of\ \ Cls(\alpha_2, \beta_2, \epsilon_2, I_c, E_c)\\
\qquad\qquad\qquad\qquad \rightarrow I_c[\delta]\ E_c\ ()\ S\ D)\\
\qquad Nil(\alpha_1, \beta_1, \delta_1) \rightarrow case\ D\ of\\
\qquad\quad Ret(\alpha_3, \beta_3, \epsilon_3, \iota_3, \delta_3, I_k, E_k, L_k, S_k, D_k) \rightarrow I_k\ E_k\ (V, L_k)\ S_k\ D_k\\
\qquad\quad Nil(\alpha_3) \rightarrow Halt[\alpha]\ V\\
Push = \Lambda\alpha.\Lambda\beta.\Lambda\epsilon.\Lambda\iota.\Lambda\delta.\\
\quad \lambda(I : \forall\delta.\epsilon \rightarrow \iota \rightarrow SpineStk(\alpha \rightarrow \beta, \delta) \rightarrow \delta \rightarrow unit).\\
\quad \lambda(E : \epsilon).\ \lambda(L : (V(\alpha), \iota)).\ \lambda(S : SpineStk(\beta, \delta)).\\
\quad \lambda(D : \delta).\ case\ L\ of\ (V, L') \rightarrow I[\delta]\ E\ L'\ Arg[\alpha, \beta, \delta](V, S)\ D\\
Reduce = \Lambda\alpha.\Lambda\beta.\Lambda\epsilon.\Lambda\iota.\Lambda\delta.\\
\quad \lambda(I : \forall\delta.\epsilon \rightarrow unit \rightarrow SpineStk(\alpha, DumpStk(\alpha, \beta, \delta))\\
\qquad\quad \rightarrow DumpStk(\alpha, \beta, \delta) \rightarrow unit).\\
\quad \lambda(I_k : \forall\delta.\epsilon \rightarrow (V(\alpha), \iota) \rightarrow SpineStk(\beta, \delta) \rightarrow \delta \rightarrow unit).\\
\quad \lambda(E : \epsilon).\ \lambda(L : \iota).\ \lambda(S : SpineStk(\beta, \delta)).\\
\quad \lambda(D : \delta).\ I[\delta]\ E\ ()\ Nil(\alpha, \beta, \delta)\ Ret(\alpha, \beta, \epsilon, \iota, \delta, I_k[\delta], E, L, S, D)\\
Access(i) = \Lambda\alpha_0. \cdots \Lambda\alpha_i.\Lambda\beta.\Lambda\epsilon.\Lambda\iota.\Lambda\delta.\\
\quad \lambda(I : \forall\delta.(\alpha_0, (\cdots, (\alpha_i, \epsilon))) \rightarrow \alpha_i * \iota \rightarrow SpineStk(\beta, \delta) \rightarrow \delta \rightarrow unit).\\
\quad \lambda(E : (\alpha_0, (\cdots, (\alpha_i, \epsilon)))).\ \lambda(L : \iota).\ \lambda(S : SpineStk(\beta, \delta)).\\
\quad \lambda(D : \delta).\ case\ E\ of (V_0, E_0) \rightarrow\ \cdots\ case\ E_{i-1}\ of (V_i, E_i)\\
\qquad\qquad\qquad \rightarrow I[\delta]\ E\ (V_i, L)\ S\ D\\
Int = \lambda(n : int).\Lambda\beta.\Lambda\epsilon.\Lambda\iota.\Lambda\delta.\\
\quad \lambda(I : \forall\delta.\epsilon \rightarrow (V(int), \iota) \rightarrow SpineStk(\beta, \delta) \rightarrow \delta \rightarrow unit).\\
\quad \lambda(E : \epsilon).\ \lambda(L : \iota).\ \lambda(S : SpineStk(\beta, \delta)).\ \lambda(D : \delta).\\
\qquad\qquad\qquad I[\delta]\ E\ (Con(n), L)\ S\ D\\
Add = \Lambda\beta.\Lambda\epsilon.\Lambda\iota.\Lambda\delta.\\
\quad \lambda(I : \forall\delta.\epsilon \rightarrow (V(int), \iota) \rightarrow SpineStk(\beta, \delta) \rightarrow \delta \rightarrow unit).\\
\quad \lambda(E : \epsilon).\ \lambda(L : (V(int), (V(int), \iota))).\ \lambda(S : SpineStk(\beta, \delta)).\ \lambda(D : \delta).\\
\qquad case\ L\ of (V_0, L_0) \rightarrow case\ L_0\ of (V_1, L_1) \rightarrow case\ V_0\ of\ Con(n_0) \rightarrow\\
\qquad case\ V_1\ of\ Con(n_1) \rightarrow I[\delta]\ E\ (Con(n_0 + n_1), L)\ S\ D\\
Halt = \Lambda\alpha.\lambda(V : V(\alpha)).()
\end{array}$$

Fig. A· 1 Typed lambda terms for ZINC instructions.

ii) With $Nil(\alpha_1, \beta_1, \delta_1) \rightarrow \cdots$, the typing yields $\{\alpha_1 = \alpha \rightarrow \beta, DumpStk(\alpha_1, \beta_1, \delta_1) = \delta\}$. This type equation leads the subsequent case expression to type safely decompose D.

ii-1) With $Ret(\alpha_2, \beta_2, \epsilon_2, \iota_2, \delta_2, I_k, E_k, L_k, L_k, S_k, D_k) \rightarrow \cdots$, the typing yields $\{\alpha_2 = \alpha_1, \beta_2 = \beta_1, \delta_2 = \delta_1\}$. This enables us to have equations $V(\alpha_2) = V(\alpha_1) = V(\alpha \rightarrow \beta)$. By value typing, $Clo[\alpha, \beta, \epsilon](\Lambda\delta.Grab[\alpha, \beta, \epsilon, \delta]\ I, E)$ has type $V(\alpha \rightarrow \beta)$. Hence, the corresponding match expression can be proved to be typeable.

ii-2) With $Nil(\alpha_2) \rightarrow \cdots$, the corresponding match expression can be shown to be typeable whatever the typing on the pattern yields.

Case Return. One can derive $\emptyset | \Gamma_0 \rhd case\ S\ of\ Arg(\cdots) \rightarrow \cdots,\ Nil(\cdots) \rightarrow \cdots : unit$ for some Γ_0 such that $\Gamma_0(S) = SpineStk(\alpha, \delta)$ and $\Gamma_0(D) = \delta$.

i) With $Arg(\alpha_1, \beta_1, \delta_1, V', S') \rightarrow \cdots$, the typing yields a set of type equations $\{\alpha_1 \rightarrow \beta_1 = \alpha, \delta_1 = \delta\}$. This forces us to assign V a closure type $V(\alpha_1 \rightarrow \beta_1)$, and so V cannot be any constants. Hence, the subsequent inner case expression has only one alternative for closure values. The rest of the typing is straightforward.

ii) With $Nil(\alpha_1,\beta_1,\delta_1) \to \cdots$, the typing yields $\{\alpha_1 = \alpha, DumpStk(\alpha_1,\beta_1,\delta_1) = \delta\}$. The rest of the typing is very similar to the case of Grab. The term can be proved to be typeable.

Case Push. The interesting part of the typing is for a spine stack S extended with V, $Arg[\alpha,\beta,\delta](V,S)$, which is of type $SpineStk(\alpha \to \beta,\delta)$.

Case Reduce. The interesting part of the typing is for a dump stack D extended with another continuation, $Ret(\alpha,\beta,\epsilon,\iota,\delta,I_k[\delta],E,L,S,D)$, which is of type $D(\alpha,\beta,\delta)$.

Case Access(i), Int, and Add. These cases can be trivially proved to be true.

Theorem 3 (Typed Compilation of the ZINC Instructions): Suppose we have $\Gamma \rightsquigarrow \sigma_e$, and $\Pi \rightsquigarrow \sigma_l$. If $\Gamma\,|\,\Pi \rhd C : \tau \rightsquigarrow e$ then $\emptyset\,|\,\emptyset \rhd e : \forall\delta.\sigma_e \to \sigma_l \to SpineStk(\tau,\delta) \to \delta \to unit$.

Proof 3: The proof is by induction on the height of type derivation on C. Base case. Case Return. By (C-Return), $\Pi = \tau \cdot \Pi'$ and $\Pi' \rightsquigarrow \sigma'_l$. Since $Return$ has type $\forall\alpha.\forall\epsilon.\forall\iota.\forall\delta.\epsilon \to (V(\alpha),\iota) \to SpineStk(\alpha,\delta) \to \delta \to unit$, one can conclude that $\Lambda\delta.Return[\tau,\sigma_e,\sigma'_l,\delta]$ has type $\forall\delta.\sigma_e \to (V(\tau),\sigma'_l) \to SpineStk(\tau,\delta) \to \delta \to unit$. By $\sigma_l = V(\tau)\cdot\sigma'_l$, the theorem is true.

Inductive cases. Case Grab $\cdot\, C'$. By assumption, there exist τ_1 and τ_2 such that $\tau_1 \to \tau_2 = \tau$. We have $\Pi = \emptyset$, which corresponds to $\sigma_l = unit$. By induction, C' is compiled to an expression e which has type $\forall\delta.(V(\tau_1),\sigma_e) \to \emptyset \to SpineStk(\tau_2,\delta) \to \delta \to unit$. The expression of interest, $\Lambda\delta.Grab[\tau_1,\tau_2,\sigma_e,\delta]\ e$, is of type $\forall\delta.\sigma_e \to unit \to SpineStk(\tau_1 \to \tau_2,\delta) \to \delta \to unit$, by (G-Abs) and (G-TAbs). Therefore, the theorem holds.

Case Reduce(C_1) $\cdot\, C_2$. By assumption, we have $\Gamma\,|\,\emptyset \rhd C_1 : \tau_0 \rightsquigarrow e_1$ and $\Gamma\,|\,\tau_0\cdot\Pi \rhd C_2 : \tau \rightsquigarrow e_2$. By induction, e_1 has type $\forall\delta.\sigma_e \to unit \to SpineStk(\tau_0,\delta) \to \delta \to unit$, and e_2 has type $\forall\delta.\sigma_e \to V(\tau_0)\cdot\sigma_l \to SpineStk(\tau,\delta) \to \delta \to unit$. For some type variables α,β, and δ', the instantiation of δ with $DumpStk(\alpha,\beta,\delta')$ in the type of e_1 results in $\forall\delta'.\sigma_e \to unit \to SpineStk(\tau_0, DumpStk(\alpha,\beta,\delta')) \to DumpStk(\alpha,\beta,\delta') \to unit$. The expression of interest, $\Lambda\delta.Reduce[\tau_0,\tau,\sigma_e,\sigma_l,\delta]\ e_1\ e_2$, has type $\forall\delta.\sigma_e \to \sigma_l \to SpineStk(\tau,\delta) \to \delta \to unit$ by (G-Abs) and (G-TAbs).

Case Push $\cdot\, C'$. By assumption, there exist τ_0 and Π_0 such that $\Pi = \tau_0 \cdot \Pi_0$. By assumption, $\Pi_0 \rightsquigarrow \sigma'_l$. By induction, C' is compiled to e' which has type $\forall\delta.\sigma_e \to \sigma'_l \to SpineStk(\tau_0 \to \tau,\delta) \to \delta \to unit$. The expression of interest, $\Lambda\delta.Push[\tau_0,\tau,\sigma_e,\sigma'_l,\delta]\ e'$, has type $\forall\delta.\sigma_e \to (V(\tau_0),\sigma'_l) \to SpineStk(\tau,\delta) \to \delta \to unit$ by (G-Abs) and (G-TAbs).

Case Access(i), Int, and Add. These cases can be similarly proved.

Theorem 4 (Semantic Correctness): Suppose $\Gamma\,|\,\Pi \rhd C : \Delta \to \tau \rightsquigarrow e$, $\models E : \Gamma \rightsquigarrow e_E : \sigma_E$, $\models L : \Pi \rightsquigarrow e_L : \sigma_L$, $\tau,\sigma_D \models S : \Delta \rightsquigarrow e_S : SpineStk(\Delta \to \tau,\sigma_D)$, $\models D : \tau \Rightarrow \tau_0 \rightsquigarrow e_D : \sigma_D$. If $(E,L,C,S,D) \longrightarrow^* v$ then $e[\sigma_D]\ e_E\ e_L\ e_S\ e_D \overset{*}{\longrightarrow}_\beta Halt[\tau](e_v)$ such that $\models v : \tau \rightsquigarrow e_v : V(\tau)$.

Proof 4: The proof is by induction on the length of execution steps. Case C = Grab $\cdot\, C'$ and $S = v' \cdot S'$. By the assumptions, $\tau,\sigma_D \models v'\cdot S' : \tau'\cdot\Delta' \to \tau \rightsquigarrow Arg(\tau_1,\tau_2,\sigma_D,e_{v'},e_{S'}) : SpineStk(\Delta \to \tau,\sigma_D)$ where $\tau_1 = \tau'$, $\tau_2 = \Delta' \to \tau$, $\Delta = \tau'\cdot\Delta'$, $e_S = Arg(\tau_1,\tau_2,\sigma_D,e_{v'},e_{S'})$. By (C-Grab), $e = \Lambda\delta.Grab[\tau_1,\tau_2,\sigma_E,\delta]e'$ where $\tau'\cdot\Gamma\,|\,\emptyset \rhd C' : \Delta' \to \tau \rightsquigarrow e'$, and $\Pi = \emptyset$. By the local stack correspondence, $L = \emptyset$, and $e_L = ()$ due to $\Pi = \emptyset$. $(E,\emptyset,\text{Grab}\cdot C', v'\cdot S', D) \to (v'\cdot E,\emptyset,C',S',D)$ by the operational semantics. $(e\ [\sigma_D]\ e')\ e_E\ e_L\ e_S\ e_D = \text{Grab}[\tau_1,\tau_2,\sigma_E,\sigma_D]e'e_Ee_Le_Se_D \to^*_\beta e'[\sigma_D](e_{v'},e_E)e_Le_{S'}e_D$ by applying β-reductions. Now one can apply induction hypothesis because $\models v'\cdot E : \tau'\cdot\Gamma \rightsquigarrow (e_{v'},e_E)$ and $\tau,\sigma_D \models S' : \Delta' \to \tau \rightsquigarrow e_{S'} : SpineStk(\Delta' \to \tau,\sigma_D)$ by the environment correspondence and the spine stack correspondence. Hence, theorem holds.

Case C = Grab $\cdot\, C'$, $S = \emptyset$, and $D = (E_0,L_0,C_0,S_0)\cdot D'$. By (C-Grab), $\Gamma\,|\,\emptyset \rhd$ Grab $\cdot\, C' : \Delta \to \tau \rightsquigarrow \Lambda\delta.Grab\ [\tau_1,\tau_2,\sigma_E,\delta]\ e'$ where $\Gamma \rightsquigarrow \sigma_E$, $\Pi = \emptyset$, $\tau_1 \to \tau_2 = \Delta \to \tau$, $\Delta = \tau_1\cdot\Delta_1$, and $\tau_2 = \Delta_1 \to \tau$ for some e' and Δ_1. By the spine stack correspondence, $\tau,\sigma_D \models \emptyset : \emptyset \rightsquigarrow Nil(\tau,\tau_3,\sigma'_D) : SpineStk(\tau,\sigma_D)$ where $\sigma_D = DumpStk(\tau,\tau_3,\sigma'_D)$. By the dump stack correspondence, $\models (E_0,L_0,C_0,S_0)\cdot D' : \tau \Rightarrow \tau_0 \rightsquigarrow e_D : DumpStk(\tau,\tau_3,\sigma'_D)$ where $e_D = Ret(\tau,\tau_3,\sigma_{E_0},\sigma_{L_0},\sigma'_D,e_{E_0},e_{L_0},e_{C_0}[\sigma'_D],e_{S_0},e'_D)$. This implies the following five correspondences on the subcomponents of the dump stack. $\models E_0 : \Gamma_0 \rightsquigarrow e_{E_0} : \sigma_{E_0}$, $\models L_0 : \Pi_0 \rightsquigarrow e_{L_0} : \sigma_{L_0}$, $\Gamma_0\,|\,\tau\cdot\Pi_0 \rhd C_0 : \tau_3 \rightsquigarrow e_{C_0}$, $\tau_4,\sigma'_D \models S_0 : \Delta_2 \rightsquigarrow e_{S_0} : SpineStk(\Delta_2 \to \tau_4,\sigma'_D)$, $\models D' : \tau_4 \Rightarrow \tau_0 \rightsquigarrow e'_D : \sigma'_D$. By the operational semantics, $(E,\emptyset,\text{Grab}\cdot C',\emptyset,(E_0,L_0,C_0,S_0)\cdot D') \to (E_0, cls(E,\text{Grab}\cdot C')\cdot L_0, C_0, S_0, D')$. $Grab\ [\tau_1,\tau_2,\sigma_E,\sigma_D]\ e'\ e_E\ e_L\ e_S\ e_D$ where $e_L = ()$ and $e_S = Nil(\tau,\tau_3,\sigma_{D'})$. This term reduces to $e_{C_0}\ [\sigma'_D]\ e_{E_0}\ (Clo[\tau_1,\tau_2,\sigma_E]\ (e,e_E),e_{L_0})\ e_{S_0}\ e'_D$. By induction hypothesis, the theorem holds.

Case C = Grab $\cdot\, C'$, $S = \emptyset$, and $D = \emptyset$. By the dump stack correspondence, $\models \tau_0 \Rightarrow \tau_0 \rightsquigarrow Nil(\tau_0) : \sigma_D$ where $\sigma_D = DumpStk(\tau_0,\tau_0,unit)$. By the spine stack correspondence, $\tau,\sigma_D \models \emptyset : \emptyset \rightsquigarrow Nil(\tau_1,\tau_2,\sigma_D) : SpineStk(\emptyset \to \tau,\sigma_D)$. By (C-Grab), $\Gamma\,|\,\emptyset \rhd$ Grab $\cdot\, C' : \emptyset \to \tau \rightsquigarrow \Lambda\delta.Grab\ [\tau_1,\tau_2,\sigma_E,\delta]\ e'$ for some τ_1, τ_2, and e' where $\tau_1 \to \tau_2 = \Delta \to \tau$. By the operational semantics, $(E,\emptyset,\text{Grab}\cdot C',\emptyset,\emptyset) \to cls(E,\text{Grab}\cdot C')$. $(\Lambda\delta.Grab\ [\tau_1,\tau_2,\sigma_E,\delta]\ e')\ [\sigma_D]\ e_E\ e_L\ Nil(\tau_1,\tau_2,\sigma_D)\ Nil(\tau_0) \to Halt[\tau_1 \to \tau_2]\ (Clo[\tau_1,\tau_2,\sigma_E](\Lambda\delta.Grab[\tau_1,\tau_2,\sigma_e,\delta]\ e', e_E))$. Since the result value of the ZINC machine corresponds to the result value in the represented term by the value correspondence, the theorem holds.

Case C = Return and $S = v'\cdot S'$. By (C-Return), $\Gamma\,|\,\tau_r\cdot\Pi' \rhd$ Return $: \Delta \to \tau \rightsquigarrow e$ where $\Pi = \tau_r\cdot\Pi'$, $\tau_r = \Delta \to \tau$ and $e = \Lambda\delta.Return\ [\tau_r,\ \sigma_E,\ \sigma_{L'},\ \delta]$. By the spine stack correspondence, $\tau,\sigma_D \models v'\cdot S' : \tau'\cdot\Delta' \rightsquigarrow Arg(\tau',\Delta' \to \tau,\sigma_D,e_{v'},e_{S'}) : SpineStk(\tau'\cdot\Delta' \to \tau,\sigma_D)$ where $\Delta = \tau'\cdot\Delta'$. By the local stack correspondence, $\models L : \Pi \rightsquigarrow e_L : \sigma_L$ where $L = v_r\cdot L'$ and $e_L = (e_{v_r},e_{L'})$. Due to $\tau_r = \Delta \to \tau = \tau' \to (\Delta' \to \tau)$, the local stack correspondence implies v_r is a closure such that $\models Clo(E_0,C_0) : \tau' \to$

$(\Delta' \to \tau) \rightsquigarrow \mathit{Cls}[\tau', \Delta' \to \tau, \sigma_{E_0}](e_{E_0}, e_{C_0}) : V(\tau' \to (\Delta' \to \tau))$. Moreover, the local stack correspondence implies $\models E_0 : \Gamma_0 \rightsquigarrow e_{E_0} : \sigma_{E_0}$ and $\Gamma_0 \,|\, \emptyset \rhd C_0 : \tau' \to (\Delta' \to \tau) \rightsquigarrow e_{C_0}$ $(E, L, C, S, D) = (E, \mathit{Clo}(E_0, C_0) \cdot L', \mathsf{Return}, v' \cdot S', D) \to (E_0, \emptyset, C_0, v' \cdot S', D)$. By the assumed correspondences, $\mathit{Return}[\tau' \to (\Delta' \to \tau), \sigma_E, \sigma_{L'}, \sigma_D]\ e_E\ e_L\ e_S\ e_D$ reduces to $e_{C_0}[\sigma_D]\ e_{E_0}\ ()\ e_S\ e_D$ because e_L is a pair, e_S is a spine stack with at least an argument, and e_{v_r} is a closure. Because of the conditions we got from the local stack correspondence and the assumed conditions on S and D, one can apply induction hypothesis. Hence the theorem holds.

Case $C = \mathsf{Return}$, $S = \emptyset$, and $D = (E_0, L_0, C_0, S_0) \cdot D'$. By the assumed conditions, $\Gamma \,|\, \tau' \cdot \Pi' \rhd \mathsf{Return} : \Delta \to \tau \rightsquigarrow \Lambda\delta.\mathit{Return}[\Delta \to \tau, \sigma_E, \sigma_{\Pi'}, \delta]$, $\models E : \Gamma \rightsquigarrow e_E : \sigma_E$, $\models v_{ret} \cdot L' : \tau_{ret} \cdot \Pi' \rightsquigarrow (e_{ret}, e_{L'}) : (\sigma_{ret}, \sigma_{L'})$, $\tau, \sigma_D \models \emptyset : \emptyset \rightsquigarrow e_S : \mathit{SpineStk}(\emptyset \to \tau, \sigma_D)$, $\Delta = \emptyset$, $e_S = \mathit{Nil}(\tau, \Delta_0 \to \tau_2, \sigma_{D'})$, $\models (E_0, L_0, C_0, S_0) \cdot D' : \tau \Rightarrow \tau_0 \rightsquigarrow e_0 : \sigma_0$, $e_0 = \mathit{Ret}(\tau, \Delta_0 \to \tau_2, \sigma_{E_0}, \sigma_{L_0}, \sigma_{D'}, e_{E_0}, e_{L_0}, e_{C_0}[\sigma'_D], e_{S_0}, e_{D'})$, and $\sigma_D = \mathit{DumpStk}(\tau, \Delta_0 \to \tau_2, \sigma_{D'})$. By the dump stack correspondence, $\models E_0 : \Gamma_0 \rightsquigarrow e_{E_0} : \sigma_{E_0}$, $\models L_0 : \Pi_0 \rightsquigarrow e_{L_0} : \sigma_{L_0}$, $\Gamma_0 \,|\, \tau \cdot \Pi_0 \rhd C_0 : \Delta_0 \to \tau_2 \rightsquigarrow e_{C_0}$, $\tau_2, \sigma_{D'} \models S_0 : \Delta_0 \rightsquigarrow e_{S_0} : \mathit{SpineStk}(\Delta_0 \to \tau_2, \sigma_{D'})$, and $\models D' : \tau_2 \Rightarrow \tau_0 \rightsquigarrow e_{D'} : \sigma_{D'}$. By the operational semantics, $(E, v_{ret} \cdot L', \mathsf{Return}, \emptyset, (E_0, L_0, C_0, S_0) \cdot D') \to (E_0, v_{ret} \cdot L_0, C_0, S_0, D')$.

Correspondingly, $(\Lambda\delta.\mathsf{Return}[\emptyset \to \tau, \sigma_E, \sigma_{L'}, \delta])[\sigma_D]$ $e_E\ (e_{ret}, e_{L'})\ \mathit{Nil}(\tau, \Delta_0 \to \tau_2, \sigma_{D'})\ \mathit{Ret}(\tau, \Delta_0 \to \tau_2, \sigma_{E_0}, \sigma_{L_0}, \sigma_{D'}, e_{E_0}, e_{L_0}, e_{C_0}[\sigma'_D], e_{S_0}, e_{D'})$ reduces to $e_{C_0}\ [\sigma_{D'}]\ e_{E_0}\ (e_{ret}, e_{L_0})\ e_{S_0}\ e_{D'}$. Since $\models v_{ret} \cdot L_0 : \tau_{ret} \cdot \Pi_0 \rightsquigarrow (e_{ret}, e_{L_0}) : (\sigma_{ret}, \sigma_{L_0})$ by the local stack correspondence, induction hypothesis can be applied. The theorem holds.

Case $C = \mathsf{Return}$, $S = \emptyset$, and $D = \emptyset$. By spine stack typing, $\Delta = \emptyset$. By the assumed conditions, $\Gamma \,|\, \tau' \cdot \Pi' \rhd \mathsf{Return} : \Delta \to \tau \rightsquigarrow \Lambda\delta.\mathit{Return}[\tau', \sigma_E, \sigma_L, \delta]$, $\models E : \Gamma \rightsquigarrow e_E : \sigma_E$, $\models v_{ret} \cdot L' : \tau_{ret} \cdot \Pi' \rightsquigarrow (e_{ret}, e_{L'}) : (\sigma_{ret}, \sigma_{\Pi'})$ where $\sigma_{ret} = V(\tau')$ and $\tau_{ret} = \tau'$, $\tau, \sigma_D \models \emptyset : \emptyset \rightsquigarrow \mathit{Nil}(\tau, \tau, \mathit{unit}) : \mathit{SpineStk}(\Delta \to \tau, \sigma_D)$, $\sigma_D = \mathit{DumpStk}(\tau, \tau, \mathit{unit})$, $\models \emptyset : \tau \Rightarrow \tau \rightsquigarrow \mathit{Nil}(\tau) : \mathit{DumpStk}(\tau, \tau, \mathit{unit})$ where $\tau = \tau_0$. By the operational semantics, $(E, v_{ret} \cdot L', \mathsf{Return}, \emptyset, \emptyset) \to v_{ret}$. $(\Lambda\delta.\mathit{Return}[\tau', \sigma_E, \sigma_L, \delta])[\sigma_D]\ e_E\ (e_{ret}, e_{L'})\ \mathit{Nil}(\tau, \tau, \mathit{unit})$ $\mathit{Nil}(\tau)$ reduces to $\mathit{Halt}[\tau']\ e_{ret}$. By the value correspondence, $\models v_{ret} : \tau \rightsquigarrow e_{ret} : \sigma_{ret}$. Therefore, the theorem holds.

Case $C = \mathsf{Access}(i) \cdot C'$. By the assumed correspondence of the environment, $\models v_i : \tau_i \rightsquigarrow e_i : \sigma_i$. With the assumed correspondence of the local stack, $\models L : \Pi \rightsquigarrow e_L : \sigma_L$, one can derive $\models v_i \cdot L_i : \tau_i \cdot \Pi_i \rightsquigarrow (e_i, e_L) : (\sigma_i, \sigma_L)$. $(E\{i : v_i\}, L, \mathsf{Access}(i) \cdot C', S, D) \to (E\{i : v_i\}, v_i \cdot L, C', S, D)$. $(\Lambda\delta.\mathit{Access}_i[\sigma_0, \cdots, \sigma_i, \Delta \to \tau, \sigma_E, \sigma_L, \delta]\ e_{C'})[\sigma_D]\ e_E\ e_L\ e_S\ e_D$ reduces to $e_{C'}[\sigma_D]\ e_E$ $(e_i, e_L)\ e_S\ e_D$. It is easy to verify the conditions necessary for applying induction hypothesis. Hence the theorem holds.

Case $C = \mathsf{Push} \cdot C'$. By the assumed correspondence of the local stack, $\models v' \cdot L' : \tau' \cdot \Pi' \rightsquigarrow (e_{v'}, e_{L'}) : (\sigma_{v'}, \sigma_{L'})$. With the assumed correspondence of the spine stack, we have $\tau, \sigma_D \models v' \cdot S : \tau' \cdot \Delta \rightsquigarrow \mathit{Arg}(\tau', \Delta \to \tau, \sigma_D, e_{v'}, e_S) : \mathit{SpineStk}(\tau' \cdot \Delta \to \tau, \sigma_D)$, $(E, v' \cdot L', \mathsf{Push} \cdot C', S, D) \to (E, L', C', v' \cdot S, D)$. $(\Lambda\delta.\mathit{Push}[\tau', \Delta \to \tau, \sigma_E, \sigma_L, \delta]\ e_{C'})[\sigma_D]\ e_E\ (e_{v'}, e_{L'})\ e_S\ e_D$ reduces to $e_{C'}[\sigma_D]\ e_E\ e'_L\ \mathit{Arg}(\tau', \Delta \to \tau, \sigma_D, e_{v'}, e_S)\ e_D$.

Case $C = \mathsf{Reduce}(C_0) \cdot C_1$. By the assumed condition for the compilation, $\Gamma \,|\, \Pi \rhd \mathsf{Reduce}(C_0) \cdot C_1 : \Delta \to \tau \rightsquigarrow e$ where $e = \Lambda\delta.\mathit{Reduce}[\tau', \Delta \to \tau, \sigma_E, \sigma_L, \delta]\ e_0\ e_1$, $\Gamma \,|\, \emptyset \rhd C_0 : \tau' \rightsquigarrow e_0$, and $\Gamma \,|\, \tau' \cdot \Pi \rhd C_1 : \Delta \to \tau \rightsquigarrow e_1$. By the assumed conditions for the correspondences, $\models E : \Gamma \rightsquigarrow e_E : \sigma_E$, $\models L : \Pi \rightsquigarrow e_L : \sigma_L$, $\tau, \sigma_D \models S : \Delta \rightsquigarrow e_S : \mathit{SpineStk}(\Delta \to \tau, \sigma_D)$, and $\models D : \tau \Rightarrow \tau_0 \rightsquigarrow e_D : \sigma_D$. By the operational semantics, $(E, L, \mathsf{Reduce}(C_0) \cdot C_1, S, D) \to (E, \emptyset, C_0, \emptyset, (E, L, C_1, S) \cdot D)$. By the local stack correspondence, $\models \emptyset : \emptyset \rightsquigarrow () : \mathit{unit}$. By the spine stack correspondence, $\tau', \mathit{DumpStk}(\tau', \Delta \to \tau, \sigma_D) \models \emptyset : \emptyset \rightsquigarrow \mathit{Nil}(\tau', \Delta \to \tau, \sigma_D) : \mathit{SpineStk}(\tau', \mathit{DumpStk}(\tau', \Delta \to \tau, \sigma_D))$. By the dump stack correspondence, $\models (E, L, C, S) \cdot D : \tau' \Rightarrow \tau_0 \rightsquigarrow \mathit{Ret}(\tau', \Delta \to \tau, \sigma_E, \sigma_L, \sigma_D, e_E, e_C, e_S, e_D) : \mathit{DumpStk}(\tau', \Delta \to \tau, \sigma_D)$. $(\Lambda\delta.\mathit{Reduce}[\tau', \Delta \to \tau, \sigma_E, \sigma_L, \delta]\ e_0\ e_1)[\sigma_D]\ e_E\ e_L\ e_S\ e_D$ reduces to $e_0[\sigma_D]\ e_E\ ()\ \mathit{Nil}(\tau', \Delta \to \tau, \sigma_D)\ \mathit{Ret}(\tau', \Delta \to \tau, \sigma_E, \sigma_L, \sigma_D, e_1[\sigma_D], e_E, e_L, e_S, e_D)$. By applying induction hypothesis to the next state and the reduced term, the theorem can be proved to be true.

Kwanghoon Choi is an Assistant Professor of Computer & Telecommunication Engineering Division at Yonsei University, Wonju. He received the B.S., the M.S., and the Ph.D. degrees in Computer Science from Korea Advanced Institute of Science and Technology (KAIST) in 1994, 1996, and 2003, respectively. His major research areas are type systems, programming languages, and compilers. Dr. Choi is a member of ACM SIGPLAN.

Seog Park is a Professor of Computer Science at Sogang University. He received the B.S. degree in Computer Science from Seoul National University in 1978, the M.S. and the Ph.D. degrees in Computer Science from Korea Advanced Institute of Science and Technology (KAIST) in 1980 and 1983, respectively. Since 1983, he has been working in the Department of Computer Science and Engineering, Sogang University. His major research areas are database security, digital library, multimedia database systems, role-based access control, Web database, and data stream management system. Dr. Park is a member of the IEEE Computer Society, ACM and the Korean Institute of Information Scientists and Engineers.