Typing ZINC Machine with Generalized Algebraic Data Types

Kwanghoon CHOI\(^{1,2}\), Nonmember and Seog PARK\(^1\), 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 \((\lambda x.M)_1 \cdots M_m\), the machine causes the following transition. \((E, S, (\lambda x.M)_1 \cdots M_m) \Rightarrow^* (E, V_1 \cdots V_m, S, \lambda x.M)\) where each \(V_i\) is the value denoted by \(M_i\) under \(E\), and \(V_1 \cdots V_m, S\) is the stack obtained by pushing \(V_m, \ldots, V_1\) 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. \((E, V_1 \cdots V_m, S, \lambda x.M) \Rightarrow (E[x : V_1], V_2 \cdots V_m, S, M)\)

As observed by Leroy \([3]\), this mechanism avoids unnecessary closure construction in evaluating the nested applications such as \((\lambda x_1, \ldots, \lambda x_n. M)_1 \cdots M_n\), 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 lightweight 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.

- \(\text{Nil} :: \forall a. GList a\)
- \(\text{Cons} :: \forall a b. a \rightarrow GList b \rightarrow GList (a \rightarrow b)\)

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 \(\text{Cons} V_1 (\text{Cons} V_2 \text{Nil})\) where \(V_1\) and \(V_2\) are an integer and a boolean respectively. \(\text{Nil}\) is a list of type \(\text{List} \text{Int}\), and so \(\text{Cons} V_2 \text{Nil}\) becomes a list of type \(\text{List} (\text{Bool} \rightarrow \text{Int})\). Note that lists of type \(\text{Int} \rightarrow \text{Bool} \rightarrow \text{Int}\) cannot be longer than two while the list
of type \( \text{List Int} \) is only \( \text{Nil} \). These heterogeneous lists typed with \( \text{GList } T \) 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.

Potier 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.

\[ I ::= \text{Return} \mid \text{Grab} \mid \text{Push} \mid \text{Access}(\mathit{i}) \mid \text{Reduce}(\mathit{C}) \mid \text{Int}(\mathit{i}) \mid \text{Add} \]

Values (\( v \)) are either a constant \( c \) or a closure \( \text{clst}(E, C) \). 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
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 $i$th value in $E$.

A ZINC machine \cite{Knoch:2010} 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 follows.

$$[[i]] = \text{Access}(i) \cdot \text{Return}$$

$$[[n]] = \text{Int}(n) \cdot \text{Return}$$

$$[[\lambda M]] = \text{Grab} \cdot [[M]]$$

$$[[M_1, M_2]] = \text{Reduce} \cdot [[M_2]] \cdot \text{Push} \cdot [[M_1]]$$

$$[[M_1 + M_2]] = \text{Reduce} \cdot [[M_1]] \cdot \text{Reduce} \cdot [[M_2]] \cdot \text{Add} \cdot \text{Return}$$

For example, $[[\lambda x. \lambda y. x \, 1]]$, using the named variable notation for convenience, can be compiled into $\text{Int}(1) \cdot \text{Push} \cdot \text{Grab} \cdot \text{Grab} \cdot \text{Access}(1) \cdot \text{Return}$ where $\text{Reduce} \cdot [[1]] \cdot C$ is replaced with $\text{Int}(1) \cdot C$. When we run the compiled code, $\text{Int}(1)$ puts $l$ on the local stack, and $\text{Push}$ moves the integer to the spine stack. The first $\text{Grab}$ will then find on the spine stack the integer to pop into the environment. However, the second $\text{Grab}$ will see the empty spine stack since the execution started with it. At this moment, the instruction will create a closure with the code $\text{Grab} \cdot \text{Access}(1) \cdot \text{Return}$ and the environment with the integer value in the $1st$ 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 $[[\lambda x. \lambda y. x \, 2]] M$, the closure obtained from running $[[M]]$ becomes bound to the $0th$ entry of the environment associated with $z$. $[[z \, 2]]$ can be compiled into $\text{Int}(2) \cdot \text{Push} \cdot \text{Access}(0) \cdot \text{Return}$ similarly. When we run $[[z]], \text{Access}(0)$ will retrieve the closure onto the local stack. After $\text{Return}$ 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 $\tau$) are defined by either $\text{int}$ or $\tau \rightarrow \tau$. Two typing contexts, environment type and local stack type (ranged over by $\Gamma$ and $\Pi$), are defined as a sequence of types.

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

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

It is important to understand that the code type $\Delta \rightarrow \tau$ 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 $\Delta$) and an associated dump stack (by $\tau$).

The set of typing rules is given in Fig. 2. By (Push), the potential spine stack grows from one specified by $\tau' \rightarrow \tau$ to another by $\tau \rightarrow \tau'$ 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 $\text{int}$, then the code type lets the spine stack empty; $\emptyset$ is the only $\Delta$ satisfying $\text{Int} \equiv \Delta \rightarrow \text{Int}$. If the return value type is $\tau_1 \rightarrow \tau_2$, then we can find the code type equivalent to $\tau_1' \cdot \Delta' \rightarrow \tau_2'$ for some $\tau_1', \tau_2'$, and $\Delta'$. (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 $\Gamma \sqsupset C : \Delta \rightarrow \tau$, $\models E : \Gamma$, $\models L : \Pi$, $\models S : \Delta$, $\models D : \tau \rightarrow \tau_0$, and $\langle E, L, C, S, D \rangle \rightarrow^* v$, then $\models v : \tau_0$. 

---

**Fig. 1** An operational semantics for ZINC machine.

**Fig. 2** Typing rules for instructions.
Value Typing:
- \( \vdash n : \text{int} \)
- \( \vdash \text{cln}(E, \text{Grab} : C) : \tau \rightarrow \tau' \) if there is some \( \Gamma \) such that \( \vdash E : \Gamma \) and \( \Gamma \mid C \triangleright \text{Grab} : C : \tau \rightarrow \tau' \).

Environment Typing: (Local stack typing \( \vdash L : \Pi \) and Spine stack typing \( \vdash S : \Delta \) are similarly defined)
- \( \vdash \emptyset : \emptyset \)
- \( \vdash v \cdot E : \tau \cdot \Gamma \) if \( \vdash v : \tau \) and \( \vdash E : \Gamma \)

Dump Stack Typing:
- \( \vdash \emptyset : \tau \rightarrow \tau \) for any \( \tau \)
- \( \vdash (E, L, C, S) : D : \tau \rightarrow \tau_0 \) if \( \vdash E : \Gamma \), \( \vdash L : \Pi \), \( \vdash S : \Delta \), \( \Gamma \mid \Pi \mid C : \Delta \rightarrow \tau' \), and \( \vdash D : \tau' \rightarrow \tau_0 \)

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

The soundness theorem ensures that \( \text{Return} \) 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 \( GL\text{ist}\ Int \) in Sect. 1 is only inhabited by \( \text{Nil} \).

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 \( c : \forall \alpha_1, \ldots, \alpha_l, \tau_1', \ldots, \tau_m' \rightarrow T(\alpha_1, \ldots, \alpha_l) \) 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 \( c : \forall \alpha_1, \ldots, \alpha_l, (\tau_1', \ldots, \tau_m') \rightarrow T(\tau_1, \ldots, \tau_n) \) 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 \( 1 \leq i \leq n \). 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 type; an assignment satisfies a conjunction of equations if and only if every assignment that satisfies each equation also satisfies the whole conjunction. A constraint \( C \) entails a constraint \( C' \) (which we write \( C \models C' \)) if and only if every assignment that satisfies \( C \) satisfies \( C' \)[5],[15].

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

\[
\begin{align*}
\sigma & ::= \alpha \mid \text{int} \mid \sigma \lor \forall \alpha.\sigma \mid T(\sigma_1, \ldots, \sigma_i) \\
e & ::= x \mid \lambda (x : \sigma).e \mid e \mid e \mid \lambda \alpha.e \mid e \mid e \mid e \\
& \mid c(\sigma_1, \ldots, \sigma_n, e_1, \ldots, e_m) \mid \text{case } e \text{ of } \{alt_1, \ldots, alt_n\} \\
alt & ::= c(\alpha_1, \ldots, \alpha_l, x_1, \ldots, x_m) \rightarrow e
\end{align*}
\]

We use the notation \( e(\sigma_1, \ldots, \sigma_n) \) to denote a multiple type application \( e \sigma_1 \cdots \sigma_n \). We regard a tuple type \( (\sigma_1, \ldots, \sigma_n) \) 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 \( C \mid \Gamma \triangleright e : \sigma \) is read as an expression \( e \) of type \( \sigma \) under the type environment \( \Gamma \) when \( C \) has an assignment. Note that \( \Gamma \) 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-
Fig. 5 GADTs for values, spine stacks, and dump stacks.

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

The declaration of $\text{Cls}$ in the generalized algebraic data type hides the type variable $\epsilon$ from $V(\alpha \rightarrow \beta)$. 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 $\sigma$, which is a feature of GADTs [5], [15]. A value of type $V(\sigma)$ must be a constant while a value of type $V(\sigma' \rightarrow \sigma')$ 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 $\text{SpineStk}(\Delta \rightarrow \sigma, \sigma_{\text{dump}})$ specifies the term representation for a spine stack that consists of a sequence of arguments specified by $\Delta$. For example, $\text{SpineStk}(\text{int} \rightarrow \text{int}, \sigma_{\text{dump}})$ can inhabit both $\text{Nil}$ and $\text{Arg}(\text{Con}(1), \text{Nil})$. The code continues to run consuming as many argument values from the spine stack $e_S$ as the length of $\Delta$, and it produces another value of type $V(\sigma')$. This value will be passed to the next saved context on the smaller dump stack $e_D$.

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, $\text{Grab}$

\[
\forall a, b, c, d. (V(a, c) \rightarrow \text{unit} \rightarrow \text{SpineStk}(b, d) \rightarrow \delta \rightarrow \text{unit}) \\
\rightarrow e \rightarrow \text{unit} \rightarrow \text{SpineStk}(a \rightarrow \beta, \delta) \rightarrow \delta \rightarrow \text{unit}
\]

that takes a return value of type $V(\sigma)$ under an environment $e_E$ of type $\epsilon$ and a local stack $e_L$ of type $\iota$. The code continues to run consuming as many argument values from the spine stack $e_S$ as the length of $\Delta$, 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 $\sigma_{\text{dump}}$.

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 \textit{Grab} in Fig. 6.

\textit{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:

\[ (I) \quad \frac{\Gamma \vdash \Pi \ni C : \tau'}{\Gamma \vdash \Pi \ni \lambda \epsilon . C : \tau} \]

which can be read as a transformation of a machine state by \((\Gamma, \Pi, \epsilon)\) into another by \((\Gamma', \Pi', \epsilon')\).

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

\[ \forall \delta. (\forall \delta'. \sigma_{\Gamma} \rightarrow \sigma_{\Pi} \rightarrow \text{SpineStk}(\tau', \delta') \rightarrow \delta' \rightarrow \text{unit}) \rightarrow \sigma_{\Gamma} \rightarrow \sigma_{\Pi} \rightarrow \text{SpineStk}(\tau, \delta) \rightarrow \delta \rightarrow \text{unit} \]

where \( \sigma_{\Gamma}, \sigma_{\Pi}, \sigma_{\Pi}, \) and \( \sigma_{\Pi} \) are the types of lambda terms for environments of type \( \Gamma \) and \( \Pi \) and local stacks of type \( \Pi \) and \( \Pi' \), respectively.

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

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

\textit{Return} is more complicated. The type of the instruction may be written as

\[ \forall \delta. e \rightarrow (V \alpha, \epsilon) \rightarrow \text{SpineStk}(\alpha, \delta) \rightarrow \delta \rightarrow \text{unit} \]

The value type and the spine stack type have the same parameter type \( \alpha \). By this, the notion of GADTs enforces the invariant of the non-uniform behavior of the instruction. When \( \alpha \) is substituted with \( \text{int} \), only the case alternative with \( \text{Nil} \) is allowed. When \( \alpha \) 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 \( \alpha \) in \( V \alpha \) becomes substituted with \( \text{int} \). This implies that the spine stack type becomes \( \text{SpineStk}(\text{int}, \delta) \), which statically disallows to use the case alternative with \( \text{Arg}(\cdots) \) in \textit{Return} due to the parameter type not unifiable with \text{int}.

When the return value is a closure, both case alternatives are allowed for use. When the spine stack \( S \) specified by \( \text{SpineStk}(\alpha, \delta) \) is \( \text{Arg}(V', S') \) (or \( \text{Arg}(\alpha_1, \beta_1, \delta_1, V', S') \)), this yields the type constraints \( \alpha = \alpha_1 \rightarrow \beta_1, \delta = \delta_1 \). The return value \( V \) on the local stack is declared to be of type \( V(\alpha) \). Under the type constraints, \( V \) is of type \( V(\alpha_1 \rightarrow \beta_1) \), and is a closure denoted by \( \text{Cls}(I_1, E_1) \) for some term variables \( I_1 \) and \( E_1 \). Then we can safely apply the code \( I_1 \) of the closure to the argument \( V' \), as in \( \text{"case } V \text{ of } \text{Cls}(I_1, E_1) \rightarrow I_1 \text{, } E_1 (\alpha) S_1 D_1 \text{"} \). Note that the \( V \) in the case expression can never be of the form \( \text{Cont}(\cdots) \), which will lead the machine to get stuck, because every case analysis on \( S \) against \( \text{Arg}(V', S') \) yields the type constraint to make \( \alpha \) be unifiable with a function type.

The term \( \text{\textit{Reduce} } e_1 \) is intended to reduce \( e_1 \) producing a value, with which we then continue to reduce \( e_k \). For the reduction of \( e_1 \), we set up a local stack and a spine stack to be empty, saving the current context with \( e_k \) on the dump stack, as this: \( \text{\textit{Reduce} } e_1 e_k \rightarrow e_1 e_E \rightarrow e_1 e_E e_S e_D \rightarrow e_1 e_E \rightarrow e_1 e_E e_S e_D \rightarrow \text{Nil} \). Nil returns the reverse by analyzing the spine stack \( S \) of type \( \text{SpineStk}(\alpha, \beta, \delta) \), and \( \text{Nil}(e_1, e_E, e_S, e_D) \) is of type \( \text{DumpStk}(\alpha, \beta, \delta) \) for some types \( \alpha, \beta, \) and \( \delta \). The spine and dump stacks \( (e_S \) and \( e_D) \) in the saved context are of type \( \text{SpineStk}(\beta, \delta) \) and \( \delta \), respectively.

The definitions of \( \text{Access}(i) \), \( \text{Int} \), and \( \text{Add} \) are straightforward. Note that we assume to have a separate definition of \( \text{Access}(i) \) for each \( i \). \( \text{Halt} \) 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.

\textbf{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 \( \sigma \leadsto V(\sigma) \). An environment type correspondence is either \( \emptyset \leadsto \text{unit} \) or \( \tau \leadsto (\sigma, \sigma') \) if \( \tau \leadsto \sigma \) and \( \Gamma \leadsto \sigma' \). A local stack type correspondence is similarly defined.

We write a compilation judgment as \( \Gamma \ni C : \tau \leadsto e \) where \( C \) is compiled to \( e \) under an environment type \( \Gamma \) and a local stack type \( \Pi \) by the algorithm given in Fig. 7. The compilation algorithm preserves typing as follows.

\textbf{Theorem 3} (Typed Compilation of the ZINC Instructions): Suppose we have \( \Gamma \leadsto \sigma_e \) and \( \Pi \leadsto \sigma_I \). If \( \Gamma \ni C : \tau \leadsto e \) then \( \emptyset \ni \emptyset \ni e : \forall \delta. \sigma_e \rightarrow \sigma_I \rightarrow \text{SpineStk}(\tau, \delta) \rightarrow \delta \rightarrow \text{unit} \).
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.

Theorem 4 (Semantic Correctness): Suppose

- $\Gamma \vdash \sigma : \tau \rightarrow e$
- $\Gamma \vdash E : \sigma_E$
- $\Gamma \vdash L : \sigma_L$
- $\Gamma, \sigma_D \vdash S : \Delta \rightarrow e_S : \text{SpineStk}(\Delta \rightarrow \tau, \sigma_D)$
- $\Gamma \vdash D : \tau \Rightarrow t_0 \Rightarrow e_D : \sigma_D$

If $(E, L, C, S, D) \rightarrow^* v$ then $e[\sigma_D] e_E e_L e_S e_D \rightarrow^\beta \text{Half}[\tau](e_v)$ such that $\models v : \tau \rightarrow e_v : V(\tau)$.

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 $\beta$ reducing the compiled term.

Value correspondence:

- $\models n : \text{Int} \rightarrow \text{Con}(n) : V(\text{Int})$
- $\models \text{cls}(E, C) : \tau \rightarrow \tau' \rightarrow \text{Class}(\tau, \tau, \sigma_G) (e_G, e_C) : V(\tau \rightarrow \tau')$

- if there is some $\Gamma$ such that $\models E : \Gamma \rightarrow e_G : \sigma_G$ and $\Gamma \vdash \emptyset : \tau \rightarrow \tau' \rightarrow e_C$

Environment correspondence: (Local stack correspondence is similarly defined)

- $\models \emptyset : \emptyset \rightarrow () : \text{unit}$
- $\models v : \tau : \Gamma \rightarrow e_v : (e_v, \sigma_g)$

- if $\models v : \tau \rightarrow e_v : \sigma_g$ and $\models E : \Gamma \rightarrow e_G : \sigma_G$

Spine stack correspondence:

- $\tau, \sigma_D \models \emptyset : \emptyset \Rightarrow \text{Nil}(\tau, \tau', \sigma') : \text{SpineStk}(\tau, \sigma_D)$ where $\sigma_D = \text{DumpStk}(\tau, \tau', \sigma')$

- $\tau, \sigma \models v : \Delta \Rightarrow \text{Arg}(\tau_1, \Delta \rightarrow \tau, \sigma, e_v, e_G) : \text{SpineStk}(\tau_1 : \Delta \rightarrow \tau, \sigma)$

- if $\models v : \tau_1 \Rightarrow e_v : V(\tau_1)$ and $\tau, \sigma \models S : \Delta \Rightarrow e_S : \text{SpineStk}(\Delta \rightarrow \tau, \sigma)$

Dump stack correspondence:

- $\models \emptyset : \emptyset \Rightarrow \tau \Rightarrow \text{Nil}(\tau) : \text{DumpStk}(\tau, \text{unit, unit})$

- $\models E, L, C, S : D : \tau_1 \Rightarrow t_0 \Rightarrow \text{Ret}(\tau_1, \Delta \rightarrow t_2, e_G, e_L, e_S, e_C[\sigma], e_E, e_D) : \text{DumpStk}(\tau_1, \Delta \rightarrow t_2, \sigma)$

- if $\models E : \Gamma \Rightarrow e_G : \sigma_G$ and $\models L : \Pi \Rightarrow e_L : e_G$ and $\Gamma \vdash \tau_1 : \Pi \rightarrow e_\sigma : e_D$

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 $\text{Grab}$ 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 \( \text{Arg}(V,Nil) \) precisely by \( \text{SpineStk}(\text{int} \rightarrow \text{int}, \text{DumpStk}(\text{int}, \beta, \delta)) \) for some types \( \beta \) and \( \delta \). The first parameter type \( \text{int} \) of the dump stack type is obtained from removing one argument type in the spine of the function type \( \text{int} \rightarrow \text{int} \). 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 \text{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


Appendix: Proofs

Theorem 1 (Soundness): If \( \Gamma \mid \Pi \vdash C : \Delta \rightarrow \tau, \models E : \Gamma \), \( \models L : \Pi, \models S : \Delta, \models D : \tau \Rightarrow \tau_0 \), and \((E,L,C,S,D) \upharpoonright v\), then \( \models v : \tau_0 \).

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

\( C = \text{return} \) By the rule (Return), \( \Gamma \mid \Pi \vdash \Delta \rightarrow \tau, \models E : \Gamma \), \( \models L : \Pi, \models S : \Delta, \models D \Rightarrow \tau_0 \), and \((E,L,C,S,D) \upharpoonright v \), then \( \models v : \tau_0 \).

Proof 2: The proof is by induction on the length of the execution steps.

\( C = \text{arg} \) By the rule (Arg), \( \Gamma \mid \Pi \vdash \Delta \rightarrow \tau, \models E : \Gamma \), \( \models L : \Pi, \models S : \Delta, \models D \Rightarrow \tau_0 \), and \((E,L,C,S,D) \upharpoonright v \), then \( \models v : \tau_0 \).

Proof 3: The proof is by induction on the length of the execution steps.

\( C = \text{if} \) By the rule (If), \( \Gamma \mid \Pi \vdash \Delta \rightarrow \tau, \models E : \Gamma \), \( \models L : \Pi, \models S : \Delta, \models D \Rightarrow \tau_0 \), and \((E,L,C,S,D) \upharpoonright v \), then \( \models v : \tau_0 \).

Proof 4: The proof is by induction on the length of the execution steps.

\( C = \text{while} \) By the rule (While), \( \Gamma \mid \Pi \vdash \Delta \rightarrow \tau, \models E : \Gamma \), \( \models L : \Pi, \models S : \Delta, \models D \Rightarrow \tau_0 \), and \((E,L,C,S,D) \upharpoonright v \), then \( \models v : \tau_0 \).

Proof 5: The proof is by induction on the length of the execution steps.

\( C = \text{case} \) By the rule (Case), \( \Gamma \mid \Pi \vdash \Delta \rightarrow \tau, \models E : \Gamma \), \( \models L : \Pi, \models S : \Delta, \models D \Rightarrow \tau_0 \), and \((E,L,C,S,D) \upharpoonright v \), then \( \models v : \tau_0 \).

Proof 6: The proof is by induction on the length of the execution steps.

\( C = \text{return} \) By the rule (Return), \( \Gamma \mid \Pi \vdash \Delta \rightarrow \tau, \models E : \Gamma \), \( \models L : \Pi, \models S : \Delta, \models D \Rightarrow \tau_0 \), and \((E,L,C,S,D) \upharpoonright v \), then \( \models v : \tau_0 \).

Proof 7: The proof is by induction on the length of the execution steps.

\( C = \text{arg} \) By the rule (Arg), \( \Gamma \mid \Pi \vdash \Delta \rightarrow \tau, \models E : \Gamma \), \( \models L : \Pi, \models S : \Delta, \models D \Rightarrow \tau_0 \), and \((E,L,C,S,D) \upharpoonright v \), then \( \models v : \tau_0 \).

Proof 8: The proof is by induction on the length of the execution steps.

\( C = \text{if} \) By the rule (If), \( \Gamma \mid \Pi \vdash \Delta \rightarrow \tau, \models E : \Gamma \), \( \models L : \Pi, \models S : \Delta, \models D \Rightarrow \tau_0 \), and \((E,L,C,S,D) \upharpoonright v \), then \( \models v : \tau_0 \).

Proof 9: The proof is by induction on the length of the execution steps.

\( C = \text{while} \) By the rule (While), \( \Gamma \mid \Pi \vdash \Delta \rightarrow \tau, \models E : \Gamma \), \( \models L : \Pi, \models S : \Delta, \models D \Rightarrow \tau_0 \), and \((E,L,C,S,D) \upharpoonright v \), then \( \models v : \tau_0 \).

Proof 10: The proof is by induction on the length of the execution steps.

\( C = \text{case} \) By the rule (Case), \( \Gamma \mid \Pi \vdash \Delta \rightarrow \tau, \models E : \Gamma \), \( \models L : \Pi, \models S : \Delta, \models D \Rightarrow \tau_0 \), and \((E,L,C,S,D) \upharpoonright v \), then \( \models v : \tau_0 \).
and $\Delta' \rightarrow \tau = \tau_2$. $(E, L, C, S, D) = (E, 0, \text{Grab} \cdot C', \text{varg} \cdot S', D) \rightarrow (\text{varg} \cdot E, 0, C', S', D')$ by the operational semantics. By environment typing, $\vdash \text{varg} \cdot E : \text{varg} \cdot \Gamma$. 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 $\vdash \text{cls}(E, \text{Grab} \cdot C') : \tau_1 \rightarrow \tau_2$ due to the environment typing and the instruction typing. $(E, L, C, S, D) = (E, 0, \text{Grab} \cdot C', 0, (E_0, L_0, C_0, S_0) \cdot D') \rightarrow (E, \text{cls}(E, \text{Grab} \cdot C') \cdot L_0, C_0, S_0, D')$ by the operational semantics. By dump stack typing, $\vdash E_0 : \Gamma_0, \vdash L_0 : \Pi_0$, $\vdash S_0 : \Delta_0, \Gamma | \tau \cdot \Pi_0 \triangleright C_0 : \Delta_0 \rightarrow \tau''$, and $\vdash D_0 : \tau'' \rightarrow \tau_0$. Since $\vdash \text{cls}(E, \text{Grab} \cdot C') : L_0 : \tau \cdot \Pi_0$, the theorem is true by induction hypothesis.

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

$(E, L, C, S, D) = (E, 0, \text{Grab} \cdot C', 0, \emptyset) \rightarrow \text{cls}(E, \text{Grab} \cdot C')$ by the operational semantics. By the instruction typing, $\Gamma | \Pi \triangleright \text{Grab} \cdot C' : \tau_1 \rightarrow \tau_2$. Due to $\Delta = \emptyset$ and $\Delta \rightarrow \tau = \tau_1 \rightarrow \tau_2$, $\tau = \tau_1 \rightarrow \tau_2$. By dump stack typing, $\tau_0$. Since $\vdash \text{cls}(E, \text{Grab} \cdot C') : C : \tau_0$.

$C = \text{Access}(j) \cdot C'$. By (Access), $\Gamma = \{\tau_i = \tau_j \mid \tau_i \cdot \tau_j \cdot \Pi' \triangleright C' \cdot \Delta \rightarrow \tau \cdot \Pi \triangleright C : \Delta \rightarrow \tau \cdot \Pi \triangleright C$.

By environment typing, $E(i) = v_i$ such that $\vdash v_i : \tau_i$. By local stack typing, $\vdash v_i : L : \tau_i$.

$(E, L, \text{Access}(j) \cdot C) \cdot (S, D) = (E \mid v_i = v_i, L, \text{Access}(j) \cdot C, S, D) \rightarrow (E, v_i \cdot L, C, S, D)$ by the operational semantics. Hence, the theorem holds by induction hypothesis.

$C = \text{Push}(\tau) \cdot C'$. By (Push), $\Pi' \triangleright \tau' \cdot \Pi'$ and $\Gamma | \Pi' \triangleright C' : \tau' \rightarrow \Delta \cdot \Pi \triangleright C : \Delta \rightarrow \tau \cdot \Pi \triangleright C$.

By spine stack typing, $\vdash v' : \tau'$ and $\vdash v' : L : \tau'$. By spine stack typing, $\tau' \rightarrow \Delta \rightarrow \tau = \tau' \cdot \Delta \rightarrow \tau$. $(E, v' \cdot L, \text{Push}(\tau) \cdot C, S, D) \rightarrow (E, L, C, S, D)$ by the operational semantics. Hence, the theorem holds by induction hypothesis.

$C = \text{Reduce}(C_1) \cdot C_2$. By (Reduce), $\Gamma | \Pi \triangleright C_1 : \tau_1$ and $\Gamma | \tau_1 : \Pi \triangleright C_2 : \Delta \rightarrow \tau$. $(E, L, \text{Reduce}(C_1) \cdot C_2, S, D) \rightarrow (E, 0, C_1, 0, (E, L, C_2, S) \cdot D)$ by the operational semantics. By dump stack typing, $\vdash (E, L, C_2, S) \cdot D : \tau_1 \rightarrow \tau_0$. Since $\tau_1 = 0 \rightarrow \tau_0$ and $\vdash \emptyset$ for the empty spine stack, the theorem holds due to induction hypothesis.

The cases with $\text{Int}(n)$ and $\text{Add}$ 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 $\emptyset | \Gamma_0 \triangleright \text{case} S$ of $\text{Arg}(\cdot \rightarrow \cdot \rightarrow \cdot) \rightarrow \cdot, \text{Nil}(\cdot \rightarrow \cdot : \text{unit})$ for some $\Gamma_0$ such that $\Gamma_0(S) = \text{SpineStk}(\alpha, \beta, \delta)$ and $\Gamma_0(D) = \delta$.

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

ii) With $\text{Nil}(\alpha_1, \beta_1, \delta_1) \rightarrow \cdot$, the typing yields $\{\alpha_1 = \alpha \rightarrow \beta, \text{DdumpStk}(1, \beta_1, \delta_1) = \delta\}$. This type equation leads the subsequent case expression to type safely decompose $D$.

ii) With $\text{Ret}(\alpha_2, \beta_2, \tau_2, \tau_2, \delta_2, \delta_2)$, the typing yields $\{\alpha_2 = \alpha_1, \beta_2 = \beta_1, \delta_2 = \delta_1\}$. This enables us to have equations $V(\alpha_2) = V(\alpha_1) = V(\beta_1)$. By value typing, $\text{Clo}[\alpha, \beta, \delta, \text{Nil}(\cdot \rightarrow \cdot \rightarrow \cdot)](\cdot, E)$ has type $V(\alpha) \rightarrow \beta$. Hence, the corresponding match expression can be proved to be typeable.

ii) With $\text{Nil}(\alpha_2) \rightarrow \cdot$, the corresponding match expression can be shown to be typeable whatever the typing on the pattern yields.

Case Return. One can derive $\emptyset | \Gamma_0 \triangleright \text{case} S$ of $\text{Arg}(\cdot \rightarrow \cdot \rightarrow \cdot) \rightarrow \cdot, \text{Nil}(\cdot \rightarrow \cdot : \text{unit})$ for some $\Gamma_0$ such that $\Gamma_0(S) = \text{SpineStk}(\alpha, \beta, \delta)$ and $\Gamma_0(D) = \delta$.

i) With $\text{Arg}(\alpha, \beta_1, \delta_1, V, S') \rightarrow \cdot$, the typing yields a set of type equations $\{\alpha_1 = \alpha_1, \beta_1 = \alpha, \delta_1 = \beta_1\}$. 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 $\text{Nil}(a_1, b_1, d_1) \to \cdots$, the typing yields $(a_1 = \alpha, \text{DumpStk}(a_1, b_1, d_1) = \delta)$. The rest of the typing is very similar to the case of $\text{Grab}$. The term can be proved to be typeable.

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

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

Case $\text{Access}(\ell)$, $\text{Int}$, and $\text{Add}$. These cases can be trivially proved to be true.

**Theorem 3** (Typed Compilation of the ZINC Instructions):
Suppose we have $\Gamma \sim \sigma_\ell$, and $\Pi \sim \sigma_I$. If $\Gamma \mid \Pi \triangleright C : t \sim e$ then $\emptyset \mid \emptyset \triangleright e : \forall \delta . \sigma_\epsilon \rightarrow \text{SpineStk}(\tau, \delta) \rightarrow \delta \rightarrow \text{unit}$.

**Proof 3:** The proof is by induction on the height of type derivation on $C$. Base case. Case Return. By (C-Return), $\Pi = \tau \cdot \Pi^\prime$ and $\Pi^\prime \sim \sigma^\prime_I$. Since Return has type $\forall \alpha . \forall \epsilon . \forall \delta . \forall \epsilon . e \rightarrow \text{SpineStk}(\alpha, \delta) \rightarrow \delta \rightarrow \text{unit}$, one can conclude that $\Delta \cdot \text{Return}(\tau, \sigma_\epsilon, \sigma^\prime_I, \delta)$ has type $\forall \delta . \sigma_\epsilon \rightarrow \text{SpineStk}(\tau, \delta) \rightarrow \delta \rightarrow \text{unit}$. By $\sigma_\epsilon = V(\tau) \cdot \sigma^\prime_I$, the theorem is true.

Inductive cases. Case $\text{Grab} \cdot C'$. By assumption, there exist $\tau_1$ and $\tau_2$ such that $\tau_1 \rightarrow \tau_2 = \tau$. We have $\Pi = \emptyset$, which corresponds to $\sigma_\ell = \text{unit}$. By induction, $C'$ is compiled to an expression $e$ which has type $\forall \delta . \text{SpineStk}(\tau_1, \delta) \rightarrow \delta \rightarrow \text{unit}$. The expression of interest, $\Delta \cdot \text{Grab}(\tau_1, \tau_2, \sigma_\epsilon, \sigma_I, \delta)$, is of type $\forall \delta . \sigma_\epsilon \rightarrow \text{SpineStk}(\tau_1, \tau_2, \delta) \rightarrow \delta \rightarrow \text{unit}$, by (G-Abs) and (G-TAbs). Therefore, the theorem holds.

Case $\text{Reduce}(C_1) \cdot C_2$. By assumption, we have $\Gamma \mid \emptyset \triangleright C_1 : t_0 \sim e_1$ and $\Gamma \mid t_0 \triangleright \Pi \triangleright C_2 : t \sim e_2$. By induction, $e_1$ has type $\forall \delta . \text{SpineStk}(\tau_1, \delta) \rightarrow \delta \rightarrow \text{unit}$, and $e_2$ has type $\forall \delta . \text{SpineStk}(t_0) \rightarrow \delta$. For some type variables $\alpha, \beta, \delta$, the instantiation of $\delta$ with $\text{DumpStk}(\alpha, \beta, \delta)$ in the type of $e_1$ results in $\forall \delta' . \text{SpineStk}(t_1, \text{DumpStk}(\alpha, \beta, \delta)) \rightarrow \delta$. By the operational semantics, $\Delta \cdot \text{Reduce}(t_0, \tau, \sigma_\epsilon, \sigma_I, \delta)$, has type $\forall \delta . \sigma_\epsilon \rightarrow \text{SpineStk}(\tau, \delta) \rightarrow \delta \rightarrow \text{unit}$, by (G-Abs) and (G-TAbs).

Case $\text{Push} \cdot C'$. By assumption, there exist $t_0$ and $\Pi_0$ such that $\Pi = t_0 \cdot \Pi_0$. By assumption, $\Pi_0 \sim \sigma^\prime_I$. By induction, $C'$ is compiled to $e'$ which has type $\forall \delta . \sigma_\epsilon \rightarrow \Delta \cdot \text{SpineStk}(t_0, \tau, \delta) \rightarrow \delta \rightarrow \text{unit}$. The expression of interest, $\Delta \cdot \text{Push}(t_0, \tau, \sigma_\epsilon, \sigma_I, \delta)$, has type $\forall \delta . \text{SpineStk}(t_0) \rightarrow \delta \rightarrow \text{unit}$, by (G-Abs) and (G-TAbs).

Case $\text{Access}(\ell)$, $\text{Int}$, and $\text{Add}$. These cases can be similarly proved.

**Theorem 4** (Semantic Correctness): Suppose $\Gamma \mid \Pi \triangleright C : \Delta \rightarrow \tau \sim e$, $e \equiv E : \Gamma \rightarrow \epsilon : \sigma_E$, $L : \Pi \sim \epsilon : \sigma_L$, $t_0 : \Delta \sim e : \text{SpineStk}(\Delta \rightarrow \tau, \sigma_D)$, $D \rightarrow \tau \rightarrow t_0 \rightarrow \epsilon : \sigma_D$. If $(E, L, C, S, D) \rightarrow v$ then $\epsilon[\sigma_D] \rightarrow \epsilon \rightarrow \text{halt}(\tau) \rightarrow (e \rightarrow \epsilon : \text{V}(\tau))$.

**Proof 4:** The proof is by induction on the length of execution steps. Case $C = \text{Grab} \cdot C'$ and $S = \epsilon' \cdot S'$. By the assumptions, $\tau, \sigma_D \rightarrow \epsilon' \cdot (\Delta') \rightarrow \tau \rightarrow \text{SpineStk}(\Delta \rightarrow \tau, \sigma_D)$. Due to $\epsilon' = \Delta' \rightarrow \tau \sim (\Delta') \rightarrow \tau$, the local stack correspondence implies $\epsilon'$ is a closure such that $\tau \rightarrow \epsilon'$. The rest of the typing is very similar to the case of $\text{Grab}$. The term can be proved to be typeable.
(Δ' → τ) ∼ CIs[τ', Δ' → τ, σ_E, E_{C_0}] : V(τ' → (Δ' → τ)).

Moreover, the local stack correspondence implies
\[ E_0; \Gamma_0 \leadsto e_{E_0} : σ_{E_0} \text{ and } \Gamma_0; \emptyset \triangleright C_0 : τ' \leadsto (Δ' → τ) \leadsto e_{C_0}. \]

(E, L, C, S, D) = (E, Cl(os(E_0, C_0), L', Return, v' : S', D') → (E_0, 0, C_0, v, S', D').

By the assumed correspondences, Return(τ' → Δ', τ, σ_E, τ, σ_D) e_E e_L e_S e_D reduces to e_{C_0}[σ_{D_0}] e_E e_L e_S e_D.

Case C = Return(0). By the assumed conditions for the compilation, \( \Gamma; \emptyset \triangleright Return(0) : \Delta \rightarrow τ \leadsto e \) where \( e = \Lambda_0 \text{Return}(τ', Δ, τ, σ_E, σ_D, δ) e_0 e_1, \) \( \Gamma; \emptyset \triangleright C_0 : τ' \leadsto e_0, \) and \( \Gamma; τ' \triangleright C_1 : Δ \leadsto τ \leadsto e_1. \) By the assumed conditions for the correspondences, \( \Gamma; \emptyset \triangleright e_0 : σ_E, \Gamma; \emptyset \triangleright L : \Pi \triangleright e_L : σ_L, \) \( \Gamma; \emptyset \triangleright C_0 : τ' \leadsto e_0 \) and \( \Gamma; τ' \triangleright C_1 : Δ \leadsto τ \leadsto e_1. \) By the operational semantics, (E, L, Return(0), C_0, C_1, S, D) → (E_0, 0, C_0, 0, (E, L, C_1, S) : D').

By the local stack correspondence, \( \emptyset : τ \triangleright (\cdot) : unit. \) By the local stack correspondence, \( τ', \text{DumpStk}(τ', Δ, τ, σ_D) : (\cdot) \leadsto Nil(τ', Δ, τ, σ_D), \) and \( \text{DumpStk}(τ', Δ, τ, σ_D) : (\cdot) \leadsto Nil(τ', Δ, τ, σ_D). \) 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, Wono. 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.