Available at
www.ComputerScienceWeb.com
POWERED BY SCIENCE DIRECT®

Information Processing Letters 87 (2003) 205–211

Information Processing Letters

www.elsevier.com/locate/ipl

A type system for the push-enter model ☆

Kwanghoon Choi *, Taisook Han

Department of Electrical Engineering & Computer Science, KAIST, 373-1 Guseong-dong, Yuseong-gu, Daejeon 305-701, Republic of Korea

Received 13 February 2002; received in revised form 28 February 2003

Communicated by H. Ganzinger

Keywords: Compilers; Programming languages; Type systems; Higher-order functions; Push-enter model; Typed compilation

1. Introduction

Compiling with typed intermediate languages has many advantages including that the properties of traditional compilation methods can be specified by types and they can be verified automatically in compile-time [5].

The push-enter model is an important class of compilation methods for higher-order functions, as has been demonstrated by the ZINC machine (which is used in the OCaml compiler) [4, §2.2] and the STG machine (which is used in the Glasgow Haskell compiler) [6, §3.2]. The compilation methods dynamically arrange arguments passed among functions, which are called with an arbitrary number of arguments according to the feature of higher-order functions. The eval-apply model is the other class of compilation methods that statically arranges those arguments.

As far as we know, there is no type system for the push-enter model. Existing type systems for the eval-apply model (e.g., one for CPS conversion [3]) never consider such a dynamic property of the push-enter model. Even though the push-enter model has its own advantages [4,6], all typed compilation approaches are obliged to use only the eval-apply model (e.g., as in a type-preserving compilation [5]) in compiling higher-order functions, which are an indispensable language feature.

A type system for the push-enter model should describe the dynamic arrangement of arguments statically. To design such a type system, it is most important to design the type of states; all compilation methods based on the model use separate states in runtime to indicate dynamic argument status. The type must specify the exact argument status that each state indicates.

In this paper, we first develop a simple compilation method based on the push-enter model in Section 2. The compilation method uses its own states to indicate dynamic argument status. In Section 3, we design generic types for the states, and we develop a type system on the basis of these generic types. We then demonstrate how the compilation method is done within our type system. In Section 4, we conclude with a discussion of future works.

☆ This work was supported by the Korea Science and Engineering Foundation (KOSEF) through the Advanced Information Technology Research Center (AITrc).

* Corresponding author.
E-mail addresses: khchoi@cs.kaist.ac.kr (K. Choi), han@cs.kaist.ac.kr (T. Han).

0020-0190/$ – see front matter
doi:10.1016/S0020-0190(03)00282-5

2. Compiling higher-order functions in the push-enter model

A compilation method based on the push-enter model is developed as shown in Fig. 1; we call this *MPS conversion*. The other compilation method based on the eval-apply model, CPS conversion [1], is also shown for comparison. The purpose of both conversions here is to compile away the feature of higher-order functions in source expressions; after conversion, the feature will not appear in target expressions any more.

Markers and demarkers, which are our primitives for the push-enter model, use tags 0 and 1 as states indicating dynamic argument status; the tag 0 indicates that no argument is available and the tag 1 indicates that one argument is available. A *zero-marker* ($0\langle v\rangle$) carries a continuation v. A *one-marker* ($1\langle v, v'\rangle$) carries an argument v and another marker v'. A *return-demarker* (case v of $0\langle x\rangle \to e$) is used to return values to a continuation x. An *argchk-demarker* (case v of $0\langle x\rangle \to e_0$; $1\langle x, x'\rangle \to e_1$) is used to check the existence of an argument and then, if there is any, to take it.

For a non-function value like integers, both conversions compile it to be returned in the same way. The MPS conversion uses a return-demarker for compiling non-function values.

For a function value, the CPS conversion compiles it to be returned in the same way as for a non-function value. The MPS conversion compiles it differently because, in the push-enter model, the existence of an argument, i.e., the tag of markers, determines whether it is returned or not. An argchk-demarker is used for compiling function values to examine if an argument is available (i.e., if the tag of markers is 1). If so, an applying operation with the argument will be implicitly done through the corresponding alternative of the demarker. As a result, the MPS conversion avoids the cost of building closures for the function value, while the CPS conversion suffers from the cost. It will build closures only when no argument turns out to be available. In general, the push-enter model has this advantage over the eval-apply model [4,6].

For an application, the CPS conversion compiles it for an applying operation to be done explicitly after the evaluation of the function and argument expressions. The MPS conversion compiles the argument expression with a zero-marker because the expression is to be evaluated without any other argument, and it compiles the function expression with a one-marker because an argument is now available from the argument expression.

For a variable, both non-function and function values can be bound to the same variable. The CPS conversion compiles a variable to be returned because both kinds of bound values are compiled uniformly to be returned. The MPS conversion compiles them differently as explained, so care must be taken to compile a variable uniformly. In the MPS conversion, every value is uniformly represented by a marker-handler function and the value itself. For a simple presentation, a dummy value (*fn*) is used to fill the second field in the representation of function values since the field is useless; such an untidy thing could be eliminated by using record types and subtyping on them. Based on this representation decision, a variable is compiled uniformly to select a marker-handler function from a pair bound to the variable and to invoke the function with a marker. This allows a faithful applying operation with a function bound to a variable in a tail position.

MPS (push-enter)	CPS (eval-apply)
$[\![x]\!] = \lambda m.\text{let } f = \pi_1(x) \text{ in } f(m)$	$[\![x]\!] = \lambda k.k(x)$
$[\![i]\!] = \text{fix } f(m).\text{case } m \text{ of}$	$[\![i]\!] = \lambda k.k(i)$
$\quad 0\langle k\rangle \to k(\langle f, i\rangle)$	$[\![\lambda x.e]\!] = \lambda k.k(\lambda(x, k').[\![e]\!]\ k')$
$[\![\lambda x.e]\!] = \text{fix } f(m).\text{case } m \text{ of}$	$[\![e_1\ e_2]\!] = \lambda k.[\![e_1]\!]\ k_1$ where
$\quad 0\langle k\rangle \to k(\langle f, fn\rangle)$	$\quad k_1 = \lambda x_1.[\![e_2]\!]\ k_2$
$\quad 1\langle x, m'\rangle \to [\![e]\!]\ m'$	$\quad k_2 = \lambda x_2.x_1(x_2, k)$
$[\![e_1\ e_2]\!] = \lambda m.[\![e_2]\!]\ 0\langle k\rangle$ where	
$\quad k = \lambda x_2.[\![e_1]\!]\ 1\langle x_2, m\rangle$	

Fig. 1. Two compilation methods for higher-order functions.

3. A type system for the push-enter model

3.1. Source and target languages

The source language λ^{F} is a higher-order, typed, and call-by-value language in a manner similar to that in [5]:

types	$t ::= \alpha \mid int \mid t_1 \to t_2 \mid \forall\alpha.t$
annotated terms	$e ::= u^t$
terms	$u ::= x \mid i \mid \lambda(x:t).e \mid e_1\, e_2$ $\mid \Lambda\alpha.e \mid e[t]$
type contexts	$\Delta ::= \alpha_1, \ldots, \alpha_n$
value contexts	$\Gamma ::= x_1 : t_1, \ldots, x_n : t_n$

The static semantics of λ^{F} is shown in Fig. 2 and it consists of the standard inference rules. In the judgments $\Delta; \Gamma \vdash_{\mathrm{F}} e : t$, Δ is a context that contains the free type variables of Γ, e, and t; Γ is a context that assigns types to the free variables of e; and t is the type of e. The judgment $\Delta \vdash_{\mathrm{F}} t$ asserts that type t has no free type variable under Δ. An empty context is denoted by $\emptyset$.

The target language λ^{M} is a *pseudo first-order* and typed language [1,5] with markers and demarkers (see Fig. 3). Each tag in markers has an annotation t. In syntax, a sequence of syntactic objects E is denoted by $\overline{E}$.

The dynamic semantics of λ^{M} in Fig. 4 is defined as a relation of terms ($\Rightarrow$), particularly to show the behavior of markers and demarkers. In λ^{M}, functions do not return values, so function calls are just jumps; "$\to$ *void*" suggests this fact. If control is to be returned to the caller, the caller must pass the callee a continuation function for it to invoke. Execution is completed by the construct $\mathrm{halt}[\tau]v$, giving a result value v of type τ. The construct fix abstracts both type and value variables, and the corresponding $\forall$ and $\to$ types are combined. Later, $\lambda[\overline{\alpha}](\overline{x} : \overline{\tau}).e$, $\lambda(\overline{x} : \overline{\tau}).e$, and $(\overline{\tau}) \to void$ are used when f or $\overline{\alpha}$ is unused in $\mathrm{fix}\, f[\overline{\alpha}](\overline{x} : \overline{\tau}).e$ and $\forall\overline{\alpha}.(\overline{\tau}) \to void$.

The static semantics of λ^{M} will be shown later in Fig. 6. It has quite similar judgments to those in the static semantics of λ^{F}, except that the new judgment $\Delta; \Gamma \vdash_{\mathrm{M}} e$ states no return type since e never returns values.

3.2. Typing markers and demarkers

Let us explain three typing problems. First, it is most important to determine the types of tags 0 and

$$\frac{}{\Delta \vdash_{\mathrm{F}} t}\ (FTV(t) \subseteq \Delta) \qquad \frac{\Delta; \Gamma \vdash_{\mathrm{F}} u : t}{\Delta; \Gamma \vdash_{\mathrm{F}} u^t : t} \qquad \frac{}{\Delta; \Gamma \vdash_{\mathrm{F}} x : t}\ (\Gamma(x) = t) \qquad \frac{}{\Delta; \Gamma \vdash_{\mathrm{F}} i : int}$$

$$\frac{\Delta \vdash_{\mathrm{F}} t_1 \quad \Delta; \Gamma, x : t_1 \vdash_{\mathrm{F}} e : t_2}{\Delta; \Gamma \vdash_{\mathrm{F}} \lambda(x : t_1).e : t_1 \to t_2}\ (x \notin \Gamma) \qquad \frac{\Delta; \Gamma \vdash_{\mathrm{F}} e_1 : t_1 \to t_2 \quad \Delta; \Gamma \vdash_{\mathrm{F}} e_2 : t_1}{\Delta; \Gamma \vdash_{\mathrm{F}} e_1\, e_2 : t_2}$$

$$\frac{\Delta, \alpha; \Gamma \vdash_{\mathrm{F}} e : t}{\Delta; \Gamma \vdash_{\mathrm{F}} \Lambda\alpha.e : \forall\alpha.t}\ (\alpha \notin \Delta) \qquad \frac{\Delta \vdash_{\mathrm{F}} t \quad \Delta; \Gamma \vdash_{\mathrm{F}} e : \forall\alpha.t'}{\Delta; \Gamma \vdash_{\mathrm{F}} e[t] : t'[t/\alpha]}$$

Fig. 2. Static semantics of λ^{F}.

types	$\tau ::= int \mid \forall\overline{\alpha}.(\overline{\tau}) \to void \mid \langle\tau_1, \ldots, \tau_n\rangle \mid \mathrm{m}\, t \mid \mathrm{v}\, t \mid \mathrm{r}\, t \mid \mathrm{c}\, t \mid fn^t$
values	$v ::= x \mid i \mid \mathrm{fix}\, f[\overline{\alpha}](\overline{x} : \overline{\tau}).e \mid \langle v_1, \ldots, v_n\rangle \mid 0^t\langle v\rangle \mid 1^t\langle v_1, v_2\rangle \mid fn^t$
terms	$e ::= \mathrm{let}\ x = v\ \mathrm{in}\ e \mid \mathrm{let}\ x = \pi_i(v)\ \mathrm{in}\ e \mid v[\overline{t}](\overline{v}) \mid \mathrm{halt}[\tau]v$ $\mid\ \mathrm{case}\ v\ \mathrm{of}\ 0\langle k : \tau\rangle \to e$ $\mid\ \mathrm{case}\ v\ \mathrm{of}\ 0\langle k : \tau\rangle \to e_0;\ 1\langle x : \tau', m : \tau''\rangle \to e_1$
type contexts	$\Delta ::= \alpha_1, \ldots, \alpha_n$
value contexts	$\Gamma ::= x_1 : \tau_1, \ldots, x_n : \tau_n$

Fig. 3. Target language λ^{M}.

let $x = v$ in e	$\Rightarrow e[v/x]$
let $x = \pi_i(\langle \ldots, v_i, \ldots \rangle)$ in e	$\Rightarrow e[v_i/x]$
$(\text{fix } f[\overline{\alpha}](\overline{x} : \overline{\tau}).e)[\overline{t}](\overline{v})$	$\Rightarrow e[\overline{t}/\overline{\alpha}][(\text{fix } f \ldots)/f, \overline{v}/\overline{x}]$
case $0^t \langle v \rangle$ of $0\langle k : \tau' \rangle \to e$	$\Rightarrow e[v/k]$
case $0^t \langle v \rangle$ of $0\langle k : \tau' \rangle \to e_0; 1\langle x : \tau'', m : \tau''' \rangle \to e_1$	$\Rightarrow e_0[v/k]$
case $1^t \langle v, v' \rangle$ of $0\langle k : \tau' \rangle \to e_0; 1\langle x : \tau'', m : \tau''' \rangle \to e_1$	$\Rightarrow e_1[v/x, v'/m]$

Fig. 4. Dynamic semantics of λ^{M}.

$$\frac{}{\tau \equiv \tau} \quad \frac{\tau_2 \equiv \tau_1}{\tau_1 \equiv \tau_2} \quad \frac{\tau_1 \equiv \tau_2 \;\; \tau_2 \equiv \tau_3}{\tau_1 \equiv \tau_3}$$

$$\frac{\tau_1 \equiv \tau_1' \;\; \ldots \;\; \tau_n \equiv \tau_n'}{\forall \overline{\alpha}.(\tau_1, \ldots, \tau_n) \to \mathit{void} \equiv \forall \overline{\alpha}.(\tau_1', \ldots, \tau_n') \to \mathit{void}}$$

$$\frac{\tau_1 \equiv \tau_1' \;\; \ldots \;\; \tau_n \equiv \tau_n'}{\langle \tau_1, \ldots, \tau_n \rangle \equiv \langle \tau_1', \ldots, \tau_n' \rangle}$$

$$\frac{}{\mathrm{v}(\mathit{int}) \equiv \mathit{int}} \quad \frac{\mathrm{m}\, t \equiv \tau}{\mathrm{v}(\forall \alpha.t) \equiv \forall \alpha.(\tau \to \mathit{void})} \quad \frac{}{\mathrm{v}(t_1 \to t_2) \equiv \mathit{fn}^{t_1 \to t_2}}$$

$$\frac{\mathrm{m}\, t \equiv \tau_1 \;\; \mathrm{v}\, t \equiv \tau_2}{\mathrm{r}\, t \equiv \langle \tau_1 \to \mathit{void}, \tau_2 \rangle} \quad \frac{\mathrm{r}\, t \equiv \tau}{\mathrm{c}\, t \equiv \tau \to \mathit{void}}$$

Fig. 5. Type equivalence rules.

1 because they substantially affect typing markers and demarkers, which use the tags to control argument passing. Every use of tags has its own type to specify components promised by the tags. For example, each tag 0 in $0\langle v_1 \rangle$ and $0\langle v_2 \rangle$ promises that v_1 and v_2 are continuations, but the type of v_1 obviously does not need to be identical to that of v_2. Therefore, the types of tags must indicate not only the kind of components promised by the tags but also the specific type of the components in each context.

Second, it is difficult to design types of markers. In order to classify markers, it is natural to identify two separate groups of markers since the two kinds of demarkers consume different markers. The first group consists of markers to be consumed only by return-demarkers. Some zero-markers will be in the group. The second group consists of markers to be consumed only by argchk-demarkers. Some other zero-markers and all one-markers will be in the group. However, it is not always possible to determine which group is assigned for all markers. For example, the MPS conversion compiles $(e_1^{\alpha \to \alpha}\, e_2^{\alpha})^{\alpha}$ into $\lambda(m : \tau).[\![e_2^{\alpha}]\!]\, 0^{\alpha}\langle \cdots \rangle$ using a zero-marker. The zero-marker may reach a return-demarker (an argchk-demarker) if α is instantiated with a non-function type (a function type); it will be in the first (the second) group, respectively. It should not be in either group right now, but it will be in some group after α is instantiated. If such a zero-marker was decided to have only type α, it would be unclear what is the type of the continuation in the marker.

Third, markers have recursive types. In $\text{fix } f(m).\ \text{case } m \text{ of } 0\langle k \rangle \to k(\langle f, i \rangle)$, for example, the type of marker m must relate to the type of k since m corresponds to $0\langle k \rangle$, the type of k must relate to the type of f since f is an argument of k, and the type of f must relate to the type of marker m since m is an argument of f. This circular relationship implies recursive marker types [2].

3.3. A type system using nonstandard types

The markers, values, representations, and continuations, which are generated by compiling an expression

$$\frac{}{\Delta \vdash_{\mathrm{M}} \tau}\ (FTV(\tau) \subseteq \Delta) \quad \frac{\Delta; \Gamma \vdash_{\mathrm{M}} v : \tau}{\Delta; \Gamma \vdash_{\mathrm{M}} v : \tau'}\ (\tau \equiv \tau')$$

$$\frac{}{\Delta; \Gamma \vdash_{\mathrm{M}} i : \mathit{int}} \quad \frac{}{\Delta; \Gamma \vdash_{\mathrm{M}} x : \tau}\ (\Gamma(x) = \tau)$$

$$\frac{\Delta; \Gamma \vdash_{\mathrm{M}} v_1 : \tau_1 \ \cdots\ \Delta; \Gamma \vdash_{\mathrm{M}} v_n : \tau_n}{\Delta; \Gamma \vdash_{\mathrm{M}} \langle v_1, \ldots, v_n \rangle : \langle \tau_1, \ldots, \tau_n \rangle} \quad \frac{}{\Delta; \Gamma \vdash_{\mathrm{M}} \mathit{fn}^t : \mathit{fn}^t}$$

$$\frac{\Delta, \overline{\alpha} \vdash_{\mathrm{M}} \tau_i \quad \Delta, \overline{\alpha}; \Gamma, f : \forall \overline{\alpha}.(\overline{\tau}) \to \mathit{void}, \overline{x} : \overline{\tau} \vdash_{\mathrm{M}} e}{\Delta; \Gamma \vdash_{\mathrm{M}} \mathrm{fix}\ f[\overline{\alpha}](\overline{x} : \overline{\tau}).e : \forall \overline{\alpha}.(\overline{\tau}) \to \mathit{void}}\ (\overline{\alpha} \notin \Delta \wedge f, \overline{x} \notin \Gamma)$$

$$\frac{\Delta; \Gamma \vdash_{\mathrm{M}} v : \mathrm{c}\, t}{\Delta; \Gamma \vdash_{\mathrm{M}} 0^t \langle v \rangle : \mathrm{m}\, t} \quad \frac{\Delta; \Gamma \vdash_{\mathrm{M}} v_1 : \mathrm{r}\, t_1 \quad \Delta; \Gamma \vdash_{\mathrm{M}} v_2 : \mathrm{m}\, t_2}{\Delta; \Gamma \vdash_{\mathrm{M}} 1^{t_1 \to t_2} \langle v_1, v_2 \rangle : \mathrm{m}(t_1 \to t_2)}$$

$$\frac{\Delta; \Gamma \vdash_{\mathrm{M}} v : \tau}{\Delta; \Gamma \vdash_{\mathrm{M}} \mathrm{halt}[\tau] v} \quad \frac{\Delta; \Gamma \vdash_{\mathrm{M}} v : \tau \quad \Delta; \Gamma, x : \tau \vdash_{\mathrm{M}} e}{\Delta; \Gamma \vdash_{\mathrm{M}} \mathrm{let}\ x = v\ \mathrm{in}\ e}\ (x \notin \Gamma)$$

$$\frac{\Delta; \Gamma \vdash_{\mathrm{M}} v : \langle \tau_1, \ldots, \tau_n \rangle \quad \Delta; \Gamma, x : \tau_i \vdash_{\mathrm{M}} e}{\Delta; \Gamma \vdash_{\mathrm{M}} \mathrm{let}\ x = \pi_i\, v\ \mathrm{in}\ e}\ (x \notin \Gamma \wedge 1 \leqslant i \leqslant n)$$

$$\frac{\Delta \vdash_{\mathrm{F}} t_i \quad \Delta; \Gamma \vdash_{\mathrm{M}} v : \forall \overline{\alpha}.(\overline{\tau}) \to \mathit{void} \quad \Delta; \Gamma \vdash_{\mathrm{M}} v_i : \tau_i[\overline{t}/\overline{\alpha}]}{\Delta; \Gamma \vdash_{\mathrm{M}} v[\overline{t}](\overline{v})}$$

$$\frac{\begin{array}{c}\Delta \vdash_{\mathrm{M}} \tau \quad \Delta; \Gamma \vdash_{\mathrm{M}} v : \mathrm{m}\, t \\ \Delta; \Gamma, x : \tau \vdash_{\mathrm{M}} x : \mathrm{c}\, t \quad \Delta; \Gamma, x : \tau \vdash_{\mathrm{M}} e\end{array}}{\Delta; \Gamma \vdash_{\mathrm{M}} \mathrm{case}\ v\ \mathrm{of}\ 0\langle x : \tau \rangle \to e}\ (t \in \{\mathit{int}, \forall \alpha.t'\} \wedge x \notin \Gamma)$$

$$\frac{\begin{array}{c}\Delta \vdash_{\mathrm{M}} \tau_i \quad \Delta; \Gamma \vdash_{\mathrm{M}} v : \mathrm{m}\,(t_1 \to t_2) \\ \Delta; \Gamma, x_0 : \tau_0 \vdash_{\mathrm{M}} e_0 \quad \Delta; \Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash_{\mathrm{M}} e_1 \\ \Delta; \Gamma, x_0 : \tau_0 \vdash_{\mathrm{M}} x_0 : \mathrm{c}\,(t_1 \to t_2) \\ \Delta; \Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash_{\mathrm{M}} x_1 : \mathrm{r}\, t_1 \quad \Delta; \Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash_{\mathrm{M}} x_2 : \mathrm{m}\, t_2\end{array}}{\Delta; \Gamma \vdash_{\mathrm{M}} \mathrm{case}\ v\ \mathrm{of}\ 0\langle x_0 : \tau_0 \rangle \to e_0; 1\langle x_1 : \tau_1, x_2 : \tau_2 \rangle \to e_1}\ (x_i \notin \Gamma)$$

Fig. 6. Static semantics of λ^{M}.

of type t through the MPS conversion, are decided to have nonstandard types $\mathrm{m}\, t$, $\mathrm{v}\, t$, $\mathrm{r}\, t$, and $\mathrm{c}\, t$.

A collection of type equivalence rules is defined in Fig. 5. If $\tau_1 \equiv \tau_2$ is derivable from the rules, τ_1 is said to be equivalent to τ_2. The first three rules are the reflexive rule, the symmetric rule, and the transitive rule. The next two rules for $\forall \overline{\alpha}.(\overline{\tau}) \to \mathit{void}$ and $\langle \overline{\tau} \rangle$ define two compound types as equivalent if their corresponding component types are all equivalent. The remaining rules define the intended interpretation for all nonstandard types except $\mathrm{m}\, t$ and $\mathrm{v}\, \alpha$, which are used as they are without any interpretation.

The defined interpretation of nonstandard types results from the MPS conversion. The $\mathrm{v}\, \mathit{int}$ rule, for example, reflects that every integer in λ^{F} is an integer in λ^{M}. The $\mathrm{r}\, t$ rule reflects our decision on representation; the representation of integers is a tuple of a marker-handler function and an integer, as explained in Section 2, and its type is $\langle \mathrm{m}\, \mathit{int} \to \mathit{void}, \mathit{int} \rangle$. By the $\mathrm{c}\, t$ rule, continuations have a function type $\mathrm{r}\, t \to \mathit{void}$, accepting a represented value of type $\mathrm{r}\, t$. For $\mathrm{m}\, t$, no separate rule is defined. Using $\mathrm{m}\, t$ as it is defined by the reflexive rule, enables us to specify implicitly the circular relationship in the third problem of Section 3.2, instead of using standard types $\mu \alpha.\tau$ [2].

The mechanical verification for equivalence between two types is possible by the following theorem. A straightforward algorithm of square complexity and its proofs could be found in [2].

Theorem 1. *There is an algorithm to determine if τ is equivalent to τ'.*

The static semantics of λ^{M} is shown in Fig. 6. First, tags 0 and 1 are decided to have generic types $\forall\alpha.\langle \mathrm{c}\,\alpha\rangle \to \mathrm{m}\,\alpha$ and $\forall\alpha\beta.\langle \mathrm{r}\,\alpha, \mathrm{m}\,\beta\rangle \to \mathrm{m}\,(\alpha \to \beta)$. Second, each use of the tags will have its own instantiated type from the generic types. Third, a *subsumption rule* is introduced to smooth the integration of non-standard types with standard types.

The inference rule for a zero-marker specifies that every zero-marker of type $\mathrm{m}\,t$ must carry a value of type $\mathrm{c}\,t$. This marker tag 0 has type $\langle \mathrm{c}\,t\rangle \to \mathrm{m}\,t$, even when t is α as in the second problem of Section 3.2. The inference rule for a one-marker specifies that every one-marker of type $\mathrm{m}\,(t_1 \to t_2)$ must carry two values of type $\mathrm{r}\,t_1$ and $\mathrm{m}\,t_2$. This marker tag 1 has type $\langle \mathrm{r}\,t_1, \mathrm{m}\,t_2\rangle \to \mathrm{m}\,(t_1 \to t_2)$. The inference rule for a return-demarker allows only markers of type $\mathrm{m}\,t$ for a non-function type t. The used tag 0 has an equivalent type to $\langle \mathrm{c}\,t\rangle \to \mathrm{m}\,t$ for some non-function type t. Note that a one-marker in a return-demarker will violate the inference rule due to its marker type $\mathrm{m}\,t$ where t is a function type. The inference rule for an argchk-demarker allows only markers of type $\mathrm{m}\,(t_1 \to t_2)$. The used tags 0 and 1 have types $\langle \mathrm{c}\,(t_1 \to t_2)\rangle \to \mathrm{m}\,(t_1 \to t_2)$ and $\langle \mathrm{r}\,t_1, \mathrm{m}\,t_2\rangle \to \mathrm{m}\,(t_1 \to t_2)$, respectively. Note that the same argchk-demarker will receive both zero-marker and one-marker if they have the same type $\mathrm{m}\,(t_1 \to t_2)$, which is a static description of the dynamic argument status (i.e., if an argument is available or not) resulting from the two different kinds of markers. By our generic types of tags 0 and 1, the first problem of Section 3.2 is thus solved.

Our subsumption rule allows v to have type τ' whenever its type τ is equivalent to τ'. For example, in "$0\langle k : \mathrm{c}\,t\rangle \to k(v)$", if v has type τ then k must have type $\tau \to \mathit{void}$ because of an application. Although k has type $\mathrm{c}\,t$, this typing will be possible if $\mathrm{c}\,t \equiv \tau \to \mathit{void}$ (see Fig. 7).

The remaining inference rules are quite standard [5] except that a dummy value fn^t is defined as having a corresponding dummy type fn^t.

The following theorem (Type Safety) ensures that well-typed λ^{M} programs will not be *stuck*, where all terminal terms other than $\mathrm{halt}[\tau]v$ are considered stuck, so no one-marker will reach return-demarkers. A detailed proof by showing Subject Reduction and Progress could be found in [2].

Theorem 2. *If $\emptyset; \emptyset \vdash_{\mathrm{M}} e$ then there is no stuck e' s.t. $e \Rightarrow^* e'$.*

Note that Type Safety can be verified by a straightforward implementation of the inference rules because

$$\frac{\dfrac{\Delta; \Gamma, k : \mathrm{c}\,t \vdash_{\mathrm{M}} k : \mathrm{c}\,t}{\Delta; \Gamma, k : \mathrm{c}\,t \vdash_{\mathrm{M}} k : \tau \to \mathit{void}}\ (\mathrm{c}\,t \equiv \tau \to \mathit{void}) \qquad \dfrac{\cdots}{\Delta; \Gamma, k : \mathrm{c}\,t \vdash_{\mathrm{M}} v : \tau}}{\Delta; \Gamma, k : \mathrm{c}\,t \vdash_{\mathrm{M}} k(v)}$$

Fig. 7.

$$\begin{aligned}
[\![x^t]\!] &= \lambda(m : \mathrm{m}\,t).\mathrm{let}\ f = \pi_1(x)\ \mathrm{in}\ f(m)\\
[\![i^t]\!] &= \mathrm{fix}\,f(m : \mathrm{m}\,t).\mathrm{case}\ m\ \mathrm{of}\ 0\langle k : \mathrm{c}\,t\rangle \to k(\langle f, i\rangle)\\
[\![(\lambda(x : t_1).u^{t_2})^t]\!] &= \mathrm{fix}\,f(m : \mathrm{m}\,t).\mathrm{case}\ m\ \mathrm{of}\\
&\qquad 0\langle k : \mathrm{c}\,t\rangle \to k(\langle f, \mathit{fn}^t\rangle)\\
&\qquad 1\langle x : \mathrm{r}\,t_1, m : \mathrm{m}\,t_2\rangle \to [\![u^{t_2}]\!]\ m\\
[\![(u_1^{t_1}\ u_2^{t_2})^t]\!] &= \lambda(m : \mathrm{m}\,t).[\![u_2^{t_2}]\!]\ (0^{t_2}\langle\lambda(x_2 : \mathrm{r}\,t_2).[\![u_1^{t_1}]\!]\ (1^{t_1}\langle x_2, m\rangle)\rangle)\\
[\![(\Lambda\alpha.u^{t'})^t]\!] &= \mathrm{fix}\,f(m : \mathrm{m}\,t).\mathrm{case}\,m\ \mathrm{of}\ 0\langle k : \mathrm{c}\,t\rangle \to k(\langle f, \lambda[\alpha](m' : \mathrm{m}\,t').[\![u^{t'}]\!]\ m'\rangle)\\
[\![(u^{t_1}[t_2])^t]\!] &= \lambda(m : \mathrm{m}\,t).[\![u^{t_1}]\!]\ (0^{t_1}\langle\lambda(x : \mathrm{r}\,t_1).\mathrm{let}\ y = \pi_2(x)\ \mathrm{in}\ y[t_2](m)\rangle)\\
[\![u^t]\!]_{\mathrm{prg}} &= [\![u^t]\!]\ (0^t\langle\lambda(x : \mathrm{r}\,t).\mathrm{halt}[\mathrm{r}\,t]\,x\rangle)
\end{aligned}$$

Fig. 8. A typed MPS conversion.

$$
\begin{aligned}
&\lambda^{\mathrm{F}} : id = (\Lambda\alpha.(\lambda(x:\alpha).x^{\alpha})^{\alpha\to\alpha})^{\forall\alpha.\alpha\to\alpha}\\
&\lambda^{\mathrm{M}} : id = \mathrm{fix}\ f[\alpha](m:\mathrm{m}(\alpha\to\alpha)).\\
&\qquad\qquad \text{case } m \text{ of}\\
&\qquad\qquad\quad 0\langle k:\mathrm{c}(\alpha\to\alpha)\rangle \to \text{let } f' = \lambda(m:\mathrm{m}(\alpha\to\alpha)).f[\alpha](m) \text{ in } k(\langle f', fn^{\alpha\to\alpha}\rangle)\\
&\qquad\qquad\quad 1\langle x:\mathrm{r}\,\alpha, m:\mathrm{m}\,\alpha\rangle \to \text{let } y = \pi_1(x) \text{ in } y(m)
\end{aligned}
$$

Fig. 9. Compiling an identity function.

they can be defined in a syntax-directed manner and their side conditions are verifiable by our algorithm.

Now, we demonstrate how the MPS conversion is done within our type system by developing a typed conversion as shown in Fig. 8. A simple compilation example is shown in Fig. 9. The following theorem ensures that the typed conversion preserves Type Correctness. The proof is by induction on the structure of e in λ^{F}. A detailed proof could be also found in [2].

Theorem 3. *If* $\emptyset;\emptyset\vdash_{\mathrm{F}} e : t$ *then* $\emptyset;\emptyset\vdash_{\mathrm{M}} [\![e]\!]_{\mathrm{prg}}$.

4. Conclusion

We developed a type system for the push-enter model. For a compilation method based on the model using states 0 and 1 to indicate dynamic argument status, we designed generic types $\forall\alpha.\langle \mathrm{c}\,\alpha\rangle \to \mathrm{m}\,\alpha$ and $\forall\alpha\beta.\langle \mathrm{r}\,\alpha, \mathrm{m}\,\beta\rangle \to \mathrm{m}\,(\alpha\to\beta)$ to describe the dynamic argument status. By assigning the generic types to the states, our type system guarantees consistency between states and dynamic argument status; and it has a type checking algorithm.

In further work, it would be possible to apply our idea on the type system to the existing compilation methods based on the push-enter model. We are currently working on a typed compilation for the ZINC machine.

References

[1] A.W. Appel, Compiling with Continuations, Cambridge Univ. Press, Cambridge, MA, 1992.

[2] K. Choi, A type system for the push-enter model, Revised Report of CS-TR-2002-175, Dept. of EE&CS, KAIST, Daejeon, Korea, February 2003.

[3] R. Harper, M. Lillibridge, Explicit polymorphism and CPS conversion, in: The 20th Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages, Charleston, SC, January 1993, pp. 206–219.

[4] X. Leroy, The ZINC experiment: An economical implementation of the ML language, Technical Report 117, INRIA, Rennes, France, 1988.

[5] G. Morrisett, D. Walker, K. Crary, N. Glew, From system F to typed assembly language, ACM Trans. Programm. Languages Systems 21 (3) (1999) 528–569.

[6] S.L. Peyton Jones, Implementing lazy functional languages on stock hardware: the spineless tagless G-machine, J. Funct. Programm. 2 (2) (1992) 127–202.