메모리 타입 분석을 통한 안전하고 효율적인 메모리 재사용*

이욱세 이광근†
한국과학기술원 전자전산학과 / 프로그램 분석 시스템 연구단‡

A Memory Type System for Safe and Efficient Memory Reuse

Oukseh Lee and Kwangkeun Yi
Dept. of EECS / Research On Program Analysis System, KAIST

요 약

메모리 재활용 방법(*garbage collection*)은 안전하고 효율적이지만, 메모리를 재사용하려면 항상 메모리를 수거해야 하는 비용이 든다. 가능하면 메모리 수거없이 즉각적으로 메모리를 재사용하게 함으로써 비용을 줄일 수 있다. 본 논문에서는 실행시간 정보 전달을 통해 효과적으로 메모리를 즉각 재사용할 수 있는 방법을 제시하고, 그러한 메모리 재사용이 안전하다는 것을 증명하는 메모리 타입 시스템을 제시한다. 제시한 방법을 사용하여 프로그램 `sieve`를 28.1% 빠르게 실행할 수 있었다.

1 서론

메모리 재활용 방법(garbage collection)[4]은 안전하고 효율적이다. 수동으로 메모리를 관리할 경우 사용자의 실수로 잘못된 포인터(dangling pointer)를 발생시켜 프로그램이 파행적으로 중단될 수 있다. 그러나, 메모리 재활용 방법은 모든 메모리 접근이 안전하다는 것이 보장된다. 게다가 메모리 재활용 방법이 메모리가 지나치게 작지 않은 경우 수동 메모리 관리 방법보다 효율적이라는 것이 증명되었다 [1]. 그래서 최근 프로그래밍 언어들(SML [6], Java [2])은 메모리 재활용 방법을 채용하고 있다.

그러나, 메모리 재활용 방법에서는 메모리를 재사용하려면 항상 쓰지 않는 메모리를 수거해야 하는 (garbage collection) 비용이 든다. 임시로 값을 메모리에 저장했다가 빠른 시간안에 사용한 후에 필요 없어지는 경우가 많은데, 이렇게 필요 없어진 메모리를 재사용하려면 항상 메모리 수거가 필요하다. 잦은 임시 메모리 할당은 메모리 수거를 자주 발생시키고 결국 전체 실행 속도를 저하시킨다.

메모리 수거를 거치지 않고 즉각적으로 메모리를 재사용하게 함으로써 불필요한 비용을 줄일 수 있다. 이미 제안된 여러 방법론 중에 즉각적인 메모리 재사용 방법이 효과적일 것이라고 예측되었지만 [3] 안전하면서도 뚜렷한 성능 향상을 보이는 방법은 아직 선보이지 않고 있다.

본 논문에서는 실행시간 정보 전달을 통해 효과적으로 메모리를 즉각 재사용할 수 있는 방법을 제시하고, 메모리 재사용이 안전하다는 것을 증명할 수 있는 메모리 타입 시스템을 제시한다.

*본 연구는 과학기술부 창의적 연구진흥사업 추진으로 얻어진 결과임
†E-mail: {cookcu; kwang}@ropas.kaist.ac.kr
‡http://ropas.kaist.ac.kr

2 메모리 재사용 언어

다음을 대상 언어로 한다:

v	::=	$x \mid i \mid \lambda\vec{\delta}.\lambda\vec{x}.e$	변수, 정수, 함수
e	::=	v	값
	$\mid$	$\kappa\,\vec{v} \mid \texttt{case}\, v\, m \cdots m$	데이타 구성자
	$\mid$	$\texttt{let}\, x = e\, \texttt{in}\, e$	let-문
	$\mid$	$v\,[\vec{f}]\,\vec{v}$	함수 호출
	$\mid$	$\kappa\,\vec{v}\,\texttt{at}\,x\,\texttt{when}\,f$	메모리 재사용
m	::=	$\kappa\,\vec{x} \Rightarrow e$	case-문
f	::=	$\text{true} \mid \text{false} \mid \delta \mid f \wedge f \mid f \vee f$	재사용 정보

여기서 벡터 표시는 여러 개를 의미한다. 재사용 명령어 "$\kappa\,\vec{v}\,\texttt{at}\,x\,\texttt{when}\,f$"는 f가 참(true)일 때 x가 가르키는 힙 셀에 $\kappa\vec{v}$를 저장하라는 뜻이다. 단약 f가 거짓(false)이라면 새로운 힙 셀을 할당하고 값을 저장하라는 뜻이다. 재사용 정보 f는 함수 호출간에 전달될 수 있어, 보다 적극적으로 메모리를 재사용할 수 있다. 정확한 의미구조는 그림 1과 같고 타입은 일반적인 단순한 타입 규칙을 따른다고 가정한다.

메모리 타입 분석의 기술을 간단히 하기 위해 다음을 가정한다. (1) 프로그램 내의 모든 변수 이름은 서로 다르다. (2) 모든 함수는 자유 변수(free variable)가 없다. (3) 모든 데이타 구성자(data constructor)는 하나 이상의 인자를 갖는다. (4) 함수는 메모리에 저장되지 않는다. 즉, 데이타 구성자의 인자로 함수가 올 수 없다. (5) 모든 데이타 구성자와 그의 인자들이 저장되는 메모리의 크기는 같다.

3 메모리 타입

요약 수준이 다른 두 메모리 타입을 통해 분석한다 (그림 2). 가능한 구체적인 메모리 타입 내에서 메모리를 분석하다가 정확한 분석이 불가능할 경우 요약하여 분석한다. 이렇게 요약된 메모리 타입은 필요에 의해 다시 구체화될 수 있다.

$$
\begin{array}{rrcl}
\textit{Addr} & a & & \\
\textit{Value} & u & ::= & i \mid a \mid \lambda\vec{\delta}.\lambda\vec{x}.e \\
\textit{Cell} & c & ::= & \kappa\,\vec{u} \\
\textit{Heap} & H & \in & \textit{Addr} \rightharpoonup \textit{Cell} \\
\textit{Dump} & D & \in & (\textit{Var} \times \textit{Expr})^{\leq k} \\
\textit{State} & & : & \textit{Expr} \times \textit{Heap} \times \textit{Dump}
\end{array}
$$

$$
\begin{array}{ll}
(\kappa\,\vec{u},\ H,\ D) & \rightarrow (a,\ H \cup \{a \mapsto \kappa\,\vec{u}\},\ D) \\
 & \quad \text{where } a \notin \mathrm{dom}(H) \\
(\kappa\,\vec{u}\ \texttt{at}\ a\ \texttt{when}\ f,\ H \cup \{a \mapsto c\},\ D) & \rightarrow (a',\ H \cup \{a' \mapsto \kappa\,\vec{u}\},\ D) \\
\quad \text{when } [\![f]\!] = \text{true and } a' \notin \mathrm{dom}(H) & \\
(\kappa\,\vec{u}\ \texttt{at}\ a\ \texttt{when}\ f,\ H,\ D) & \rightarrow (\kappa\,\vec{u},\ H,\ D) \\
\quad \text{when } [\![f]\!] = \text{false} & \\
(\texttt{case}\ a\ m_1 \cdots m_n,\ H,\ D) & \rightarrow (e[u_i/x_i],\ H,\ D) \\
\quad \text{if } H(a) = \kappa\,\vec{u} & \\
\quad \text{and } m_i = \kappa\,\vec{x} \Rightarrow e & \\
(\texttt{let}\ x = e_1\ \texttt{in}\ e_2,\ H,\ D) & \rightarrow (e_1,\ H,\ (x, e_2) \cdot D) \\
((\lambda\vec{\delta}.\lambda\vec{x}.e)\ [\vec{f}]\ \vec{u},\ H,\ D) & \rightarrow (e[f_i/\delta_i][u_i/x_i],\ H,\ D) \\
(u,\ H,\ (x, e) \cdot D) & \rightarrow (e[u/x],\ H,\ D)
\end{array}
$$

그림 1: 의미 구조 (semantics).

요약할 때 각 힙 셀(heap cell)을 종류별로 분류하고 공유 정보(sharing flag)를 유지함으로써 보다 정확한 구체화를 가능케 한다.

구체적인 메모리 타입은 힙의 구조를 그대로 기술한다. 구체적인 메모리 타입 $\langle L, \kappa, \vec{\mu}\rangle$이 의미하는 바로 도달 가능한 힙 셀들의 이름들은 L이고 그 종류(데이타 구성자)는 κ이며 그 셀이 가지고 있는 원소들의 메모리 타입은 $\vec{\mu}$이다.

요약된 메모리 타입은 도달 가능한 힙 셀을 종류(데이타 구성자)별로 구분한 것이다. 추가적으로 힙 셀이 공유되었는지를 나타내는 논리값($\mathit{Sharing}$)을 유지한다. 여기서 공유란 특정 힙 셀이 요약된 타입 내에서 두 가지 이상의 경로로 접근 가능할 때를 말한다.

함수 타입은 인자들의 메모리 타입과 결과의 메모리 타입, 그리고 함수가 호출되어 실행되면서 사용, 재사용하는 이름들로 구성된다. 사용되는 이름들(U)은 이름들의 집합이고 재사용되는 이름들(R)은 재사용이 선택적이므로 이름에서 재사용 조건으로 가는 맵이다. 즉, l의 재사용 조건이 참이면 무조건 l이 재사용 된다는 것이고 거짓이면 무조건 재사용 안 한다는 것 그리고 δ이면 전달되는 δ에 따라 재사용이 결정된다는 것이다.

4 메모리 타입 시스템

메모리 타입 시스템은 그림 2와 같다. "$\Delta \vdash v : \mu$"는 주어진 메모리 환경 Δ 아래에서 값 v가 메모리 타입 μ를 갖는다는 것이다. "$\Delta \vdash e : \mu, U, R$"은 주어진 메모리 환경 Δ 아래에서 프로그램 식 e는 수행되어 메모리 타입 μ의 값을 갖고, 수행되면서 U를 사용하고 R을 재사용한다는 뜻이다.

변수의 경우 (var) 타입 환경의 메모리 타입을 갖고, 정수일 경우 (int) 정수 타입을 갖는다. 함수일 경우 (fun) 함수 내용(function body)이 갖는 타입을 결과 타입으로 한다. 값의 경우 (value) 메모리를 사용도 재사용도 하지 않는다. 데이타 구성자의 경우 (data) 새로운 구체적인 메모리 타입을 만든다. 메모리 재사용의 경우 (reuse) 데이타 구성자의 경우와 같으나 x가 가르키는 메모리를 재사용한다.

let-문의 경우 (let) 메모리 사용/재사용 패턴이 옳바른지 검증한다. e_2가 e_1 실행 이후에 실행되므로, e_1에서 재사용한 것은 e_2에서 사용 또는 재사용하면 안된다.

$$
\begin{array}{rrcl}
\textit{Loc} & l & & \\
 & L, U & \in & \wp(\textit{Loc}) \\
 & R & \in & \textit{Loc} \rightharpoonup \textit{Flag} \\
\textit{Sharing} & \pi & \in & \textit{Bool} \\
\textit{Const} & \kappa & & \\
\textit{CollapTy} & \bar{\mu} & \in & \textit{Const} \rightharpoonup \wp(\textit{Loc}) \times \textit{Sharing} \\
\textit{MemTy} & \mu & ::= & \mathrm{int} \mid \langle L, \kappa, \vec{\mu}\rangle \mid \bar{\mu} \mid \lambda\vec{\delta}.\vec{\mu} \overset{U,R}{\rightarrow} \mu \\
\textit{MemTyEnv} & \Delta & \in & \textit{Var} \rightharpoonup \textit{MemTy} \\
\textit{Subst} & S & \in & \textit{Loc} \rightharpoonup \wp(\textit{Loc}) \times \textit{Sharing} \\
\textit{Alias} & A & \in & \wp(\textit{Loc} \times \textit{Loc})
\end{array}
$$

$\boxed{\Delta \vdash v : \mu}$

$$\Delta \vdash x : \Delta(x) \quad \text{(var)} \qquad \Delta \vdash i : \mathrm{int} \quad \text{(int)}$$

$$\frac{\Delta \cup \{x_i \mapsto \mu_i\} \vdash e : \mu, U, R}{\Delta \vdash \lambda\vec{\delta}.\lambda\vec{x}.e : \lambda\vec{\delta}.\vec{\mu} \overset{U,R}{\rightarrow} \mu} \quad \text{(fun)}$$

$\boxed{\Delta \vdash e : \mu, U, R}$

$$\frac{\Delta \vdash v : \mu}{\Delta \vdash v : \mu, \emptyset, \emptyset} \quad \text{(value)} \qquad \frac{\forall i.\Delta \vdash v_i : \mu_i}{\Delta \vdash \kappa\,\vec{v} : \langle\{l\}, \kappa, \vec{\mu}\rangle, \{l\}, \emptyset} \quad \text{(data)}$$

$$\frac{\Delta(x) = \langle L, \kappa', \vec{\mu}'\rangle \quad \forall i.\Delta \vdash v_i : \mu_i}{\Delta \vdash \kappa\,\vec{v}\ \texttt{at}\ x\ \texttt{when}\ g : \langle\{l\}, \kappa, \vec{\mu}\rangle, \{l\}, \{l' \mapsto g \mid l' \in L\}} \quad \text{(reuse)}$$

$$\frac{\begin{array}{l}\Delta \vdash e_1 : \mu_1, U_1, R_1 \\ \Delta \cup \{x \mapsto \mu_1\} \vdash e_2 : \mu_2, U_2, R_2 \\ \forall l \in \mathrm{dom}(R_1).[\![R_1(l)]\!] \Rightarrow (l \notin U_2 \wedge \neg[\![R_2(l)]\!])\end{array}}{\Delta \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \mu_2, U_1 \cup U_2, R_1 \vee R_2} \quad \text{(let)}$$

$$\frac{\begin{array}{l}\Delta \vdash v : \lambda\vec{\delta}.\mu_0 \quad \mu_0[\vec{f}/\vec{\delta}] = \vec{\mu} \overset{U,R}{\rightarrow} \mu \quad \Delta \vdash v_i : S\mu_i \\ \{l_1 \sim l_2 \mid l_1, l_2 \in \mathrm{dom}(S),\ S(l_1) \cap S(l_2) \neq \emptyset\} \rhd U, R\end{array}}{\Delta \vdash v\ [\vec{f}]\ \vec{v} : S\mu, SU, SR} \quad \text{(app)}$$

$$\frac{\begin{array}{l}\Delta \vdash y : \langle L, \kappa, \vec{\mu}\rangle \quad m_j = (\kappa\,\vec{x} \Rightarrow e) \\ \Delta \cup \{x_i \mapsto \mu_i\} \vdash e : \mu, U, R\end{array}}{\Delta \vdash \texttt{case}\ y\ m_1 \cdots m_n : \mu, U \cup L, R} \quad \text{(case)}$$

$$\frac{\begin{array}{l}\forall \kappa \in \mathrm{kind}(\mathrm{typeof}(x)). \\ \quad (\mu'_\kappa, S_\kappa) = \gamma_\kappa(\bar{\mu}) \\ \quad \Delta \cup \{x \mapsto \mu'_\kappa\} \vdash e : \mu_\kappa, U_\kappa, R_\kappa \\ \quad \{l \sim l',\ l' \sim l \mid (l \mapsto L^\pi) \in S,\ l' \in L\} \rhd U, R\end{array}}{\Delta \cup \{x \mapsto \bar{\mu}\} \vdash e : \bigsqcup S_\kappa\mu_\kappa, \bigcup S_\kappa U_\kappa, \bigvee S_\kappa R_\kappa} \quad \text{(conc)}$$

$\boxed{A \rhd U, R}$

$$\frac{\forall (l_1 \sim l_2) \in A.[\![R(l_1)]\!] \Rightarrow (l_2 \notin U \wedge \neg[\![R(l_2)]\!])}{A \rhd U, R} \quad \text{(robust)}$$

그림 2: 메모리 타입 시스템.

함수 호출의 경우 (app) 다른 이름이 같은 메모리를 가르킬 수 있어 이에 따라 추가적으로 메모리 사용/재사용 패턴이 옳바른지 검증한다. 다른 인자에 같은 값을 전달할 때 문제가 생긴다. 함수 호출에서 인자 전달은 이름 치환 함수(S)에 의해 기술되므로 치환 함수의 결과에 교집합이 있을 때 한 쪽이 재사용되었으면 다른 쪽은 사용도 재사용도 불가한다 (robust).

case-문의 경우 (case) 타입이 구체적이면 바로 타입 유추가 가능하지만 타입이 요약되어 있으면 (conc) 규칙에 의해 가능한 데이타 구성자별로 구체화하여 타입 유추한다. 여러 데이타 구성자로 구체화 가능하므로 각각 타입 유추한 후 결과를 모두 포함하는 타입을 결과로 한다. 구체화할 때

$$\bar{\mu}(\kappa) = \begin{cases} \emptyset^{\text{false}}, & \text{if } \kappa \notin \text{dom}(\bar{\mu}) \\ L^{\pi}, & \text{if } (\kappa \mapsto L^{\pi}) \in \bar{\mu} \end{cases}$$

$$R(l) = \begin{cases} \text{false}, & \text{if } l \notin \text{dom}(R) \\ f, & \text{if } (l \mapsto f) \in R \end{cases}$$

$$R_1 \vee R_2 = \{l \mapsto R_1(l) \vee R_2(l) \mid l \in \text{dom}(R_1) \cup \text{dom}(R_2)\}$$

$$L_1^{\pi_1} \sqcup L_2^{\pi_2} = (L_1 \cup L_2)^{\pi_1 \vee \pi_2}$$

$$\text{int} \sqcup \text{int} = \text{int}$$

$$\langle L_1, \kappa, \vec{\mu}\rangle \sqcup \langle L_2, \kappa, \vec{\mu}'\rangle = \langle L_1 \cup L_2, \kappa, (\mu_1 \sqcup \mu_1', \cdots, \mu_n \sqcup \mu_n')\rangle$$

$$\bar{\mu} \sqcup \bar{\mu}' = \{\kappa \mapsto \bar{\mu}(\kappa) \sqcup \bar{\mu}'(\kappa) \mid \kappa \in \text{dom}(\bar{\mu}) \cup \text{dom}(\bar{\mu}')\}$$

$$(\lambda\vec{\delta}.\vec{\mu} \xrightarrow{U_1,R_1} \mu_1) \sqcup (\lambda\vec{\delta}.\vec{\mu} \xrightarrow{U_2,R_2} \mu_2) = \lambda\vec{\delta}.\vec{\mu} \xrightarrow{U_1 \cup U_2, R_1 \vee R_2} (\mu_1 \sqcup \mu_2)$$

$$\text{otherwise, } \mu_1 \sqcup \mu_2 = \alpha(\mu_1) \sqcup \alpha(\mu_2)$$

$$L_1^{\pi_1} \oplus L_2^{\pi_2} = (L_1 \cup L_2)^{\pi_1 \vee \pi_2 \vee (L_1 \cap L_2 \neq \emptyset)}$$

$$\bar{\mu} \oplus \bar{\mu}' = \{\kappa \mapsto \bar{\mu}(\kappa) \oplus \bar{\mu}'(\kappa) \mid \kappa \in \text{dom}(\bar{\mu}) \cup \text{dom}(\bar{\mu}')\}$$

$$\alpha(\text{int}) = \emptyset$$

$$\alpha(\bar{\mu}) = \bar{\mu}$$

$$\alpha(\langle L, \kappa, \vec{\mu}\rangle) = \{\kappa \mapsto L^{\text{false}}\} \oplus (\oplus\alpha(\mu_i))$$

$$\gamma_{\kappa'}(\bar{\mu}) = (\langle L_0, \kappa', \vec{\mu}\rangle, \{l_{i\kappa} \mapsto \bar{\mu}(\kappa)\})$$

where

$$\text{typeof}(\kappa') = \vec{\tau} \to t$$

$$L_0 = \{l_{0\kappa'}\} \text{ if } \bar{\mu}(\kappa') = L^{\text{false}}, \text{ or } L \text{ if } \bar{\mu}(\kappa') = L^{\text{true}}$$

$$\mu_i = \{\kappa \mapsto \{l_{i\kappa}\}^{\text{false}} \mid \kappa \in \text{kinds}(\tau_i),\ \bar{\mu}(\kappa) = L^{\text{false}}\} \cup \{\kappa \mapsto \bar{\mu}(\kappa) \mid \kappa \in \text{kinds}(\tau_i),\ \bar{\mu}(\kappa) = L^{\text{true}}\}$$

$$S(l) = \begin{cases} \{l\}^{\text{false}} & \text{if } l \notin \text{dom}(S) \\ S(l) & \text{if } l \in \text{dom}(S) \end{cases}$$

$$S(L^{\pi}) = L_1^{\pi \vee \pi_1} \text{ where } L_1^{\pi_1} = \oplus_{l \in L} S(l)$$

$$S(L) = L_1 \text{ where } L_1^{\pi} = \oplus_{l \in L} S(l)$$

$$S(\text{int}) = \text{int}$$

$$S(\langle L, \kappa, \vec{\mu}\rangle) = \langle S(L), \kappa, (S\mu_1, \cdots, S\mu_n)\rangle$$

$$S(\bar{\mu}) = \{\kappa \mapsto S(L^{\pi}) \mid (\kappa \mapsto L^{\pi}) \in \bar{\mu}\}$$

$$S(\lambda\vec{\delta}.\vec{\mu} \xrightarrow{U,R} \mu) = \lambda\vec{\delta}.\vec{\mu} \xrightarrow{U,R} \mu$$

$$[\![\text{true}]\!] = \text{true}$$

$$[\![\text{false}]\!] = \text{false}$$

$$[\![\delta]\!] = \text{true}$$

$$[\![f_1 \vee f_2]\!] = [\![f_1]\!] \vee [\![f_2]\!]$$

$$[\![f_1 \wedge f_2]\!] = [\![f_1]\!] \wedge [\![f_2]\!]$$

그림 3: 연산자들의 정의.

기존의 이름들을 새로운 이름으로 명명하는 경우가 있다. 이런 경우 새로운 이름과 기존의 이름 사이에 같은 메모리를 가르키게 되는 경우가 발생한다. 함수 호출 시와 유사하게 한쪽이 재사용되면 다른 쪽의 사용도 재사용도 불가한다.

요약 함수 α는 도달할 수 있는 모든 힙 셀을 모아서 종류별로 구분지어 주는 것이다. 여기서 같은 이름의 힙 셀이 두 번 이상 모이면 그 힙 셀은 공유되고 있다는 뜻이므로 공유 논리값을 참(true)으로 한다.

구체화 함수 γ는 각각의 종류별로 분류되어 있는 이름들을 배분해서 새로이 만드는 힙 구조에 적절히 배치해 준다. 문제는 어떻게 배분하느냐이다. 예를 들어 l_1, l_2, l_3에 도달할 수 있는 이진 나무구조(binary tree)를 좌측과 우측으로 자르는 문제를 생각해 보자. 공유가 되어있으면 두 부분으로 자를 수 없으므로 좌측 우측 모두 l_1, l_2, l_3가 가능하다고 한다. 공유가 안 되어 있으면 새로운 이름 l_4, l_5를 이름지어 l_4는 좌측, l_5는 우측으로 해서 타입 유추한다. 그러면 좌측과 우측이 이름이 다르므로 좌측을 재사용한 후에 우측을 사용할 수 있다. 결과 타입이 나오면 l_4, l_5를 모두 원래의 l_1, l_2, l_3로 치환해 준다.

5 메모리 안전성

타입 안전성(type soundness)[8]과 유사하게 메모리 타입 시스템에 대한 안전성을 증명할 수 있다.

정리 1 (메모리 안전성) *프로그램 e에 대해 "$\emptyset \vdash e : \tau$"나 "$\emptyset \vdash e : \mu, U, R$"가 성립하면, e는 끝나지 않거나 나미구조에 나쁘게 $(e, \emptyset, \epsilon) \to^* (u, H, \epsilon)$로 끝난다.*

즉, 타입이 옳바르고 메모리 타입이 옳바르면 프로그램이 중간에 파행적으로 중단하지 않는다.

6 결론

초기 단계의 실험에 의하면 본 논문에서 제시한 방법이 효과적일 듯 하다. Objective Caml 컴파일러[5]에서 사용되는 실험용 프로그램들은 수행 시간의 0–51.6%가 메모리 수거 시간인데, 특히 리스트나 데이타 구성자 연산이 주인 프로그램의 경우 메모리 수거 비용이 높았다. 작은 실험용 프로그램 `sieve`의 경우는 제시한 메모리 타입 시스템에 맞는 재사용 명령어를 삽입한 경우 원래의 프로그램보다 28.1% 빨리 수행되었다.

제시한 메모리 타입 시스템은 검증되어야 하고 확장되어야 한다. 메모리 타입 시스템의 안전성 뿐만 아니라 얼마나 효과적인지 검증되어야 한다. 그러기 위해서는 재사용 명령어를 자동으로 유추해내는 알고리즘도 개발하여야 한다. 제시된 타입 시스템은 자동 유추를 염두에 두고 고안되었기 때문에 검증을 위한 다른 메모리 타입 시스템들[7]과는 달리 자동 유추가 가능할 것으로 보인다. 또한 다형 타입 (polymorphic type), 참조 값 (reference value) 등에 대해서도 메모리 타입 시스템을 확장할 필요가 있다.

참고 자료

[1] Andrew W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4):275–279, June 1987.

[2] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, 1997.

[3] G. W. Hamilton. Garbage recycling: Transforming programs to reuse garbage. Technical Report TR95-13, Department of Computer Science, University of Keele, Staffordshire, ST5 5BG, U. K., July 1995.

[4] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, 1996.

[5] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The objective caml system release 3.02. Institut National de Recherche en Informatique et en Automatique, July 2001. `http://caml.inria.fr`.

[6] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[7] Frederick Smith, David Walker, and Greg Morrisett. Alias types. In *European Symposium on Programming*, volume 1782 of *Lecture Notes in Computer Science*, pages 366–381, March 2000.

[8] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. Technical Report TR91-160, 1992.