e ::=	expression
c	constant
$op(e_1, \ldots, e_n)$	primitive operation
`if` e_1 `then` e_2 `else` e_3	conditional
`let` $x = e_1$ `in` e_2	variable declaration
x	variable reference
`let rec` $x\ y_1\ \ldots\ y_n = e_1$ `in` e_2	declaration of recursive function
$e\ e_1\ \ldots\ e_n$	function call
$(e_1, \ldots, e_n)$	creation of a tuple
`let` $(x_1, \ldots, x_n) = e_1$ `in` e_2	pattern matching agains a tuple
`Array.create` $e_1\ e_2$	creation of an array
$e_1.(e_2)$	array element dereference
$e_1.(e_2) \leftarrow e_3$	assignment to an array element

図 1: MinCaml の抽象構文（型は省略）

Abstract syntax of MinCaml

τ ::=	type
π	primitive type
$\tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau$	functional type
$\tau_1 \times \ldots \times \tau_n$	tuple type
τ `array`	array type
α	type variable (for type inference)

図 2: MinCaml の型

MinCaml types

c is a constant of the π type

$$\frac{c \text{ は } \pi \text{ 型の定数}}{\Gamma \vdash c : \pi}$$

op is a primitive operation that takes values whose types are π1, ..., πn and gives a value of π type

$$\frac{\Gamma \vdash e_1 : \pi_1 \quad \ldots \quad \Gamma \vdash e_n : \pi_n \qquad op \text{ は } \pi_1, \ldots, \pi_n \text{ 型の値を受け取って } \pi \text{ 型の値を返すプリミティブ演算}}{\Gamma \vdash op(e_1, \ldots, e_n) : \pi}$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let } x = e_1 \texttt{ in } e_2 : \tau_2} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma, x : \tau_1 \to \ldots \to \tau_n \to \tau, y_1 : \tau_1, \ldots, y_n : \tau_n \vdash e_1 : \tau \quad \Gamma, x : \tau_1 \to \ldots \to \tau_n \to \tau \vdash e_2 : \tau'}{\Gamma \vdash \texttt{let rec } x\ y_1\ \ldots\ y_n = e_1 \texttt{ in } e_2 : \tau'} \qquad \frac{\Gamma \vdash e : \tau_1 \to \ldots \to \tau_n \to \tau \quad \Gamma \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash e\ e_1\ \ldots\ e_n : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash (e_1, \ldots, e_n) : \tau_1 \times \ldots \times \tau_n} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \times \ldots \times \tau_n \quad \Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e_2 : \tau}{\Gamma \vdash \texttt{let } (x_1, \ldots, x_n) = e_1 \texttt{ in } e_2 : \tau}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{Array.create } e_1\ e_2 : \tau\ \texttt{array}}$$

$$\frac{\Gamma \vdash e_1 : \tau\ \texttt{array} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1.(e_2) : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau\ \texttt{array} \quad \Gamma \vdash e_2 : \texttt{int} \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash e_1.(e_2) \leftarrow e_3 : \texttt{unit}}$$

図 3: MinCaml の型つけ規則

MinCaml's typing rule

$$\begin{array}{l} e\ ::= \\ \quad c \\ \quad op(x_1, \ldots, x_n) \\ \quad \texttt{if } x = y \texttt{ then } e_1 \texttt{ else } e_2 \\ \quad \texttt{if } x \leq y \texttt{ then } e_1 \texttt{ else } e_2 \\ \quad \texttt{let } x = e_1 \texttt{ in } e_2 \\ \quad x \\ \quad \texttt{let rec } x\ y_1\ \ldots\ y_n = e_1 \texttt{ in } e_2 \\ \quad x\ y_1\ \ldots\ y_n \\ \quad (x_1, \ldots, x_n) \\ \quad \texttt{let } (x_1, \ldots, x_n) = y \texttt{ in } e \\ \quad x.(y) \\ \quad x.(y) \leftarrow z \end{array}$$

図 4: MinCaml の K 正規形（外部配列・外部関数適用は省略）

MinCaml's K normal form (external arrays and external function calls are not included)

$\mathcal{K}$: `Syntax.t` $\rightarrow$ `KNormal.t`

$$
\begin{array}{lcl}
\mathcal{K}(c) & = & c \\
\mathcal{K}(\texttt{not}(e)) & = & \mathcal{K}(\texttt{if } e \texttt{ then false else true}) \\
\mathcal{K}(e_1 = e_2) & = & \mathcal{K}(\texttt{if } e_1 = e_2 \texttt{ then true else false}) \\
\mathcal{K}(e_1 \leq e_2) & = & \mathcal{K}(\texttt{if } e_1 \leq e_2 \texttt{ then true else false}) \\
\mathcal{K}(op(e_1, \ldots, e_n)) & = & \texttt{let } x_1 = \mathcal{K}(e_1) \texttt{ in } \ldots \texttt{ let } x_n = \mathcal{K}(e_n) \texttt{ in } op(x_1, \ldots, x_n) \\
 & & \text{when op is not a logical nor comparison operator} \\
\mathcal{K}(\texttt{if not } e_1 \texttt{ then } e_2 \texttt{ else } e_3) & = & \mathcal{K}(\texttt{if } e_1 \texttt{ then } e_3 \texttt{ else } e_2) \\
\mathcal{K}(\texttt{if } e_1 = e_2 \texttt{ then } e_3 \texttt{ else } e_4) & = & \texttt{let } x = \mathcal{K}(e_1) \texttt{ in let } y = \mathcal{K}(e_2) \texttt{ in} \\
 & & \texttt{if } x = y \texttt{ then } \mathcal{K}(e_3) \texttt{ else } \mathcal{K}(e_4) \\
\mathcal{K}(\texttt{if } e_1 \leq e_2 \texttt{ then } e_3 \texttt{ else } e_4) & = & \texttt{let } x = \mathcal{K}(e_1) \texttt{ in let } y = \mathcal{K}(e_2) \texttt{ in} \\
 & & \texttt{if } x \leq y \texttt{ then } \mathcal{K}(e_3) \texttt{ else } \mathcal{K}(e_4) \\
\mathcal{K}(\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3) & = & \mathcal{K}(\texttt{if } e_1 = \texttt{false then } e_3 \texttt{ else } e_2) \\
 & & \text{when e1 is not a logical nor a comparison operator} \\
\mathcal{K}(\texttt{let } x = e_1 \texttt{ in } e_2) & = & \texttt{let } x = \mathcal{K}(e_1) \texttt{ in } \mathcal{K}(e_2) \\
\mathcal{K}(x) & = & x \\
\mathcal{K}(\texttt{let rec } x\ y_1\ \ldots\ y_n = e_1 \texttt{ in } e_2) & = & \texttt{let rec } x\ y_1\ \ldots\ y_n = \mathcal{K}(e_1) \texttt{ in } \mathcal{K}(e_2) \\
\mathcal{K}(e\ e_1\ \ldots\ e_n) & = & \texttt{let } x = \mathcal{K}(e) \texttt{ in let } y_1 = \mathcal{K}(e_1) \texttt{ in } \ldots \texttt{ let } y_n = \mathcal{K}(e_n) \texttt{ in} \\
 & & x\ y_1\ \ldots\ y_n \\
\mathcal{K}(e_1, \ldots, e_n) & = & \texttt{let } x_1 = \mathcal{K}(e_1) \texttt{ in } \ldots \texttt{ let } x_n = \mathcal{K}(e_n) \texttt{ in } (x_1, \ldots, x_n) \\
\mathcal{K}(\texttt{let } (x_1, \ldots, x_n) = e_1 \texttt{ in } e_2) & = & \texttt{let } y = \mathcal{K}(e_1) \texttt{ in let } (x_1, \ldots, x_n) = y \texttt{ in } \mathcal{K}(e_2) \\
\mathcal{K}(\texttt{Array.create } e_1\ e_2) & = & \texttt{let } x = \mathcal{K}(e_1) \texttt{ in let } y = \mathcal{K}(e_2) \texttt{ in create\_array } x\ y \\
\mathcal{K}(e_1.(e_2)) & = & \texttt{let } x = \mathcal{K}(e_1) \texttt{ in let } y = \mathcal{K}(e_2) \texttt{ in } x.(y) \\
\mathcal{K}(e_1.(e_2) \leftarrow e_3) & = & \texttt{let } x = \mathcal{K}(e_1) \texttt{ in let } y = \mathcal{K}(e_2) \texttt{ in let } z = \mathcal{K}(e_3) \texttt{ in} \\
 & & x.(y) \leftarrow z
\end{array}
$$

図 5: K 正規化（論理値の整数化と、`insert_let` による最適化は省略）。右辺に出現していて左辺に出現していない変数は、すべて新しい (fresh) とする。

K normal for (conversion of logical values to numbers and oplimization using insert_let is abbreviated). A variable that occurs in RHS but not in LHS should be considered a new/ fresh.

α : `Id.t M.t` $\to$ `KNormal.t` $\to$ `KNormal.t`

$$
\begin{array}{lcl}
\alpha_\varepsilon(c) & = & c \\
\alpha_\varepsilon(op(x_1, \ldots, x_n)) & = & op(\varepsilon(x_1), \ldots, \varepsilon(x_n)) \\
\alpha_\varepsilon(\texttt{if } x = y \texttt{ then } e_1 \texttt{ else } e_2) & = & \texttt{if } \varepsilon(x) = \varepsilon(y) \texttt{ then } \alpha_\varepsilon(e_1) \texttt{ else } \alpha_\varepsilon(e_2) \\
\alpha_\varepsilon(\texttt{if } x \leq y \texttt{ then } e_1 \texttt{ else } e_2) & = & \texttt{if } \varepsilon(x) \leq \varepsilon(y) \texttt{ then } \alpha_\varepsilon(e_1) \texttt{ else } \alpha_\varepsilon(e_2) \\
\alpha_\varepsilon(\texttt{let } x = e_1 \texttt{ in } e_2) & = & \texttt{let } x' = \alpha_\varepsilon(e_1) \texttt{ in } \alpha_{\varepsilon, x \mapsto x'}(e_2) \\
\alpha_\varepsilon(x) & = & \varepsilon(x) \\
\alpha_\varepsilon(\texttt{let rec } x\ y_1\ \ldots\ y_n = e_1 \texttt{ in } e_2) & = & \texttt{let rec } x'\ y'_1\ \ldots\ y'_n = \alpha_{\varepsilon, x \mapsto x', y_1 \mapsto y'_1, \ldots, y_n \mapsto y'_n}(e_1) \texttt{ in } \\
 & & \alpha_{\varepsilon, x \mapsto x'}(e_2) \\
\alpha_\varepsilon(x\ y_1\ \ldots\ y_n) & = & \varepsilon(x)\ \varepsilon(y_1)\ \ldots\ \varepsilon(y_n) \\
\alpha_\varepsilon((x_1, \ldots, x_n)) & = & (\varepsilon(x_1), \ldots, \varepsilon(x_n)) \\
\alpha_\varepsilon(\texttt{let } (x_1, \ldots, x_n) = y \texttt{ in } e) & = & \texttt{let } (x'_1, \ldots, x'_n) = \varepsilon(y) \texttt{ in } \alpha_{\varepsilon, x_1 \mapsto x'_1, \ldots, x_n \mapsto x'_n}(e) \\
\alpha_\varepsilon(x.(y)) & = & \varepsilon(x).(\varepsilon(y)) \\
\alpha_\varepsilon(x.(y) \leftarrow z) & = & \varepsilon(x).(\varepsilon(y)) \leftarrow \varepsilon(z)
\end{array}
$$

図 6: α 変換。ε は α 変換前の変数を受け取って、α 変換後の変数を返す写像。右辺に出現していて左辺に出現していない変数（x' など）は、すべて fresh とする。

α conversion: ε is a mapping that takes a variable name and gives its α-converted name.
A name that occurs only in RHS should be considers new/fresh name.

β : `Id.t M.t` $\to$ `KNormal.t` $\to$ `KNormal.t`

$$
\begin{array}{lcll}
\beta_\varepsilon(c) & = & c & \\
\beta_\varepsilon(op(x_1, \ldots, x_n)) & = & op(\varepsilon(x_1), \ldots, \varepsilon(x_n)) & \\
\beta_\varepsilon(\texttt{if } x = y \texttt{ then } e_1 \texttt{ else } e_2) & = & \texttt{if } \varepsilon(x) = \varepsilon(y) \texttt{ then } \beta_\varepsilon(e_1) \texttt{ else } \beta_\varepsilon(e_2) & \\
\beta_\varepsilon(\texttt{if } x \leq y \texttt{ then } e_1 \texttt{ else } e_2) & = & \texttt{if } \varepsilon(x) \leq \varepsilon(y) \texttt{ then } \beta_\varepsilon(e_1) \texttt{ else } \beta_\varepsilon(e_2) & \\
\beta_\varepsilon(\texttt{let } x = e_1 \texttt{ in } e_2) & = & \beta_{\varepsilon, x \mapsto y}(e_2) & \beta_\varepsilon(e_1) \text{ is a variable "y"} \\
\beta_\varepsilon(\texttt{let } x = e_1 \texttt{ in } e_2) & = & \texttt{let } x = \beta_\varepsilon(e_1) \texttt{ in } \beta_\varepsilon(e_2) & \beta_\varepsilon(e_1) \text{ is not a variable} \\
\beta_\varepsilon(x) & = & \varepsilon(x) & \\
\beta_\varepsilon(\texttt{let rec } x\ y_1\ \ldots\ y_n = e_1 \texttt{ in } e_2) & = & \texttt{let rec } x\ y_1\ \ldots\ y_n = \beta_\varepsilon(e_1) \texttt{ in } \beta_\varepsilon(e_2) & \\
\beta_\varepsilon(x\ y_1\ \ldots\ y_n) & = & \varepsilon(x)\ \varepsilon(y_1)\ \ldots\ \varepsilon(y_n) & \\
\beta_\varepsilon((x_1, \ldots, x_n)) & = & (\varepsilon(x_1), \ldots, \varepsilon(x_n)) & \\
\beta_\varepsilon(\texttt{let } (x_1, \ldots, x_n) = y \texttt{ in } e) & = & \texttt{let } (x_1, \ldots, x_n) = \varepsilon(y) \texttt{ in } \beta_\varepsilon(e) & \\
\beta_\varepsilon(x.(y)) & = & \varepsilon(x).(\varepsilon(y)) & \\
\beta_\varepsilon(x.(y) \leftarrow z) & = & \varepsilon(x).(\varepsilon(y)) \leftarrow \varepsilon(z) &
\end{array}
$$

図 7: β 簡約。ε は β 簡約前の変数を受け取って、β 簡約後の変数を返す写像。$\varepsilon(x)$ が定義されていない場合は、$\varepsilon(x) = x$ とみなす。

β reduction: ε is a mapping that takes a variable name and gives its β-converted name.
We consider ε(x) = x when ε is not defined for x.

$\mathcal{A}$: `KNormal.t` $\rightarrow$ `KNormal.t`

$$
\begin{array}{lcl}
\mathcal{A}(c) & = & c \\
\mathcal{A}(op(x_1, \ldots, x_n)) & = & op(x_1, \ldots, x_n) \\
\mathcal{A}(\texttt{if } x = y \texttt{ then } e_1 \texttt{ else } e_2) & = & \texttt{if } x = y \texttt{ then } \mathcal{A}(e_1) \texttt{ else } \mathcal{A}(e_2) \\
\mathcal{A}(\texttt{if } x \leq y \texttt{ then } e_1 \texttt{ else } e_2) & = & \texttt{if } x \leq y \texttt{ then } \mathcal{A}(e_1) \texttt{ else } \mathcal{A}(e_2) \\
\mathcal{A}(\texttt{let } x = e_1 \texttt{ in } e_2) & = & \texttt{let } \ldots \texttt{ in let } x = e_1' \texttt{ in } \mathcal{A}(e_2) \\
 & & \quad \text{A(e1) has a form let ... in e' (let ... is a sequence of one or more let's) and e1' is not a let form.} \\
\mathcal{A}(x) & = & x \\
\mathcal{A}(\texttt{let rec } x\ y_1\ \ldots\ y_n = e_1 \texttt{ in } e_2) & = & \texttt{let rec } x\ y_1\ \ldots\ y_n = \mathcal{A}(e_1) \texttt{ in } \mathcal{A}(e_2) \\
\mathcal{A}(x\ y_1\ \ldots\ y_n) & = & x\ y_1\ \ldots\ y_n \\
\mathcal{A}((x_1, \ldots, x_n)) & = & (x_1, \ldots, x_n) \\
\mathcal{A}(\texttt{let } (x_1, \ldots, x_n) = y \texttt{ in } e) & = & \texttt{let } (x_1, \ldots, x_n) = y \texttt{ in } \mathcal{A}(e) \\
\mathcal{A}(x.(y)) & = & x.(y) \\
\mathcal{A}(x.(y) \leftarrow z) & = & x.(y) \leftarrow z
\end{array}
$$

図 8: Reduction of nested let's

$\mathcal{I}$: (`Id.t list` $\times$ `KNormal.t`) `M.t` $\rightarrow$ `KNormal.t` $\rightarrow$ `KNormal.t`

$$
\begin{array}{lcl}
\mathcal{I}_\varepsilon(c) & = & c \\
\mathcal{I}_\varepsilon(op(x_1, \ldots, x_n)) & = & op(x_1, \ldots, x_n) \\
\mathcal{I}_\varepsilon(\texttt{if } x = y \texttt{ then } e_1 \texttt{ else } e_2) & = & \texttt{if } x = y \texttt{ then } \mathcal{I}_\varepsilon(e_1) \texttt{ else } \mathcal{I}_\varepsilon(e_2) \\
\mathcal{I}_\varepsilon(\texttt{if } x \leq y \texttt{ then } e_1 \texttt{ else } e_2) & = & \texttt{if } x \leq y \texttt{ then } \mathcal{I}_\varepsilon(e_1) \texttt{ else } \mathcal{I}_\varepsilon(e_2) \\
\mathcal{I}_\varepsilon(\texttt{let } x = e_1 \texttt{ in } e_2) & = & \texttt{let } x = \mathcal{I}_\varepsilon(e_1) \texttt{ in } \mathcal{I}_\varepsilon(e_2) \\
\mathcal{I}_\varepsilon(x) & = & x \\
\mathcal{I}_\varepsilon(\texttt{let rec } x\ y_1\ \ldots\ y_n = e_1 \texttt{ in } e_2) & = & \varepsilon' = \varepsilon, x \mapsto ((y_1, \ldots, y_n), e_1) \text{ として} \\
 & & \texttt{let rec } x\ y_1\ \ldots\ y_n = \mathcal{I}_{\varepsilon'}(e_1) \texttt{ in } \mathcal{I}_{\varepsilon'}(e_2) \\
 & & \qquad\qquad size(e_1) \leq th \\
\mathcal{I}_\varepsilon(\texttt{let rec } x\ y_1\ \ldots\ y_n = e_1 \texttt{ in } e_2) & = & \texttt{let rec } x\ y_1\ \ldots\ y_n = \mathcal{I}_\varepsilon(e_1) \texttt{ in } \mathcal{I}_\varepsilon(e_2) \\
 & & \qquad\qquad size(e_1) > th \\
\mathcal{I}_\varepsilon(x\ y_1\ \ldots\ y_n) & = & \alpha_{y_1 \mapsto z_1, \ldots, y_n \mapsto z_n}(e) \qquad \varepsilon(x) = ((z_1, \ldots, z_n), e) \\
\mathcal{I}_\varepsilon(x\ y_1\ \ldots\ y_n) & = & x\ y_1\ \ldots\ y_n \qquad \varepsilon(x) \text{ is undefined} \\
\mathcal{I}_\varepsilon((x_1, \ldots, x_n)) & = & (x_1, \ldots, x_n) \\
\mathcal{I}_\varepsilon(\texttt{let } (x_1, \ldots, x_n) = y \texttt{ in } e) & = & \texttt{let } (x_1, \ldots, x_n) = y \texttt{ in } \mathcal{I}_\varepsilon(e) \\
\mathcal{I}_\varepsilon(x.(y)) & = & x.(y) \\
\mathcal{I}_\varepsilon(x.(y) \leftarrow z) & = & x.(y) \leftarrow z
\end{array}
$$

$$
\begin{array}{lcl}
size(c) & = & 1 \\
size(op(x_1, \ldots, x_n)) & = & 1 \\
size(\texttt{if } x = y \texttt{ then } e_1 \texttt{ else } e_2) & = & 1 + size(e_1) + size(e_2) \\
size(\texttt{if } x \leq y \texttt{ then } e_1 \texttt{ else } e_2) & = & 1 + size(e_1) + size(e_2) \\
size(\texttt{let } x = e_1 \texttt{ in } e_2) & = & 1 + size(e_1) + size(e_2) \\
size(x) & = & 1 \\
size(\texttt{let rec } x\ y_1\ \ldots\ y_n = e_1 \texttt{ in } e_2) & = & 1 + size(e_1) + size(e_2) \\
size(x\ y_1\ \ldots\ y_n) & = & 1 \\
size((x_1, \ldots, x_n)) & = & 1 \\
size(\texttt{let } (x_1, \ldots, x_n) = y \texttt{ in } e) & = & 1 + size(e) \\
size(x.(y)) & = & 1 \\
size(x.(y) \leftarrow z) & = & 1
\end{array}
$$

Inline expansion: ε is a mapping that takes the name of a smaller-sized function and gives its formal arguments and body. "th" is a user-specified threshold, that specifies the maximum function size that can be expanded.

$\mathcal{F}$: KNormal.t M.t → KNormal.t → KNormal.t

X の場合: when X
それ以外の場合: otherwise

$\mathcal{F}_\varepsilon(c)$	=	c	
$\mathcal{F}_\varepsilon(op(x_1, \ldots, x_n))$	=	c	$op(\varepsilon(x_1), \ldots, \varepsilon(x_n)) = c$ の場合
$\mathcal{F}_\varepsilon(op(x_1, \ldots, x_n))$	=	$op(x_1, \ldots, x_n)$	それ以外の場合
$\mathcal{F}_\varepsilon(\texttt{if } x = y \texttt{ then } e_1 \texttt{ else } e_2)$	=	$\mathcal{F}_\varepsilon(e_1)$	$\varepsilon(x)$ と $\varepsilon(y)$ are the same constants
$\mathcal{F}_\varepsilon(\texttt{if } x = y \texttt{ then } e_1 \texttt{ else } e_2)$	=	$\mathcal{F}_\varepsilon(e_2)$	$\varepsilon(x)$ と $\varepsilon(y)$ are different constants
$\mathcal{F}_\varepsilon(\texttt{if } x = y \texttt{ then } e_1 \texttt{ else } e_2)$	=	$\texttt{if } x = y \texttt{ then } \mathcal{F}_\varepsilon(e_1) \texttt{ else } \mathcal{F}_\varepsilon(e_2)$	それ以外の場合
$\mathcal{F}_\varepsilon(\texttt{if } x \leq y \texttt{ then } e_1 \texttt{ else } e_2)$	=	$\mathcal{F}_\varepsilon(e_1)$	$\varepsilon(x)$ と $\varepsilon(y)$: constants & $\varepsilon(x) \leq \varepsilon(y)$ の場合
$\mathcal{F}_\varepsilon(\texttt{if } x \leq y \texttt{ then } e_1 \texttt{ else } e_2)$	=	$\mathcal{F}_\varepsilon(e_2)$	$\varepsilon(x)$ と $\varepsilon(y)$: constants & $\varepsilon(x) > \varepsilon(y)$ の場合
$\mathcal{F}_\varepsilon(\texttt{if } x \leq y \texttt{ then } e_1 \texttt{ else } e_2)$	=	$\texttt{if } x \leq y \texttt{ then } \mathcal{F}_\varepsilon(e_1) \texttt{ else } \mathcal{F}_\varepsilon(e_2)$	それ以外の場合
$\mathcal{F}_\varepsilon(\texttt{let } x = e_1 \texttt{ in } e_2)$	=	$e_1' = \mathcal{F}_\varepsilon(e_1)$ として $\texttt{let } x = e_1' \texttt{ in } \mathcal{F}_{\varepsilon, x \mapsto e_1'}(e_2)$	
$\mathcal{F}_\varepsilon(x)$	=	x	
$\mathcal{F}_\varepsilon(\texttt{let rec } x\ y_1\ \ldots\ y_n = e_1 \texttt{ in } e_2)$	=	$\texttt{let rec } x\ y_1\ \ldots\ y_n = \mathcal{F}_\varepsilon(e_1) \texttt{ in } \mathcal{F}_\varepsilon(e_2)$	
$\mathcal{F}_\varepsilon(x\ y_1\ \ldots\ y_n)$	=	$x\ y_1\ \ldots\ y_n$	
$\mathcal{F}_\varepsilon((x_1, \ldots, x_n))$	=	$(x_1, \ldots, x_n)$	
$\mathcal{F}_\varepsilon(\texttt{let } (x_1, \ldots, x_n) = y \texttt{ in } e)$	=	$\texttt{let } x_1 = y_1 \texttt{ in } \ldots \texttt{ let } x_n = y_n \texttt{ in } \mathcal{F}_\varepsilon(e)$	$\varepsilon(y) = (y_1, \ldots, y_n)$ の場合
$\mathcal{F}_\varepsilon(\texttt{let } (x_1, \ldots, x_n) = y \texttt{ in } e)$	=	$\texttt{let } (x_1, \ldots, x_n) = y \texttt{ in } \mathcal{F}_\varepsilon(e)$	
$\mathcal{F}_\varepsilon(x.(y))$	=	$x.(y)$	
$\mathcal{F}_\varepsilon(x.(y) \leftarrow z)$	=	$x.(y) \leftarrow z$	

Constant folding: ε is a mapping that takes a variable ~~and gives a constant.~~ its associated expression

$\mathcal{E}$: KNormal.t $\rightarrow$ KNormal.t

$\mathcal{E}(c)$	$=$	c	"X の場合" (X no baai): "when X"
$\mathcal{E}(op(x_1, \ldots, x_n))$	$=$	$op(x_1, \ldots, x_n)$	"それ以外の場合" (sore igai no baai): "otherwise"
$\mathcal{E}(\texttt{if } x = y \texttt{ then } e_1 \texttt{ else } e_2)$	$=$	$\texttt{if } x = y \texttt{ then } \mathcal{E}(e_1) \texttt{ else } \mathcal{E}(e_2)$	
$\mathcal{E}(\texttt{if } x \leq y \texttt{ then } e_1 \texttt{ else } e_2)$	$=$	$\texttt{if } x \leq y \texttt{ then } \mathcal{E}(e_1) \texttt{ else } \mathcal{E}(e_2)$	
$\mathcal{E}(\texttt{let } x = e_1 \texttt{ in } e_2)$	$=$	$\mathcal{E}(e_2)$	$effect(\mathcal{E}(e_1)) = false$ かつ $x \notin FV(\mathcal{E}(e_2))$ の場合
$\mathcal{E}(\texttt{let } x = e_1 \texttt{ in } e_2)$	$=$	$\texttt{let } x = \mathcal{E}(e_1) \texttt{ in } \mathcal{E}(e_2)$	それ以外の場合
$\mathcal{E}(x)$	$=$	x	
$\mathcal{E}(\texttt{let rec } x\ y_1\ \ldots\ y_n = e_1 \texttt{ in } e_2)$	$=$	$\mathcal{E}(e_2)$	$x \notin FV(\mathcal{E}(e_2))$ の場合
$\mathcal{E}(\texttt{let rec } x\ y_1\ \ldots\ y_n = e_1 \texttt{ in } e_2)$	$=$	$\texttt{let rec } x\ y_1\ \ldots\ y_n = \mathcal{E}(e_1) \texttt{ in } \mathcal{E}(e_2)$	それ以外の場合
$\mathcal{E}(x\ y_1\ \ldots\ y_n)$	$=$	$x\ y_1\ \ldots\ y_n$	
$\mathcal{E}((x_1, \ldots, x_n))$	$=$	$(x_1, \ldots, x_n)$	
$\mathcal{E}(\texttt{let } (x_1, \ldots, x_n) = y \texttt{ in } e)$	$=$	$\mathcal{E}(e)$	$\{x_1, \ldots, x_n\} \cap FV(\mathcal{E}(e)) = \emptyset$ の場合
$\mathcal{E}(\texttt{let } (x_1, \ldots, x_n) = y \texttt{ in } e)$	$=$	$\texttt{let } (x_1, \ldots, x_n) = y \texttt{ in } \mathcal{E}(e)$	それ以外の場合
$\mathcal{E}(x.(y))$	$=$	$x.(y)$	
$\mathcal{E}(x.(y) \leftarrow z)$	$=$	$x.(y) \leftarrow z$	

effect : KNormal.t $\rightarrow$ bool

$effect(c)$	$=$	$false$
$effect(op(x_1, \ldots, x_n))$	$=$	$false$
$effect(\texttt{if } x = y \texttt{ then } e_1 \texttt{ else } e_2)$	$=$	$effect(e_1) \vee effect(e_2)$
$effect(\texttt{if } x \leq y \texttt{ then } e_1 \texttt{ else } e_2)$	$=$	$effect(e_1) \vee effect(e_2)$
$effect(\texttt{let } x = e_1 \texttt{ in } e_2)$	$=$	$effect(e_1) \vee effect(e_2)$
$effect(x)$	$=$	$false$
$effect(\texttt{let rec } x\ y_1\ \ldots\ y_n = e_1 \texttt{ in } e_2)$	$=$	$effect(e_2)$
$effect(x\ y_1\ \ldots\ y_n)$	$=$	$true$
$effect((x_1, \ldots, x_n))$	$=$	$false$
$effect(\texttt{let } (x_1, \ldots, x_n) = y \texttt{ in } e)$	$=$	$effect(e)$
$effect(x.(y))$	$=$	$false$
$effect(x.(y) \leftarrow z)$	$=$	$true$

図 11: 不要定義削除 (1/2)

Elimination of redundant definition

FV : `KNormal.t` $\rightarrow$ `S.t`

$$
\begin{array}{lcl}
FV(c) & = & \emptyset \\
FV(op(x_1, \ldots, x_n)) & = & \{x_1, \ldots, x_n\} \\
FV(\texttt{if } x = y \texttt{ then } e_1 \texttt{ else } e_2) & = & \{x, y\} \cup FV(e_1) \cup FV(e_2) \\
FV(\texttt{if } x \leq y \texttt{ then } e_1 \texttt{ else } e_2) & = & \{x, y\} \cup FV(e_1) \cup FV(e_2) \\
FV(\texttt{let } x = e_1 \texttt{ in } e_2) & = & FV(e_1) \cup (FV(e_2) \setminus \{x\}) \\
FV(x) & = & \{x\} \\
FV(\texttt{let rec } x\ y_1\ \ldots\ y_n = e_1 \texttt{ in } e_2) & = & ((FV(e_1) \setminus \{y_1, \ldots, y_n\}) \cup FV(e_2)) \setminus \{x\} \\
FV(x\ y_1\ \ldots\ y_n) & = & \{x, y_1, \ldots, y_n\} \\
FV((x_1, \ldots, x_n)) & = & \{x_1, \ldots, x_n\} \\
FV(\texttt{let } (x_1, \ldots, x_n) = y \texttt{ in } e) & = & \{y\} \cup (FV(e) \setminus \{x_1, \ldots, x_n\}) \\
FV(x.(y)) & = & \{x, y\} \\
FV(x.(y) \leftarrow z) & = & \{x, y, z\}
\end{array}
$$

図 12: 不要定義削除 (2/2)

Elimination of redundant definition

P ::= The whole program

$(\{D_1, \ldots, D_n\}, e)$ Definitions of the top-level functions and the main routine

D ::= Definitions of the top-level functions

$\mathtt{L}_x(y_1, \ldots, y_m)(z_1, \ldots, z_n) = e$ Function: label/formal arguments/free variables/body

e ::=

- c
- $op(x_1, \ldots, x_n)$
- `if` $x = y$ `then` e_1 `else` e_2
- `if` $x \leq y$ `then` e_1 `else` e_2
- `let` $x = e_1$ `in` e_2
- x
- $make\_closure\ x = (\mathtt{L}_x, (y_1, \ldots, y_n))$ `in` e Closure creation
- $apply\_closure(x, y_1, \ldots, y_n)$ Function call using a closure
- $apply\_direct(\mathtt{L}_x, y_1, \ldots, y_n)$ Function call without using a closure (for known functions)
- $(x_1, \ldots, x_n)$
- `let` $(x_1, \ldots, x_n) = y$ `in` e
- $x.(y)$
- $x.(y) \leftarrow z$

図 13: The closure language

$\mathcal{C}$: `KNormal.t` $\rightarrow$ `Closure.t`

$$
\begin{array}{lcl}
\mathcal{C}(c) & = & c \\
\mathcal{C}(op(x_1, \ldots, x_n)) & = & op(x_1, \ldots, x_n) \\
\mathcal{C}(\texttt{if } x = y \texttt{ then } e_1 \texttt{ else } e_2) & = & \texttt{if } x = y \texttt{ then } \mathcal{C}(e_1) \texttt{ else } \mathcal{C}(e_2) \\
\mathcal{C}(\texttt{if } x \leq y \texttt{ then } e_1 \texttt{ else } e_2) & = & \texttt{if } x \leq y \texttt{ then } \mathcal{C}(e_1) \texttt{ else } \mathcal{C}(e_2) \\
\mathcal{C}(\texttt{let } x = e_1 \texttt{ in } e_2) & = & \texttt{let } x = \mathcal{C}(e_1) \texttt{ in } \mathcal{C}(e_2) \\
\mathcal{C}(x) & = & x \\
\mathcal{C}(\texttt{let rec } x\ y_1\ \ldots\ y_n = e_1 \texttt{ in } e_2) & = & \mathcal{D} \text{ に } \mathtt{L}_x(y_1, \ldots, y_n)(z_1, \ldots, z_m) = e_1' \text{ を加え、} \\
 & & make\_closure\ x = (\mathtt{L}_x, (z_1, \ldots, z_m)) \texttt{ in } e_2' \text{を返す} \\
 & & \quad \text{ただし } e_1' = \mathcal{C}(e_1),\ e_2' = \mathcal{C}(e_2), \\
 & & \quad FV(e_1') \setminus \{x, y_1, \ldots, y_n\} = \{z_1, \ldots, z_m\} \\
\mathcal{C}(x\ y_1\ \ldots\ y_n) & = & apply\_closure(x, y_1, \ldots, y_n) \\
\mathcal{C}((x_1, \ldots, x_n)) & = & (x_1, \ldots, x_n) \\
\mathcal{C}(\texttt{let } (x_1, \ldots, x_n) = y \texttt{ in } e) & = & \texttt{let } (x_1, \ldots, x_n) = y \texttt{ in } \mathcal{C}(e) \\
\mathcal{C}(x.(y)) & = & x.(y) \\
\mathcal{C}(x.(y) \leftarrow z) & = & x.(y) \leftarrow z
\end{array}
$$

FV : `Closure.t` $\rightarrow$ `S.t`

$$
\begin{array}{lcl}
FV(c) & = & \emptyset \\
FV(op(x_1, \ldots, x_n)) & = & \{x_1, \ldots, x_n\} \\
FV(\texttt{if } x = y \texttt{ then } e_1 \texttt{ else } e_2) & = & \{x, y\} \cup FV(e_1) \cup FV(e_2) \\
FV(\texttt{if } x \leq y \texttt{ then } e_1 \texttt{ else } e_2) & = & \{x, y\} \cup FV(e_1) \cup FV(e_2) \\
FV(\texttt{let } x = e_1 \texttt{ in } e_2) & = & FV(e_1) \cup (FV(e_2) \setminus \{x\}) \\
FV(x) & = & \{x\} \\
FV(make\_closure\ x = (\mathtt{L}_x, (y_1, \ldots, y_n)) \texttt{ in } e) & = & \{y_1, \ldots, y_n\} \cup (FV(e) \setminus \{x\}) \\
FV(apply\_closure(x, y_1, \ldots, y_n)) & = & \{x, y_1, \ldots, y_n\} \\
FV(apply\_direct(\mathtt{L}_x, y_1, \ldots, y_n)) & = & \{y_1, \ldots, y_n\} \\
FV((x_1, \ldots, x_n)) & = & \{x_1, \ldots, x_n\} \\
FV(\texttt{let } (x_1, \ldots, x_n) = y \texttt{ in } e) & = & \{y\} \cup (FV(e) \setminus \{x_1, \ldots, x_n\}) \\
FV(x.(y)) & = & \{x, y\} \\
FV(x.(y) \leftarrow z) & = & \{x, y, z\}
\end{array}
$$

図 14: 賢くない Closure 変換 $\mathcal{C}(e)$。$\mathcal{D}$ はトップレベル関数定義の集合を記憶しておくためのグローバル

Simply-minded closure conversion.
The variable D collects the set of top-level function definitions.

$\mathcal{C}$: S.t $\rightarrow$ KNormal.t $\rightarrow$ Closure.t

$$
\begin{array}{lcll}
\mathcal{C}_s(\texttt{let rec}\ x\ y_1\ \ldots\ y_n = e_1\ \texttt{in}\ e_2) & = & \mathcal{D} \text{ に } \mathtt{L}_x(y_1,\ldots,y_n)() = e_1' \text{ を加え、} & \\
 & & \mathit{make\_closure}\ x = (\mathtt{L}_x, ())\ \texttt{in}\ e_2' \text{を返す} & \\
 & & \quad \text{ただし } e_1' = \mathcal{C}_{s'}(e_1),\ e_2' = \mathcal{C}_{s'}(e_2),\ s' = s \cup \{x\}, & \\
 & & \quad FV(e_1') \setminus \{y_1,\ldots,y_n\} = \emptyset \text{ の場合} & \\
\mathcal{C}_s(\texttt{let rec}\ x\ y_1\ \ldots\ y_n = e_1\ \texttt{in}\ e_2) & = & \mathcal{D} \text{ に } \mathtt{L}_x(y_1,\ldots,y_n)(z_1,\ldots,z_m) = e_1' \text{ を加え、} & \\
 & & \mathit{make\_closure}\ x = (\mathtt{L}_x, (z_1,\ldots,z_m))\ \texttt{in}\ e_2' \text{を返す} & \\
 & & \quad \text{ただし } e_1' = \mathcal{C}_s(e_1),\ e_2' = \mathcal{C}_s(e_2), & \\
 & & \quad FV(e_1') \setminus \{y_1,\ldots,y_n\} \neq \emptyset, & \\
 & & \quad FV(e_1') \setminus \{x, y_1,\ldots,y_n\} = \{z_1,\ldots,z_m\} \text{ の場合} & \\
\mathcal{C}_s(x\ y_1\ \ldots\ y_n) & = & \mathit{apply\_closure}(x, y_1,\ldots,y_n) & x \notin s \text{ の場合} \\
\mathcal{C}_s(x\ y_1\ \ldots\ y_n) & = & \mathit{apply\_direct}(\mathtt{L}_x, y_1,\ldots,y_n) & x \in s \text{ の場合}
\end{array}
$$

図 15: やや賢い Closure 変換 $\mathcal{C}_s(e)$。s は自由変数がないとわかっている関数の名前の集合。

$\mathcal{C}$: S.t $\rightarrow$ KNormal.t $\rightarrow$ Closure.t

$$
\begin{array}{lcll}
\mathcal{C}_s(\texttt{let rec}\ x\ y_1\ \ldots\ y_n = e_1\ \texttt{in}\ e_2) & = & \mathcal{D} \text{ に } \mathtt{L}_x(y_1,\ldots,y_n)() = e_1' \text{ を加え、} & \\
 & & \mathit{make\_closure}\ x = (\mathtt{L}_x, ())\ \texttt{in}\ e_2' \text{を返す} & \\
 & & \quad \text{ただし } e_1' = \mathcal{C}_{s'}(e_1),\ e_2' = \mathcal{C}_{s'}(e_2),\ s' = s \cup \{x\}, & \\
 & & \quad FV(e_1') \setminus \{y_1,\ldots,y_n\} = \emptyset \text{ かつ } x \in FV(e_2') \text{ の場合} & \\
\mathcal{C}_s(\texttt{let rec}\ x\ y_1\ \ldots\ y_n = e_1\ \texttt{in}\ e_2) & = & \mathcal{D} \text{ に } \mathtt{L}_x(y_1,\ldots,y_n)() = e_1' \text{ を加え、} e_2' \text{を返す} & \\
 & & \quad \text{ただし } e_1' = \mathcal{C}_{s'}(e_1),\ e_2' = \mathcal{C}_{s'}(e_2),\ s' = s \cup \{x\}, & \\
 & & \quad FV(e_1') \setminus \{y_1,\ldots,y_n\} = \emptyset \text{ かつ } x \notin FV(e_2') \text{ の場合} & \\
\mathcal{C}_s(\texttt{let rec}\ x\ y_1\ \ldots\ y_n = e_1\ \texttt{in}\ e_2) & = & \mathcal{D} \text{ に } \mathtt{L}_x(y_1,\ldots,y_n)(z_1,\ldots,z_m) = e_1' \text{ を加え、} & \\
 & & \mathit{make\_closure}\ x = (\mathtt{L}_x, (z_1,\ldots,z_m))\ \texttt{in}\ e_2' \text{を返す} & \\
 & & \quad \text{ただし } e_1' = \mathcal{C}_s(e_1),\ e_2' = \mathcal{C}_s(e_2), & \\
 & & \quad FV(e_1') \setminus \{y_1,\ldots,y_n\} \neq \emptyset, & \\
 & & \quad FV(e_1') \setminus \{x, y_1,\ldots,y_n\} = \{z_1,\ldots,z_m\} \text{ の場合} & \\
\mathcal{C}_s(x\ y_1\ \ldots\ y_n) & = & \mathit{apply\_closure}(x, y_1,\ldots,y_n) & x \notin s \text{ の場合} \\
\mathcal{C}_s(x\ y_1\ \ldots\ y_n) & = & \mathit{apply\_direct}(\mathtt{L}_x, y_1,\ldots,y_n) & x \in s \text{ の場合}
\end{array}
$$

図 16: もっと賢い Closure 変換 $\mathcal{C}_s(e)$

P ::=
  $(\{D_1, \ldots, D_n\}, E)$
D ::=
  $\mathtt{L}_x(y_1, \ldots, y_n) = E$

E ::=	命令の列	Instruction sequence
$x \leftarrow e; E$	代入	Assignment
e	返値	Return a value
e ::=	式	Expressions
c	即値	Immediate values
$\mathtt{L}_x$	ラベル	Label
$op(x_1, \ldots, x_n)$	算術演算	Arithmetic operation
$\mathtt{if}\ x = y\ \mathtt{then}\ E_1\ \mathtt{else}\ E_2$	比較＆分岐	Comparison and branch
$\mathtt{if}\ x \leq y\ \mathtt{then}\ E_1\ \mathtt{else}\ E_2$	比較＆分岐	
x	`mov` 命令	"mov" operation
$apply\_closure(x, y_1, \ldots, y_n)$	クロージャを用いた関数呼び出し	Function call via a closure
$apply\_direct(\mathtt{L}_x, y_1, \ldots, y_n)$	クロージャを用いない関数呼び出し	Function call
$x.(y)$	ロード	Load/store a value from/to
$x.(y) \leftarrow z$	ストア	a memory address indexed by y
$\mathtt{save}(x, y)$	変数 x の値をスタック位置 y に退避する	
$\mathtt{restore}(y)$	スタック位置 y から値を復元する	
		Save/restore the value to/from the stack.

図 17: 仮想マシンコードの構文

Virtual machine code language

$\mathcal{V}$: `Closure.prog` $\rightarrow$ `SparcAsm.prog`

$\mathcal{V}((\{D_1, \ldots, D_n\}, e)) = (\{\mathcal{V}(D_1), \ldots, \mathcal{V}(D_n)\}, \mathcal{V}(e))$

$\mathcal{V}$: `Closure.fundef` $\rightarrow$ `SparcAsm.fundef`

$\mathcal{V}(\mathtt{L}_x(y_1, \ldots, y_n)(z_1, \ldots, z_n) = e) = \mathtt{L}_x(y_1, \ldots, y_n) = z_1 \leftarrow \mathtt{R}_0.(4); \ldots; z_n \leftarrow \mathtt{R}_0.(4n); \mathcal{V}(e)$

$\mathcal{V}$: `Closure.t` $\rightarrow$ `SparcAsm.t`

$$
\begin{array}{lcl}
\mathcal{V}(c) & = & c \\
\mathcal{V}(op(x_1, \ldots, x_n)) & = & op(x_1, \ldots, x_n) \\
\mathcal{V}(\mathtt{if}\ x = y\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2) & = & \mathtt{if}\ x = y\ \mathtt{then}\ \mathcal{V}(e_1)\ \mathtt{else}\ \mathcal{V}(e_2) \\
\mathcal{V}(\mathtt{if}\ x \leq y\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2) & = & \mathtt{if}\ x \leq y\ \mathtt{then}\ \mathcal{V}(e_1)\ \mathtt{else}\ \mathcal{V}(e_2) \\
\mathcal{V}(\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2) & = & x \leftarrow \mathcal{V}(e_1); \mathcal{V}(e_2) \\
\mathcal{V}(x) & = & x \\
\mathcal{V}(make\_closure\ x = (\mathtt{L}_x, (y_1, \ldots, y_n))\ \mathtt{in}\ e) & = & x \leftarrow \mathtt{R}_{\mathtt{hp}}; \mathtt{R}_{\mathtt{hp}} \leftarrow \mathtt{R}_{\mathtt{hp}} + 4(n+1); z \leftarrow \mathtt{L}_x; x.(0) \leftarrow z; \\
 & & x.(4) \leftarrow y_1; \ldots; x.(4n) \leftarrow y_n; \mathcal{V}(e) \\
\mathcal{V}(apply\_closure(x, y_1, \ldots, y_n)) & = & apply\_closure(x, y_1, \ldots, y_n) \\
\mathcal{V}(apply\_direct(\mathtt{L}_x, y_1, \ldots, y_n)) & = & apply\_direct(\mathtt{L}_x, y_1, \ldots, y_n) \\
\mathcal{V}((x_1, \ldots, x_n)) & = & y \leftarrow \mathtt{R}_{\mathtt{hp}}; \mathtt{R}_{\mathtt{hp}} \leftarrow \mathtt{R}_{\mathtt{hp}} + 4n; \\
 & & y.(0) \leftarrow x_1; \ldots; y.(4(n-1)) \leftarrow x_n; y \\
\mathcal{V}(\mathtt{let}\ (x_1, \ldots, x_n) = y\ \mathtt{in}\ e) & = & \{x_1, \ldots, x_n\} \cap FV(e) = \{x_{i_1}, \ldots, x_{i_m}\}\ \text{として} \\
 & & x_{i_1} \leftarrow y.(4(i_1 - 1)); \ldots; x_{i_m} \leftarrow y.(4(i_m - 1)); \mathcal{V}(e) \\
\mathcal{V}(x.(y)) & = & y' \leftarrow 4 \times y; x.(y') \\
\mathcal{V}(x.(y) \leftarrow z) & = & y' \leftarrow 4 \times y; x.(y') \leftarrow z
\end{array}
$$

図 18: 仮想マシンコード生成 $\mathcal{V}(P)$, $\mathcal{V}(D)$ および $\mathcal{V}(e)$。右辺に出現して左辺に出現しない変数は fresh とする。$\mathtt{R}_{\mathtt{hp}}$ はヒープポインタ（専用レジスタ）。$e_1; e_2$ はダミーの変数 x について $x \leftarrow e_1; e_2$ の略記。$x \leftarrow E_1; E_2$ は、$E_1 = (x_1 \leftarrow e_1; \ldots; x_n \leftarrow e_n; e)$ として、$x_1 \leftarrow e_1; \ldots; x_n \leftarrow e_n; x \leftarrow e; E_2$ の略記。

English translation of the caption is found in the slide

$$
\begin{array}{lcl}
FV : \texttt{S.t} \to \texttt{SparcAsm.t} \to \texttt{S.t} & & \\
FV_s(x \leftarrow e; E) & = & s' = FV_s(E) \setminus \{x\} \text{ として } FV_{s'}(e) \\
FV_s(e) & = & FV_s(e) \\
 & & \\
FV : \texttt{S.t} \to \texttt{SparcAsm.exp} \to \texttt{S.t} & & \\
FV_s(c) & = & s \\
FV_s(\texttt{L}_x) & = & s \\
FV_s(op(x_1, \ldots, x_n)) & = & \{x_1, \ldots, x_n\} \cup s \\
FV_s(\texttt{if } x = y \texttt{ then } E_1 \texttt{ else } E_2) & = & \{x, y\} \cup FV_s(E_1) \cup FV_s(E_2) \\
FV_s(\texttt{if } x \leq y \texttt{ then } E_1 \texttt{ else } E_2) & = & \{x, y\} \cup FV_s(E_1) \cup FV_s(E_2) \\
FV_s(x) & = & \{x\} \cup s \\
FV_s(\mathit{apply\_closure}(x, y_1, \ldots, y_n)) & = & \{x, y_1, \ldots, y_n\} \cup s \\
FV_s(\mathit{apply\_direct}(\texttt{L}_x, y_1, \ldots, y_n)) & = & \{y_1, \ldots, y_n\} \cup s \\
FV_s(x.(y)) & = & \{x, y\} \cup s \\
FV_s(x.(y) \leftarrow z) & = & \{x, y, z\} \cup s \\
FV_s(\texttt{save}(x, y)) & = & \{x\} \cup s \\
FV_s(\texttt{restore}(y)) & = & s
\end{array}
$$

図 19: 命令の列 E および式 e において生きている変数の集合 $FV_s(E)$ および $FV_s(e)$。s は E や e の後で使われる変数の集合。以後の $FV(E)$ は $FV_\emptyset(E)$ の略記。

$\mathcal{R}$: SparcAsm.prog → SparcAsm.prog

$$\mathcal{R}((\{D_1, \ldots, D_n\}, E)) = (\{\mathcal{R}(D_1), \ldots, \mathcal{R}(D_n)\}, \mathcal{R}_\emptyset(E, x, ()))$$

x is a fresh dummy variable

$\mathcal{R}$: SparcAsm.fundef → SparcAsm.fundef

$$\mathcal{R}(\mathtt{L}_x(y_1, \ldots, y_n) = E) = \mathtt{L}_x(\mathtt{R}_1, \ldots, \mathtt{R}_n) = \mathcal{R}_{x \mapsto \mathtt{R}_0, y_1 \mapsto \mathtt{R}_1, \ldots, y_n \mapsto \mathtt{R}_n}(E, \mathtt{R}_0, \mathtt{R}_0)$$

$\mathcal{R}$: Id.t M.t → SparcAsm.t × Id.t × SparcAsm.t → SparcAsm.t × Id.t M.t

$$\begin{aligned}
\mathcal{R}_\varepsilon((x \leftarrow e; E), z_{\mathtt{dest}}, E_{\mathtt{cont}}) = {} & E'_{\mathtt{cont}} = (z_{\mathtt{dest}} \leftarrow E; E_{\mathtt{cont}}), \\
& \mathcal{R}_\varepsilon(e, x, E'_{\mathtt{cont}}) = (E', \varepsilon'), \\
& r \notin \{\varepsilon'(y) \mid y \in FV(E'_{\mathtt{cont}})\}, \quad \text{choose an unused register} \\
& \mathcal{R}_{\varepsilon', x \mapsto r}(E, z_{\mathtt{dest}}, E_{\mathtt{cont}}) = (E'', \varepsilon'') \text{ として} \\
& ((r \leftarrow E'; E''), \varepsilon'') \quad \text{when x is not a register} \\
\mathcal{R}_\varepsilon((r \leftarrow e; E), z_{\mathtt{dest}}, E_{\mathtt{cont}}) = {} & E'_{\mathtt{cont}} = (z_{\mathtt{dest}} \leftarrow E; E_{\mathtt{cont}}), \\
& \mathcal{R}_\varepsilon(e, r, E'_{\mathtt{cont}}) = (E', \varepsilon'), \\
& \mathcal{R}_{\varepsilon'}(E, z_{\mathtt{dest}}, E_{\mathtt{cont}}) = (E'', \varepsilon'') \text{ として} \\
& ((r \leftarrow E'; E''), \varepsilon'') \\
\mathcal{R}_\varepsilon(e, x, E_{\mathtt{cont}}) = {} & \mathcal{R}_\varepsilon(e, x, E_{\mathtt{cont}}) \quad \text{(次図参照)}
\end{aligned}$$

図 20: 単純なレジスタ割り当て $\mathcal{R}(P)$, $\mathcal{R}(D)$ および $\mathcal{R}_\varepsilon(E, z_{\mathtt{dest}}, E_{\mathtt{cont}})$。$\varepsilon$ は変数からレジスタへの写像、$z_{\mathtt{dest}}$ は E の結果をセットする変数、$E_{\mathtt{cont}}$ は E の後に実行される命令の列。$\mathcal{R}_\varepsilon(E, x, E_{\mathtt{cont}})$ の返り値はレジスタ割り当てされた命令の列 E' と、E の後のレジスタ割り当てを表す写像 ε' の組。[ファイル `regAlloc.notarget-nospill.ml` 参照]

a simple register allocation R(P), R(D), and R(E, z_dest, E_cont).
- ε is a mapping from variables to registers.
- z_dest is the variable where the result of E should be stored.
- E_cont is an instruction sequence that should be executed after the execution of E.
- The result of R_ε (E, x, E_cont) is a tuple of a register-allocated instruction sequence (E') and variable-register mapping (ε').

$\mathcal{R}$: Id.t M.t → SparcAsm.exp × Id.t × SparcAsm.t → SparcAsm.t × Id.t M.t

$\mathcal{R}_\varepsilon(c, z_{\mathtt{dest}}, E_{\mathtt{cont}}) = (c, \varepsilon)$

$\mathcal{R}_\varepsilon(\mathtt{L}_x, z_{\mathtt{dest}}, E_{\mathtt{cont}}) = (\mathtt{L}_x, \varepsilon)$

$\mathcal{R}_\varepsilon(op(x_1, \ldots, x_n), z_{\mathtt{dest}}, E_{\mathtt{cont}}) = (op(\varepsilon(x_1), \ldots, \varepsilon(x_n)), \varepsilon)$

$\mathcal{R}_\varepsilon(\mathtt{if}\ x = y\ \mathtt{then}\ E_1\ \mathtt{else}\ E_2, z_{\mathtt{dest}}, E_{\mathtt{cont}}) =$
$\mathcal{R}_\varepsilon(E_1, z_{\mathtt{dest}}, E_{\mathtt{cont}}) = (E_1', \varepsilon_1)$,
$\mathcal{R}_\varepsilon(E_2, z_{\mathtt{dest}}, E_{\mathtt{cont}}) = (E_2', \varepsilon_2)$,
$\varepsilon' = \{z \mapsto r \mid \varepsilon_1(z) = \varepsilon_2(z) = r\}$,
$\{z_1, \ldots, z_n\} =$
$(FV(E_{\mathtt{cont}}) \setminus \{z_{\mathtt{dest}}\} \setminus dom(\varepsilon')) \cap dom(\varepsilon)$ として
$((\mathtt{save}(\varepsilon(z_1), z_1); \ldots; \mathtt{save}(\varepsilon(z_n), z_n);$
$\mathtt{if}\ \varepsilon(x) \le \varepsilon(y)\ \mathtt{then}\ E_1'\ \mathtt{else}\ E_2'), \varepsilon')$

ε': env subset of variables consistently used for both "then" and "else" clause

$\mathcal{R}_\varepsilon(\mathtt{if}\ x \le y\ \mathtt{then}\ E_1\ \mathtt{else}\ E_2, z_{\mathtt{dest}}, E_{\mathtt{cont}}) =$ 同様

$\mathcal{R}_\varepsilon(x, z_{\mathtt{dest}}, E_{\mathtt{cont}}) = (\varepsilon(x), \varepsilon)$

$\mathcal{R}_\varepsilon(apply\_closure(x, y_1, \ldots, y_n), z_{\mathtt{dest}}, E_{\mathtt{cont}}) =$
$\{z_1, \ldots, z_n\} = (FV(E_{\mathtt{cont}}) \setminus \{z_{\mathtt{dest}}\}) \cap dom(\varepsilon)$ として
$((\mathtt{save}(\varepsilon(z_1), z_1); \ldots; \mathtt{save}(\varepsilon(z_n), z_n);$
$apply\_closure(\varepsilon(x), \varepsilon(y_1), \ldots, \varepsilon(y_n))), \emptyset)$

z's: live variables except z_dest

$\mathcal{R}_\varepsilon(apply\_direct(\mathtt{L}_x, y_1, \ldots, y_n), z_{\mathtt{dest}}, E_{\mathtt{cont}}) =$ 同様

$\mathcal{R}_\varepsilon(x.(y), z_{\mathtt{dest}}, E_{\mathtt{cont}}) = (\varepsilon(x).(\varepsilon(y)), \varepsilon)$

$\mathcal{R}_\varepsilon(x.(y) \leftarrow z, z_{\mathtt{dest}}, E_{\mathtt{cont}}) = (\varepsilon(x).(\varepsilon(y)) \leftarrow \varepsilon(z), \varepsilon)$

$\mathcal{R}_\varepsilon(\mathtt{save}(x, y), z_{\mathtt{dest}}, E_{\mathtt{cont}}) = (\mathtt{save}(\varepsilon(x), y), \varepsilon)$

$\mathcal{R}_\varepsilon(\mathtt{restore}(y), z_{\mathtt{dest}}, E_{\mathtt{cont}}) = (\mathtt{restore}(y), \varepsilon)$

図 21: 単純なレジスタ割り当て $\mathcal{R}_\varepsilon(e, z_{\mathtt{dest}}, E_{\mathtt{cont}})$。$\mathcal{R}_\varepsilon(e)$ の右辺で変数 x のレジスタ $\varepsilon(x)$ が定義されていない場合は、$\mathcal{R}_\varepsilon(e) = \mathcal{R}_\varepsilon(x \leftarrow \mathtt{restore}(x); e)$ とする。ただしレジスタ r については $\varepsilon(r) = r$ とする。[ファイル regAlloc.notarget-nospill.ml 参照]

A simple register allocation R_ε (e, z_dest, E_cont)
- when ε (x) is not defined for a variable x in the RHS of R_ε (e),
R_ε (e) = R_ε (x ← restore(x); e)
- ε (r) = r for all the register

E_cont is used to test liveness of variables

$\mathcal{T}$: `Id.t` $\to$ `SparcAsm.t` $\times$ `Id.t` $\to$ `bool` $\times$ `S.t`

$$
\begin{array}{lcl}
\mathcal{T}_x((y \leftarrow e; E), z_{\texttt{dest}}) & = & \mathcal{T}_x(e, y) = (c_1, s_1) \text{ として、もし } c_1 \text{ならば } (true, s_1) \\
 & & \text{そうでなければ } \mathcal{T}_x(E, z_{\texttt{dest}}) = (c_2, s_2) \text{ として } (c_2, s_1 \cup s_2) \\
\mathcal{T}_x(e, z_{\texttt{dest}}) & = & \mathcal{T}_x(e, z_{\texttt{dest}})
\end{array}
$$

$\mathcal{T}$: `Id.t` $\to$ `SparcAsm.exp` $\times$ `Id.t` $\to$ `bool` $\times$ `S.t`

$$
\begin{array}{lcl}
\mathcal{T}_x(x, z_{\texttt{dest}}) & = & (false, \{z_{\texttt{dest}}\}) \\
\mathcal{T}_x(\texttt{if } y = z \texttt{ then } E_1 \texttt{ else } E_2, z_{\texttt{dest}}) & = & \mathcal{T}_x(E_1, z_{\texttt{dest}}) = (c_1, s_1), \\
 & & \mathcal{T}_x(E_2, z_{\texttt{dest}}) = (c_2, s_2) \text{ として} \\
 & & (c_1 \wedge c_2, s_1 \cup s_2) \\
\mathcal{T}_x(\texttt{if } y \leq z \texttt{ then } E_1 \texttt{ else } E_2, z_{\texttt{dest}}) & = & \text{同上} \\
\mathcal{T}_x(apply\_closure(y_0, y_1, \ldots, y_n), z_{\texttt{dest}}) & = & (true, \{\texttt{R}_i \mid x = y_i\}) \\
\mathcal{T}_x(apply\_direct(\texttt{L}_y, y_1, \ldots, y_n), z_{\texttt{dest}}) & = & \text{同上} \\
\mathcal{T}_x(e, z_{\texttt{dest}}) & = & (false, \emptyset) \qquad \text{それ以外の場合}
\end{array}
$$

図 22: 変数 x に割り当てるレジスタ r を選ぶときに使う targeting $\mathcal{T}_x(E, z_{\texttt{dest}})$ および $\mathcal{T}_x(e, z_{\texttt{dest}})$。$E$ や e で関数呼び出しがあったかどうかを表す論理値 c と、x を割り当てると良いレジスタの集合 s の組を返す。前々図の「x がレジスタでない場合」において、$\mathcal{T}_x(E'_{\texttt{cont}}, z_{\texttt{dest}}) = (c, s)$ として、できれば $r \in s$ とする。[ファイル `regAlloc.target-nospill.ml` 参照]

Register selection scheme for a variable x.
- c: if E or e issues function application
- s: recommended set of registers for use of variable x

The result of register targeting can be used in choosing a fresh register r for in Fig. 20

$\mathcal{R}$: `Id.t M.t` $\to$ `SparcAsm.t` $\times$ `Id.t` $\times$ `SparcAsm.t` $\to$ `SparcAsm.t` $\times$ `Id.t M.t`

$$
\begin{array}{lcl}
\mathcal{R}_\varepsilon((x \leftarrow e; E), z_{\texttt{dest}}, E_{\texttt{cont}}) & = & E'_{\texttt{cont}} = (z_{\texttt{dest}} \leftarrow E; E_{\texttt{cont}}), \\
 & & \mathcal{R}_\varepsilon(e, x, E'_{\texttt{cont}}) = (E', \varepsilon'), \\
 & & y \in FV(E'_{\texttt{cont}}), \\
 & & \mathcal{R}_{\varepsilon' \setminus \{y \mapsto \varepsilon'(y)\}, x \mapsto \varepsilon'(y)}(E, z_{\texttt{dest}}, E_{\texttt{cont}}) = (E'', \varepsilon'') \text{ として} \\
 & & \begin{cases} ((\texttt{save}(\varepsilon(y), y); \varepsilon'(y) \leftarrow E'; E''), \varepsilon'') & y \in dom(\varepsilon) \text{ のとき} \\ ((\varepsilon'(y) \leftarrow E'; E''), \varepsilon'') & y \notin dom(\varepsilon) \text{ のとき} \end{cases} \\
 & & \qquad x \text{ がレジスタでなく、} \\
 & & \qquad r \notin \{\varepsilon'(y) \mid y \in FV(E'_{\texttt{cont}})\} \text{ なる } r \text{ がない場合}
\end{array}
$$

図 23: spilling をするレジスタ割り当て $\mathcal{R}_\varepsilon(E, z_{\texttt{dest}}, E_{\texttt{cont}})$ [ファイル `regAlloc.target-latespill.ml` 参照]

$\mathcal{S}$: `SparcAsm.prog` $\rightarrow$ `string`

$$
\begin{array}{lcl}
\mathcal{S}((\{D_1, \ldots, D_n\}, E)) & = & \texttt{.section ".text"} \\
 & & \mathcal{S}(D_1) \\
 & & \ldots \\
 & & \mathcal{S}(D_n) \\
 & & \texttt{.global min\_caml\_start} \\
 & & \texttt{min\_caml\_start:} \\
 & & \texttt{save \%sp, -112, \%sp} \\
 & & \mathcal{S}(E, \texttt{\%g0}) \\
 & & \texttt{ret} \\
 & & \texttt{restore}
\end{array}
$$

$\mathcal{S}$: `SparcAsm.fundef` $\rightarrow$ `string`

$$
\begin{array}{lcl}
\mathcal{S}(\mathtt{L}_x(y_1, \ldots, y_n) = E) & = & x\texttt{:} \\
 & & \mathcal{S}(E, \mathtt{R}_0) \\
 & & \texttt{retl} \\
 & & \texttt{nop}
\end{array}
$$

$\mathcal{S}$: `SparcAsm.t` $\times$ `Id.t` $\rightarrow$ `string`

$$
\begin{array}{lcl}
\mathcal{S}((x \leftarrow e; E), z_{\texttt{dest}}) & = & \mathcal{S}(e, x) \\
 & & \mathcal{S}(E, z_{\texttt{dest}}) \\
\mathcal{S}(e, z_{\texttt{dest}}) & = & \mathcal{S}(e, z_{\texttt{dest}})
\end{array}
$$

図 24: 単純なアセンブリ生成 $\mathcal{S}(P)$, $\mathcal{S}(D)$ および $\mathcal{S}(E, z_{\texttt{dest}})$

$\mathcal{S}$: SparcAsm.exp $\times$ Id.t $\to$ string

$\mathcal{S}(c, z_{\mathtt{dest}})$	=	set $c, z_{\mathtt{dest}}$
$\mathcal{S}(\mathtt{L}_x, z_{\mathtt{dest}})$	=	set $\mathtt{L}_x, z_{\mathtt{dest}}$
$\mathcal{S}(op(x_1, \ldots, x_n), z_{\mathtt{dest}})$	=	op $x_1, \ldots, x_n, z_{\mathtt{dest}}$
$\mathcal{S}(\mathtt{if}\ x = y\ \mathtt{then}\ E_1\ \mathtt{else}\ E_2, z_{\mathtt{dest}})$	=	cmp x, y
		bne b_1
		nop
		$\mathcal{S}(E_1, z_{\mathtt{dest}})$
		b b_2
		nop
		b_1:
		$\mathcal{S}(E_2, z_{\mathtt{dest}})$
		b_2:
$\mathcal{S}(\mathtt{if}\ x \leq y\ \mathtt{then}\ E_1\ \mathtt{else}\ E_2, z_{\mathtt{dest}})$	=	同様
$\mathcal{S}(x, z_{\mathtt{dest}})$	=	mov $x, z_{\mathtt{dest}}$
$\mathcal{S}(apply\_closure(x, y_1, \ldots, y_n), z_{\mathtt{dest}})$	=	$shuffle((x, y_1, \ldots, y_n), (\mathtt{R}_0, \mathtt{R}_1, \ldots, \mathtt{R}_n))$
		st $\mathtt{R}_{\mathtt{ra}}, [\mathtt{R}_{\mathtt{st}} + 4\#\varepsilon]$
		ld $[\mathtt{R}_0], \mathtt{R}_{n+1}$
		call $\mathtt{R}_{n+1}$
		add $\mathtt{R}_{\mathtt{st}}, 4(\#\varepsilon + 1), \mathtt{R}_{\mathtt{st}}$! *delay slot*
		sub $\mathtt{R}_{\mathtt{st}}, 4(\#\varepsilon + 1), \mathtt{R}_{\mathtt{st}}$
		ld $[\mathtt{R}_{\mathtt{st}} + 4\#\varepsilon], \mathtt{R}_{\mathtt{ra}}$
		mov $\mathtt{R}_0, z_{\mathtt{dest}}$
$\mathcal{S}(apply\_direct(\mathtt{L}_x, y_1, \ldots, y_n), z_{\mathtt{dest}})$	=	$shuffle((y_1, \ldots, y_n), (\mathtt{R}_1, \ldots, \mathtt{R}_n))$
		st $\mathtt{R}_{\mathtt{ra}}, [\mathtt{R}_{\mathtt{st}} + 4\#\varepsilon]$
		call x
		add $\mathtt{R}_{\mathtt{st}}, 4(\#\varepsilon + 1), \mathtt{R}_{\mathtt{st}}$! *delay slot*
		sub $\mathtt{R}_{\mathtt{st}}, 4(\#\varepsilon + 1), \mathtt{R}_{\mathtt{st}}$
		ld $[\mathtt{R}_{\mathtt{st}} + 4\#\varepsilon], \mathtt{R}_{\mathtt{ra}}$
		mov $\mathtt{R}_0, z_{\mathtt{dest}}$
$\mathcal{S}(x.(y), z_{\mathtt{dest}})$	=	ld $[x + y], z_{\mathtt{dest}}$
$\mathcal{S}(x.(y) \leftarrow z, z_{\mathtt{dest}})$	=	st $z, [x + y]$
$\mathcal{S}(\mathtt{save}(x, y), z_{\mathtt{dest}})$	=	もし $y \notin dom(\varepsilon)$ なら ε に $y \mapsto 4\#\varepsilon$ を加えて
		st $x, [\mathtt{R}_{\mathtt{st}} + \varepsilon(y)]$
$\mathcal{S}(\mathtt{restore}(y), z_{\mathtt{dest}})$	=	ld $[\mathtt{R}_{\mathtt{st}} + \varepsilon(y)], z_{\mathtt{dest}}$

図 25: 単純なアセンブリ生成 $\mathcal{S}(e, z_{\mathtt{dest}})$。$\varepsilon$ はスタック位置を記憶するグローバル変数。$\#\varepsilon$ は ε の要素の個数。$shuffle((x_1, \ldots, x_n), (r_1, \ldots, r_n))$ は $x_1, \ldots, x_n$ を $r_1, \ldots, r_n$ に適切な順序で移動する命令。

$\mathcal{S} : \texttt{S.t} \to \texttt{SparcAsm.t} \times \texttt{Id.t} \to \texttt{S.t} \times \texttt{string}$

$\mathcal{S}_s((x \leftarrow e; E), z_{\texttt{dest}}) = \mathcal{S}_s(e, x) = (s', S)$,
$\mathcal{S}_{s'}(E, z_{\texttt{dest}}) = (s'', S')$ として
(s'', SS')

$\mathcal{S}_s(e, z_{\texttt{dest}}) = \mathcal{S}_s(e, z_{\texttt{dest}})$

$\mathcal{S} : \texttt{S.t} \to \texttt{SparcAsm.exp} \times \texttt{Id.t} \to \texttt{S.t} \times \texttt{string}$

$\mathcal{S}_s(\texttt{if } x = y \texttt{ then } E_1 \texttt{ else } E_2, z_{\texttt{dest}}) = \mathcal{S}_s(E_1, z_{\texttt{dest}}) = (s_1, S_1)$,
$\mathcal{S}_s(E_2, z_{\texttt{dest}}) = (s_2, S_2)$ として
$(s_1 \cap s_2,$
$\texttt{cmp } x, y$
$\texttt{bne } b_1$
$\texttt{nop}$
S_1
$\texttt{b } b_2$
$\texttt{nop}$
$b_1\texttt{:}$
S_2
$b_2\texttt{:})$

$\mathcal{S}_s(\texttt{if } x \leq y \texttt{ then } E_1 \texttt{ else } E_2, z_{\texttt{dest}}) =$ 同様

$\mathcal{S}_s(\texttt{save}(x, y), z_{\texttt{dest}}) = (s, \texttt{nop})$ $y \in s$ の場合

$\mathcal{S}_s(\texttt{save}(x, y), z_{\texttt{dest}}) =$ もし $y \notin dom(\varepsilon)$ なら ε に $y \mapsto 4\#\varepsilon$ を加えて
$(s \cup \{y\}, \texttt{st } x, [\texttt{R}_{\texttt{st}} + \varepsilon(y)])$ $y \notin s$ の場合

$\mathcal{S}_s(e, z_{\texttt{dest}}) = (s,$ 以前と同様$)$ 上述以外の場合

図 26: 無駄な save を省略するアセンブリ生成 $\mathcal{S}_s(E, z_{\texttt{dest}})$ および $\mathcal{S}_s(e, z_{\texttt{dest}})$。$s$ はすでに save された変数の名前の集合。以前の $\mathcal{S}(E, z_{\texttt{dest}})$ は $\mathcal{S}_\emptyset(E, z_{\texttt{dest}}) = (s, S)$ として S の略記とする。

$\mathcal{S}$: SparcAsm.fundef $\rightarrow$ string

$\mathcal{S}(\mathtt{L}_x(y_1, \ldots, y_n) = E)$ = $\mathcal{S}_\emptyset(E, \mathtt{tail}) = (s, S)$ として

x:

S

$\mathcal{S}$: S.t $\rightarrow$ SparcAsm.exp $\times$ Id.t $\rightarrow$ S.t $\times$ string

$\mathcal{S}_s(\mathtt{if}\ x = y\ \mathtt{then}\ E_1\ \mathtt{else}\ E_2, \mathtt{tail})$ = $\mathcal{S}_s(E_1, \mathtt{tail}) = (s_1, S_1)$,

$\mathcal{S}_s(E_2, \mathtt{tail}) = (s_2, S_2)$ として

($\emptyset$,

cmp x, y

bne b

nop

S_1

b:

S_2)

$\mathcal{S}_s(\mathtt{if}\ x \leq y\ \mathtt{then}\ E_1\ \mathtt{else}\ E_2, \mathtt{tail})$ = 同様

$\mathcal{S}_s(\mathit{apply\_closure}(x, y_1, \ldots, y_n), \mathtt{tail})$ = ($\emptyset$,

$\mathit{shuffle}((x, y_1, \ldots, y_n), (\mathtt{R}_0, \mathtt{R}_1, \ldots, \mathtt{R}_n))$

ld $[\mathtt{R}_0], \mathtt{R}_{n+1}$

jmp $\mathtt{R}_{n+1}$

nop)

$\mathcal{S}_s(\mathit{apply\_direct}(\mathtt{L}_x, y_1, \ldots, y_n), \mathtt{tail})$ = ($\emptyset$,

$\mathit{shuffle}((y_1, \ldots, y_n), (\mathtt{R}_1, \ldots, \mathtt{R}_n))$

b x

nop)

$\mathcal{S}_s(e, \mathtt{tail})$ = $\mathcal{S}_s(e, \mathtt{R}_0) = (s', S)$ として

($\emptyset$,

S

retl

nop) 上述以外の場合

図 27: 末尾呼び出し最適化をするアセンブリ生成 $\mathcal{S}_s(D)$ および $\mathcal{S}_s(e, z_{\mathtt{dest}})$。$z_{\mathtt{dest}} = \mathtt{tail}$ の場合が末尾。