Flow Analytic Type System for Array Bound Checks

MATSUNO Yutaka [1]

Department of Frontier Informatics
The University of Tokyo
Tokyo, Japan

SATO Hiroyuki [2]

Information Technology Center
The University of Tokyo
Tokyo, Japan

Abstract

It is widely recognized that formulating program analysis by a type system is a promising approach because of its clarity and rigidity. However, there remains much to be done for applying them to practical use. One of the problems is that it is not trivial what kind of type systems is appropriate for low level languages. To solve the problem, the type systems must be closely related to data flow analysis because it has been the major method for analyzing low level languages. In this paper, taking array bound checks as an example, first we propose a framework for type systems for low level languages derived from data flow analysis. Second, we propose a type system for analyzing programs as a network of blocks(especially loops), dealing with SSA form and induction variables.

1 Introduction

It is widely recognized that formulating program analysis by a type system is a promising approach because of its clarity and rigidity. Morrisett et al. introduce *typed assembly language* in which labels are typed([10,11]). [10] shows that type systems are useful for analyzing low level languages. For example,

[1] Email: matsu@pi.cc.u-tokyo.ac.jp
[2] Email: schuko@pi.cc.u-tokyo.ac.jp

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

certifying assembly codes with types protects memory from violations by unsafe codes. However, there remains much to be done for applying them to practical use. One of the problems is that it is not trivial what kind of type is appropriate for low level languages. Specifically, we must associate some kind of types with *jumps*, which characterize low level languages. A hint is that data flow analysis in conventional compilers is a major concern for analyzing programs written in low level languages. Therefore, type systems for low level languages must be based on data flow analysis. Many previous works of type systems for low level languages essentially use data flow analysis though they do not address explicitly, or their systems would be much simpler if they use data flow analysis more directly.

In this paper, we take a low level language as our target, and analyze array bound checks in type theoretical way. Because array bound checks essentially use data flow analysis(for example, [6] uses *available expressions* for eliminating redundant checks), this is a typical example of type theory for low level languages.

The object for analyzing array bound checks is ranges of array subscripts. If a compiler detects that the value of subscripts are within the array bounds, no checks need to be inserted. Otherwise we must insert codes for run time checks. Therefore we introduce *range types* for registers. *Range types* are of the form $r : [1, 10]$, which means that the value of r is within $1 \leq r \leq 10$ at that program point. A program of our target is represented by a network of basic blocks. With each basic block in a program, a type is associated, which we call *basic block types*. If all basic blocks have types, then the program is well typed. Next, we extend our types so that previous studies on compiler optimizations can be formulated in type theoretical way. For an example, we introduce *IVrange* types. If a register is detected as an *induction variable*, it is typed as $IVrange$, a subtype of range types. Because many of array subscripts in loops are affine conjunctions of induction variables, these types are useful enough in the fields of science and engineering programs which consist mainly of nested loops such as Gaussian elimination(Program 1).

Program 1 (Gaussian Elimination)

```
for(k=0;k<=N-2;k++){
  for(i=k+1;i<=N-1;i++){
    m[i][k]=a[i][k]/a[k][k];
    for(j=k;j<=N-1;j++)
      a[i][j]=a[i][j]-m[i][k]*a[k][j];
  }
}
```

As mentioned in [15,5] which propose an algorithm to detect induction variables, *Static Single Assignment form*(SSA form) is useful for analyzing induction variables and loops. Therefore we define typing rules on SSA form, which is not mentioned in previous type theoretical papers. The compilation stages

are as follows.

Target Language → SSA form → Target Language

On our target language, we analyze basic blocks and complete programs. In SSA form, we analyze induction variables in loops. Translation to SSA form and set back to original languages are explained in [4].

Our Contribution

We list our contributions as follows.

- We propose a framework of type systems for low level languages which corresponds to traditional data flow analysis. This will be the core of type systems for low level languages
- Besides, as an extension of the framework, we take a program as a network of basic blocks and analyze them in a way as usual in studies of compiler optimizations. Specially, we focus on loop analysis, dealing with induction variables and SSA form.

The rest of this paper is: in section 2, we show preliminaries for this paper. Section 3 introduces type systems for basic blocks and for complete programs, which we call flow analytic type system. Then in section 4 we introduce type inference rules for induction variables in loops. Section 5 shows soundness of our type system. Section 6 lists related works. In section 7, we give a summary of this paper and future works.

2 Preliminaries

2.1 Syntax for Target Language

We define syntax of our target language as Fig 1. The language is a small subset of RISC-like assembly languages. Register file consists of n_r registers and heap consists of a sequence of n_h heap elements with addresses which can contain word size values. To focus on our interest we assume:

- n_r and n_h are large enough.
- Arrays are already allocated in the heap.
- Registers which contain heap addresses point to valid offset of the arrays at the beginning of the program.

We explain here `load` and `store`, which are our main concerns.

- `load` and `store`
  "`load` $r_s(v), r_d$" loads a heap value whose heap address is calculated by adding v to the heap address stored in register r_s, the pointer for an array. if v is greater than the length of the array, memory violation occurs. Our type system tries to detect whether this will happen or not in executions of `load` and `store`. If it cannot be proved that the array access is not within the bound of the array, our type checker reports error.

$$
\begin{array}{rrl}
\textit{integers} & i & \in I \\
\textit{labels} & l & \in L \\
\\
\textit{heap address} & h & ::= h_0 \mid h_1 \mid \ \ldots \ \mid h_{n_h-1} \\
\textit{registers} & r & ::= r_0 \mid r_1 \mid \ \ldots \ \mid r_{n_r-1} \\
\textit{word values} & wv & ::= i \mid h \\
\textit{values} & v & ::= i \mid r \\
\\
\textit{heap} & H & ::= [h_0 \mapsto wv_0, h_1 \mapsto wv_1, \ldots, h_{n_h-1} \mapsto wv_{n_h-1}] \\
\textit{register file} & R & ::= [r_0 \mapsto wv_0, r_1 \mapsto wv_1, \ldots, r_{n_r-1} \mapsto wv_{n_r-1}] \\
\textit{stack} & S & ::= [] \mid wv :: S \\
\\
\textit{arithmetic ops} & \texttt{aop} & ::= \texttt{add} \mid \texttt{sub} \mid \texttt{mul} \mid \texttt{div} \\
\textit{branch ops} & \texttt{bop} & ::= \texttt{beq} \mid \texttt{bne} \mid \texttt{blt} \mid \texttt{blte} \mid \texttt{bgt} \mid \texttt{bgte} \\
\textit{branch instructions} & \textit{bins} & ::= \texttt{bop}\ r, v, l \mid \texttt{jmp}\ l \mid \texttt{halt} \\
\textit{instructions} & \textit{ins} & ::= \texttt{aop}\ r_s, v, r_d \mid \texttt{mov}\ v, r \mid \\
 & & \quad \texttt{load}\ r_s(v), r_d \mid \texttt{store}\ v_s, r_d(v) \mid \\
\\
\textit{instruction sequences} & I & ::= [\, \textit{empty instructions} \,] \mid \textit{ins} \mid \textit{ins}; I \\
\textit{basic blocks} & B & ::= I \mid I; \textit{bins} \\
\textit{programs} & P & ::= l_0 : B_0 \ ; \ l_1 : B_1 \ ; \ \ldots \ ; \ l_{n-1} : B_{n-1}
\end{array}
$$

Fig. 1. Syntax for Target Language

2.1.1 Abstract Machine

We consider an abstract machine by a standard approach. A machine state M is a triple (pc, H, R). pc stands for program counter which starts from 0 to n_i-1 where n_i is the number of instructions. $P[pc]$ denotes an instruction whose order is pc in program P. H and R are finite mappings which stand for a heap and a register file respectively.

Notation 1 (map functions)

Let g be a map, $Dom(g)$ be the domain of g; for $x \in Dom(g)$, $g[x]$ is the value of g at x, and $g[x \to v]$ is the map with the same domain as g defined by the following equation, for all $y \in Dom(g)$: $(g[x \to v])[y] = if\ y \neq x\ then\ g[y]\ else\ v$.

2.1.2 Dynamic Semantics

We model the state of an execution as follows.

- Initial machine state: $M_0 = (0, H_0, R_0)$ where H_0 and R_0 correspond to the initial state of the heap and register.
- Program executions are denoted as $P \vdash (pc, H, R) \rightarrow (pc', H', R')$, which means that program P can, in one step, go from state (pc, H, R) to state (pc', H', R').
- If the program reaches the `halt` instruction, the state transfers to *HALT* state.

Fig. 2 contains a small-step operational semantics for some instructions where $M = (pc, H, R)$. We also define M as maps: if $v \in Dom(R)$ then $M[v] = R[v]$ else $M[v] = i$ when v is $i \in I$. Furthermore, We define a set of labels $L = \{l_0, l_1, \ldots, l_{n-1}\}$ as finite maps to the set of order of instruction with which they are associated.

$$\frac{P[pc] = \texttt{add}\ r_s, v, r_d \quad M[r_s] = i \quad M[v] = j}{P \vdash M \rightarrow (pc+1, H, R[r_d \mapsto (i+j)])}$$

$$\frac{P[pc] = \texttt{mov}\ v, r_d}{P \vdash M \rightarrow (pc+1, H, R[r_d \mapsto M[v]])}$$

$$\frac{P[pc] = \texttt{load}\ r_s(v), r_d \quad R[r_s] = h \quad M[v] = i}{P \vdash M \rightarrow (pc+1, H, R[r_d \mapsto (H[h_{(R[r_s]+M[v])}])])}$$

$$\frac{P[pc] = \texttt{store}\ v_s, r_d(v) \quad R[r_d] = h \quad M[v] = i}{P \vdash M \rightarrow (pc+1, H[h_{(R[r_d]+M[v])} \mapsto M[v_s]], R)}$$

$$\frac{P[pc] = \texttt{jmp}\ l \quad l \in L}{P \vdash M \rightarrow (L[l], H, R)}$$

$$\frac{P[pc] = \texttt{bop}\ r, v, l \quad R[r] = i \quad M[v] = j}{P \vdash M \rightarrow (pc+1, H, R)} \quad (\texttt{bop} - \textbf{False})$$

$$\frac{P[pc] = \texttt{bop}\ r, v, l \quad l \in L \quad R[r] = i \quad M[v] = j}{P \vdash M \rightarrow (L[l], H, R)} \quad (\texttt{bop} - \textbf{True})$$

$$\frac{P[pc] = \texttt{halt}}{P \vdash M \rightarrow HALT} \quad (\texttt{HALT})$$

Fig. 2. Dynamic semantics for some instructions from $M = (pc, H, R)$

2.2 Types

We define types as Fig.4:

- basic types
  - · *int*. In this paper we only consider integers as a data type for simplicity.
  - · *const*. When analyzing programs, sometimes it is useful to assume that the value of a register as invariant. For such registers we use *const*. Spe-

cially, this type is essential for loop analysis as in section 4. Note that a register typed as *const* should not be the destination register for arithmetic, move and load instructions. We omit the details in this paper.

- range types
  If the range of a register can be deduced, we use *range types*. This is a kind of dependent types. For expressions of range types, we allow affine conjunctions of integers and registers which are typed as *const*. Also, we allow min and max functions as Fig.3.
- IV range types
  These types are for induction variables. If a register which has range type is deduced as an induction variable, we annotate its range type as IV. We will explain them in section 5.
- dimension types
  These types are given to registers which are pointers for arrays. e denotes the length of the array. We allow multi-dimensional arrays. For simplicity we assume that typing dimension types to a register is valid.
- heap types and register file types
  We define these types as maps, such as $R_\tau(r)$ equals to the type of register r in the register file.
- basic block types
  A basic block type consists of a register file type at the entry of the block.

$$\textit{range expressions} \quad e ::= i \mid r \mid i \cdot e \mid e_1 + e_2 \mid e_1 - e_2 \mid \min(e_1, e_2, \cdots) \mid \max(e_1, e_2, \cdots)$$

Fig. 3. Range Expressions

2.2.1 Subtyping Rule

Relations between types and a subtyping rule are defined in Fig.5. A range type $[e_1, e_2]$ is a subtype of another range type $[e_3, e_4]$ if $e_1 \geq e_3$ and $e_2 \leq e_4$ can be deduced. Range types are subtypes of *int*. IV range types are subtypes of range types and *const* is a subtype of *int*. We can extend these relations to heap types and register file types easily. For example, if $R_\tau = [r_1 : [1, 2], r_2 : IV[1, 2]]$ and $R'_\tau = [r_1 : int, r_2 : [1, 2]]$, we can deduce that $R_\tau \leq R'_\tau$.

3 Static Semantics

Static semantics of our type system consists of two levels: semantics inside a basic blocks and the one for complete programs.

$$
\begin{array}{rl}
\textit{basic types} & \tau_b ::= \textit{int} \mid \textit{const} \mid \bot \mid \top \\
\textit{range types} & \tau_r ::= [e_1, e_2] \\
\textit{IVrange types} & \tau_{iv} ::= IV\tau_r \\
\textit{dimension types} & \tau_d ::= \textit{int array}(e) \mid \tau_d \textit{ array}(e) \\
 & \\
\textit{types} & \tau ::= \tau_b | \tau_r \mid \tau_{iv} \mid \tau_d \\
\textit{heap types} & H_\tau ::= [h_0 : \tau_0, h_1 : \tau_1, \ldots, h_{n_h-1} : \tau_{n_h-1}] \\
\textit{register file types} & R_\tau ::= [r_0 : \tau_0, r_1 : \tau_1, \ldots, r_{n_r-1} : \tau_{n_r-1}] \\
\textit{basic block types} & \phi ::= R_\tau
\end{array}
$$

Fig. 4. Types

$$
\frac{e_1 \geq e_3 \quad e_2 \leq e_4}{[e_1, e_2] \leq [e_3, e_4]} \qquad \frac{}{\textit{const} \leq \textit{int}} \qquad \frac{}{IV\tau_r \leq \tau_r}
$$

$$
\frac{}{\bot \leq \tau} \qquad \frac{}{\tau \leq \top} \qquad \frac{}{\tau_r \leq \textit{int}} \qquad \frac{}{\tau \leq \tau}
$$

$$
\frac{\phi \vdash r : \tau_1 \quad \tau_1 \leq \tau_2}{\phi \vdash r : \tau_2}
$$

Fig. 5. Subtyping Rules

3.1 For Basic Blocks

We show static semantics for basic blocks as Fig.6. ϕ_{init} denotes the register file type at the entry of the basic blocks. This semantics uses three judgments:

- $\phi \vdash r : \tau$
  This ensures that if $\phi = R_\tau$, then $R_\tau(r) = \tau$.
- $\phi \vdash I$
  This judgment says that instruction sequence I is well typed under ϕ.
- $\phi \vdash B(\phi)$
  We denote $B(\phi)$ as a register file type at the exit of block B. This judgment says that $B(\phi)$ can be deduced if we assume that the register file type at the entry of B is ϕ.

$\phi \vdash I$ is derived from the following static semantics.

- int and const rules
  Integers can be typed as *int* or $[i, i]$ for any ϕ. Also, register r typed as *const* for ϕ can be typed as $[r, r]$ for ϕ.
- arithmetic instructions

Range types are introduced for deducing ranges of registers value statically. However, it is impossible to deduce ranges for all calculations: for example, it involves complicated judgments for typing register r_d in `mul` r_s, v, r_d where r_s and v have range types. Therefore we only focus on range types of affine conjunctions on integers and registers. This is ensured by the following lemma.

Lemma 3.1 *If $\phi \vdash r : \tau_r$, then expressions appearing in τ_r are restricted to affine conjunctions of registers typed as const and integers.*

Proof. By inspections on static semantics of arithmetic instructions. □

If type checker cannot use this semantics for typing arithmetic instructions, it is allowed to use trivial semantics such as

$$\frac{\phi \vdash r_s : int, v : int \quad \phi \vdash I}{\phi[r_d : int] \vdash I \,;\, \texttt{mul}\ r_s, v, r_d} \quad \textbf{(type-mul-int)}$$

- branch instructions and the last instructions of basic blocks
  These derive a judgment $\phi \vdash B(\phi)$. This ensures that inside of the basic block is well typed.
- `load` and `store`
  These require that array references are within bounds. For this to be satisfied, $\tau_r \sqsubseteq [0, e-1]$(see Definition 3.4) must be proved.

3.2 For Complete Programs

For complete programs, we define *Flow Analytic Type System* derived from data flow analysis. In syntax of our target language, we define a program as a network of basic blocks as usual in data flow analysis whereas in previous works it is defined as just a sequence of instructions.

Notation 2

A basic block which has predecessors $P_1, P_2, \ldots P_n$ and successors S_1, S_2 is denoted as:

$$(P_1, P_2, \ldots P_n)B(S_1, S_2)$$

Sometimes we abbreviate it just as B.

Definition 3.2 [$\sqcup$]

- Case 1: τ and τ' are range types.

$$[e_1, e_2] \sqcup [e_3, e_4] = [\min(e_1, e_3), \max(e_2, e_4)]$$

- Case 2: $\tau \sqcup \tau'$ where τ and τ' are dimension types.
  Only if two types are equivalent we get $\tau \sqcup \tau = \tau$, otherwise $\tau_d \sqcup \tau'_d = \top$

Definition 3.3 [$\bigsqcup$]

$$\frac{}{\phi \vdash i : int} \quad \frac{}{\phi \vdash i : [i,i]} \quad \textbf{(int rule)}$$

$$\frac{\phi \vdash r : const}{\phi \vdash r : [r,r]} \quad \textbf{(const rule)}$$

$$\frac{}{\phi_{init} \ \vdash \ [\, empty\ instructions \,]} \quad \textbf{(start rule)}$$

$$\frac{\phi \vdash r_d : [e_1,e_2],\ v : [e_3,e_4] \quad \phi \vdash I}{\phi[r_d : [e_1+e_3, e_2+e_4]] \vdash I \ ;\ \texttt{add}\ r_s, v, r_d} \quad \textbf{(type-add)}$$

$$\frac{\phi \vdash r_s : [e_1,e_2],\ v : [e_3,e_4] \quad \phi \vdash I}{\phi[r_d : [e_1-e_4, e_2-e_3]] \vdash I \ ;\ \texttt{sub}\ r_s, v, r_d} \quad \textbf{(type-sub)}$$

$$\frac{\phi \vdash r_s : [e_1,e_2],\ v = i \geq 0 \quad \phi \vdash I}{\phi[r_d : [i \cdot e_1, i \cdot e_2]] \vdash I \ ;\ \texttt{mul}\ r_s, v, r_d} \quad \textbf{(type-mul} \geq \textbf{0)}$$

$$\frac{\phi \vdash r_s : [e_1,e_2],\ v = i < 0 \quad \phi \vdash I}{\phi[r_d : [i \cdot e_2, i \cdot e_1]] \vdash I \ ;\ \texttt{mul}\ r_s, v, r_d} \quad \textbf{(type-mul} < \textbf{0)}$$

$$\frac{\phi \vdash r_s : int,\ v = i \neq 0 \quad \phi \vdash I}{\phi[r_d : int] \vdash I \ ;\ \texttt{div}\ r_s, v, r_d} \quad \textbf{(type-div)}$$

$$\frac{B = I \ ;\ \texttt{bop}\ r, l \quad \phi_{exit} \vdash I}{\phi_{init} \vdash B(\phi_{init})(\equiv \phi_{exit})} \quad \textbf{(type-bop)}$$

$$\frac{\phi \vdash r_s : \tau\ array(i),\ v : \tau_r,\ r_s(v) : \tau \quad \tau_r \sqsubseteq [0, i-1] \quad \phi \vdash I}{\phi[r_d : \tau] \vdash I \ ;\ \texttt{load}\ r_s(v), r_d} \quad \textbf{(type-load)}$$

$$\frac{\phi \vdash r_s : \tau,\ r_d : \tau\ array(i),\ v : \tau_r \quad \tau_r \sqsubseteq [0, i-1] \quad \phi \vdash I}{\phi \ \vdash \ I \ ;\ \texttt{store}\ r_s, r_d(v)} \quad \textbf{(type-store)}$$

$$\frac{\phi \vdash v : \tau \quad \phi \vdash I}{\phi[r : \tau] \vdash I \ ;\ \texttt{mov}\ v, r} \quad \textbf{(type-mov)} \quad \frac{B = I; \texttt{jmp}\ l \quad \phi_{exit} \vdash I}{\phi_{init} \vdash B(\phi_{init})(\equiv \phi_{exit})} \quad (\textbf{type-jmp})$$

$$\frac{B = I; \texttt{halt} \quad \phi_{exit} \vdash I}{\phi_{init} \vdash B(\phi_{init})(\equiv \phi_{exit})} \quad \textbf{(type-halt)} \quad \frac{B = I \quad \phi_{exit} \vdash I}{\phi_{init} \vdash B(\phi_{init})(\equiv \phi_{exit})} \quad \textbf{(end rule)}$$

Fig. 6. Static Semantics

$$\bigsqcup\{\tau_1, \ldots \tau_n\} = \bigsqcup_{i=1}^{n} \tau_i$$

$$\bigsqcup\{R_{\tau_1}, \ldots, R_{\tau_n}\} = [\bigsqcup_{i=1}^{n} R_{\tau_i}(r_1), \ldots, \bigsqcup_{i=1}^{n} R_{n_r-1}(r_{n_r-1})]$$

Definition 3.4 [$\sqsubseteq$]

- $\tau_1 \sqsubseteq \tau_2$ iff $\tau_1 \leq \tau_2$, i.e. τ_1 is a subtype of τ_2.

- If $\forall r.R_\tau(r) \sqsubseteq R'_\tau(r)$, then $R_\tau \sqsubseteq R'_\tau$.
- Assume that $\phi = R_\tau$ and $\phi' = R'_\tau$. $\phi \sqsubseteq \phi'$ iff $R_\tau \sqsubseteq R'_\tau$.

Our main type system is shown in Fig.7. Consider a basic block B which has predecessors $P_1, P_2, \cdots, P_n$ and successors S_1 and S_2. Note that blocks have at most two successors because branch instructions allow only two branches. Conditions for the typability of B are (we consider here two successors): (1) $P_1, P_2, \cdots, P_n$ have types $\phi_{p_1}, \phi_{p_2}, ...\phi_{p_n}$, (2) successors have types ϕ_{s_1} and ϕ_{s_2} , (3) $B(\phi)$ can be deduced from ϕ where $\phi = \bigsqcup P_i(\phi_{p_i})$, and (4) ϕ_{s_1} and ϕ_{s_2} are greater than $B(\phi)$. If all basic blocks inside a program have types, the program is well typed. Well-typedness of a program ensures that there is no out of array reference occurred in the execution of the program. Clearly, this framework can be applied for other analysis by defining types and several operators such as $\sqcup$, $\bigsqcup$ and $\sqsubseteq$ appropriately: traditional data flow analysis are subsumed to our framework.

$$\frac{\begin{array}{c} P_1 : \phi_{p_1}, P_2 : \phi_{p_2}, \ldots, P_n : \phi_{p_n} \\ S_1 : \phi_{s_1}, S_2 : \phi_{s_2} \\ \phi = \bigsqcup P_i(\phi_{p_i}) \\ \phi \vdash B(\phi) \\ B(\phi) \sqsubseteq \phi_{s_1},\ \phi_{s_2} \end{array}}{(P_1, P_2, \ldots, P_n)B(S_1, S_2) : \phi} \qquad \frac{\begin{array}{c} P = l_1 : B_1; l_2 : B_2; \cdots; l_n : B_n \\ B_1 : \phi_1, B_2 : \phi_2, \cdots, B_n : \phi_n \end{array}}{P\ \ [well\ \ typed]}$$

Fig. 7. Flow Analytic Type System

Type inference in our framework corresponds to known methods for data flow equations such as MFP algorithms([13]).

4 Induction Variable Analysis in Loops

In this section we sketch to give a type untyped programs with loops and induction variables. We consider here only *natural loops*, which consist of a backedge be, $B_{exit} \rightarrow B_{entry}$, where B_{entry} dominates B_{exit}, and a set of blocks $\mathbf{B}$ such that B_{entry} dominates $\mathbf{B}$ and there is a path from $\mathbf{B}$ to B_{exit} not containing B_{entry}. We denote a natural loop by a triple : $P_l = (B_{entry}, B_{exit}, be)$.

4.1 Translation from Target Language into SSA form

We introduce SSA form as follows. Registers are extended for SSA values. A register r_i is divided to $r_{i0}, r_{i1}, \ldots$ for each definitions of r_i. Also we add

another pseudo instruction $\texttt{loadssa}\,(r_{ip_0}, r_{ip_1}, \ldots)\; r_{id}$ which corresponds to ϕ functions(i.e., $r_{id} = \phi(r_{ip_0}, r_{ip_1}, \ldots)$). The static semantics is :

$$\frac{\phi \vdash r_{ip_j} : \tau_{ip_j} \quad \phi \vdash I}{\phi[r_{id} : \bigsqcup \tau_{ip_j}] \vdash I \;;\; \texttt{loadssa}\,(r_{ip_0}, r_{ip_1}, \ldots)\; r_{id}}$$

As a benefit of SSA form, we can state following typing rules(Fig.8), because in SSA form all assignments of variables are determined uniquely(there is only one assignment for a variable). These rules say that if a register(r_d) is assigned by using another register(r_s) and r_s is typed a type at the top level of the program, the type of r_d can be deduced easily.

$$\frac{\vdash r_s : \tau_s \qquad \texttt{mov}\; r_s, r_d \in P}{\vdash r_d : \tau_s} \;\textbf{(type-mov-init)}$$

$$\frac{\vdash r_s : [e_1, e_2] \qquad \texttt{add}\; r_s, i, r_d \in P}{\vdash r_d : [e_1 + i, e_2 + i]} \;\textbf{(type-add-init)}$$

$$\frac{\vdash r_s : [e_1, e_2] \qquad \texttt{sub}\; r_s, i, r_d \in P}{\vdash r_d : [e_1 - i, e_2 - i]} \;\textbf{(type-sub-init)}$$

Fig. 8. Some Typing Rules for program P

4.2 Single Loop Analysis

In [15,5], if an assignment such as `add r,i,r` is in a program, then `r` is detected as a *basic induction variable*. In Program 2, `r1` and `r2` are basic induction variables(note that `r0` is an assignment from outside of the loop. `r0,r1,r2` are originally the same registers.)

Program 2

```
L: loadssa (r0,r2) r1;
   ...
   add r1,1,r2;
   ...
   blt r2,100,L;
```

In single loops such as Program 2, to deduce ranges of registers, we focus on conditions for backedges. We define such conditions as *Backedge Conditions*.

Definition 4.1 [Backedge Condition] Given a backedge *be* we define its associated condition, *Backedge Condition* as Fig.9 corresponding to the last instruction of the exit block of the loop. The notation $BEC_{be} \vdash \ldots$ express what can be derived from the backedge condition of *be*.

We assume that some registers are typed as *const* to deduce backedge condition. We call such assumptions as *constant assumption*.

Definition 4.2 [Constant Assumptions]

$$\frac{B_{exit} = I; \texttt{blt}\ r, i, l}{BEC_{be} \vdash r < i} \quad \frac{B_{exit} = I; \texttt{blte}\ r, i, l}{BEC_{be} \vdash r < i + 1}$$

$$\frac{B_{exit} = I; \texttt{bgt}\ r, i, l}{BEC_{be} \vdash r > i} \quad \frac{B_{exit} = I; \texttt{bgte}\ r, i, l}{BEC_{be} \vdash r > i - 1}$$

$$\frac{B_{exit} = I; \texttt{blt}\ r_1, r_2, l}{BEC_{be}, [r_2] \vdash r_1 < r_2} \quad \frac{B_{exit} = I; \texttt{blte}\ r_1, r_2, l}{BEC_{be}, [r_2] \vdash r_1 < r_2 + 1}$$

$$\frac{B_{exit} = I; \texttt{bgt}\ r_1, r_2, l}{BEC_{be}, [r_2] \vdash r_1 > r_2} \quad \frac{B_{exit} = I; \texttt{bgte}\ r_1, r_2, l}{BEC_{be}, [r_2] \vdash r_1 > r_2 - 1}$$

$$\frac{B_{exit} = I; \texttt{blt}\ r_1, r_2, l}{BEC_{be}, [r_1] \vdash r_2 > r_1} \quad \frac{B_{exit} = I; \texttt{blte}\ r_1, r_2, l}{BEC_{be}, [r_1] \vdash r_2 > r_1 - 1}$$

$$\frac{B_{exit} = I; \texttt{bgt}\ r_1, r_2, l}{BEC_{be}, [r_1] \vdash r_2 < r_1} \quad \frac{B_{exit} = I; \texttt{bgte}\ r_1, r_2, l}{BEC_{be}, [r_1] \vdash r_2 < r_1 + 1}$$

Fig. 9. Backedge Conditions with $be = B_{exit} \rightarrow B_l$

Constant Assumption $C = \{r_0, r_1, \ldots, r_c\}$ is a finite set of registers to be typed as *const* in assumptions.

Notation 3 $C[r], C \backslash r$

$C[r]$ means appending r to C. $C \backslash r$ means deleting r from C.

Fig. 10 shows typing rules for linearly incrementing induction variables. We abbreviate other kinds of induction variables in this paper for simplicity.

- **type-inv-IV**
  If there is a move instruction"`move` i, r_d" in a loop of backedge be, r_d is typed as induction variables.
- **type-basic-IV-$*$**
  - `loadssa` assignment
    By the property of SSA form, assignments from outside of the loop and from the backedge are merged by `loadssa` assignment at loop header of the entry block.
  - Assignment from backedge
    Registers such as r_2 in `add` r_1, i, r_2 in loops are used in `loadssa` assignment. Thus, assignments make a chain. This chain is called *strongly connected component*[15,5].
  - Backedge Condition
    ¿From backedge conditions and taking r_0 as *const*, we can deduce that r_2 is at most $e + i - 1$ and r_1 is at most $e - 1$ because the value of r_1 is one iteration before the execution exits the loop.
- **type-derived-IV-$*$**

$$\frac{\texttt{mov}\ i, r \in P_l}{BEC_{be} \vdash r : IV[i,i]} \quad \textbf{(type-inv-IV)}$$

$$\frac{\begin{array}{c}\texttt{loadssa}\ (r_0, r_2)\ r_1\ \in\ P_l \\ \texttt{add}\ r_1, i, r_2\ \in P_l \quad BEC_{be}, C\ \vdash r_2 < e\end{array}}{BEC_{be}, C[r_0] \vdash r_2 : IV[r_0, e+i-1]} \quad \textbf{(type-basic-IV-1)}$$

$$\frac{\begin{array}{c}\texttt{loadssa}\ (r_0, r_2)\ r_1\ \in\ P_l \\ \texttt{add}\ r_1, i, r_2\ \in P_l \quad BEC_{be}, C\ \vdash r_2 < e\end{array}}{BEC_{be}, C[r_0] \vdash r_1 : IV[r_0, e-1]} \quad \textbf{(type-basic-IV-2)}$$

$$\frac{\begin{array}{c}\texttt{mov}\ r_s, r_d \in\ P_l \quad r_d \notin C \\ BEC_{be}, C\ \vdash r_s : IV\tau_r\end{array}}{BEC_{be}, C \vdash r_d : IV\tau_r} \quad \textbf{(type-derived-mov-IV)}$$

$$\frac{\begin{array}{c}\texttt{add}\ r_s, i, r_d \in\ P_l \quad r_d \notin C \\ BEC_{be}, C\ \vdash r_s : IV[e_1, e_2]\end{array}}{BEC_{be}, C \vdash r_d : IV[e_1+i, e_2+i]} \quad \textbf{(type-derived-add-IV)}$$

Fig. 10. Some Typing Rules for $P_l = (B_{entry}, B_{exit}, be)$

If r is assigned by `mov` or `add` instructions, and an induction variable is used, r is defined as a *derived induction variable*. One condition is that r is not assumed to be *const* for the derivation of the induction variable.

4.3 Nested Loop Analysis

First, we define partial order for backedges.

Definition 4.3 [$\leq$ for backedges]

Two backedges be and be' satisfy $be \leq be'$ iff incoming point of be' dominates both outgoing point and incoming point of be.

In the above typing rules, registers are typed as $IVrange$ by assuming some registers as *const*. It is necessary to substitute them for their actual ranges. We introduce two substitution rules as follows: a register can be substituted for its range type if it is deduced without any assumptions or typed as $IVrange$ by the outer loops backedge condition.

For example, if r_d is typed as $r_d : IV[r_s, 100 + r_s]$ and r_s is typed as $r_s : IV[10, 20]$ in the outer loops, then r_d can be typed as $r_d : IV[10, 120]$.

$$\frac{\vdash r_1 : \tau_{r_1} \quad r_1 \in C \quad BEC_{be}, C \vdash r_2 : \tau_{r_2}}{BEC_{be}, C \backslash r_1 \vdash r_2 : \tau_{r_2}[r_1/\tau_{r_1}]}$$

(type-subst-1)

$$\frac{be \leq be' \quad BEC_{be'}, C' \vdash r_{out} : IV\tau_{r_{out}} \quad r_{out} \in C \quad BEC_{be}, C \vdash r_{in} : IV\tau_{r_{in}}}{BEC_{be}, \{C \cup C'\} \backslash r_{out} \vdash r_{in} : IV\tau_{r_{in}}[r_{out}/\tau_{r_{out}}]}$$

(type-subst-2)

Fig. 11. Substitution of registers typed as *const* in Nested Loops

4.4 Type Inference of Gaussian Elimination

We show type inference of Gaussian Elimination in our target language. We adopt C-like notations to represent array references instead of combination of `load`'s and `store`'s. The program consists of three nested loops: the outermost loop(loop1 with backedge $be1$), the middle loop(loop2 with backedge $be2$), and the innermost loop(loop3 with backedge $be3$). In Fig.12 r_{11},r_{21}, and r_{31} are used for array references. We focus on deducing the ranges of them.

```
L0: mov 0,r10;
-------------------------------------------------------------|
L1: loadssa (r10,r12) r11;           loop1(be1)              |
    add r11,1,r20;                                           |
-------------------------------------------------------------| |
L2: loadssa (r20,r22) r21;                 loop2(be2)      | |
    m[r21][r11]=a[r21][r11]/a[r11][r11];                   | |
    mov r11,r30;                                           | |
-------------------------------------------------------------| | |
L3: loadssa (r30,r32),r31;                                   | | |
    a[r21][r31]=a[r21][r31]-m[r21][r11]*a[r11][r31];       | | |
    add r31,1,r32;                         loop3(be3)      | | |
    blt r32,r00,L3;                                        | | |
-------------------------------------------------------------| | |
L4: add r21,1,r22;                                           | |
    blt r22,r00,L2;                                        | |
-------------------------------------------------------------| |
L5: add r11,1,r12;                                           |
    sub r00,1,r40;                                           |
    blt r12,r40,L1;                                          |
-------------------------------------------------------------|
```

Fig. 12. Gaussian Elimination Program P

r_{00} corresponds to N in Program 1, the length of arrays $a[], a[][], m[]$, and $m[][]$. r_{10} is assigned by 0 outside of loops. We assume $\vdash r_{00} : const$ and

$\vdash r_{10} : [0, 0]$ at the top level of the program. Backedge conditions are:

$$BEC_{be1}, \{r_{40}\} \vdash r_{12} < r_{40}$$

$$BEC_{be2}, \{r_{00}\} \vdash r_{22} < r_{00}$$
$$BEC_{be3}, \{r_{00}\} \vdash r_{32} < r_{00}$$

By using typing rules for induction variables, we get

$$BEC_{be1} \vdash \ r_{11} : IV[0, r_{00} - 2]$$

$$BEC_{be2} \vdash \ r_{21} : IV[1, r_{00} - 1]$$
$$BEC_{be3} \vdash \ r_{31} : IV[0, r_{00} - 1]$$

For example, the derivation of r_{21} is shown in Fig.13. By these typing it is

$$\frac{\texttt{loadssa}\,(r_{20}, r_{22})\ r_{11} \ \in \ loop2 \qquad \texttt{add}\,r_{21}, 1, r_{22} \ \in loop2 \quad BEC_{be2}, \{r_{00}\} \ \vdash r_{22} < r_{00}}{BEC_{be2}, \{r_{00}, r_{20}\} \vdash r_{21} : IV[r_{20}, r_{00} - 1]}$$
(type-basic-IV-2)

$$\frac{\texttt{add}\,r_{11}, 1, r_{20} \in \ loop1 \quad r_{20} \notin \emptyset \qquad BEC_{be1} \ \vdash r_{11} : IV[0, r_{00} - 2]}{BEC_{be1} \ \vdash r_{20} : IV[1, r_{00} - 1]}$$
(type-derived-add-IV)

$$\frac{be2 \leq be1 \quad BEC_{be1} \vdash r_{20} : IV[1, r_{00} - 1] \quad r_{20} \in \{r_{00}, r_{20}\} \qquad BEC_{be2}, \{r_{00}, r_{20}\} \vdash r_{21} : IV[r_{20}, r_{00} - 1]}{BEC_{be2}, \{r_{00}\} \vdash r_{21} : IV[1, r_{00} - 1]}$$
(type-subst-2)

$$\frac{\vdash r_{00} : [r_{00}, r_{00}] \quad BEC_{be2}, [r_{00}] \vdash r_{21} : IV[1, r_{00} - 1]}{BEC_{be2} \vdash r_{21} : IV[1, r_{00} - 1]}$$
(type-subst-1)

Fig. 13. Derivation of `r21`(i in Program 1)

straightforward to type the whole program.

5 Soundness

Cleary we can prove the following theorem for our framework described in section 3.

Theorem 5.1 (Soundness) *If P[well typed], out of array references never occur in P.*

This theorem also holds when we extend our framework with $IVrange$ types described in section 4 by the following way.

Theorem 5.2 *If $BEC_{be} \vdash r : IV[e_1, e_2]$ where be is a backedge from the loop exit B_{exit} to the loop entry B_{entry}, then $B_{exit}(\phi_{exit})[r] \sqsubseteq [e_1, e_2]$ is satisfied.*

Assume that $be = B_{exit} \rightarrow B_{entry}$. When the program execution goes to B_{entry} from B_{exit} via be and the range of r is $[e'_1, e'_2]$ where $[e'_1, e'_2]$ denotes the range of one iteration before the execution exits the loop(the last time the execution goes to loop entry). Note that $[e'_1, e'_2]$ can be deduced if we change typing rules for basic induction variable to deduce range types of one iteration before the execution exits. To get the least upper bound, by lemma 5.2, we break down the condition for r as:

$$\phi[r] = [e'_1, e'_2] \sqcup \bigsqcup_{j \neq exit} P_j(\phi_{p_j})[r]$$

As this extension, other specialized analysis done in studies of compiler optimization can be applied to our framework.

6 Related Works

Recently there have been many papers of type system for low level languages ([14,7,16]). For example, Necula et al. introduced *proof carrying code*. In their system all codes are sent with proofs of their safety. The host certifies the proof and then determines to execute the codes([12,3]). Xi et al. introduced *practical dependent types* for the type system of ML. As one of the application of dependent type, they tried to detect array bound violations ([18,17]). However, their type systems are rather involved or require strict restrictions on languages. Our framework described in section 3 has much clarity than their paper. Also, there are few papers focusing on loop analysis in low level languages as in section 4.
There have been many papers for array bound checks such as [8,2,9]. [8] is an extension of [6]. [2] uses the constraint system of SSA graph. However, it is not clear which methods of such papers are better. In [9] Midkiff et al. focus on *for loops*. However, they are not considering general loops. As shown in our paper, detecting induction variables is essential.

7 Concluding Remarks

In this paper we have proposed flow analytic type system for array bound checks. In section 3 we have introduced the type system for basic blocks and complete programs. Then in section 4 we have focused on loops where

optimizing array bound checks is crucial. In section 5 we have stated soundness of our type system. Section 6 has listed related works.
The most important future work is memory management analysis, especially stack analysis with `pop` and `push` instructions. Recent researches have been trying to overcome this task by applying logics such as linear logic([1]). We regard this approach is promising, and in addition to the approach we are considering the relation between logics and data flow analysis.

References

[1] Ahmed, A., and D. Walker, *The Logical Approach to Stack Typing*, in: *The ACM SIGPLAN Workshop on Types in Language Design and Implementation*, 2003, to appear.

[2] Bodik, R., R. Gupta and V. Sarkar, *ABCD: eliminating array bounds checks on demand*, in: *SIGPLAN Conference on Programming Language Design and Implementation*, 2000, pp. 321–333.

[3] Colby, C., P. Lee, G. C. Necula, F. Blau, M. Plesko and K. Cline, *A certifying compiler for Java*, ACM SIGPLAN Notices **35** (2000), pp. 95–107.

[4] Cytron, R., J. Ferrante, B. K. Rosen, M. N. Wegman and F. K. Zadeck, *Efficiently computing static single assignment form and the control dependence graph*, ACM Transactions on Programming Languages and Systems **13-4** (1991), pp. 451–490.

[5] Gerlek, M. P., E. Stoltz and M. Wolfe, *Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA form*, ACM Transactions on Programming Languages and Systems **17-1** (1995), pp. 85–122.

[6] Gupta, R., *Optimizing array bound checks using flow analysis*, ACM Letters on Programming Languages and Systems **2-1-4** (1993), pp. 135–150.

[7] Hagiya, M. and A. Tozawa, *On a new method for dataflow analysis of java virtual machine subroutines*, Lecture Notes in Computer Science **1503** (1998), pp. 15–37.

[8] Kolte, P. and M. Wolfe, *Elimination of redundant array subscript checks*, in: *SIGPLAN Conference on Programming Language Design and Implementation*, 1995, pp. 270–278.

[9] Midkiff, S. P., J. E. Moreira and M. Snir, *Optimizing array reference checking in java programs*, IBM Systems Journal **37-3** (1998), pp. 409–453.

[10] Morrisett, G., D. Walker, K. Crary and N. Glew, *From System F to typed assembly language*, ACM Transactions on Programming Languages and Systems **21** (1999), pp. 527–568.

[11] Morrisett, J. G., K. Crary, N. Glew and D. Walker, *Stack-Based Typed Assembly Language*, in: *The ACM SIGPLAN Workshop on Types in Compilation*, 1998, pp. 28–52.

[12] Necula, G. C., *Proof-carrying code*, in: *Conference Record of POPL 97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges*, Paris, 1997, pp. 106–119.

[13] Nielson, F., H. R. Nielson and C. Hankin, "Princeples of Program Analysis," Springer Verlag, Berlin, Heidelberg, 1998.

[14] Stata, R. and M. Abadi, *A type system for Java bytecode subroutines*, in: *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, New York, NY, 1998, pp. 149–160.

[15] Wolfe, M., *Beyond induction variables*, SIGPLAN Notices **27** (1992), pp. 162–174.

[16] Xi, H. and R. Harper, *Dependently typed assembly language*, in: *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, Florence, 2001, pp. 169–180.

[17] Xi, H. and F. Pfenning, *Eliminating array bound checking through dependent types*, in: *SIGPLAN Conference on Programming Language Design and Implementation*, 1998, pp. 249–257.

[18] Xi, H. and F. Pfenning, *Dependent Types in Practical Programming*, in: *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, New York, NY, 1999, pp. 214–227.