A Type System Equivalent to Static Single Assignment *

Yutaka Matsuno † Atsushi Ohori

Research Institute of Electrical Communication
Tohoku University
{matsu, ohori}@riec.tohoku.ac.jp

Abstract

This paper develops a static type system equivalent to static single assignment (SSA) form. In this type system, a type of a variable at some program point represents the control flows from the assignment statements that reach the program point. For this type system, we show that a derivable typing of a program corresponds to the program in SSA form. By this result, any SSA transformation can be interpreted as a type inference process in our type system. By adopting a result on efficient SSA transformation, we develop a type inference algorithm that reconstructs a type annotated code from a given code. These results provide a static alternative to SSA based compiler optimization without performing code transformation. Since this process does not change the code, it does not incur overhead due to insertion of ϕ functions. Another advantage of this type based approach is that it is not constrained to naming mechanism of variables and can therefore be combined with other static properties useful for compilation and code optimization such as liveness information of variables. As an application, we express optimizations as type-directed code transformations.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.4 [*Programming Languages*]: Processors—Optimization; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Language, Theory

Keywords Static Single Assignment form, Type System, Compiler Optimization

1. Introduction

Static Single Assignment (SSA) form [9, 1, 24] is an intermediate representation in which the use-def relation on variables is explicit in the syntax of the code by guaranteeing that each variable is assigned exactly once. This property significantly simplifies various optimization process such as constant propagation, induction variables analysis, and dead code elimination. For example, in SSA form, constant propagation for assignment $x = c$ can be performed simply by substituting c for x without considering the scope of this assignment. SSA form also makes optimization more efficient by simplifying the def-use analysis. Due to these benefits, SSA form is widely accepted as an intermediate representation for code optimization.

`L01:x = 0;`	L01: $\mathtt{x}_{l_{01}}$ = 0;
`L02:y = 1;`	L02: $\mathtt{y}_{l_{02}}$ = 1;
`L03:goto L11;`	L03:goto L10;
	L10: $\mathtt{x}_{l_{10}} = \phi(\mathtt{x}_{l_{01}}, \mathtt{x}_{l_{11}})$;
`L11:x = x + y;`	L11: $\mathtt{x}_{l_{11}} = \mathtt{x}_{l_{10}} + \mathtt{y}_{l_{02}}$;
`L12:if x goto L21;`	L12:if $\mathtt{x}_{l_{11}}$ goto L21;
`L13:goto L11;`	L13:goto L10;
`L21:return x;`	L21:return $\mathtt{x}_{l_{11}}$;

Figure 1. A Simple Example of SSA Form

To take advantage of these benefits, a compiler needs to transform an ordinary intermediate code into SSA form. This is done by encoding the def-use relation in variable names. For a straight-line program, this is simply done by introducing a new name for each assignment of a variable and mechanically renaming all the subsequent uses of the variable up to another assignment point. For a branching program in which a use of a variable corresponds to multiple assignments, yet another new name is introduced for each join point and the joining assignments are recorded by a ϕ function. Figure 1 shows an example code and the corresponding SSA program. The term $\phi(\mathtt{x}_{l_{01}}, \mathtt{x}_{l_{11}})$ denotes $\mathtt{x}_{l_{01}}$ when the execution comes from the preceding block (the statement L03) and denotes $\mathtt{x}_{l_{11}}$ when the execution come from the backedge (the statement L12).

As seen from the example, an assignment of the form $\mathtt{x}_j = \phi(\mathtt{x}_{i_1}, \ldots, \mathtt{x}_{i_n})$ is introduced only to record the property that the subsequent uses of x refer to multiple different definitions made at location i_1 trough i_n, and does not contribute to computation. This pseudo instruction must therefore be eliminated after optimizations are completed. However, this "reverse transformation" is a non trivial task as shown in [5, 25], and some redundant instructions may remain. We regard this an unfortunate consequence of the ad-hoc encoding of static information in the names of variables through the conventional SSA transformation. This encoding of def-use relation also make it difficult to apply some optimizations such as partial redundancy elimination. Kennedy et al. proposed an algorithm to apply partial redundancy elimination in SSA form [12], but the algorithm is much more complicated than the original one [13].

* This research was partially supported by the Japan MEXT (Ministry of Education, Culture, Sports, Science and Technologies) leading project of "Comprehensive Development of Foundation Software for E-Society" under the title "dependable software development technology based on static program analysis."
This is the author's version of the paper to appear in ACM PPDP 2006.

† Part of this work was done while the first author was studying at Department of Frontier Informatics, the University of Tokyo.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PPDP'06 July 10–12, 2006, Venice, Italy.
Copyright © 2006 ACM 1-59593-388-3/06/0007. . . $5.00.

The motivation of the present work is to develop a static type system that can serve as an alternative basis for SSA transformation. As the term indicates, the property of a code that is made explicit by SSA transformation is *static*, and can therefore be regarded as a static type of the code. The SSA transformation can then be represented as a type inference process. Based on this general observation, we define a type system that is equivalent to SSA form. Any SSA transformation can be interpreted as a type inference algorithm in our type system. By adopting a result on efficient SSA transformation, we develop a type inference algorithm that reconstructs the type annotated code from the given code. Since this process does not change the code, it does not incur overhead due to insertion of ϕ functions. The reverse transformation is a trivial process of erasing type information. Another advantage of this type based approach is that it is not constrained to naming mechanism of variables and can therefore be combined with other static properties useful for compilation and code optimization such as liveness information of variables.

The rest of the paper is organized as follows. In Section 2, we explain our key ideas and introduce our type system. Section 3 shows the relationship between our type system and SSA form. Section 4 develops a type inference algorithm. In section 5 we express typical optimizations as type-directed code transformations and show the feasibility of our framework by a prototype implementation. Section 6 states concluding remarks.

2. Type System

We consider an intermediate language with the following restrictions: a basic block ends with an unconditional jump; a conditional branch is followed by an unconditional jump. These restrictions are introduced only to make the presentation simple. Our type system should be applicable to various intermediate languages, as SSA form can be.

Let $x, y, \ldots$ range over a given countably infinite set of variables and let c range over a given set of atomic constants. Let l range over a given set of *labels*. The set of instructions (ranged over by I), basic blocks (ranged over by B), and programs (ranged over by P) are defined by the following syntax.

$$I ::= x = c \mid x = y \mid x = y + z \mid x = x + y$$
$$B ::= l : \texttt{return}\ x \mid l : \texttt{goto}\ l \mid l : \texttt{if}\ x\ \texttt{goto}\ l; l : \texttt{goto}\ l \mid l : I; B$$
$$P ::= \{l : B, \ldots, l : B\}$$

We require that if $l : B \in P$ for a program P then l is the label of the first instruction of B.

We observe that, for each use of a variable, the set of definitions that reach the use of the variable forms a tree whose paths are control flows from the definitions. We consider such a tree as a *type*. For each definition, we introduce a unique atomic type. A definition is generated by an assignment statement or given as an initial value through a live-in variable. The former is represented by a label of the defining instruction. In order to represent the latter, we introduce a countably infinite set of *definition constants* (ranged over by d). Labels and definition constants are atomic types. Types are trees generated from those atomic types. To deals with possibly infinite regular trees, we represent a tree by *type variables* (ranged over by φ^l) and an associated structure mapping $\mathcal{M}$ that assigns each type variable to the set of its children. $\varphi^l \mapsto \{\tau_1, \ldots, \tau_n\} \in \mathcal{M}$ represents the fact that a type variable φ^l is a tree node with the children $\tau_1, \ldots, \tau_n$. We write $dom(\mathcal{M})$ for the domain of the structure mapping $\mathcal{M}$. We only consider *closed* structure mappings, i.e., if a type variable φ^l appears in $\mathcal{M}$ then $\varphi^l \in dom(\mathcal{M})$. The label l in a type variable φ^l indicates the program point where the type variable is first introduced. This is needed to avoid accidental name collisions in the choice of type variables. The set of types is then given by the following syntax.

$$\tau ::= d \mid l \mid \varphi^l$$

We set up a type system so that if a variable x is given a type variable φ^l and $\varphi^l \mapsto \{\tau_1, \ldots, \tau_n\} \in \mathcal{M}$, then the set of reaching definitions is represented by a (possibly infinite regular) tree composed of subtrees τ_1 through τ_n. This is our type-based analogue of ϕ functions in the conventional SSA form: $\varphi^l \mapsto \{\tau_1, \ldots, \tau_n\} \in \mathcal{M}$ corresponds to a ϕ function $x_l = \phi(x_1, \ldots, x_n)$ where φ^l corresponds to x_l and $\tau_1, \ldots, \tau_n$ correspond to $x_1, \ldots, x_n$, respectively.

We define a type system for blocks as a proof system to derive a typing judgement of the form

$$\mathcal{M}, \mathcal{L}, \Gamma \vdash B$$

where Γ is a *type context* which is a function from a finite set of variables to types, and $\mathcal{L}$ is a *label environment* which is a function from a finite set of labels to type contexts. Γ describes the types of variables in B at the entry of B. $\mathcal{L}$ describes the type context of the entry point of each block in the program. We write $\Gamma\{x : \tau\}$ for the function Γ' such that $dom(\Gamma') = dom(\Gamma) \cup \{x\}$, $\Gamma'(x) = \tau$ and $\Gamma'(y) = \Gamma(y)$ for any $y \in (dom(\Gamma) \setminus \{x\})$.

We adopt the idea underlying the logical interpretation of machine code [22, 23] and define the type system as a set of rules of the form

$$\frac{\mathcal{M}, \mathcal{L}, \Gamma_1 \vdash B}{\mathcal{M}, \mathcal{L}, \Gamma_2 \vdash I; B}$$

representing the fact that the static semantics of I is to change the (static) status of the set of live variables from Γ_2 to Γ_1.

At an instruction that branches to some block labeled with l, the type context Γ at this instruction and the Γ' at the target block must satisfy the following conditions.

1. The set of variables used in the target must be live at the source.
2. For each variable, the target type information must represents the fact that the control flows from the source to the target $l : B$.

To precisely specify these conditions, we define the following relations on types and type contexts.

DEFINITION 1.

- $\mathcal{M} \vdash \tau \leq_l \tau'$ *if one of the following conditions holds.*
  - τ *is* τ'.
  - τ' *is* φ^l *and* $\tau \in \mathcal{M}(\varphi^l)$.
- $\mathcal{M} \vdash \Gamma_1 \Subset_l \Gamma_2$ *if*
  - $dom(\Gamma_1) \subseteq dom(\Gamma_2)$, *and*
  - *for each* x *in* $dom(\Gamma_1)$, $\mathcal{M} \vdash \Gamma_2(x) \leq_l \Gamma_1(x)$.

Using these definitions, the set of typing rules for blocks is given in Figure 2. We note that a type variable φ^l may be introduced to satisfy sub conditions for (**Type-If**) and (**Type-Goto**). This corresponds to the fact that ϕ functions are introduced at the entry of a basic block in conventional SSA form. Also note that there is no explicit rule to introduce a definition constant. A definition constant is implicitly given to a variable in a type context. Here we assume the following convention. A definition constant has been given to a variable in a type context if there is no typing constraint associated with that type. It is easily checked that such case occurs only for live-in variables in the entry block of a program.

A type judgment for a program is of the form:

$$\mathcal{M}, \mathcal{L} \vdash \{l_1 : B_1, \ldots, l_n : B_n\},$$

$$\frac{}{\mathcal{M},\mathcal{L},\Gamma\{x:\tau\}\vdash l:\texttt{return}\ x}\ (\textbf{Type-Ret})$$

$$\frac{\mathcal{M},\mathcal{L},\Gamma\{x:l\}\vdash B}{\mathcal{M},\mathcal{L},\Gamma\vdash l:x=c;B}\ (\textbf{Type-Const})$$

$$\frac{\mathcal{M},\mathcal{L},\Gamma\{x:l,y:\tau\}\vdash B}{\mathcal{M},\mathcal{L},\Gamma\{y:\tau\}\vdash l:x=y;B}\ (\textbf{Type-Mov})$$

$$\frac{\mathcal{M},\mathcal{L},\Gamma\{x:l,y:\tau_1,z:\tau_2\}\vdash B}{\mathcal{M},\mathcal{L},\Gamma\{y:\tau_1,z:\tau_2\}\vdash l:x=y+z;B}\ (\textbf{Type-Assign-1})$$

$$\frac{\mathcal{M},\mathcal{L},\Gamma\{x:l,y:\tau_2\}\vdash B}{\mathcal{M},\mathcal{L},\Gamma\{x:\tau_1,y:\tau_2\}\vdash l:x=x+y;B}\ (\textbf{Type-Assign-2})$$

$$\frac{\mathcal{M},\mathcal{L},\Gamma\{x:\tau\}\vdash B}{\mathcal{M},\mathcal{L},\Gamma\{x:\tau\}\vdash l:\texttt{if}\ x\ \texttt{goto}\ l';B}\ (\textbf{Type-If})$$

(if $\mathcal{L}(l')=\Gamma'$ such that $\mathcal{M}\vdash\Gamma'\Subset_{l'}\Gamma\{x:\tau\}$.)

$$\frac{}{\mathcal{M},\mathcal{L},\Gamma\vdash l:\texttt{goto}\ l'}\ (\textbf{Type-Goto})$$

(if $\mathcal{L}(l')=\Gamma'$ such that $\mathcal{M}\vdash\Gamma'\Subset_{l'}\Gamma$.)

Figure 2. The Typing Rules for Blocks

$$\frac{\mathcal{M},\mathcal{L},\Gamma_i\vdash B_i\ \ (1\le i\le n)\quad \mathcal{L}(l_i)=\Gamma_i}{\mathcal{M},\mathcal{L}\vdash\{l_1:B_1,\ldots,l_n:B_n\}}\ (\textbf{Type-Prog})$$

Figure 3. The Typing Rule for a Program

whose typing rule is given in Figure 3. The mechanism of (**Type-Prog**) is the same as the typing rules for mutually recursive function definitions.

Figure 4 shows the typing derivation of the example program in Figure 1, which shows the structure mapping, the label environment, the type derivations for three basic blocks inside the example program, and type constraints associated to branching instructions.

We note that the type system so far defined is a proof system that describes the required relationship between the type contexts Γ_1 and Γ_2 before and after the execution of each instruction with respect to the given structure mapping ($\mathcal{M}$) and label environment ($\mathcal{L}$). In order to use this type system for static analysis of a code, we need to construct $\mathcal{M}$ and $\mathcal{L}$ for the code and Γ for each instruction step. In Section 4, we shall give one such type inference algorithm after we establish the relationship between this type system and SSA transformation, to which we now turn.

3. Relationship to SSA Transformation

Let us first analyze the relationship between typing derivations and SSA programs. Figure 5 shows the sequence of instructions of the example program with the inferred type contexts, and the corresponding SSA program. From this, we see that a new type variable φ^l is introduced at the program point where a ϕ function is inserted in the corresponding SSA program. Although this relationship might appear obvious, to formally establish this relationship requires a certain amount of technical development. This is mainly due to the fact that the properties of SSA form itself is rather involved. In this section, we review the notion of SSA form, introduce a series of technical definitions, and establish the above relationship.

SSA form can be defined declaratively as follows.

> A program is in SSA form if each of its variables is defined exactly once, and each use of a variable is dominated by the variable's definition.

This "definition" only specifies the property that SSA programs must satisfy. A constructive definition that can be used for program optimization is obtained by giving an SSA transformation that inserts ϕ functions at those program points where different definitions for the same variable merge. There can be more than one such definitions, which differ in their strategies of ϕ function insertion.

Our type system is designed so that it precisely represents the class of SSA forms that satisfy the above declarative definition under the interpretation that each occurrence of a variable x is regarded as a variable x^τ decorated with its type τ. This desired property is shown by establishing the following two properties.

1. Any decorated variable x^τ has exactly one definition identified by the type τ.
2. Each use of a decorated variable x^τ is dominated by the unique program point that generates the type τ .

Since each assignment always introduce a new type either identified by the label of the assignment or a new type variable, the first property is obvious. To formally establish the second property, we introduce some notions that are our type-theoretical analogues of domination relations.

The type of x divides the code into regions in such a way that if $\Gamma\vdash B$ and $\Gamma'\vdash B'$ are two typings in the code and $\Gamma(x)=\Gamma'(x)$, then the entry points of B and B' are in the same region. We let $R(\tau)$ be the set of labels of the region identified by τ.

DEFINITION 2. *Assume that $\mathcal{M},\mathcal{L}\vdash\{l_1:B_1,\ldots,l_n:B_n\}$. The type region set for a type τ of x, $R_{\mathcal{M},\mathcal{L}}(\tau,x)$, is the following set:*

$$R_{\mathcal{M},\mathcal{L}}(\tau,x)=\{l\ \mid\ \exists\Gamma\exists B.\mathcal{M},\mathcal{L},\Gamma\vdash l:B\ \textit{and}\ x:\tau\in\Gamma\}$$

We write $R(\tau,x)$ instead of $R_{\mathcal{M},\mathcal{L}}(\tau,x)$ if it is clear from context. $R(\tau,x)$ corresponds to the set of labels of basic blocks which are *dominated* by an assignment(s) to x represented by τ. Using these definitions, we can then show the following property.

LEMMA 1. *Assume that $\mathcal{M},\mathcal{L}\vdash P$.*

- *If $l_i\in R(l,x)$, then $l:B$ dominates $l_i:B_i$.*
- *If $l_j\in R(\varphi^l,x)$, then $l:B$ dominates $l_j:B_j$.*

Proof. We consider the case $l_i\in R(l,x)$. The case for $l_j\in R(\varphi^l,x)$ is similar. By the typing rule, $l_i\in R(l,x)$ indicates that there is a path from $l:B$ to $l_i:B_i$. If $l=l_i$, the required result holds by the definition of the dominance relation. Otherwise, the type derivation of B_i must be of the form:

$$\mathcal{M},\mathcal{L},\Gamma\{x:l\}\vdash l_i:B_i.$$

The structure mapping:

$$\mathcal{M} = \{\varphi^{l_{11}} \mapsto \{l_{01}, l_{11}\}\}$$

The label environment:

$$\mathcal{L} = \{\texttt{L01} : \emptyset,\ \texttt{L11} : \{x : \varphi^{l_{11}}, y : l_{02}\},\ \texttt{L21} : \{x : l_{11}\}\}$$

Basic block typings:

L01:
$$\dfrac{\dfrac{\overline{\mathcal{M}, \mathcal{L}, \{x : l_{01}, y : l_{02}\} \vdash \texttt{L03: goto L11}}\ (1)}{\mathcal{M}, \mathcal{L}, \{x : l_{01}\} \vdash \texttt{L02: y=1;goto L11}}}{\mathcal{M}, \mathcal{L}, \emptyset \vdash \texttt{L01:x=0;y=1;goto L11}}$$

L11:
$$\dfrac{\dfrac{\overline{\mathcal{M}, \mathcal{L}, \{x : l_{11}, y : l_{02}\} \vdash \texttt{L13: goto L11}}\ (2)}{\mathcal{M}, \mathcal{L}, \{x : l_{11}, y : l_{02}\} \vdash \texttt{L12: if x goto L21;goto L11}}\ (3)}{\mathcal{M}, \mathcal{L}, \{x : \varphi^{l_{11}}, y : l_{02}\} \vdash \texttt{L11: x=x+y;if x goto L21;goto L11}}$$

L21: $\overline{\mathcal{M}, \mathcal{L}, \{x : l_{11}\} \vdash \texttt{L21: return x}}$

with the following constraints associated to branching instructions (1) through (3):

$$\begin{aligned}
\mathcal{M} \vdash \{x : \varphi^{l_{11}}, y : l_{02}\} \Subset_{l_{11}} \{x : l_{01}, y : l_{02}\} \quad (1) \\
\mathcal{M} \vdash \{x : \varphi^{l_{11}}, y : l_{02}\} \Subset_{l_{11}} \{x : l_{11}, y : l_{02}\} \quad (2) \\
\mathcal{M} \vdash \{x : l_{11}, y : l_{02}\} \Subset_{l_{11}} \{x : l_{11}, y : l_{02}\} \quad (3)
\end{aligned}$$

Figure 4. Example of Typing Derivation

Typing	Label	Instruction
$\mathcal{M}, \mathcal{L}, \emptyset \vdash$	`L01:`	`x = 0;`
$\mathcal{M}, \mathcal{L}, \{x : l_{01}\} \vdash$	`L02:`	`y = 1;`
$\mathcal{M}, \mathcal{L}, \{x : l_{11}, y : l_{02}\} \vdash$	`L03:`	`goto L11;`
$\mathcal{M}, \mathcal{L}, \{x : \varphi^{l_{11}}, y : l_{02}\} \vdash$	`L11:`	`x = x + y;`
$\mathcal{M}, \mathcal{L}, \{x : l_{11}, y : l_{02}\} \vdash$	`L12:`	`if x goto L21;`
$\mathcal{M}, \mathcal{L}, \{x : l_{11}, y : l_{02}\} \vdash$	`L13:`	`goto L11;`
$\mathcal{M}, \mathcal{L}, \{x : l_{11}\} \vdash$	`L21:`	`return x;`

Label	SSA instruction
`L01:`	$\texttt{x}_{l_{01}}$ `= 0;`
`L02:`	$\texttt{y}_{l_{02}}$ `= 1;`
`L03:`	`goto L10;`
`L10:`	$\texttt{x}_{l_{10}} = \phi(\texttt{x}_{l_{01}}, \texttt{x}_{l_{11}})$;
`L11:`	$\texttt{x}_{l_{11}} = \texttt{x}_{l_{10}} + \texttt{y}_{l_{02}}$;
`L12:`	`if` $\texttt{x}_{l_{11}}$ `goto L21;`
`L13:`	`goto L10;`
`L21:`	`return` $\texttt{x}_{l_{11}}$;

Figure 5. SSA Typing and the corresponding SSA form

For this type derivation, all type derivations for instructions of the form `goto` l_i and `if` y `goto` l_i must be either of the form:

$$\frac{\mathcal{M}, \mathcal{L}, \Gamma_1\{x : l\} \vdash B}{\mathcal{M}, \mathcal{L}, \Gamma_1\{x : l\} \vdash l_1 : \texttt{if}\ y\ \texttt{goto}\ l_i; B}\ (\textbf{Type-If})$$

or

(if $\mathcal{L}(l_i) = \Gamma_i$ such that $\mathcal{M} \vdash \Gamma_i \Subset_{l_i} \Gamma_1\{x : l\}$.)

$$\frac{}{\mathcal{M}, \mathcal{L}, \Gamma_2\{x : l\} \vdash l_2 : \texttt{goto}\ l_i}\ (\textbf{Type-Goto}),$$

(if $\mathcal{L}(l_i) = \Gamma_i$ such that $\mathcal{M} \vdash \Gamma_i \Subset_{l_i} \Gamma_2\{x : l\}$.)

where $\Gamma_i = \Gamma\{x : l\}$, because otherwise the sub conditions $\mathcal{M} \vdash \Gamma_i \Subset_{l_i} \Gamma_1\{x : l\}$ and $\mathcal{M} \vdash \Gamma_i \Subset_{l_i} \Gamma_2\{x : l\}$ can not be satisfied. Therefore, all predecessors are in $R(l, x)$. This implies that all paths to predecessors of $l_i : B_i$ pass $l : B$. Hence, $l : B$ dominates $l_i : B_i$. □

To conclude our proof of the property 2, suppose that there is a use of a variable x and x is typed with τ at some label l. τ is either l' or $\varphi^{l'}$ for some label l'. Then by the definition of R, $l \in R(\tau, x)$ and therefore l' dominates the use site at l, as desired.

We have established that our type system is equivalent to the declarative notion of SSA form. As we have noted earlier, however, the declarative definition does not yield any constructive algorithm. As such, the above correspondence alone is of limited interests in practice. The real value of SSA form is the existence of an efficient algorithm that transforms any program into one that satisfies the SSA property. In order for our type system to serve as a static alternative to SSA form, it is essential to establish that our type system can also represent a practical SSA transformation algorithm. Here we base our development on the standard SSA transformation algorithm presented in [9]. We believe that our type system is expressive enough to represent most of the other algorithms (we shall comment on this later in Section 7.)

Our goal now is to set up a proper restriction on our type system so that a derivable typing of a program corresponds to the result of the SSA transformation of the program by the algorithm of Cytron et al. The first simple restriction we impose is to require the stronger property of $dom(\Gamma') = dom(\Gamma)$ for item 1 in the Definition 1. This restriction corresponds to the ϕ placement of Cytron et al's algorithm, in which dead ϕ assignments may be inserted.

In addition to this, our type system only requires that for each branching instruction, the target type of a variable must be greater with the source type, respect to the relation $\leq_l$. As a result, a typing derivation may contain unrelated label constants in the associated structure mapping. For example, the typing derivation obtained from the one shown in Figure 4 by inserting some unrelated label constant l to the structure mapping of the form $\varphi^{l_{11}} \mapsto \{l_{01}, l_{11}, l\}$ remains to be a valid typing. Also, a cycle in a structure

mapping may contain redundant types. For example, in Figure 6, $\mathcal{M}, \mathcal{L}, \Gamma\{x : \varphi^l\} \vdash l : B$ where $\varphi^l \mapsto \{\varphi^l, l_1\} \in \mathcal{M}$ is derivable. However, the derivation should be of the form $\mathcal{M}, \mathcal{L}, \Gamma\{x : l_1\} \vdash l : B$ (this is the case when one of the dominance frontiers of a basic block is itself.)

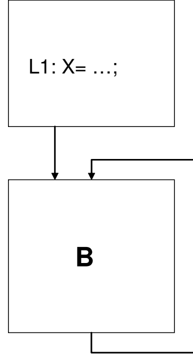

Figure 6. A Typical Example

To eliminate the flexibility exhibited in the above analyzes and to constrain the type system in such a way that it only derives those typings that correspond to SSA programs, we introduce a *minimality condition* to our type system.

Since insertion of ϕ functions is done for each variable separately, in the subsequent development we fix one variable x and only consider typing for x. By this convention, $R(\tau, x)$ is simply written as $R(\tau)$.

For a type substitution $S = [\tau_1/\varphi_1, \ldots, \tau_n/\varphi_n]$, we write $dom(S)$ for $\{\varphi_1, \ldots, \varphi_n\}$ and $range(S)$ for $\{\tau_1, \ldots, \tau_n\}$. A substitution S is applied to a structure as $S(\{\tau_1, \ldots, \tau_n\}) = \{S(\tau_1), \ldots, S(\tau_n)\}$.

DEFINITION 3. *$\mathcal{M}_1, \mathcal{L}_1 \vdash P$ is smaller than $\mathcal{M}_2, \mathcal{L}_2 \vdash P$ if there is an injection map $i : dom(\mathcal{M}_1) \to dom(\mathcal{M}_2)$ and a substitution S such that $dom(S) = dom(\mathcal{M}_2)$, $range(S) \subseteq dom(\mathcal{M}_1) \cup \{l_1, l_2, \ldots\} \cup \{d_1, d_2, \ldots\}$, and $\mathcal{M}_1(\varphi) \subseteq S(\mathcal{M}_2(i(\varphi)))$ for any $\varphi \in \mathcal{M}_1$.*

We set up several definitions for SSA form. Let P_{SSA} be the SSA-transformed intermediate code of P by the algorithm of [9]. The syntax is given as follows. The subscript label of a variable indicates the places where the value is assigned.

$$\begin{aligned} I &::= x_l = c \mid x_l = y_l \mid x_l = y_l + z_l \mid x_l = x_l + y_l \\ &\quad \mid x_l = \phi(x_l, \ldots, x_l) \\ B^{\mathsf{SSA}} &::= l : \mathtt{return}\ x_l \mid l : \mathtt{goto}\ l \\ &\quad \mid l : \mathtt{if}\ x_l\ \mathtt{goto}\ l; l : \mathtt{goto}\ l \mid l : I; B^{\mathsf{SSA}} \\ P_{\mathsf{SSA}} &::= \{l : B^{\mathsf{SSA}}, \ldots, l : B^{\mathsf{SSA}}\} \end{aligned}$$

For a code,

$$P = \{l_1 : B_1, l_2 : B_2, \ldots l_n : B_n\},$$

the corresponding SSA program is written as follows:

$$P_{\mathsf{SSA}} = \{l_1 : B_1^{\mathsf{SSA}}, l_2 : B_2^{\mathsf{SSA}}, \ldots, l_n : B_n^{\mathsf{SSA}}\}.$$

The following theorem establishes that any minimal typing corresponds to an SSA program.

THEOREM 1. *Assume that there is a minimal typing $\mathcal{M}, \mathcal{L} \vdash P$. If there is a block typing of the form $\mathcal{M}, \mathcal{L}, \Gamma\{x : \varphi^l\} \vdash l : B$ in $\mathcal{M}, \mathcal{L} \vdash P$ such that $\varphi^l \mapsto \{\tau_1, \ldots, \tau_n\} \in \mathcal{M}$, then there is a ϕ function $x = \phi(x_1, \ldots, x_n)$ in B^{SSA} of the SSA translated program P_{SSA}. Moreover, for each corresponding τ_i and x_i ($1 \leq i \leq n$), if $\tau_i = l_i$, then there is the assignment to x_l in P_{SSA} not by a ϕ assignment, or if $\tau_i = \varphi_i^{l_i}$, then there is the assignment to x_{l_i} by the ϕ function in P_{SSA}.*

Proof. We use the merge relation introduced in [4], whose detailed definition is given in Appendix A.

Assume that $\mathcal{M}, \mathcal{L}, \Gamma\{x : \varphi^l\} \vdash l : B$ and $\varphi^l \mapsto \{\tau_1, \ldots, \tau_n\} \in \mathcal{M}$. We distinguish two cases.

- The case where a definition constant d is in $\{\tau_1, \ldots, \tau_n\}$. It is enough to consider the simplest case: φ^l is mapped to $\{d_1, l_1\}$ for some d_1 and l_1. By the minimality requirement, there must be some derivation of the form:

  $$\mathcal{M}, \mathcal{L}, \Gamma_1\{x : \tau_1'\} \vdash l_1' : \mathtt{goto}\ l$$
  or
  $$\mathcal{M}, \mathcal{L}, \Gamma_2\{x : \tau_2'\} \vdash l_2' : \mathtt{if}\ y\ \mathtt{goto}\ l; B_2,$$

  where τ_1' and τ_2' are either d_1 or l_1.

  The proof uses the following.

  LEMMA 2. $l : B \in J(\{\textbf{\textit{Entry}}, l_1 : B_1\})$.

  Proof. By Lemma 1, all the predecessor blocks are dominated by either $l_1 : B_1$ or **Entry** node. Also $R(l_1)$ and $R(d_1)$ are disjoint sets, thus predecessors in $R(l_1)$ and $R(d_1)$ only intersect at $l : B$. Therefore $l : B \in J(\{\textbf{Entry}, l_1 : B_1\})$. □

  From this lemma, $l : B \in Merge(l_1 : B_1)$ and there is an M-path $l_1 : B_1 \xrightarrow{+} l : B$. By Theorem 4, $l : B \in DF^+$, hence there is a ϕ function $x = \phi(x_0, x_1)$ in $l : B^{\mathsf{SSA}}$, where x_0 is the initial value. Clearly the assignment to x_{l_1} exists in $l_1 : B_1^{\mathsf{SSA}}$.

- The case where $\{\tau_1, \ldots, \tau_n\}$ does not contain d. Because of the minimality requirement, to derive $\mathcal{M}, \mathcal{L}, \Gamma\{x : \varphi^l\} \vdash l : B$ (where $\varphi^l \mapsto \{\tau_1, \ldots, \tau_n\} \in \mathcal{M}$), there must exist some derivation of the form:

  $$\mathcal{M}, \mathcal{L}, \Gamma_{p_1}\{x : \tau_{p_1}\} \vdash l_{p_1} : \mathtt{goto}\ l$$
  or
  $$\mathcal{M}, \mathcal{L}, \Gamma_{p_2}\{x : \tau_{p_2}\} \vdash l_{p_2} : \mathtt{if}\ y\ \mathtt{goto}\ l; B$$

  where τ_{p_1} and τ_{p_2} are included in $\{\tau_1, \ldots, \tau_n\}$.

  Let $\{l_1, \ldots, l_n\}$ be the set of labels of basic blocks where $\tau_1, \ldots, \tau_n$ are introduced. We argue that there is at least one M path from one of $l_1 : B_1, \ldots, l_n : B_n$ to $l : B$. Otherwise the immediate dominator of $l : B$ appears in all paths from $l_1 : B_1, \ldots, l_n : B_n$ to $l : B$. Let $l' : B'$ be the immediate dominator. $l' : B'$ dominates $l : B$ thus all paths reaching to $l : B$ pass $l' : B'$. Because $R(\tau_1), \ldots, R(\tau_n)$ are disjoint sets, all blocks from $l' : B'$ to $l : B$ are not in $R(\tau_1), \ldots, R(\tau_n)$. Therefore, the predecessors of $l : B$ are not in $R(\tau_1), \ldots, R(\tau_n)$, which is a contradiction. By Theorem 4, $l : B \in DF^+$.

  If some of $\{\tau_1, \ldots, \tau_n\}$ are of the form $\varphi_i^{l_i}$, by the same argument, there exists the ϕ function in $l_i : B_i^{\mathsf{SSA}}$ in P_{SSA} for such τ_i. Therefore we can place a ϕ function of the form $x = \phi(x_1, \ldots, x_n)$ in $l : B_l^{\mathsf{SSA}}$ which satisfies the requirements with $\tau_1, \ldots, \tau_n$. □

We state the direction from SSA form to our type system.

THEOREM 2. *For an SSA program P_{SSA} transformed by Cytron et al's algorithm, there exists a minimal typing for the original program P: $\mathcal{M}, \mathcal{L} \vdash P$ such that if $x_l = \phi(x_{l_1}, \ldots, x_{l_n})$ is in B_l^{SSA}, then $\mathcal{M}, \mathcal{L}, \Gamma\{x : \varphi^l\} \vdash l : B$ is derivable for some φ^l and Γ in the minimal typing. Moreover, for each $l_i : B_i$ $(1 \leq i \leq n)$, $\mathcal{M}, \mathcal{L}, \Gamma_i\{x : \tau_i\} \vdash l_i : B_i$ for some Γ_i where $\varphi^l \mapsto \{\tau_1, \ldots, \tau_n\} \in \mathcal{M}$ and τ_i is of the form φ^{l_i} if the assignment to x_i is by the ϕ function (otherwise τ_i is either of the form d_i or l_i.)*

A proof sketch is as follows.

From a given SSA program, we construct a typing derivation in the following two steps.

1. Reconstructing label environment. For each ϕ assignment of the form $x = \phi(x_1, \ldots, x_n)$ in a basic block $l : B^{\mathsf{SSA}}$, we put a type assignment of the form $x : \varphi^l$ (φ is a fresh type variable) in the type context of $l : B$. For each type variable φ^l, from the next instruction, we put $x : \varphi^l$ in the type contexts successively until there is an assignment instructions of the form $l' : x = \ldots$ (we put $x : \varphi^l$ in the type context for $l' : B'$), or in a type context, x is already given a type variable. For each assignment of the forms $l : x = \ldots$, from the next instruction, we put type assignments of the form $x : l$ in the type contexts successively until there is an assignment instructions, or x is already given a type variable. After these steps, if there is a type context in which a variable is not given a type yet, we give a definition constant to the variable.
2. Reconstructing structure mapping. For each φ^l introduced in item 1, we reconstruct its tree structure as follows. If φ^l corresponds to the ϕ assignment of the form $x = \phi(x_1, \ldots, x_m)$, we add $\varphi^l \mapsto \{\tau_1, \ldots, \tau_m\}$ to the structure mapping where $\{\tau_1, \ldots, \tau_m\}$ corresponds to $\{x_1, \ldots, x_m\}$.

We check that this reconstruction follows the typing rules. For the inside of a basic block, it is easy to see that the successive placement of types follows (**Type-Const**), (**Type-Mov**), **(Type-Assign-1)**, and (**Type-Assign-2**). At the entry of a basic block $l : B$, for each variable, the type derivations must satisfy the condition of Definition 1: $\mathcal{M} \vdash \tau \leq_l \tau'$ if one of the following conditions holds.

1. τ is τ'.
2. τ' is φ^l and $\tau \in \mathcal{M}(\varphi^l)$.

If x is given a label type l' (or a type variable $\varphi_1^{l'}$) in the type context of $l : B$, this placement is only possible by successively placing a type from where it is first placed. Therefore, types for x in the type contexts of all the predecessor instructions are l' ($\varphi_1^{l'}$). Thus condition is satisfied by the item 1. If x is given a type variable φ^l, clearly the condition is satisfied by the item 2.

Next, we check that this typing derivations satisfies the minimality condition according to Definition 3.

Let $\mathcal{M}, \mathcal{L} \vdash P$ is the reconstructed type derivation. Assume that there is another type derivation $\mathcal{M}', \mathcal{L}' \vdash P$ where $\mathcal{M}', \mathcal{L}' \vdash P$ is smaller than $\mathcal{M}, \mathcal{L} \vdash P$ i.e., there is an injection map $i : dom(\mathcal{M}') \to dom(\mathcal{M})$ and a substitution S such that $dom(S) = dom(\mathcal{M})$, $range(S) \subseteq dom(\mathcal{M}') \cup \{l_1, l_2, \ldots\} \cup \{d_1, d_2, \ldots\}$, and $\mathcal{M}'(\varphi') \subseteq S(\mathcal{M}(i(\varphi')))$ for any $\varphi' \in \mathcal{M}'$.

Note that the numbers of type variables in $\mathcal{M}$ and $\mathcal{M}'$ are equal. If $\mathcal{M}$ has more type variables than $\mathcal{M}'$ has, there is a basic block for which a type variable is redundant in $\mathcal{M}, \mathcal{L} \vdash P$. However, it can not happen. The type variable is introduced where the ϕ assignment is placed in P_{SSA}, i.e., dominance frontier. Therefore different definitions merge at the basic block, thus the type variable is not redundant. This implies that the injection map i is a bijection. Then, the substitution S needs to substitute $i^{-1}(\varphi)$ for each $\varphi \in \mathcal{M}$, thus S is also a bijection between $dom(\mathcal{M})$ and $dom(\mathcal{M}')$. We check the inclusion constraint $\mathcal{M}'(\varphi') \subseteq S(\mathcal{M}(i(\varphi')))$. Because the structures of $\mathcal{M}$ exactly corresponds to arguments of ϕ functions in P_{SSA}, clearly each type variable is mapped to the least structure. Therefore the inclusion constraint is satisfied by the equality. Hence, $\mathcal{M}$ is a minimal structure mapping.

4. Type Inference Algorithm

In the previous section we have proved that our type system is equivalent to SSA form. Therefore, the type inference algorithm exactly corresponds to SSA transformation algorithms. In this section we develop an inference algorithm based on conventional SSA transformation algorithms.

The first efficient algorithm for computing where to place ϕ functions is introduced by Cytron et al [9]. Several efficient algorithms have been proposed since the publication of [9]. Bilardi and Pingari [4] introduced the merge relation (See appendix A) and proposed efficient algorithms. In this section we introduce a type inference algorithm based on the most recent algorithm [10], which uses the merge relation. For details refer to the paper [10].

In the following development, we regard each instruction as a node in the control flow graph. We denote a program P as $P = \{l_1 : I_1, \ldots, l_n : I_n\}$ where $l_1 : I_1$ is the entry point. This is a typical representation for SSA transformation algorithms. In this setting, the type inference algorithm infers the structure mapping and label environment for a given program.

4.1 Computing merge sets

The algorithm of [10] first computes the merge sets for all nodes. We show the algorithm in Figure 7. In the algorithm the ***DJ*** graph is used. The DJ graph of a program is the graph that consists of the dominator tree with edges (called J edges) $s \to d$ where s is not the immediate dominator of t. Taking the DJ graph of a program P as its input, the algorithm returns the merge set. It is recursively executed until the flag *RequireAnotherPass* is to be *False*. In the algorithm, $level(node)$ denotes the level of the node at the dominator tree (the root node has level 0).

4.2 Placing type variables

In the previous subsection the Merge sets for each node in the control flow graph are computed. If a node is in the merge set of a node, then a type variable (i.e. a ϕ function) needs to be inserted. The algorithm to place type variables is shown in Fig.8. In the algorithm the following special type contexts are exploited. One is $\Gamma_d = \{x : d, y : d, \ldots\}$ for the set of variables $\{x, y, \ldots\}$ of P. This type context is for the entry block. At the entry block, all variables have their initial values and they are represented by definition constants d. Another is $\Gamma_\perp = \{x : \perp, y : \perp, \ldots\}$ for $\{x, y, \ldots\}$. $x : \perp$ indicates that the type of x is still not inferred in the type context. In this step only labeled type variables are inferred. The types for variables which are given $\perp$ are inferred in the next step.

4.3 Inferring structure mapping and label environment

After type variables (i.e. ϕ functions) are inserted, we construct the structure mapping and label environment. We need to determine which definitions come into the program point where type variables are inserted. This is done by conventional reaching definition analysis. We adopt a classical method to solve reaching definition equations known as MFP (for Maximal Fixed Point) [21] solution for our type inference algorithm. The algorithm is shown in Figure 9. In the algorithm, the following transfer function $t_{l,P}^x(\tau)$ is exploited.

```
Merge(DJ) =
{
RequireAnotherPass = False
while (in Breadth First Order) do
 Let n =NextNode in the BFO list
 for (all incoming edges to n) do
  Let e =Incoming edge
  if(e is a J-edge ∧ e not visited ) then
   Visit(e)
  Let snode = Source Node of e
  Let tnode = Target Node of e
  Let tmp = snode
  Let lnode=NULL
  while (level(tmp) ≤ level(tnode)) do
        Merge(tmp) = Merge(tmp)
      ∪Merge(tnode) ∪ {tnode}
    lnode = tmp
    tmp = parent(tmp) // dominator tree parent
  end while
  for (all incoming edges to lnode) do
   // lnode ancestor of snode
   Let e′ =Incoming edge
   if (e′ is a J-edge ∧ e′ visited) then
    Let snode′ = Source Node of e′
    if (Merge(snode′) ⊉ Merge(lnode)) then
     RequireAnotherPass=True
    end if
   end if
  end for
  end if
 end for
end while
return (RequireAnotherPass, Mergeset)
}
```

Figure 7. An Algorithm to Compute Merge Sets

```
PhiPlacement(Merge) =
{
L = {l1 : Γd, l2 : Γ⊥, ..., ln : Γ⊥}
for (every assignment node n of the form x = v in P) do
 for (every node k in Merge(n)) do
 L = {l1 : Γ1, ..., lk : Γk{x : φ^lk}, ..., ln : Γn} (φ fresh)
 end for
end for
return L
}
```

Figure 8. An Algorithm to Place Type Variables

- If $l : x = c, l : x = y, l : x = y + z$, or $l : x = x + z$ is in P. Then $t^x_{l,P}(\tau) = l$.
- Otherwise $t^x_{l,P}(\tau) = \tau$

The correctness of the type inference algorithm corresponds to the correctness of the SSA transformation algorithm [10] and that of MFP algorithm [21].

5. Compiler Optimizations as Type-directed Code Transformations

In this section we test the feasibility of our type system. First we show that our type system is expressive enough to represent typ-

```
Infer(P) =
{
Merge = Merge(DJ_P)
L = PhiPlacement(Merge)
M = {φ1 ↦ {}, ..., φm ↦ {}}
where φ1, ..., φm appear in L
(V, E) = CFG_P
W=nil
for all (l, l′) in E do W = cons((l, l′), W)
while work-list W is not empty do
 Remove node (lp, l) from W;
 for each x ∈ dom(L(l)) do
  case 1 L(l)(x) = d or L(l)(x) = l : {}
  case 2 L(l)(x) = φ1^l′ : {}
  case 3 L(l)(x) = ⊥: {L(l)(x) = t^x_{lp,P}(L(lp)(x))}
  case 4 L(l)(x) = φ^l :
   case 4.a t^x_{lp,P}(L(lp)(x)) = ⊥ : {}
   case 4.b t^x_{lp,P}(L(lp)(x)) = φ^l : {}
   case 4.c Otherwise: { If φ^l ↦ {, ..., } ∈ M,
      update M as φ^l ↦ {, ..., t^x_{lp,P}(L(lp)(x))} ∈ M}
 end for
 if case 3 is taken:
  for all l′ with (l, l′) in E do W = cons(W, (l, l′))
end while
return (M, L)
}
```

Figure 9. An Algorithm to Reconstruct Structure Mapping and Label Environment

ical optimizations. Second, we implement a prototype of the type inference algorithm developed in Section 4. Then we implement optimizations in the prototype, and check the practical benefits of our framework.

It is known that representing compiler optimizations in a formal way is a hard task. There have been many previous works on the task. For example, Lacey et al [14] used temporal logic to specify the conditions for applying a compiler optimization. For example, dead code elimination is defined as follows.

$$x := e \Longrightarrow skip \text{ if } AX\ \neg E(\,true\,U\,use(x)).$$

This rule intuitively says that an assignment $x := e$ in the control graph can be rewritten to a skip instruction if in all paths from $x := e$, there is no use of x. Also Benton [3] exploited Hoare logic and defined a type system which ensures partial equivalence relation between the source and optimized codes. However, currently SSA form has not been widely used in the literature, although SSA form makes many program analyses simple. It seems this is due to the fact that data dependence analyses such as SSA form are difficult to be expressed in a formal framework such as temporal logic or Hoare logic. In SSA form, there has not been well accepted semantics of ϕ functions, and dynamic semantics of SSA form has not been well defined in the literature.

In our type system, we lift SSA information to types. This avoids such subtle issues discussed above. In this section we express dead code elimination and common subexpression as type-directed code transformations in the following form:

$$P \Longrightarrow P' \ \ (\text{if } \Psi),$$

where P and P' are the codes before and after applying the optimization, respectively. Ψ denotes the condition to apply the optimization, where Ψ is represented by type information.

5.1 Dead Code Elimination

To apply dead code elimination, liveness analysis (which variables are used in conditional jumps and the return instructions) is required. In our framework, liveness analysis is easily incorporated with the typing rules. First, we introduce *the live set* Π as the set of labels of assignment instructions to live variables. We incorporate liveness analysis into type derivation in the following way.

$$(\mathcal{M}, \mathcal{L}, \Gamma \vdash B) : \Pi,$$

where Π denotes the set of labels of live variables at the program point immediately after B. Figure 10 shows typing rules incorporated with liveness analysis. In Figure 10, a function $\mathsf{flat}_{\mathcal{M}}$ for a structure mapping $\mathcal{M}$ is used. $\mathsf{flat}_{\mathcal{M}}$ flattens the tree structure of type variables. In SSA form, algorithms for dead code elimination become efficient because there is only one definition for a variable. In our framework, the same effect is obtained by $\mathsf{flat}_{\mathcal{M}}$ which derives all possible definitions without further flow analysis. $\mathsf{flat}_{\mathcal{M}}(\tau)$ is defined as:

- $\mathsf{flat}_{\mathcal{M}}(d) = \{d\}$ and $\mathsf{flat}_{\mathcal{M}}(l) = \{l\}$.
- $\mathsf{flat}_{\mathcal{M}}(\varphi) = \mathsf{flat}_{\mathcal{M}}(\tau_1) \cup \ldots \cup \mathsf{flat}_{\mathcal{M}}(\tau_n)$, where $\varphi \mapsto \{\tau_1, \ldots, \tau_n\} \in \mathcal{M}$.

Based on the liveness analysis, we express dead code elimination as a typed code transformation rule for a given program P. We denote whole type derivations with liveness analysis for a program P as Δ_P^{Π}. If $(\mathcal{M}, \mathcal{L}, \Gamma \vdash B) : \Pi$ is in Δ_P^{Π}, we denote $\Delta_P^{\Pi} \vdash (\mathcal{M}, \mathcal{L}, \Gamma \vdash B) : \Pi$. The rule is as follows.

$$P[l : x = v; B] \Longrightarrow P[l : \text{skip}; B]$$
$$\text{if } \Delta_P^{\Pi} \vdash (\mathcal{M}, \mathcal{L}, \Gamma\{x : l\} \vdash B) : \Pi \ \wedge \ l \notin \Pi.$$

This rule intuitively says that if there exists a type derivation with liveness analysis as

$$\frac{(\mathcal{M}, \mathcal{L}, \Gamma\{x : l\} \vdash B) : \Pi}{(\mathcal{M}, \mathcal{L}, \Gamma \vdash l : x = v; B) : \Pi'}$$

in Δ_P and l is not in Π (i.e., the value of x assigned in the instruction labeled l is dead), then we can replace the instruction with a skip instruction.

5.2 Common Subexpression Elimination

Common subexpression elimination replaces an expression with more simple expression. For example, in the following program:

```
w=...
...
x=y+z   (*) -> x=w
```

, if the value of `w` and `y+z` are equal immediately before `(*)`, then we can replace the instruction `x=y+z` with `x=w`. As in this example, common subexpression elimination is based on value equivalence among variables. Two variables are said to be equivalent at a program point p if those variables contain the same values whenever control reaches p during any possible execution of the program. For the value equivalence problem, SSA form is very applicable [1]. Alpern et al [1] exploited SSA form and introduced *value graph* (also known as SSA graph [8]) for each variable. In [1], it is proved that if the value graphs of two variables are *congruent*, then they contain the same value in a program point. In this paper we introduce *the value type tree* for each variable which corresponds to value graph. To introduce value type trees, we extend structure mapping as type variables are mapped to the set of labelled types of the form $\varphi^l \mapsto \{l_1 : \tau_1, \ldots, l_n : \tau_n\}$ where $\{l_1, \ldots, l_n\}$ are the labels of predecessor blocks of $l : B$. $l_i : \tau_i \in \mathcal{M}(\varphi^l)$ indicates that the value represented by τ_i is derived from the predecessor block $l_i : B_i$. This labelling is easily obtained by extending the relation in Definition 1 as:

$\mathcal{M} \vdash \tau \leq_{(l_s, l_t)} \tau'$ if one of the following conditions holds.

- τ is τ'.
- τ' is φ^{l_t} and $l_s : \tau \in \mathcal{M}(\varphi^{l_t})$,

and then the sub conditions for **(Type-If)** and **(Type-Goto)** are replaced with $\mathcal{M} \vdash \Gamma' \Subset_{(l,l')} \Gamma\{x : \tau\}$ and $\mathcal{M} \vdash \Gamma' \Subset_{(l,l')} \Gamma$, respectively. This labeling is needed because the information where a value comes from is essential for value equivalence problem.

A value type tree $VT_{\mathcal{M}}^P(\tau)$ in a type derivation $\mathcal{M}, \mathcal{L} \vdash P$ is defined as follows.

- $\tau = d$. Then $VT_{\mathcal{M}}^P(\tau) = (d)$.
- $\tau = l$ and $\Delta_P \vdash \mathcal{M}, \mathcal{L}, \Gamma \vdash l : x = c; B$. Then $VT_{\mathcal{M}}^P(\tau) = (c)$.
- $\tau = l$ and $\Delta_P \vdash \mathcal{M}, \mathcal{L}, \Gamma\{y : \tau_1\} \vdash l : x = y; B$. Then $VT_{\mathcal{M}}^P(\tau) = VT_{\mathcal{M}}^P(\tau_1)$.
- $\tau = l$ and $\Delta_P \vdash \mathcal{M}, \mathcal{L}, \Gamma\{y : \tau_1, z : \tau_2\} \vdash l : x = y + z; B$ (same as for the case $x = x + y$).

$$VT_{\mathcal{M}}^P(\tau) = \begin{array}{c} (+) \\ \swarrow \quad \searrow \\ VT_{\mathcal{M}}^P(\tau_1) \quad VT_{\mathcal{M}}^P(\tau_2). \end{array}$$

- $\tau = \varphi^l$ where $\varphi^l \mapsto \{l_1 : \tau_1, \ldots, l_n : \tau_n\} \in \mathcal{M}$.

$$VT_{\mathcal{M}}^P(\tau) = \begin{array}{c} (l) \\ \swarrow \ \cdots \ \searrow \\ l_1 : VT_{\mathcal{M}}^P(\tau_1) \ \ldots \ l_n : VT_{\mathcal{M}}^P(\tau_n). \end{array}$$

We denote $\Delta_P \vdash VT_1 \cong VT_2$ if two value type trees VT_1 and VT_2 have the same tree structure in the type derivation Δ_P. This corresponds to the notion of congruence defined in [1].

We express common subexpression elimination as a typed code transformation as follows.

$$P[l : x = y + z; B] \Longrightarrow P[l : x = w; B]$$
$$\text{if } \Delta_P \vdash (\mathcal{M}, \mathcal{L}, \Gamma\{y : \tau_1, z : \tau_2\} \vdash l : x = y + z; B)$$
$$\wedge \exists w : \tau_3 \in \Gamma. \Delta_P \vdash VT_{\mathcal{M}}^P(\tau_3) \cong (+\, VT_{\mathcal{M}}^P(\tau_1)\, VT_{\mathcal{M}}^P(\tau_2)).$$

The problem of determining whether two value type trees have the same tree structure can be solved by well known partitioning algorithms [1].

5.3 Prototype implementation

We have implemented a prototype of the inference algorithm developed in Section 4 using SML/NJ and tested the feasibility of our framework. A function `INFER` reconstructs the structure mapping and type contexts for all instructions in an input program. Figure 11 shows the simple program of Figure 1 and the output of `INFER`. In the prototype implementation, a program is defined as a list of pairs of a label and an instruction. A label is defined by an ML datatype constructor `L`. A variable is defined by a constructor `V`. Types are implemented as an ML type `SSAtype` which consists of definition constants (`D`), label types (`Lt`), and type variables (`Phi`). Type variables are identified by the annotated label and the assigned variable.

By exploiting reconstructed type information, we have implemented prototype compiler optimizations introduced in the previ-

$$(\mathcal{M},\mathcal{L},\Gamma\{x:\tau\}\vdash \texttt{return}\ x):\mathsf{flat}_{\mathcal{M}}(\tau) \qquad \frac{(\mathcal{M},\mathcal{L},\Gamma\{x:l\}\vdash B):\Pi}{(\mathcal{M},\mathcal{L},\Gamma\vdash l:x=c;B):\Pi}$$

$$\frac{(\mathcal{M},\mathcal{L},\Gamma\{x:l,y:\tau\}\vdash B):\Pi}{(\mathcal{M},\mathcal{L},\Gamma\{y:\tau\}\vdash l:x=y;B):\Pi}\ (l\notin\Pi) \qquad \frac{(\mathcal{M},\mathcal{L},\Gamma\{x:l,y:\tau\}\vdash B):\Pi}{(\mathcal{M},\mathcal{L},\Gamma\{y:\tau\}\vdash l:x=y;B):\Pi\cup\mathsf{flat}_{\mathcal{M}}(\tau)}\ (l\in\Pi)$$

$$\frac{(\mathcal{M},\mathcal{L},\Gamma\{x:l,y:\tau_2,z:\tau_3\}\vdash B):\Pi}{(\mathcal{M},\mathcal{L},\Gamma\{y:\tau_1,z:\tau_2\}\vdash l:x=y+z;B):\Pi}\ (l\notin\Pi) \qquad \frac{(\mathcal{M},\mathcal{L},\Gamma\{x:l,y:\tau_2,z:\tau_3\}\vdash B):\Pi}{(\mathcal{M},\mathcal{L},\Gamma,x\{y:\tau_1,z:\tau_2\}\vdash l:x=y+z;B):\Pi\cup\mathsf{flat}_{\mathcal{M}}(\tau_1)\cup\mathsf{flat}_{\mathcal{M}}(\tau_2)}\ (l\in\Pi)$$

$$\frac{(\mathcal{M},\mathcal{L},\Gamma\{x:l,y:\tau_2\}\vdash B):\Pi}{(\mathcal{M},\mathcal{L},\Gamma\{x:\tau_1,y:\tau_2\}\vdash l:x=x+y;B):\Pi}\ (l\notin\Pi) \qquad \frac{(\mathcal{M},\mathcal{L},\Gamma\{x:l,y:\tau_2\}\vdash B):\Pi}{(\mathcal{M},\mathcal{L},\Gamma\{x:\tau_1,y:\tau_2\}\vdash l:x=x+y;B):\Pi\cup\mathsf{flat}_{\mathcal{M}}(\tau_1)\cup\mathsf{flat}_{\mathcal{M}}(\tau_2)}\ (l\in\Pi)$$

$$\frac{(\mathcal{M},\mathcal{L},\Gamma\{x:\tau\}\vdash B):\Pi}{(\mathcal{M},\mathcal{L},\Gamma\{x:\tau\}\vdash l:\texttt{if}\ x\ \texttt{goto}\ l';B):\Pi\cup\mathsf{flat}_{\mathcal{M}}(\tau)}$$

(if $\Delta_P^{\Pi}\vdash(\mathcal{M},\mathcal{L},\Gamma'\vdash l':I';B'):\Pi'$ such that $\Gamma'\Subset\Gamma\{x:\tau\}$ and $\Pi'\subseteq\Pi\cup\mathsf{flat}_{\mathcal{M}}(\tau)$.)

$$\overline{(\mathcal{M},\mathcal{L},\Gamma\vdash l:\texttt{goto}\ l'):\Pi}$$

(if $\Delta_P^{\Pi}\vdash(\mathcal{M},\mathcal{L},\Gamma'\vdash l':I';B'):\Pi'$ such that $\Gamma'\Subset\Gamma$ and $\Pi'\subseteq\Pi$.)

Figure 10. Liveness Analysis for Dead Code Elimination

```
# simple_prog;
val it =  [(L 1, Const (V "x", 0)),(L 2, Const (V "y", 1 )),
           (L 3, Goto L 11),(L 11, Add2 (V "x", V "y" )),
           (L 12, If (V "x", L 21)),(L 13, Goto Label 11),
           (L 21, Ret V "x")] : prog

# INFER simple_prog;
val it = ([(Phi (L 11, V "x"), [Lt 11,Lt 1])],

           [([(V "y", D),(V "x", D)],                   (L  1, Const (V "x", 0))),
            ([(V "y", D),(V "x", Lt 1)],                (L  2, Const (V "y", 1))),
            ([(V "y", Lt 2),(V "x", Lt 1)],             (L  3, Goto L 11)),
            ([(V "y", Lt 2),(V "x", Phi (L 11, V "x"))], (L 11, Add2 (V "x", V "y"))),
            ([(V "y", Lt 2),(V "x", Lt 11)],            (L 12, If (V "x", L 21))),
            ([(V "y", Lt 2),(V "x", Lt 11)],            (L 13, Goto L 11)),
            ([(V "y", Lt 2),(V "x", Lt 11)],            (L 21, Ret V "x"))])
                    : (SSAtype * SSAtype list) list * ((var * SSAtype) list * prog) list
```

Figure 11. A Simple Program and the Type Reconstruction

ous subsections. We implemented a function `DeadCodeElim` which replaces dead instructions with skip instructions. `DeadCodeElim` collects types of variables which are used in conditional jumps and return instructions. If such a type is a label type, it indicates that the instruction of the label is live, and then `DeadCodeElim` checks types of used variables in the instruction. Otherwise, if such a type is a type variable, then `DeadCodeElim` checks its structure and collects all types appearing in the structure. This process exactly corresponds to the rule introduced in subsection 5.1. A use of `DeadCodeElim` is shown in Figure. 12.

Although the prototype and the optimization are rather toy implementations, we observe that our framework has the following practical benefits.

- Various static information such as liveness and value information of variables can be easily obtained from structure mapping and label environment, as shown in the previous subsections. Unlike in conventional SSA form, we do not need to solve further data flow equations, such as available expression and reaching definition equations.
- We do not need to implement un-SSA transformation algorithms. It would take some time to understand SSA transformation algorithms (type reconstruction algorithms in our framework) and implement one of them. Because it will take similar costs for implementing un-SSA transformation algorithms [5, 25], this seems to be considerable.

Currently, we are planning to implement our type system into an optimizer of the SML# compiler [28] being developed at Tohoku University.

6. Related Work

A recent work by Menon et al [18] presented a type-based approach to verify memory safety of SSA programs, and discussed its application to compiler optimization. Their type system, however, requires that the SSA property of a program is checked separately outside the typing framework. In contrast, the major contribution of the present paper is to have developed a type theoretical framework to represent the SSA property itself. To our knowledge, there does not seem to exist any existing attempt to represent SSA property in a static type system.

Our type system is a form of static verification system for a low-level code language. In this general perspective, it is similar to many recent proposals of type systems for low level languages including [19, 20, 26]. See also [17] for a survey on related topic.

```
# example_program;
val it = [(L 1, Const (V "x", 3)),(L 2, Const (V "y", 4)),(L 3, If (V "x", L 8)),
          (L 4, Add2 (V "x", V "y")),(L 5, If (V "y", L 10)),(L 6, Add1 (V "y", V "x", V "z")),
          (L 7, Goto L 12),(L 8, Const (V "y", 6)),(L 9, Goto L 12),
          (L 10, Add1 (V "z", V "x", V "y")),(L 11, Goto L 7),(L 12, Ret V "x")] : prog

# DeadCodeElim example_program;
val it = [(L 1, Const (V "x", 3)),(L 2, Const (V "y", 4)),(L 3, If (V "x", L 8)),
          (L 4, Add2 (V "x", Var "y")),(L 5, If (V "y", L 10)),(L 6, Skip),
          (L 7, Goto L 12),(L 8, Skip),(L 9, Goto L 12),
          (L 10, Skip),(L 11, Goto L 7),(L 12, Ret V "x")] : prog
```

Figure 12. An Example of Dead Code Elimination

Since SSA from is widely used as an low-level intermediate representation, it is an interesting future issue to consider the integration of our type theoretical representation of SSA property with those static verification systems. A further investigation in this line of work would be to combine static typing of low-level code with a logical verification system such as those proposed in [3, 15, 16].

Several papers [11, 2, 6] discussed correspondences between SSA form and restricted form of functional languages (CPS and A-normal forms.) Since the logical interpretation [22, 23] on which our type system is roughly based, is shown to be equivalent to the type system of the lambda calculus, our SSA typing could be used to extend these results to typed calculi.

7. Concluding Remarks

We have developed a static type system that represents the essential properties of static single assignment (SSA) form without performing code transformation. In our type system, the SSA property is represented in a typing derivation of an original program. We have shown that derivable typings precisely correspond to SSA programs. Furthermore, our type system is expressive enough to represent a practical SSA transformation algorithm. In our approach, a particular SSA transformation algorithm can be represented as a type reconstruction process that satisfies some additional constraint on allowable typing. We have constructed a type inference algorithm that is equivalent to the SSA transformation based on "dominance frontier." We have also shown that typical optimizations can be expressed as type-directed code transformations in our framework.

This is our first step towards a type theoretical approach to SSA form and SSA based compiler optimization. There are a number of interesting issues remain to be investigated. Here we only mention two of them below.

Representing other SSA algorithms There have been several variants of SSA transformation algorithms. *Pruned* SSA form proposed in [7] avoids dead ϕ functions, and inserts only live ϕ functions. In our type system, this strategy corresponds to choosing a minimum type context Γ with respect to the inclusion constraint ($dom(\Gamma_1) \subseteq dom(\Gamma_2)$ in Definition 1) at each branching instruction. Actually the type derivation depicted in the left hand side of Figure 5 is a such typing. In this way, we believe that our type system can represent various SSA transformation algorithms such as *semi-pruned* SSA form [5].

Generalizing the typing rule We introduced label restrictions for the relations $\leq$ and $\Subset$ in Definition 1. If labels are not annotated to type variables, our type system does not preserve dominance relation. For example, we consider the program in Figure 13. If label restrictions are not considered, a type derivation of the form $\mathcal{M}, \{x : \varphi\} \vdash l_3 : B_3$ and $\mathcal{M}, \{x : \varphi\} \vdash l_4 : B_4$ where $\mathcal{M} = \{\varphi \mapsto \{l_1, l_2\}\}$ is derivable. In Figure 13, although $l_3 : B_3$ does not dominate $l_4 : B_4$, x is given the same type variable φ in both of them. Therefore the unrestricted type system does not represent dominance relation. However, while label annotation is useful for some static analyses, the unrestricted type system might work as well, because atomic types consist of labels. Note that we only introduce one type variable φ in the type derivation, while Cytron et al's algorithms requires two different ϕ assignments in $l_3 : B_3$ and $l_4 : B_4$. In this sense, it can be said that Cytron et al's SSA form is not the "minimal" one.

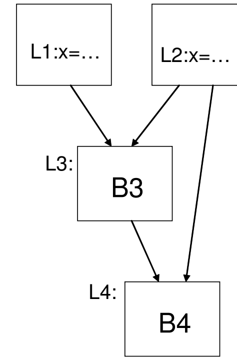

Figure 13. A Typical Example

Acknowledgments

We thank reviewers for their comments and suggestions that have been useful in improving the paper. The first author also thanks Hiroyuki Sato for his relevant comments on this work.

References

[1] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *POPL*, pages 1–11, 1988.

[2] Andrew W. Appel. SSA is functional programming. *ACM SIGPLAN Notices*, 33(4):17–20, 1998.

[3] Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In *POPL*, pages 14–25, 2004.

[4] Gianfranco Bilardi and Keshav Pingali. Algorithms for computing the static single assignment form. *J. ACM*, 50(3):375–425, 2003.

[5] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical improvements to the construction and destruction of static single assignment form. *Software Practice and Experience*, 28(8):859–881, 1998.

[6] M. Chakravarty, P. Keller, and P. Zadarnowski. A functional perspective on SSA optimisation algorithms. Technical Report 0217, University of New South Wales, 2003.

[7] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *POPL*, pages 55–66, 1991.

[8] Keith D. Cooper, L. Taylor Simpson, and Christopher A. Vick. Operator strength reduction. *TOPLAS*, 23(5):603–625, 2001.

[9] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 13(4):451–490, 1991.

[10] Dibyendu Das and U. Ramakrishna. A practical and fast iterative algorithm for phi-function computation using dj graphs. *TOPLAS*, 27(3):426–440, 2005.

[11] Richard A. Kelsey. A correspondence between continuation passing style and static single assignment form. *ACM SIGPLAN Notices*, 30(3):13–22, 1995.

[12] Robert Kennedy, Sun Chan, Shin-Ming Liu, Raymond Lo, Peng Tu, and Fred Chow. Partial redundancy elimination in SSA form. *TOPLAS*, 21(3):627–676, 1999.

[13] Jens Knoop, Oliver Ruthing, and Bernhard Steffen. Lazy code motion. In *PLDI*, pages 224–234, 1992.

[14] David Lacey, Neil Jones, Eric Van Wyk, and Carl Christian Frederikson. Proving correctness of compiler optimizations by temporal logic. *Higher-Order and Symbolic Computation*, 17(2), 2004.

[15] Sorin Lerner, Todd D. Millstein, and Craig Chambers. Automatically proving the correctness of compiler optimizations. In *PLDI*, pages 220–231, 2003.

[16] Sorin Lerner, Todd D. Millstein, Erika Rice, and Craig Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *POPL*, pages 364–377, 2005.

[17] Xavier Leroy. Java bytecode verification: Algorithms and formalizations. *J. Autom. Reasoning*, 30(3-4):235–269, 2003.

[18] Vijay S. Menon, Neal Glew, Brian R. Murphy, Andrew McCreight, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, and Leaf Petersen. A verifiable ssa program representation for aggressive compiler optimization. In *POPL*, pages 397–408, 2006.

[19] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *TOPLAS*, 21(3):527–568, 1999.

[20] George Necula. Proof-carrying code. In *POPL*, pages 106–119, 1997.

[21] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[22] Atsushi Ohori. The logical abstract machine: A curry-howard isomorphism for machine code. In *Fuji International Symposium on Functional and Logic Programming*, pages 300–318, 1999. (An extended version is available under the title "A Proof Theory for Machine Code" from the author's home page: http://www.pllab.riec.tohoku.ac.jp/~ohori/.)

[23] Atsushi Ohori. Register allocation by proof transformation. *Sci. Comput. Program (also in ESOP'03).*, 50(1-3):161–187, 2004.

[24] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. In *POPL*, pages 12–27, 1988.

[25] Vugranam C. Sreedhar, Roy Dz-Ching Ju, David M. Gillies, and Vatsa Santhanam. Translating out of static single assignment form. In *SAS*, pages 194–210, 1999.

[26] Raymie Stata and Martín Abadi. A type system for Java bytecode subroutines. In *POPL*, pages 149–160, 1998.

[27] Michael Weiss. The transitive closure of control dependence: The iterated join. *LOPLAS*, 1(2):178–190, 1992.

[28] The SML# Home page. http://www.pllab.riec.tohoku.ac.jp/smlsharp/

A. Static Single Assignment(SSA) Form [9, 1, 24]

A.1 Definitions

In this subsection we follow [9] and state definitions needed for SSA transformation.

Cytron et al [9] developed an efficient SSA transformation algorithm which places ϕ functions in *dominance frontiers* of the program.

DEFINITION 4 (Domination). *If all paths incoming to a node B_2 pass a node B_1, then B_1 is said to dominate B_2. We denote this relation by $B_1 \geq B_2$ (every node dominates itself.)*

DEFINITION 5 (Strict domination). *If a node B_1 dominates B_2 and $B_1 \neq B_2$, B_1 is said to strictly dominate B_2. We denote this relation by $B_1 \gg B_2$.*

DEFINITION 6 (Immediate dominator). *If a node B_1 strictly dominates B_2 and there is no other nodes that strictly dominates B_2 in any paths from B_1 to B_2, B_1 is said to be the immediate dominator of B_2.*

Dominance relation can be expressed by the *dominator tree*, in which each edge corresponds to the immediate dominance relation. The root of the tree is the entry node (**Entry**). Dominance frontier is defined as follows.

DEFINITION 7 (Dominance Frontier). *The dominance frontier $DF(B)$ of a CFG node B is the set of all CFG nodes B' such that B dominates a predecessor of B' but does not strictly dominate B':*

$$DF(B) = \{B' | (\exists P \in Pred(B'))(B \geq P \wedge B \not\gg B')\}.$$

A.2 Merge relation and dominance frontier [27, 4]

Dominance frontiers are obtained by computing *join sets*.

DEFINITION 8 (Join sets J). *Given a CFG $G = (V, E)$ and a set $\mathcal{I} \subseteq V$ of its nodes such that **entry** $\in \mathcal{I}$, $J(\mathcal{I})$ is the set of all nodes v for which there are distinct nodes $u, w \in \mathcal{I}$ such that there is a pair of paths $u \xrightarrow{+} v$ and $w \xrightarrow{+} v$, intersecting only at v.*

Cytron et al defined the join set for a set of nodes $\mathcal{I}$ as iterated sets [9]. However, later Weiss discovered that $J(\mathcal{I}) = J(\mathcal{I} \cup J(\mathcal{I}))$ [27]. Hence, the ϕ-assignments in the nodes $J(\mathcal{I})$ are sufficient: $DF^+(\mathcal{I}) = J(\mathcal{I})$. Furthermore, Bilardi and Pingari introduced another relation for calculating DF^+: the merge relation $Merge$ that holds between nodes v and w of the CFG whenever $v \in J(\{\mathbf{Entry}, w\})$; that is, v is a ϕ-node for a variable assigned only at w and **Entry**. The $Merge$ relation is proved to satisfy the following properties.

1. If $\{\mathbf{Entry}\} \subseteq \mathcal{I} \subseteq V$, then $J(\mathcal{I}) = \bigcup_{w \in \mathcal{I}} Merge(w)$ ($v \in Merge(w)$ if $v \in J(\{\mathbf{Entry}, w\})$.)
2. $v \in Merge(w)$ if and only if there is a so-called M-path from w to v in the CFG.
3. $Merge$ is a transitive relation.

We state some results from [4].

THEOREM 3. *$v \in Merge(w)$ iff there is a path $w \xrightarrow{+} v$ not containing $idom(v)$. Such a path is called an M path.*

The notion of M-path is proved to be closely related to the notion of dominance frontier.

PROPOSITION 1. *There exists a prime M-path from w to v iff $(w, v) \in DF$.*

THEOREM 4. $Merge = DF^+$.