A Type System for Proving Correctness of Compiler Optimizations

MATSUNO Yutaka and SATO Hiroyuki

The University of Tokyo, Japan
FAX: +81-3-3814-7231; {`matsu,schuko`}`@satolab.itc.u-tokyo.ac.jp`

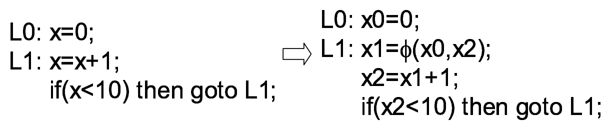
Abstract. A type system for proving correctness of compiler optimizations is proposed. We introduce *assignment types* for variables, which abstract the calculation process of the value. Soundness of our type system ensures that if types of return values are equal in two programs, the programs are equivalent. Then by extending the notion of type equality to order relation, we redefine several optimizations and prove that they preserve program semantics.

1 Introduction

Today, as compiler optimizations become essential parts for high performance computing, verification of compiler optimizations is recognized as an important issue. Several compiler verification systems have been proposed[14, 11]. However, they are still in early stage of the development. The fundamental matter is that only a few compiler optimizations have been proved correct in formal way. If there is no proof of the correctness, then how a verification system can prove that the optimization is correct? Several papers for formally proving correctness of compiler optimizations have been proposed[10, 3]. However, such papers have some problems for practical use, for example, they require complicated settings even to prove simple optimizations.

To prove correctness of compiler optimizations formally, first we consider how optimizations are usually applied. As stated by Allen and Kennedy [1], it is known that data dependence relations are sufficient to ensure that optimizations preserve program semantics, as conventional compilers verify that dependences among variables are preserved through optimizations. The process is known as *def-use analysis*. The introduction of *Static Single Assignment form (SSA form)*[6] has greatly simplified def-use analysis, and SSA form has become a standard intermediate form.

Following above observation, we propose a type theoretical framework for compiler optimizations which is based on the notion of def-use analysis in SSA form. The information of def-use analysis is represented by types for variables which we call *assignment types*. An assignment type is a record of how the value of the variable is calculated during the execution of program. For example, if a value is assigned to a variable x by an instruction $x = y + z$, and y and z are



```
L0: x=0;
L1: x=x+1;
    if(x<10) then goto L1;
```

```
L0: x0=0;
L1: x1=ϕ(x0,x2);
    x2=x1+1;
    if(x2<10) then goto L1;
```

Fig. 1. Source and SSA form of a program P where `x1` : α and `x2` : β

given types τ_1 and τ_2 respectively, then x is given a type $(\tau_1, \tau_2)_+$, intuitively meaning that a value is assigned to x by adding two variables of types τ_1 and τ_2. Note that this typing is only possible in SSA form, because there is only one assignment to a variable in SSA form. Sometimes a use-def chain makes a circle, which means that some variables are recursively calculated in a loop. In such a case their assignment types become recursive types. In Fig.1, let types of `x1` and `x2` be α and β, respectively. A variable whose value is assigned by a ϕ function is given a type $\{l_1 : \tau_1, l_2 : \tau_2\}$ where l_1 and l_2 are the labels of basic blocks from which the assignments of variables of type τ_1 and τ_2 come. $int(0)$ and $int(1)$ are singleton types[16] of integer whose values are 0 and 1, respectively. Type equations for α and β are:

$$\begin{cases} \alpha = \{L0 : int(0), L1 : \beta\} & (1) \\ \beta = (\alpha, int(1))_+ & (2) \end{cases}$$

By substituting the right hand side of (2) for β in (1), we get $\alpha = \{L0 : int(0), L1 : (\alpha, int(1))_+\}$, in other words `x1` has a recursive type of the form $\mu\alpha.\{L0 : int(0), L1 : (\alpha, int(1))_+\}$. We extract recursive calculations from unstructured programs as recursive types.

Our contributions are as follows.

– We prove soundness of our type system that if types of return values of source and optimized programs are equal, then they are equivalent programs. Soundness ensures that many structure preserving optimizations preserve program semantics.
– By extending the notion of type equality to order relation, we redefine several structure preserving optimizations, and formally prove that they also preserve program semantics.

Comparing with previous work, our framework has the following merits.

– Proofs are formal in the sense of type theory, and simple, using only standard inductive method.
– It is extensive as many structure preserving optimizations proved correct in uniform way.
– By introducing recursive types we avoid difficulty of analyzing unstructured loops. Some related works about equivalence of programs can only handle while-language, or need intricate settings for loops [7, 15, 2].

In Section 2, we give preliminaries for the paper. In Section 3 we define program equivalence. In Section 4, we introduce our type system. In Section 5 we state soundness of our type system. In Section 6, we define constant folding/propagation and code motion, and state that they preserve program semantics. Also we show that many structure preserving optimizations can be covered in our framework. Then in Section 7 we discuss related work and give a summary in Section 8.

2 Preliminaries

We define a simple assembler-like intermediate language in SSA form (Fig.2), which satisfies the following definition of SSA form.

Definition 1 (SSA form). *A program is said to be in SSA form if each of its variables is defined exactly once, and each use of a variable is dominated by that variable's definition.*

The meaning of domination is as follows. In the control flow graph of a program, if an instruction I_2 is dominated by an instruction I_1, then all control flow paths from the entry to I_2 pass I_1.

In order to provide a simple framework this language has no memory operations and function calls, as in some previous works such as [10]. We expect that our formalization can be extended to include such features, and leave them as future work. We define ordinary instructions such as $\texttt{add}\, x, y, v$ which means the value of $(y + v)$ where v is either a variable or an integer is assigned to x. Also a pseudo instruction $\texttt{ssa}\, x, (l_1 : x_1, l_2 : x_2)$ is defined for an assignment by ϕ functions. l_1 and l_2 are the labels of the predecessor basic blocks which the assignments of x_1 and x_2 come from. Though labels are not annotated in conventional SSA form, we annotate labels because the information of where the assignments come from is important for the verification. Also for simplicity we restrict the number of arguments of ϕ functions by two. Initial values are assigned to variables $x_0, y_0, \ldots$. Branch instructions such as `beq`, `bne`, `blt`, ... correspond to $=, \neq, <, \ldots$, respectively. For example, $\texttt{beq}\, x, v, l$ transfers an execution to a basic block labeled l if $x = v$. A program is represented as a sequence of instructions, and divided into a set of basic blocks. The first instruction of each basic block is labeled by a label $l \in \mathit{Label}$. Also Label includes l_{entry} and l_{exit} which are the labels for the unique entry and unique exit of a program, respectively. A program ends with a unique `return` instruction.

Program execution is modeled as follows. A machine state M has one of two forms: a terminate state $M = \texttt{HALT}$ or a tuple $(pc, V, (l_p, l_c))$ where pc is the order of the instruction to be executed, V is a map[1] of variables to their values, l_p is the label of the (immediate) predecessor block that the program execution has passed, and l_c is the label of the current block.

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

$$
\begin{array}{rl}
\textit{integers} & i \in I \\
\textit{labels} & l \in \textit{Label} \\
\textit{variables} & x, y, z, \ldots \in Var \\
\textit{values} & v ::= i \mid x \\
\textit{arithmetic ops} & \texttt{aop} ::= \texttt{add} \mid \texttt{sub} \mid \texttt{mul} \\
\textit{branch ops} & \texttt{bop} ::= \texttt{beq} \mid \texttt{bne} \mid \texttt{blt} \mid \texttt{blte} \mid \texttt{bgt} \mid \texttt{bgte} \\
\textit{branch instructions} & \textit{bins} ::= \texttt{bop}\ x, v, l \mid \texttt{jmp}\ l \mid \texttt{return}\ x \\
\textit{instructions} & \textit{ins} ::= \texttt{aop}\ x, y, v \mid \texttt{mov}\ x, v \mid \texttt{ssa}\ x, (l_1 : x_1, l_2 : x_2) \\
\textit{instruction sequences} & I ::= I_\emptyset \mid \textit{ins} \mid \textit{ins}; I \\
\textit{basic blocks} & B ::= I \mid I; \textit{bins} \\
\textit{programs} & P ::= l_0 : B_0, l_1 : B_1, \ldots, l_{n-1} : B_{n-1}
\end{array}
$$

Fig. 2. Syntax for Intermediate Language in SSA form

- Initial machine state: $M_0 = (0, V_0, (l_{entry}, l_0))$ where V_0 is the map from variables to the initial values. l_0 is the label of the first basic block to be executed.
- Program executions are denoted as $P \vdash (pc, V, (l_p, l_c)) \to (pc', V', (l'_p, l'_c))$, which means that program P can, in one step, go from state $(pc, V, (l_p, l_c))$ to state $(pc', V', (l'_p, l'_c))$.
- If the program reaches the `return` x instruction, the state transfers to `HALT` state.

Fig.3 contains a one-step operational semantics for instructions where `aop` is either `add`, `sub` or `mul` and aop is either $+$, $-$, or $\times$, respectively. $M = (pc, V, (l_p, l_c))$ is also defined as a map: if $v \in Dom(V)$ then $M[v] = V[v]$ else $M[v] = i$ when v is $i \in I$; L is defined as a map from labels to the order of the first instructions of basic blocks with which they are associated. A function $NL(pc, (l_p, l_c))$ checks whether the next instruction is the first instruction of the next basic block, and returns the new pair of previous and current labels.

Definition 2 **($NL(pc, (l_p, l_c))$).** *Assume that there is a program with a label mapping L.*

$$NL(pc, (l_p, l_c)) = \begin{cases} (l_c, l) & \exists l. L[l] = pc + 1 \\ (l_p, l_c) & \textit{otherwise} \end{cases}$$

3 Program Equivalence

We define program equivalence which is closely related to the definition in [10]. To verify the equivalence, we check all possible paths that source and optimized codes may take, and verify that each path in source code corresponds to a path in

$$\frac{M = (pc, V, (l_p, l_c)) \quad P[pc] = \texttt{aop}\, x, y, v \quad V[y] = i \quad M[v] = j}{P \vdash M \to (pc+1, V[x \mapsto (i \text{ aop } j)], NL(pc, (l_p, l_c)))}$$

$$\frac{M = (pc, V, (l_p, l_c)) \quad P[pc] = \texttt{mov}\, x, v \quad M[v] = i}{P \vdash M \to (pc+1, V[x \mapsto i], NL(pc, (l_p, l_c)))}$$

$$\frac{M = (pc, V, (l_i, l_c)) \quad P[pc] = \texttt{ssa}\, x, (l_1 : x_1, l_2 : x_2)}{P \vdash M \to (pc+1, V[x \mapsto V[x_i]], NL(pc, (l_i, l_c)))}\ (i \in \{1, 2\})$$

$$\frac{M = (pc, V, (l_p, l_c)) \quad P[pc] = \texttt{jmp}\, l}{P \vdash M \to (L[l], V, (l_c, l))}$$

$$\frac{M = (pc, V, (l_p, l_c)) \quad P[pc] = \texttt{bop}\, x, v, l \quad V[x] = i \quad M[v] = j \quad i \text{ bop } j = \mathit{true}}{P \vdash M \to (L[l], V, (l_c, l))}$$

$$\frac{M = (pc, V, (l_p, l_c)) \quad P[pc] = \texttt{bop}\, x, v, l \quad \exists l'.L[l'] = pc+1 \quad V[x] = i \quad M[v] = j \quad i \text{ bop } j = \mathit{false}}{P \vdash M \to (pc+1, V, (l_c, l'))}$$

$$\frac{M = (pc, V, (l_p, l_c)) \quad P[pc] = \texttt{return}\ x}{P \vdash M \to \texttt{HALT}}$$

Fig. 3. Dynamic Semantics

the optimized code and vice versa. First, such paths are defined as *computation prefixes.*

Definition 3 (Computation Prefix). *For a program P, a computation prefix is a sequence (finite or infinite)*

$$P \vdash M_0 \to M_1 \to M_2 \to \cdots$$

where $P \vdash M_i \to M_{i+1}$ for $i = 0, 1, 2, \ldots$.

Program equivalence is determined by the return value. We call the variable whose value is the return value of a program P as the observable variable, and denote it as $OV(P)$. If a program P ends with $\texttt{return}\, x$, then $OV(P)$ is x. Program equivalence is defined as follows.

Definition 4 (Program Equivalence). *Assume that there are two programs P and Q such that $OV(P) = OV(Q) = x$ which share a label set Label and a variable set Var.*

$$P \approx Q$$

if and only if,
for any finite computation prefix of P:

$$P \vdash M_0 \to M_1 \to M_2 \to \cdots M_i \overset{return\ x}{\to} \mathit{HALT},$$

there exists a computation prefix of Q:

$$Q \vdash M_0 \to M'_1 \to M'_2 \to \cdots M'_j \overset{return\ x}{\to} \mathit{HALT}$$

such that $M_i[x] = M'_j[x]$, and vice versa.

Note that the length of computation prefixes are not necessarily the same.

In this paper we require that the structure of control flow graph of a program is not modified by optimizations. Optimizations which preserve this property are known as *structure preserving optimizations* in which almost all global optimizations are included.

Definition 5. *Two programs P and Q:*

$$P = l_0 : B_0, l_1 : B_1, \ldots, l_{n-1} : B_{n-1}$$
$$Q = l_0 : B'_0, l_1 : B'_1, \ldots, l_{n-1} : B'_{n-1}$$

which share a label set Label and a variable set Var are structurally equivalent:

$$P \cong Q$$

if and only if for each B_i and B'_i, one of the following conditions holds.

- *Either B_i or B'_i does not end with branch instructions.*
- *Both B_i and B'_i end with the same* **bop** x, v, l *where $x \in Var$, $v \in Var$ if v is a variable, and $l \in Label$.*
- *Both B_i and B'_i end with the same* **jmp** l *where $l \in Label$.*
- *Both B_i and B'_i end with the same* **return** x *where $x \in Var$.*

As shown in Definition 5 we need to analyze variables used in branch instructions. We define such variables as *control variables.*

Definition 6. *The control variable set for program P, $CV(P)$ is a set of all variables which are used in branch instructions of P.*

4 Type System

4.1 Assignment Types

We define assignment types as follows.

$$\begin{array}{rl} \textit{type variables} & \alpha, \beta, \gamma, \ldots \\ \textit{types} & \tau ::= \alpha \mid \top \mid \mathit{int}(i) \mid (\tau_1, \tau_2)_+ \mid (\tau_1, \tau_2)_- \mid (\tau_1, \tau_2)_\times \\ & \quad \mid \{l_1 : \tau_1, l_2 : \tau_2\} \mid \mu\alpha.\{l_1 : \tau_1, l_2 : \tau_2\} \\ \textit{type environments} & \Gamma ::= \cdot \mid x : \tau \mid \Gamma_1, \Gamma_2 \end{array}$$

Type variables are represented by $\alpha, \beta, \ldots$. $\top$ is the super type of all other types. $\mathit{int}(i)$ is a singleton type of a variable or an integer whose value is i. Sometimes we use a notation $\mathit{int}(x)$ for $\mathit{int}(i)$ when the value of x is i. $(\tau_1, \tau_2)_+$, $(\tau_1, \tau_2)_-$, and $(\tau_1, \tau_2)_\times$ are assignment types of variables to which values are assigned by add, subtract, and multiply operations, respectively. $\{l_1 : \tau_1, l_2 : \tau_2\}$ is the type of a variable to which a value is assigned by an `ssa` instruction. Variables to which values are assigned by `mov` instructions also have these types. A type environment is defined as a sequence of type declaration of variables as usual. The exchange rule for type environments holds, such that $\Gamma_1, x : \tau_1, \Gamma_2, y : \tau_2$ and $\Gamma_1, y : \tau_2, \Gamma_2, x : \tau_1$ are the same. We denote $Dom(\Gamma)$ for the set of variables whose types are declared in Γ.

4.2 Type System

The type system is shown in Fig.4. We explain the meaning of judgments as follows.

- $\vdash \Gamma$
  This judgment says that Γ is a valid type environment. Specially, we define Γ_0 as the type environment for initial values, $x_0, y_0, \ldots$. They are given types $int(i), int(j), \ldots$ in Γ_0 if their initial values are $i, j, \ldots$ respectively, and we assume that $\vdash \Gamma_0$.
- $\Gamma \vdash_P v : \tau$
  This judgment says that v is given a type τ in a program P under a type environment Γ.

$\Gamma \vdash_P v : \tau$ is derived by the following rules. Rule (**type-int**) is applied when v is an integer. Rule (**type-var**) is applied when $v : \tau$ is declared in Γ. Rule (**type-aop**) is applied when the value is assigned by an arithmetic operation in a program P. (**type-mov**) and (**type-ssa**) are similar to (**type-aop**). Intuitively (**type-mu**) means that if a variable x is given a type $\{l_1 : \tau_1, l_2 : \tau_2\}$ under a type environment in which x itself is given a type variable α, then x is given a type $\mu\alpha.\{l_1 : \tau_1, l_2 : \tau_2\}$, where the type declaration $x : \alpha$ is deleted from the type environment. We restrict the body of μ types to ssa types. The cut rule allows substituting a valid type of a variable x for an occurrence of a type variable α in a valid type judgment where x is given α in the type environment. In our type system, $\oplus$ operation is used for type environments.

Definition 7 **($\Gamma_1 \oplus \Gamma_2$).** *Assume that $\Gamma_1 = x_1 : \tau_1, x_2 : \tau_2, \ldots$, $\Gamma_2 = x'_1 : \tau'_1, x'_2 : \tau'_2, \ldots$, and $\{x''_1, x''_2, \ldots\} = \{x_1, x_2, \ldots\} \cup \{x'_1, x'_2, \ldots\}$.*

$$\Gamma_1 \oplus \Gamma_2 = x''_1 : \tau''_1, x''_2 : \tau''_2, \ldots$$

where for each x''_i

- *if $x''_i = x_j = x'_k$*
  - *if $\tau_j = \tau'_k$ then $\tau''_i = \tau_j$*
  - *else $\tau''_i = \top$*
- *else*
  - *if $x''_i = x_j$ then $\tau''_i = \tau_j$*
  - *else if $x''_i = x'_k$ then $\tau''_i = \tau'_k$*

Because only type environments of the form Γ_0 augmented with some type declarations in which variables are given type variables are valid (such as $\Gamma_0, x : \alpha, y : \beta, \ldots$), $\top$ never appears in valid type derivation trees. As an example of typing, the type derivation for `x1` in Fig.1 is shown in Fig.5 (we assume that $\Gamma_0 = x_0 : int(0)$).

$$\boxed{\vdash \mathit{\Gamma}}$$

$$\frac{}{\vdash \mathit{\Gamma}_0} \qquad \frac{\vdash \mathit{\Gamma} \quad x \notin \mathit{Dom}(\mathit{\Gamma}) \quad \alpha \text{ fresh in } \mathit{\Gamma}}{\vdash \mathit{\Gamma}, x : \alpha}$$

$$\boxed{\mathit{\Gamma} \vdash_P v : \tau}$$

$$\frac{\vdash \mathit{\Gamma}}{\mathit{\Gamma} \vdash_P i : \mathit{int}(i)}\ (\textbf{type-int}) \qquad \frac{\vdash \mathit{\Gamma} \quad x : \tau \in \mathit{\Gamma}}{\mathit{\Gamma} \vdash_P x : \tau}\ (\textbf{type-var})$$

$$\frac{\texttt{aop}\ x, y, v \in P \quad \mathit{\Gamma}_1 \vdash_P y : \tau_1 \quad \mathit{\Gamma}_2 \vdash_P v : \tau_2}{\mathit{\Gamma}_1 \oplus \mathit{\Gamma}_2 \vdash_P x : (\tau_1, \tau_2)_{\mathrm{aop}}}\ (\textbf{type-aop})$$

$$\frac{\texttt{mov}\ x, v \in P \quad \mathit{\Gamma} \vdash_P v : \tau}{\mathit{\Gamma} \vdash_P x : \tau}\ (\textbf{type-mov})$$

$$\frac{\texttt{ssa}\ x, (l_1 : x_1, l_2 : x_2) \in P \quad \mathit{\Gamma}_1 \vdash_P x_1 : \tau_1 \quad \mathit{\Gamma}_2 \vdash_P x_2 : \tau_2}{\mathit{\Gamma}_1 \oplus \mathit{\Gamma}_2 \vdash_P x : \{l_1 : \tau_1, l_2 : \tau_2\}}\ (\textbf{type-ssa})$$

$$\frac{\mathit{\Gamma}, x : \alpha, \mathit{\Gamma}' \vdash_P x : \{l_1 : \tau_1, l_2 : \tau_2\}}{\mathit{\Gamma}, \mathit{\Gamma}' \vdash_P x : \mu\alpha.\{l_1 : \tau_1, l_2 : \tau_2\}}\ (\textbf{type-mu}) \qquad \frac{\mathit{\Gamma} \vdash_P x : \tau_1 \quad \mathit{\Gamma}', x : \alpha, \mathit{\Gamma}'' \vdash_P y : \tau_2}{\mathit{\Gamma} \oplus (\mathit{\Gamma}', \mathit{\Gamma}'') \vdash_P y : \tau_2[\tau_1/\alpha]}\ (\textbf{cut})$$

Fig. 4. Typing Rules

$$\frac{\texttt{add}\, x_2, x_1, 1 \in P \qquad \dfrac{\dfrac{\vdash \mathit{\Gamma}_0 \quad x_1 \notin \mathit{dom}(\mathit{\Gamma}_0) \quad \alpha \text{ fresh}}{\vdash \mathit{\Gamma}_0, x_1 : \alpha} \qquad x_1 : \alpha \in \mathit{\Gamma}_0, x_1 : \alpha}{\mathit{\Gamma}_0, x_1 : \alpha \vdash_P x_1 : \alpha, 1 : \mathit{int}(1)}\ (\textbf{type-var}), (\textbf{type-int})}{\mathit{\Gamma}_0, x_1 : \alpha \vdash_P x_2 : (\alpha, \mathit{int}(1))_+ \quad (D_1)}\ (\textbf{type-add})$$

$$\frac{\texttt{ssa}\, x_1, (L0 : x_0, L1 : x_2) \in P \qquad \dfrac{\dfrac{\vdash \mathit{\Gamma}_0 \quad x_2 \notin \mathit{dom}(\mathit{\Gamma}_0) \quad \beta \text{ fresh}}{\vdash \mathit{\Gamma}_0, x_2 : \beta} \qquad x_2 : \beta \in \mathit{\Gamma}_0, x_2 : \beta}{\mathit{\Gamma}_0, x_2 : \beta \vdash_P 0 : \mathit{int}(0), x_2 : \beta}\ (\textbf{type-int}), (\textbf{type-var})}{\mathit{\Gamma}_0, x_2 : \beta \vdash_P x_1 : \{L_0 : \mathit{int}(0), L_1 : \beta\} \quad (D_2)}\ (\textbf{type-ssa})$$

$$\frac{\dfrac{D_1 \quad D_2}{\mathit{\Gamma}_0, x_1 : \alpha \vdash_P x_1 : \{L0 : \mathit{int}(0), L1 : (\alpha, \mathit{int}(1))_+\}}\ (\textbf{cut})}{\mathit{\Gamma}_0 \vdash_P x_1 : \mu\alpha.\{L0 : \mathit{int}(0), L1 : (\alpha, \mathit{int}(1))_+\}}\ (\textbf{type-mu})$$

Fig. 5. Type derivation of `x1` in Fig.1

4.3 Cut Elimination Theorem

In our type system, cut elimination theorem holds. The (**cut**) rule corresponds to substitution operation for solving type equations as shown in Section 1. By Theorem 1 it is ensured that all type equations can be solved, no matter what way we use (**cut**) rule. Also, for proving soundness of our type system and other theorems, it suffices to prove the case of cut free derivations.

Theorem 1 (Cut Elimination). *If $\Gamma_0 \vdash_P x : \tau$ is derivable, then $\Gamma_0 \vdash_P x : \tau$ can be derived without any (**cut**) rules.*

4.4 Type Equality

In our type system, a recursive type takes several forms for the same variable. Though they look different, they are equal by the following definition. First, we define $T(\tau)$ to be the regular (possibly infinite) tree obtained by completely unfolding all occurrences of $\mu\alpha.\tau$ to $\tau[\mu\alpha.\tau/\alpha]$.

Definition 8 (Regular Trees $T(\tau)$). *Regular trees are defined as follows(Fig.6). Nodes are of the form $(+)$,$(-)$, $(\times)$, (ssa), or leaves of the form $(int(i))$. If $T(\tau_1)$ and $T(\tau_2)$ have the same structure, we denote $T(\tau_1) = T(\tau_2)$.*

$$T(int(i)) = (int(i))$$

$$\begin{tikzcd}[column sep=tiny] T((\tau_1,\tau_2)_{aop}) = & & \text{(aop)} \arrow[dl] \arrow[dr] & \\ & T(\tau_1) & & T(\tau_2) \end{tikzcd}$$

$$\begin{tikzcd}[column sep=tiny] T(\{l_1:\tau_1, l_2:\tau_2\}) = & & \text{(ssa)} \arrow[dl] \arrow[dr] & \\ & l_1 : T(\tau_1) & & l_2 : T(\tau_2) \end{tikzcd}$$

$$T(\mu\alpha.\{l_1:\tau_1, l_2:\tau_2\}) = T(\{l_1:\tau_1, l_2:\tau_2\}[\mu\alpha.\{l_1:\tau_1, l_2:\tau_2\}/\alpha])$$

Fig. 6. Regular Trees

Type equality is defined by the structure of regular trees [4].

Definition 9 (Type Equality). *If $T(\tau_1) = T(\tau_2)$, then $\tau_1 = \tau_2$.*

Note that this equality is obviously decidable. The following theorem ensures that our type system is consistent.

Theorem 2. *For any derivations of $\Gamma_0 \vdash_P x : \tau$ and $\Gamma_0 \vdash_P x : \tau'$, $\tau = \tau'$.*

5 Soundness

The main contribution of this paper is the following proof of soundness.

Theorem 3 (Soundness). *If $P \cong Q$, $OV(P) = OV(Q) = x$ and*

– $\exists\tau\exists\tau'.(\Gamma_0 \vdash_P x : \tau \wedge \Gamma_0 \vdash_Q x : \tau' \wedge \tau = \tau')$
– $\forall y_j \in CV(P)\exists\tau_j\exists\tau_j'.(\Gamma_0 \vdash_P y_j : \tau_j \wedge \Gamma_0 \vdash_Q y_j : \tau_j' \wedge \tau_j = \tau_j')$

then $P \approx Q$.

To prove soundness, we prove the following lemmas.

Lemma 1. *Assume that $P \cong Q$ and*

– $\forall y_j \in CV(P)\exists\tau_j\exists\tau_j'.(\Gamma_0 \vdash_P y_j : \tau_j \wedge \Gamma_0 \vdash_Q y_j : \tau_j' \wedge \tau_j = \tau_j')$.

Then for any computation prefix of P:

$$P \vdash M_0 \to \dots \overset{bins_1}{\to} \dots \overset{bins_2}{\to} \dots \overset{bins_n}{\to} \dots,$$

where $bins_1, \dots, bins_n$ are branch instructions, there exists a computation prefix of Q:

$$Q \vdash M_0 \to \dots \overset{bins_1}{\to} \dots \overset{bins_2}{\to} \dots \overset{bins_n}{\to} \dots$$

and conversely.

Lemma 2. *Assume that $P \cong Q$ and*

– $\forall y_j \in CV(P)\exists\tau_j\exists\tau_j'.(\Gamma_0 \vdash_P y_j : \tau_j \wedge \Gamma_0 \vdash_Q y_j : \tau_j' \wedge \tau_j = \tau_j')$.

Also assume that for any computation prefix of P:

$$P \vdash M_0 \to M_1 \dots \overset{assignment\ to\ x}{\to} M$$

there exists a computation prefix of Q:

$$Q \vdash M_0 \to M_1' \dots \overset{assignment\ to\ x}{\to} M',$$

and conversely. If

– $\exists\tau\exists\tau'.(\Gamma_0 \vdash_P x : \tau \wedge \Gamma_0 \vdash_Q x : \tau' \wedge \tau = \tau')$

, then $M[x] = M'[x]$.

We prove Lemma 1 and Lemma 2 simultaneously.
Proof
Note that by $P \cong Q$, $CV(P) = CV(Q)$. For Lemma 2, we prove by induction on the sum of length of computation prefix to the assignment to x in P and Q. For Lemma 1, we prove by induction on the number of branch instructions. Because of cut elimination theorem, in the following proof, we have only to consider cut free derivations.

$\boxed{\textbf{Base Case}}$

There are two cases for minimum length of computation prefix for the assignment to x (1 and 2) and three cases for the least number of branches (3, 4, and 5). Assume that these computation prefixes are taken in P.

1. $P \vdash M_0 \overset{\texttt{aop}\ x,y_0,v_0}{\rightarrow} M_1$
2. $P \vdash M_0 \overset{\texttt{mov}\ x,v}{\rightarrow} M_1$
3. $P \vdash M_0 \overset{\texttt{jmp}\ l}{\rightarrow} M_1$
4. $P \vdash M_0 \overset{\texttt{bop}\ y_0,v_0,l}{\rightarrow} M_1$
5. $P \vdash M_0 \overset{\texttt{return}\ x_0}{\rightarrow} \texttt{HALT}$

In case 1, the type derivation for x is $\Gamma_0 \vdash_P x : (int(y_0), int(v_0))_{\text{aop}}$. Because x has the same type in P and Q, by considering the type derivation of x, clearly x has the same value in both P and Q. Case 2 is the same as case 1. In case 3, because $P \cong Q$, P and Q are identical. In case 4 because $P \cong Q$ and y_0 and v_0 are initial values, if P takes true or false branch then Q takes the same branch. In case 5, because $P \cong Q$, P and Q are identical.

$\boxed{\textbf{Induction Step}}$

First we prove Lemma 1. Assume that for any computation prefix of P,

$$P \vdash M_0 \rightarrow \ldots \overset{bins_1}{\rightarrow} \ldots \overset{bins_2}{\rightarrow} \ldots \overset{bins_k}{\rightarrow} \ldots$$

there is a computation prefix of Q

$$Q \vdash M_0 \rightarrow \ldots \overset{bins_1}{\rightarrow} \ldots \overset{bins_2}{\rightarrow} \ldots \overset{bins_k}{\rightarrow} \ldots$$

and conversely. We consider each variable z which is used in the $(k+1)$ th branch instruction. Because z is used in the $(k+1)$ th branch instruction, the assignment to z is in the computation prefixes of P and Q before reaching the $(k+1)$ th branch (or z is an initial value. in this case clearly the value of z is the same in both P and Q). By Lemma 2, the value of z in P and Q are the same since z is a control variable before reaching the $(k+1)$ th branch. Hence both computation prefixes pass the same branch at the $(k+1)$ th branch instruction. This proves Lemma 1. Then we prove Lemma 2. The case when $\tau = int(i)$ is easy. We consider other cases.

$\boxed{\textbf{Case 1}}$

$\tau = (\tau_1, \tau_2)_+$ (similar for other arithmetic types). In this case a value is assigned to x by either an `add` instruction or a `mov` instruction. There are two cases for the last typing rule: (**type-mov**) and (**type-add**) rules. When (**type-mov**) is used, the type derivation is as follows.

$$\frac{\texttt{mov}\ \ x, v \in P \quad \Gamma_0 \vdash_P v : (\tau_1, \tau_2)_+}{\Gamma_0 \vdash_P x : (\tau_1, \tau_2)_+} \quad \textbf{(type-mov)}$$

In this case the value of x depends on v. Therefore we proceed the proof for v. When a value is assigned to x by `add` instructions in both P and Q, the type derivation of x in P is (the same as in Q though the variable used in `add` instruction is not necessarily the same (but the types are equal). For notational convenience we assume that the value of x is assigned by the same variable in P and Q):

$$\frac{\texttt{add}\ \ x, y, v \in P \quad \Gamma_0 \vdash_P y : \tau_1, v : \tau_2}{\Gamma_0 \vdash_P x : (\tau_1, \tau_2)_+} \quad \textbf{(type-add)}$$

Since the assignment to y dominates the assignment to x in both P and Q, the assignment to y must be in the computation prefixes of both P and Q before the assignment to x. Also the type of y is the same in P and Q, and the value is assigned by shorter prefix than that to the assignment to x, by induction hypothesis the value of y is the same in P and Q before the assignment to x. We consider the case v is a variable and denote it as z (it is easy when v is an integer). By the same argument, the value of z is the same before the assignment to x. Therefore the value of x is the same in P and Q. Hence $M[x] = M'[x]$.

$\boxed{\textbf{Case 2}}$

$\tau = \{l_1 : \tau_1, l_2 : \tau_2\}$. There is a following type derivation for x in P (the same as in Q):

$$\frac{\texttt{ssa}\ \ x, (l_1 : x_1, l_2 : x_2) \in P \quad \varGamma_0 \vdash_P x_1 : \tau_1, x_2 : \tau_2}{\varGamma_0 \vdash_P x : \{l_1 : \tau_1, l_2 : \tau_2\}}$$

(type-ssa)

where the types of x_i $(i = \{1, 2\})$ in P and Q are equal. By Lemma 1, for any computation prefix of P (denoted as CP)

$$P \vdash M_0 \to \ldots \overset{bins_1}{\to} \ldots \overset{bins_k}{\to} \ldots \overset{\texttt{ssa}\ \ x,(l_1:x_1,l_2:x_2)}{\to} M$$

there is a computation prefix of Q (denoted as CP'),

$$Q \vdash M_0 \to \ldots \overset{bins_1}{\to} \ldots \overset{bins_k}{\to} \ldots \overset{\texttt{ssa}\ \ x,(l_1:x_1,l_2:x_2)}{\to} M'$$

and conversely. Because both CP and CP' pass the same sequence of branch instructions, both CP and CP' pass $l_i : B_i$ (and then reach to $\texttt{ssa}\ \ x, (l_1 : x_1, l_2 : x_2)$). Hence there must be assignments for x_i in both CP and CP'. Because types of x_i in P and Q are equal and the value is assigned by shorter prefix than that of x, by induction hypothesis $M[x] = M'[x]$ where $M[x_i] = M'[x_i]$.

$\boxed{\textbf{Case 3}}$

$\tau = \mu\alpha.\{l_1 : \tau_1, l_2 : \tau_2\}$. In this case values may be assigned to x in more than one time by the same instruction in a computation prefix (i.e. assigned in a loop):

$$P \vdash M_0 \to \ldots \overset{\text{assignment to } x^1}{\to} \ldots \overset{\text{assignment to } x^i}{\to} M,$$

where superscripts for x are iteration times. Assume that there also exists the following computation prefix of Q:

$$Q \vdash M_0 \to \ldots \overset{\text{assignment to } x^1}{\to} \ldots \overset{\text{assignment to } x^i}{\to} M'.$$

We prove $M[x]$=$M'[x]$ by induction on the iteration times. We also have to consider the case that a branch instruction is executed more than one time: we prove that for any computation prefix of P:

$$P \vdash M_0 \to \ldots \overset{bins^1}{\to} \ldots \overset{bins^2}{\to} \ldots \overset{bins^j}{\to} \ldots$$

there exists a computation prefix of Q,

$$Q \vdash M_0 \to \ldots \overset{bins^1}{\to} \ldots \overset{bins^2}{\to} \ldots \overset{bins^j}{\to} \ldots,$$

and conversely. In this case we prove by induction on the iteration times. Base cases for these two proofs are already proved in previous cases. By assumption all types of x and variables in $CV(P)$ have the same tree structure. Therefore $\Gamma_0 \vdash_P x^i : \tau_i \wedge \Gamma_0 \vdash_Q x^i : \tau'_i$ and $\Gamma_0 \vdash_P y^j : \tau''_j \wedge \Gamma_0 \vdash_Q y^j : \tau'''_j$ for all $y \in CV(P)$ where τ_i and τ'_i, and τ''_j and τ'''_j have the same tree structures. By proving simultaneously as in previous cases, we get the required result. □

Then we prove soundness.

$\boxed{\textbf{Proof of Soundness}}$
Assume that there exists a computation prefix of P:

$$P \vdash M_0 \ldots \overset{bins_1}{\rightarrow} \ldots \overset{bins_n}{\rightarrow} \ldots \overset{\text{return } x}{\rightarrow} \texttt{HALT},$$

with $bins_1, \ldots, bins_n$ branch instructions. By Lemma 1, there exists a computation prefix of Q:

$$Q \vdash M_0 \ldots \overset{bins_1}{\rightarrow} \ldots \overset{bins_n}{\rightarrow} \ldots \overset{\text{return } x}{\rightarrow} \texttt{HALT}$$

and conversely. Since x is used by `return` x in both computation prefixes, there must be the assignments to x in both computation prefixes (or x is the initial value. In this case the value of x is the same). By Lemma 2, the values of x are equal before the `return` instructions. Hence $P \approx Q$. □

Many optimizations which preserve types such as simple constant propagation, common subexpression elimination, loop invariant hoisting and value numbering satisfy soundness of our type system. For example, the following common subexpression elimination preserves program semantics because $(\tau_1, \tau_2)_\times$, which is the type of `d` (assuming `b`:τ_1 and `c`:τ_2) is unchanged.

```
a = b * c;     a = b * c;
  ...      =>    ...
d = b * c;     d = a;
```

6 Several Optimizations

6.1 Constant Folding

Constant folding rewrites an expression with constant operands by its evaluated value, thus improves run-time performance and reduces code size. The following transformation is an example of constant folding.

```
a = 3 * 2; -> a = 6;
```

Constant folding is a simple optimization. However, proving the correctness is not easy. In this paper, we redefine an optimization by an order relation between types of variables before and after the optimization is applied.

Definition 10 (Optimizing Order). *Optimizing order on the set of types is an order relation defined as transitive closure of:*

$$\frac{\tau = \tau'}{\tau \succeq \tau'} \qquad \frac{\tau_1 \succeq \tau_2 \quad \tau_3 \succeq \tau_4}{(\tau_1, \tau_3)_{aop} \succeq (\tau_2, \tau_4)_{aop}} \qquad \frac{\tau_1 \succeq \tau_2 \quad \tau_3 \succeq \tau_4}{\{l_1 : \tau_1, l_2 : \tau_3\} \succeq \{l_1 : \tau_2, l_2 : \tau_4\}}$$

In the above program $(int(3), int(2))_\times$, which is the type of a changes into $int(6)$ and $((int(3), int(2))_\times, int(4))_+$, which is the type of b changes into $int(10)$. Constant folding is redefined as a program transformation which changes types in this way.

Definition 11 (Constant Folding). *Assume that $P \cong Q$ and $OV(P) = OV(Q) = x$.*

$$P \to_{fp} Q$$

iff

- $\exists \tau_i \exists \tau'_i.(\mathit{\Gamma}_0 \vdash_P x : \tau \wedge \mathit{\Gamma}_0 \vdash_Q x : \tau' \wedge \tau \succeq_{fp} \tau')$
- $\forall y_j \in CV(P) \exists \tau_j \exists \tau'_j.(\mathit{\Gamma}_0 \vdash_P y_j : \tau_j \wedge \mathit{\Gamma}_0 \vdash_Q y_j : \tau'_j \wedge \tau_j \succeq_{fp} \tau'_j)$

where an optimizing order $\succeq_{fp}$ is defined as the transitive closure of:

$$(int(i), int(j))_{aop} \succeq_{fp} int(k) \ (k = i \ aop \ j)$$

Theorem 4. *If $P \to_{fp} Q$, then $P \approx Q$.*

The proof is done in the similar way of that of soundness.

The type information can be used for specifying conditions for applying an optimization as temporal logic is used in [10]. For example, if $\mathit{\Gamma}_0 \vdash_P x : (int(5), int(4))_+$ is derivable, then the assignment to x in P can be transformed to mov $x, 9$.

By the same way optimizations using arithmetic laws (associative, distributive, and commutative laws) can be defined by optimizing orders which represent those laws, and proved correct. For example, the following transformation is proved to preserve program semantics by introducing distributive law as an optimizing order.

```
a1 = a0 * 2;     a1 = a0 + b0;
b1 = b0 * 2; =>
a2 = a1 + b1;    a2 = a1 * 2;
```

Assuming that a0:τ_1 and b0:τ_2,

$$((\tau_1, int(2))_\times, (\tau_2, int(2))_\times)_+$$

, which is the type of a2, changes into $((\tau_1, \tau_2)_+, int(2))_\times$.

6.2 Code Motion

The central idea of code motion is to obtain *computationally optimal* results by placing computations *as early as possible* in a program [9]. For example, in Fig.7 the computation of a1+b1 for the assignment to z1 is hoisted to the left predecessor block so that the partial redundancy is eliminated (this kind of placement is called *lazy code motion*). In Fig.7 $(\tau_1, \tau_2)_+$, the type of z1 changes into $\{l_1 : (\tau_1, \tau_2)_+, l_2 : (\tau_1, \tau_2)_+\}$ (assuming a1:τ_1 and b1:τ_2). We redefine code

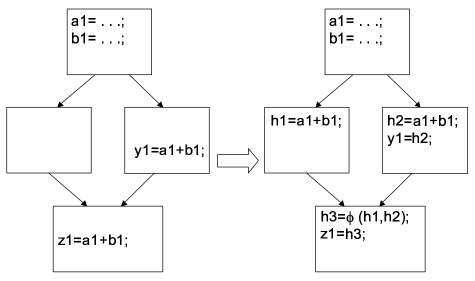


Fig. 7. An Example of Lazy Code Motion

motion as a program transformation which changes types of variables as in Fig.7 (currently we assume that there are not any *critical edges*[9] in source codes). The optimizing order for code motion (denoted as $\succeq_m$) is defined as:

$$(\tau_1, \tau_2)_{\mathrm{aop}} \succeq_m \{l_1 : (\tau_1, \tau_2)_{\mathrm{aop}}, l_2 : (\tau_1, \tau_2)_{\mathrm{aop}}\}.$$

The correctness can be proved in the same way of that of constant folding. This definition would not subsume all code motions which have been proposed so far [9, 8]. However, this definition captures the basic idea of code motion.

6.3 Strength Reduction

Strength reduction is a transformation that a compiler uses to replace costly instructions with cheaper ones [5]. The following program transformation shows an example of strength reduction, in which the assignment to `j2` by multiplication is replaced with the assignment to `j2'` by addition.

```
   i0 = 1;              i0 = 1;
   j0 = 0;              j0 = 0;
                        j0' = 4;
 L:i1 = phi(i0,i2);=> L:i1 = phi(i0,i2);
   j1 = phi(j0,j2);     j1 = phi(j0,j2);
   j2 = i1 * 4;         j1'= phi(j0',j2');
   i2 = i1 + 1;         j2 = j1';
   if(...) goto L;      i2 = i1 + 1;
                        j2'= j1' + 4;
                        if(...) goto L;
```

This transformation can be represented by the following optimizing order (denoted as $\succeq_{sr}$).

$$(\mu\alpha.\{l_1 : int(i), l_2 : (\alpha, int(j))_+\}, int(k))_\times \succeq_{sr} \mu\alpha.\{l_1 : int(i{\cdot}k), l_2 : (\alpha, int(j{\cdot}k))_+\}$$

7 Related Work

There have been many papers for proving correctness of compiler optimizations, which are based on well known frameworks e.g., denotational semantics[3] and temporal logic [10, 11]. Benton [3] proposed a denotational semantics based type system which can express dead code elimination and constant propagation on simple while-language. Its problems are that the type system only tracks constancy and limited dependency (it can not express recursive calculation) and extending its system for unstructured programs might not be easy. Lacey et al[10] showed that temporal logic is sufficient to express data dependence among variables and conditions for applying optimizations, and defined dead code elimination, constant folding, and simple code motion. However, their framework requires different simulation relation between source and optimized codes for each optimization, and finding such relations seems not easy whereas our framework can define optimizations by just order relations on types, and can be proved correct in uniform way.

Typed assembly language (TAL) [12] and proof carrying code (PCC) [13] are well known type systems for low level languages. Extending their system with our recursive types seems worth studying.

Translation validation[14, 17] is a technique for proving source and optimized codes are equivalent. The strategy of translation validation is to check whether two codes are bisimilar on simple operational semantics automatically, irrespectively how the optimization is applied. However, it seems that works on translation validation do not concern proving correctness of compiler optimization.

8 Conclusion

In this paper we have proposed a type-theoretical formalization for proving correctness of compiler optimizations. In Section 2, we have given preliminaries for the paper. In Section 3 we have defined program equivalence. In Section 4 we have introduced our type system. In Section 5 we have proved soundness of our type system. In Section 6, we have defined constant folding, code motion, and strength reduction and stated that they preserve program semantics.

References

1. R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures.* Morgan Kaufmann Publishers, 2001.
2. Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Proc. POPL'88*, 1988.
3. Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In *Proc. POPL'04*, Venice, Italy, 2004.
4. Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. In Roger Hindley, editor, *Proc. 3d Int'l Conf. on Typed Lambda Calculi and Applications (TLCA), Nancy, France, April 2–4, 1997*, volume 1210, pages 63–81. Springer-Verlag, 1997.

5. Keith D. Cooper, L. Taylor Simpson, and Christopher A. Vick. Operator strength reduction. *ACM Transactions on Programming Languages and Systems*, 23(5):603–625, 2001.
6. 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. *ACM Transactions on Programming Languages and Systems*, 13-4:451–490, 1991.
7. S. Horwitz, J.Prins, and T.Reps. On the adequacy of program dependence graphs for representing programs. In *Proc. POPL'88*, 1988.
8. Robert Kennedy, Sun Chan, Shin-Ming Liu, Raymond Lo, Peng Tu, and Fred Chow. Partial redundancy elimination in SSA form. *ACM Transactions on Programming Languages and Systems*, 21(3):627–676, 1999.
9. Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Optimal code motion: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 16(4):1117–1155, July 1994.
10. 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.
11. Sorin Lerner, Todd Millstein, and Craig Chambers. Automatically proving the correctness of compiler optimizations. In *Proc. PLDI'03*, San Diego, California, 2003.
12. Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.
13. George Necula. Proof-carrying code. In *Proc. POPL'97*, pages 106–119, Paris, 1997.
14. George Necula. Translation validation for an optimizing compiler. In *Proc. PLDI'00*, 2000.
15. T. Reps. Algebraic properties of program integration. In *Proc. ESOP'90*, 1990.
16. Hongwei Xi and Robert Harper. Dependently typed assembly language. In *Proc. ICFP'01*, pages 169–180, 2001.
17. Lenore Zuck, Amir Pnueli, Yi Fang, and Benjamin Goldberg. Voc: A translation validator for optimizing compilers. In Jens Knoop and Wolf Zimmermann, editors, *Electronic Notes in theoretical Computer Science*, volume 65. Elsevier, 2002.