A Type System for Optimization Verifying Compilers

Yutaka Matsuno  Hiroyuki Sato

This paper presents a type theoretical framework for the verification of compiler optimizations. Although today’s compiler optimizations are fairly advanced, there is still not an appropriate theoretical framework for the verification of compiler optimizations. To establish a generalized theoretical framework, we introduce assignment types for variables that represent how the value of variables will be calculated in a program. In this paper, first we introduce our type system. Second we prove soundness of our type system. This implies that the given two programs are equivalent if their return values are equal in types. Soundness ensures that many structure preserving optimizations preserve program semantics. Furthermore, by extending the notion of type equality to order relations, we redefine several optimizations and prove that they also preserve program semantics.

Key Words: theoretical framework for compiler optimizations, type system, recursive types, def-use analysis, SSA form.

1 Introduction

Compiler optimizations have become essential parts of high performance computing on modern CPU architectures, on which advanced aggressive optimizations are indispensable to obtain performance. However, this makes the verification of compiler optimizations extremely difficult. Actually, some advanced optimizing compilers have been released without formally proving that they preserve program semantics. Besides practical problems, the most crucial problem is that there is still not an appropriate theoretical framework for the verification of compiler optimizations. To overcome this difficulty, some previous papers exploited several theoretical frameworks such as denotational semantics [2] and temporal logic [10]. However, because such frameworks are originally not introduced for compiler optimizations, they are not necessarily suitable for compiler optimizations. To be worse, such frameworks sometimes require substantial amount of work even to prove the correctness of a simple optimization.

Our major objective is to provide a formal framework for proving correctness of compiler optimizations. For this goal, 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. In fact, conventional compilers mostly 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) [5] has greatly simplified def-use analysis. Because of this, SSA form has become a standard intermediate form.

In this paper, we propose a type theoretical framework for compiler optimizations based on the notion of def-use analysis in SSA form. Moreover, we apply it to verification of compiler optimizations. The information of def-use analysis is rep-
L0: x=0;
L1: x=x+1;
if(x<10) then goto L1;
[
\begin{align*}
L0: & x=0; \\
L1: & x=x+1;
\end{align*}
]

Fig. 1 Source program $P$ and its SSA form. 

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 given types $\tau_1$ and $\tau_2$ respectively, then $x$ is given a type $(\tau_1, \tau_2)_+$. Intuitively, this means that a value is assigned to $x$ by adding two variables of types $\tau_1$ and $\tau_2$. 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 $x_1$ and $x_2$ be $\alpha$ and $\beta$, respectively. A variable whose value is assigned by a $\phi$ 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 $\tau_1$ and $\tau_2$ come. $\text{int}(0)$ and $\text{int}(1)$ are singleton types [15] of integer whose values are 0 and 1, respectively. Type equations for $\alpha$ and $\beta$ are:

\[
\alpha = \{L0 : \text{int}(0), L1 : \beta\} \quad (1)
\]

\[
\beta = (\alpha, \text{int}(1))_+ \quad (2)
\]

By substituting the right hand side of (2) for $\beta$ in (1), we obtain $\alpha = \{L0 : \text{int}(0), L1 : (\alpha, \text{int}(1))_+\}$. In other words, $x_1$ has a recursive type of the form $\mu\alpha.\{L0 : \text{int}(0), L1 : (\alpha, \text{int}(1))_+\}$.

Our contributions are summarized 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 relations, we redefine several optimizations, and prove that they also preserve program semantics. Furthermore, by applying our methodology to strength reduction [4] and induction variable analysis [6], we show that our framework is an appropriate framework for compiler optimization theories.

Specifically, we prove the correctness of the following structure preserving optimizations:

- Dead code elimination, constant folding, constant propagation, loop invariant hoisting, common subexpression elimination, value numbering, code motion (partial redundancy elimination), strength reduction, instruction reordering optimizations, optimizations using arithmetic laws, and their combinations. Conventionally, correctness proofs of above optimizations have been done only in informal ways. Formal proof of them needs complicated setting, and therefore is never trivial, as shown in [10][11][2]. Optimizations which can not be covered in our current framework are loop optimizations such as loop unrolling, loop tiling, loop fusion, etc. Note that formally proving the correctness of these loop optimizations is even much harder. However, because our framework is based on dependence, the most fundamental notion of compiler optimizations, we believe that our framework can be extended for these loop optimizations.

Our goal is to establish an Optimization Verifying Compiler, a compiler that automatically verifies its optimization process, guaranteeing correctness of semantics and of performance improvement. Our type system will be the theoretical framework for guaranteeing correctness of semantics.

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. Furthermore, we show that many structure preserving optimizations can be covered in our framework. Then in Section 7 we discuss related work and Section 8 gives a summary and future work.

2 Preliminaries

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

**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...
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 \( l \in \text{Label} \). Furthermore, Label includes \( l_{\text{entry}} \) and \( l_{\text{exit}} \) 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 \( \text{HALT} \) 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.

- Initial machine state: \( M_0 = (0,V_0,(l_{\text{entry}},l_0)) \) where \( V_0 \) is the map of variables to the initial values. \( l_0 \) is the label of the first basic block to be executed.
- Program executions are denoted as \( P \models (pc,V,(l_p,l_c)) \rightarrow (pc',V',(l'_p,l'_c)) \), 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 \( \text{HALT} \) state.

Fig. 3 contains a one-step operational semantics for instructions where \( \texttt{aop} \) is either \texttt{add}, \texttt{sub} or \texttt{mul} and \( \texttt{aop} \) is either +, −, or ×, respectively. \( M = (pc,V,(l_p,l_c)) \) is also defined as a map: if \( v \in \text{Dom}(V) \) then \( M[v] = V[v] \), else \( M[v] = i \) when \( v \) is \( i \in I \); \( L \) is defined as a map of 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

\(^1\) Let \( g \) be a map, \( \text{Dom}(g) \) be the domain of \( g \); for \( x \in \text{Dom}(g) \), \( g[x] \) is the value of \( x \) at \( g \), and \( g[x \rightarrow v] \) is the map with the same domain as \( g \) defined by the following equation: for all \( y \in \text{Dom}(g) \):

\[ g[x \rightarrow v][y] = \begin{cases} \text{if } y \neq x & \text{then } g[y] \text{ else } v. \end{cases} \]
\[
M = (pc, V, (l_{p}, l_c)) \quad P[pc] = aop \ x, y, v \quad V[y] = i \quad M[v] = j \\
\quad P \vdash M \rightarrow (pc + 1, V[x \mapsto (i \ aop \ j)], NL(pc, (l_{p}, l_c)))
\]

\[
M = (pc, V, (l_{p}, l_c)) \quad P[pc] = mov \ x, v \quad M[v] = i \\
\quad P \vdash M \rightarrow (pc + 1, V[x \mapsto i], NL(pc, (l_{p}, l_c)))
\]

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

\[
M = (pc, V, (l_{p}, l_c)) \quad P[pc] = jmp \ \\
\quad P \vdash M \rightarrow (L[l], V, (l_c, l_i))
\]

\[
M = (pc, V, (l_{p}, l_c)) \quad P[pc] = bop \ x, v, l \quad V[x] = i \quad M[v] = j \quad i \ bop \ j = true \\
\quad P \vdash M \rightarrow (pc + 1, V, (l_c, l'))
\]

\[
M = (pc, V, (l_{p}, l_c)) \quad P[pc] = bop \ x, v, l \quad \exists ! l. L[l'] = pc + 1 \quad V[x] = i \quad M[v] = j \quad i \ bop \ j = false \\
\quad P \vdash M \rightarrow (pc + 1, V, (l_c, l'))
\]

\[
M = (pc, V, (l_{p}, l_c)) \quad P[pc] = return \ x \\
\quad P \vdash M \rightarrow HALT
\]

Fig. 3 Dynamic Semantics

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.

\[
NL(pc, (l_{p}, l_c)) = \begin{cases} (l_{c}, l) & \exists ! L[l] = pc + 1 \\ (l_{p}, l_{c}) & \text{otherwise} \end{cases}
\]

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 original and optimized codes may take, and verify that each path in the original code corresponds to a path in the optimized code and vice versa.

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

\[
P \vdash M_0 \rightarrow M_1 \rightarrow M_2 \rightarrow \cdots
\]

where \(P \vdash M_i \rightarrow M_{i+1}\) 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 return x, then OV(P) is x.

Then we define program equivalence 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.

\[
P \equiv Q
\]

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

\[
P \vdash M_0 \rightarrow M_1 \rightarrow M_2 \rightarrow \cdots M_i \ \text{return} \ x \ \text{HALT}
\]

there exists a computation prefix of Q:

\[
Q \vdash M_0 \rightarrow M'_1 \rightarrow M'_2 \rightarrow \cdots M'_j \ \text{return} \ x \ \text{HALT}
\]

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 \(x, v, l\) instruction where \(x \in \text{Var}, v \in \text{Var}\) if \(v\) is a
variable, and $l \in \text{Label}$.

- Both $B_i$ and $B'_i$ end with the same jmp $l$ instruction where $l \in \text{Label}$.
- Both $B_i$ and $B'_i$ end with the same return $x$ instruction where $x \in \text{Var}$.

Note that this definition excludes some simple optimizations, for example replacing $\text{bop} \ x, v, l$ with $\text{bop} \ x, 3, l$ when $v$ is ensured to be 3 in all program executions. Extending Definition 5 to such optimizations or simple branch optimizations is easy. However, for simplicity we leave the definition of structural equivalence as Definition 5.

As shown in Definition 5, variables used in branch instructions play an important role. We define such variables as control variables.

**Definition 6** The control variable set for a program $P$, $\text{CV}(P)$ 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 shown in Fig.4. Type variables are represented by $\alpha, \beta, \ldots$. $\top$ is the super type of all other types. $\text{int}(i)$ is a singleton type of a variable or an integer that has value $i$. We sometimes use a notation $\text{int}(x)$ for $\text{int}(i)$ when the value of $x$ is $i$. $(\tau_1, \tau_2)_+$, $(\tau_1, \tau_2)_-$, and $(\tau_1, \tau_2)_x$ 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. We require that the exchange rule for type environments, meaning 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 $\text{Dom}(\Gamma)$ for the set of variables whose types are declared in $\Gamma$.

### 4.2 Type System

The type system is shown in Fig.5. We explain the meaning of judgments:

- $\vdash \Gamma$

  This judgment says that $\Gamma$ is a valid type environment. Specially, we define $\Gamma_0$ as the type environment for initial values, $x_0, y_0, \ldots$. They are given types $\text{int}(i), \text{int}(j), \ldots$ in $\Gamma_0$ if their initial values are $i, j, \ldots$ respectively. Moreover, $\vdash \Gamma_0$ is given as an axiom.

- $\Gamma \vdash_P v : \tau$

  This judgment says that $v$ is given a type $\tau$ in a program $P$ under a type environment $\Gamma$.

  $\Gamma \vdash_P v : \tau$ 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 $\Gamma$.
  - Rule (type-aop) is applied when the value is assigned by an arithmetic operation in a program $P$. ($(\text{type-mov})$ and (type-ssa) are similar to (type-aop).)
  - Rule (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 $\alpha$, 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 $\mu$ types to ssa types.)

  - The cut rule allows substituting a valid type of a variable $x$ for an occurrence of a type variable $\alpha$ in a valid type judgment where $x$ is given $\alpha$ 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 $\Gamma_0$ augmented with some type declarations in which variables are given type variables are valid (e.g. $\Gamma_0, x : \alpha, y : \beta, \ldots$), $\top$ never appears in valid type derivation trees. As an example of typing, the type derivation for $x_1$ in Fig.1 is shown in Fig.6 (we assume that $\Gamma_0 = x_0 : \text{int}(0)$).

### 4.3 Cut Elimination Theorem

In our type system, cut elimination theorem holds. The (cut) rule corresponds to the substitu-
type variables \( \alpha, \beta, \gamma, \ldots \)

types \( \tau ::= \alpha | \top | \text{int}(i) | (\tau_1, \tau_2)_+ | (\tau_1, \tau_2)_- | (\tau_1, \tau_2)_\times \)

\( \{l_1 : \tau_1, l_2 : \tau_2\} \) \( \mu \alpha. \{l_1 : \tau_1, l_2 : \tau_2\} \)

\( \Gamma ::= \cdot | x : \tau | \Gamma_1, \Gamma_2 \)

Fig. 4 Types

\[
\frac{}{\Gamma \vdash} \\
\frac{}{\Gamma_0 \vdash} \\
\frac{\Gamma \vdash} {\Gamma \vdash x \notin \text{Dom}(\Gamma)} \quad (\text{fresh in } \Gamma) \\
\frac{\Gamma \vdash x : \alpha} {\Gamma \vdash p \ v : \tau} \quad (\text{type-var}) \\
\frac{\Gamma \vdash p \ i : \text{int}(i)} {\Gamma \vdash p \ x : \tau} \quad (\text{type-int}) \\
\frac{\Gamma \vdash p \ x : \tau} {\Gamma \vdash p \ x : \tau} \\
\frac{\Gamma \vdash p \ x : \tau} {\Gamma \vdash p \ x : \tau} \quad (\text{type-mov}) \\
\frac{\Gamma \vdash p \ x : \tau} {\Gamma \vdash p \ x : \tau} \quad (\text{type-ssa}) \\
\frac{\Gamma, x : \alpha, \Gamma' \vdash p \ x : \{l_1 : \tau_1, l_2 : \tau_2\}} {\Gamma, \Gamma' \vdash p \ x : \{l_1 : \tau_1, l_2 : \tau_2\}} \quad (\text{type-mu}) \\
\frac{\Gamma \vdash p \ x : \tau_1 \quad \Gamma', x : \alpha, \Gamma'' \vdash p \ y : \tau_2} {\Gamma \vdash (\Gamma', \Gamma'') \vdash p \ y : \tau_2[\tau_1/\alpha]} \quad (\text{cut})
\]

Fig. 5 Typing Rules

\[
\frac{\Gamma_1 \vdash \alpha \quad \Gamma_2 \vdash \beta} {\Gamma_1 \oplus \Gamma_2 \vdash \alpha \times \beta} \quad (\text{type-add}) \\
\frac{\Gamma_0 \vdash x_1 : \alpha \quad \Gamma_0 \vdash x_2 : \alpha} {\Gamma_0 \vdash \alpha \vdash p \ x_1 : \alpha, 1 : \text{int}(1)} \quad (\text{D1}) \\
\frac{\Gamma_0 \vdash x : \alpha \quad \Gamma_0 \vdash x_2 : \alpha} {\Gamma_0 \vdash x : \alpha, \Gamma' \vdash p \ x_2 : (\alpha, \text{int}(1))_\times} \quad (\text{D2}) \\
\frac{\Gamma_0 \vdash x : \alpha \quad \Gamma_0 \vdash x_2 : \alpha} {\Gamma_0 \vdash x : \alpha \quad \Gamma_0 \vdash x_2 : \alpha} \quad (\text{cut}) \\
\frac{\Gamma_0 \vdash x_1 : \alpha \quad \Gamma_0 \vdash x : \{L_0 : \text{int}(0), L_1 : \beta\}} {\Gamma_0 \vdash x_1 : \{L_0 : \text{int}(0), L_1 : \beta\}} \quad (\text{type-mu})
\]

Fig. 6 Type derivation of \( x_1 \) in Fig.1
rules.

Cut elimination theorem is a corollary to this theorem.  

Proof. Without loss of generality we can assume that \( (\text{cut}) \) is used in the last derivation step for \( \Gamma \vdash \alpha \vdash x : \tau \).  
We prove Theorem 2 by induction on the structure of \( \tau \) and the length of derivation trees.

**Base Cases**

The minimum structure of types are \( \text{int}(i) \) and a type variable.  
We prove each cases that the least length of derivation using \( (\text{cut}) \) can be transformed to \( (\text{free}) \) free derivations.

**Case 1** \( \tau \equiv \text{int}(i) \).

Possible type derivations of the least length are shown in Fig. 7.  
Case (1) is that the cut type declaration \( (y : \alpha) \) is distinct from \( x : \text{int}(i) \), i.e., \( x \neq y \).  
Case (2) is the case of \( x = y \).  
In (1), the premise that \( x : \text{int}(i) \in \Gamma_0 \), \( y : \alpha \) implies \( x : \text{int}(i) \in \Gamma_0 \).  
Furthermore, \( \vdash \Gamma_0 \) holds by axiom.  
Therefore
\[
\begin{align*}
\Gamma_0 \vdash & x : \text{int}(i) \\
& \vdash \Gamma_0 \vdash \alpha \\
\end{align*}
\]
are derivable without \( (\text{cut}) \).  
In the second case, because \( x \) is given a type variable \( \alpha \) under a type environment where \( y \) is given \( \alpha \),  
there must be a \( \text{mov} \) instruction which assigns the value of \( y \) to \( x \), i.e., \( \text{mov} x, y \in P \).  
Because \( \Gamma_0 \vdash \alpha \vdash y : \alpha \) can be derived without \( (\text{cut}) \),  
we have the following \( (\text{cut}) \)-free type derivation:
\[
\begin{align*}
\vdash & \Gamma_0 \vdash \alpha \\
& \vdash \Gamma_0, x : \alpha \vdash x : \alpha \\
& \vdash \Gamma_0, y : \alpha \vdash x : \alpha \\
\end{align*}
\]

**Induction Step**

**Case 1** \( \tau \equiv \text{int}(i) \).

There are two cases:
\[
\begin{align*}
\Gamma_1 \vdash & \alpha \vdash \tau' \\
\Gamma_1, y : \alpha \vdash & \Gamma_2 \vdash \alpha \vdash \Gamma_3 \vdash x : \text{int}(i) \\
\end{align*}
\]
are derivable without \( (\text{cut}) \).  
In the first case, we have a derivation \( (\ast) \) of a judgment \( \Gamma_2 \vdash \alpha \vdash \Gamma_3 \vdash x : \text{int}(i) \).  
By induction hypothesis on length of derivation tree, \( \Gamma_0 \vdash x : \text{int}(i) \) is derivable without \( (\text{cut}) \).  
In the second case we have a derivation \( (\ast\ast) \) of a judgment \( \Gamma_1 \vdash x : \text{int}(i) \).  
Also by induction hypothesis on length of derivation tree, \( \Gamma_0 \vdash x : \text{int}(i) \) is derivable without \( (\text{cut}) \).  

**Case 2** \( \tau \equiv (\tau_1, \tau_2)_+ \)  
(the proofs for other arithmetic types and ssa types are similar).

When \( (\text{cut}) \) rule is used, the type derivation is of
the form:

\[ \frac{\Gamma_1 \vdash y : \tau_3}{\Gamma \vdash x : (\tau_1, \tau_2)_+} \quad \frac{\Gamma_2, y : \alpha, \Gamma_3 \vdash p : \tau''}{\Gamma \vdash x : (\tau_1, \tau_2)_+} \]  

(cut),

where \( \tau''[\tau_3/\alpha] \equiv (\tau_1, \tau_2)_+ \) and \( \Gamma \equiv \Gamma_1 \cup (\Gamma_2, \Gamma_3) \).

Sub Case 1 \( (\tau'' \equiv \alpha) \).

Then the derivation is of the form:

\[ \frac{\vdots}{\Gamma \vdash x : (\tau_1, \tau_2)_+} \quad \frac{\Gamma_1 \vdash y : (\tau_1, \tau_2)_+}{\Gamma \vdash x : (\tau_1, \tau_2)_+} \quad \frac{\Gamma_2, y : \alpha, \Gamma_3 \vdash x : \alpha}{\Gamma \vdash x : (\tau_1, \tau_2)_+} \]  

(cut).

Because \( x \) is given a type \( \alpha \) under a type environment where \( y \) is given \( \alpha \), there must be a \texttt{mov} instruction which assigns the value of \( y \) to \( x \), i.e., \texttt{mov} \( x, y \in P \).

Also by induction hypothesis the derivation \((***)\) of \( \Gamma_1 \vdash y : (\tau_1, \tau_2)_+ \) can be transformed to a cut-free derivation of \( \Gamma' \vdash y : (\tau_1, \tau_2)_+ \) where \( FV(\Gamma') = FV((\tau_1, \tau_2)_+) \). Therefore we have

\[ \texttt{mov} : y \in P \quad \Gamma' \vdash y : (\tau_1, \tau_2)_+ \]  

\[ \Gamma' \vdash x : (\tau_1, \tau_2)_+ \]  

without (cut).

Sub Case 2 \( (\alpha \text{ does not appear freely in } \tau'', \text{i.e., } \tau'' \equiv (\tau_1, \tau_2)_+) \).

Similar to Case 1.

Otherwise

We denote \( \tau'' \) as \( (\tau_1', \tau_2')_+ \). By induction hypothesis on the length of derivation we have cut free derivations of the premises

\[ \Gamma_4 \vdash y : \tau_3 \quad \Gamma_5 \vdash x : (\tau_1', \tau_2')_+ \]  

where \( FV(\Gamma_4) = FV(\tau_3) \), \( FV(\Gamma_5) = FV((\tau_1', \tau_2')_+) \), and \( y : \alpha \in \Gamma_5 \). To derive \( \Gamma_5 \vdash x : (\tau_1', \tau_2')_+ \) without (cut), (type-add) must be used in the following way:

\[ \text{add} : x, z, v \in P \quad \Gamma_6 \vdash p : \tau' \quad \Gamma_7 \vdash v : \tau' \]  

\[ \Gamma_5 \vdash x : (\tau_1', \tau_2')_+ \]  

where \( \Gamma_5 = \Gamma_6 \cup \Gamma_7 \). We apply (cut) with \( \Gamma_4 \vdash y : \tau_3 \) and obtain

\[ \Gamma_4 \oplus \Gamma_6 \vdash p : \tau_1 \quad \Gamma_4 \oplus \Gamma_7 \vdash v : \tau_2 \]  

Where \( \text{dom}(\Gamma_6) - \text{dom}(\Gamma_4) = \{y\} \) and \( \text{dom}(\Gamma_7) - \text{dom}(\Gamma_4) = \{y\} \). By induction hypothesis on the structure of types, we obtain

\[ \Gamma_8 \vdash p : \tau_1 \quad \Gamma_9 \vdash v : \tau_2 \]  

without (cut) where \( FV(\Gamma_8) = FV(\tau_1) \) and \( FV(\Gamma_9) = FV(\tau_2) \). By using (type-aop), we obtain the required result.

Case 3 \( (\tau \equiv \mu \alpha.\{l_1 : \tau_1, l_2 : \tau_2\}) \).

4.4 Type Equality

In our type system, a recursive type takes several forms for the same variable. For example, in Fig.8, two type derivations are possible for \( x_1 \), as shown in Fig.9. Though they look different, they are equal by the following definition. 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 \( T(\tau) \)) Given a type \( \tau \), we define its regular tree \( T(\tau) \) in Fig.10. Nodes are of the form \((+, -, \times), (\text{ssa})\), or leaves of the form \((\text{int}(i))\). \( T(\tau_1) = T(\tau_2) \) means that \( T(\tau_1) \) and \( T(\tau_2) \) have the same structure as trees.

Type equality is defined by the structure of regular trees [3].

Definition 9 (Type Equality) We define \( \tau_1 = \tau_2 \) if \( T(\tau_1) = T(\tau_2) \).

Note that this equality is obviously decidable. Fig.11 shows the same tree structure of both 1 and 2 of Fig.9.

Theorem 3 Given a program \( P \) and a variable \( x \) in \( P \), if \( \Gamma_0 \vdash p : \tau \) and \( \Gamma_0 \vdash x : \tau' \) are derivable, then \( \tau = \tau' \).

Because of cut elimination theorem, we have only to consider cut free derivations. For types in which type variables appear, we check whether there exist a substitution \( \theta = \{\alpha \mapsto \tau_1, \beta \mapsto \tau_2, \ldots\} \) such that \( \theta(\tau) = \theta(\tau') \). We prove the following theorem, which is a generalization of Theorem 3.

Theorem 4 Given a program \( P \) and a variable \( x \) in \( P \), if \( \Gamma \vdash x : \tau \) and \( \Gamma' \vdash x : \tau' \) are derivable, then there exists a substitution \( \theta = \{\alpha \mapsto \tau_1, \beta \mapsto \tau_2, \ldots\} \).
\[ \Gamma_0, x_1 : \alpha \vdash Q \ x_2 : (\alpha, \text{int}(4)) \quad \Gamma_0, x_1 : \alpha, x_2 : \gamma \vdash Q \ x_1 : \{L_0 : \text{int}(0), L_4 : \{L_2 : (\alpha, \text{int}(2))_+, L_3 : \gamma\}\} \] (cut)
\[ \Gamma_0, x_1 : \alpha \vdash Q \ x_1 : \{L_0 : \text{int}(0), L_4 : \{L_2 : (\alpha, \text{int}(2))_+, L_3 : (\alpha, \text{int}(4))_+\}\} \] (type-mu)
\[ \Gamma_0 \vdash Q \ x_1 : \mu\alpha.\{L_0 : \text{int}(0), L_4 : \{L_2 : (\{L_0 : \text{int}(0), L_4 : \alpha, \text{int}(4)\}_+, L_3 : (\{L_0 : \text{int}(0), L_4 : \alpha, \text{int}(4)\}_+)\}\} \] (type-mu)

**Fig. 9** Type Derivations for \( x_1 \) in Fig. 8

\[ T(\text{int}(i)) = (\text{int}(i)) \]
\[ T((\tau_1, \tau_2)_{aop}) = (\text{aop}) \]
\[ T(\tau_1) \quad T(\tau_2) \]
\[ T(\{l_1 : \tau_1, l_2 : \tau_2\}) = (\text{ssa}) \]
\[ l_1 : T(\tau_1) \quad l_2 : T(\tau_2) \]
\[ T(\mu\alpha.\{l_1 : \tau_1, l_2 : \tau_2\}) = T(\{l_1 : \tau_1, l_2 : \tau_2\}[\mu\alpha.\{l_1 : \tau_1, l_2 : \tau_2\}/\alpha]) \]

**Fig. 10** Regular Trees

\[ \tau_2, \ldots \] such that \( \theta(\tau) = \theta(\tau') \).

**Proof**

We prove Theorem 4 by induction on the sum of length of derivations for \( \Gamma \vdash P x : \tau \) and \( \Gamma' \vdash P x : \tau' \).

**Base Case**

The shortest derivations for \( \Gamma \vdash P x : \tau \) is the case when \( x \) is an initial value. In this case the only possible derivation is:

\[ \vdash \Gamma_0 \quad x : \text{int}(i) \in \Gamma_0 \]
\[ \Gamma_0 \vdash P x : \text{int}(i) \]

Clearly the required result holds.

**Induction Step**

**Case 1** (\( \tau \equiv \alpha \)).

Then there exists a substitution \( \theta = \{\alpha \mapsto \tau'\} \).
\[
\begin{align*}
\text{Case 2} & \quad (\tau \equiv \text{int}(i)). \\
& \text{The possible last step of the derivation is by (type-var) or (type-mov). Therefore, possible types are int}(i) \text{ and } \alpha. \text{ If } \tau' = \alpha \text{ there exists a substitution } \theta = \{\alpha \mapsto \text{int}(i)\}. \\
\text{Case 3} & \quad (\tau \equiv (\tau_1, \tau_2)_{aop}). \\
& \text{In this case the last applied rule is either (type-aop) or (type-mov). If (type-mov) is used, the last step is as follows.}
\end{align*}
\]

\[
\begin{align*}
\text{mov } x, v & \quad \Gamma \vdash \tau \vdash (\tau_1, \tau_2)_{aop} \\
\Gamma \vdash \tau \vdash x : (\tau_1, \tau_2)_{aop} \\
\end{align*}
\]

There may be successive uses of (type-mov), but a (type-aop) must be used because only (type-aop) can give a variable a type of the form \((\tau_1, \tau_2)_{aop}\). Since (type-mov) does not change types, we do not have to consider (type-mov).

Consider the other case where (type-aop) is used:

\[
\begin{align*}
\text{aop } x, y, v & \quad \Gamma_1 \vdash \tau \vdash \tau_1 \quad \Gamma_2 \vdash \tau \vdash \tau_2 \\
\Gamma \vdash \tau \vdash x : (\tau_1, \tau_2)_{aop} \\
\end{align*}
\]

where \(\Gamma_1 \oplus \Gamma_2 = \Gamma\).

\textbf{Sub Case 1} \quad (\tau' \equiv \alpha).

We have a substitution \(\theta = \{\alpha \mapsto (\tau_1, \tau_2)_{aop}\}\).

\textbf{Otherwise}

We consider the case that the last derivation for \(\Gamma' \vdash \tau \vdash \tau'\) is (type-aop).

\[
\begin{align*}
\text{aop } x, y, v & \quad \in P \quad \Gamma' \vdash \tau \vdash \tau_1' \quad \Gamma' \vdash \tau \vdash \tau_2' \\
\Gamma' \vdash \tau \vdash x : (\tau_1', \tau_2'_{aop}(= \tau')) \\
\end{align*}
\]

where \(\Gamma' = \Gamma_1 \oplus \Gamma_2\). By induction hypothesis, there exist two substitutions \(\theta_1\) and \(\theta_2\) such that \(\theta_1(\tau_1) = \theta_1(\tau_1') \) and \(\theta_2(\tau_2) = \theta_2(\tau_2')\). Consider the case that \(\theta_1\) maps a type variable \(\beta\) to a type \(\tau_3\). This implies that there is a judgment \(\Gamma_3 \vdash \tau \vdash \tau_3\) in either the derivations of \(\Gamma_1 \vdash \tau \vdash \tau_1\) or \(\Gamma_1 \vdash \tau \vdash \tau_1'\), and that \(\Gamma_4 \vdash \tau \vdash \beta \vdash \beta\) by the other derivation. Because the definition of a variable is unique in SSA form, all type derivation trees for the type judgment always have the same structure, except the use of (type-mu). Therefore, \(\beta\) and \(\tau_3\) are given to the same variable. If \(\theta_2\) maps \(\beta\) to a type \(\tau_4\), then there is a type derivation \(\Gamma_5 \vdash \tau \vdash \tau_4\) in either type derivation for \(\Gamma_2 \vdash \tau \vdash \tau_2\) or \(\Gamma_2' \vdash \tau \vdash \tau_2'\). By induction hypothesis, there exists a substitution \(\theta'\) such that \(\theta'(\tau_3) = \theta'(\tau_4)\). Because \(\theta' \circ \theta_1(\beta) = \theta'(\tau_3)\) and \(\theta' \circ \theta_2(\beta) = \theta'(\tau_4)\), we can conclude that there exists a substitution \(\theta_3 \equiv \theta' \circ \theta_1 = \theta' \circ \theta_2\) which satisfies \(\theta_3((\tau_1, \tau_2)_{aop}) = \theta_3((\tau_1', \tau_2')_{aop})\).

\textbf{Case 4} \quad (\tau \equiv (\tau_1, \tau_2)_{aop}).

The last type derivation is (again we do not have to consider the case when (type-mov) is used) as shown in (1) of Fig.12.

\textbf{Sub Case 1} \quad (\tau' \equiv \alpha).

We have a substitution \(\theta = \{\alpha \mapsto (\tau_1, \tau_2)_{aop}\}\).

\textbf{Sub Case 2} \quad (\tau' \equiv (\tau_1', \tau_2'_{aop})\).

The proof is the same as the proof for \textbf{Case 3}.

\textbf{Sub Case 3} \quad (\tau' \equiv \text{recursive type}).

The type derivation is as (2) of Fig.12. By induction hypothesis, there exist substitutions \(\theta_1\) and \(\theta_2\) such that \(\theta_1(\tau_1) = \theta_1(\tau_1')\) and \(\theta_2(\tau_2) = \theta_2(\tau_2')\).

As in \textbf{Case 3}, there exists a substitution \(\theta_3\) which agrees with \(\theta_1\) and \(\theta_2\) such that \(\theta_3\{(\tau_1, \tau_2)_{aop}\} = \theta_3\{(\tau_1', \tau_2'_{aop})\}\). We consider a substitution \(\theta_3 = \{\alpha \mapsto (\tau_1, \tau_2)_{aop}\}\). If \(\theta_3(\alpha) = \tau''\), then there exists a type derivation \(\Gamma'' \vdash \tau'' \vdash \tau''\) in the type derivation for \(\Gamma \vdash \tau \vdash (\tau_1, \tau_2)_{aop}\).

By induction hypothesis, \(\tau'' = \{\tau_1, \tau_2\}\). Therefore \(\theta_4\) agrees with \(\theta_3\). \(\mu\alpha\{\tau_1, \tau_2\}\) is derived by applying \(\theta_4\) to \(\{\tau_1, \tau_2\}\). Assume that there exist subtrees of \(\{\tau_1, \tau_2\}\) which can not be matched by \(\theta_4\). However, by induction hypothesis, such subtrees can not exist because the length of the derivations of \(\tau_1, \tau_1'\) and \(\tau_2, \tau_2'\) are less than that of \(\{\tau_1, \tau_2\}\) and \(\{\tau_1', \tau_2'\}\). Therefore we have the required result.

\textbf{Case 5} \quad (\tau \equiv \mu\alpha\{\tau_1, \tau_2\}_{aop}).

The same as of \textbf{Case 4}. \(\Box\)
5 Soundness

The soundness of our type system says that if types of the observable variable and control variables of source and optimized codes are equal, the optimization preserves program semantics.

Theorem 5 (Soundness) If \( P \cong Q \), \( OV(P) = OV(Q) = x \) and
- \( \exists \tau \exists \tau' \).
- \( (\Gamma_0 \vdash p \ x : \tau \land \Gamma_0 \vdash q \ x : \tau' \land \tau = \tau') \), and
- \( \forall y_j \in CV(P) \exists \tau_j \exists \tau'_j \).
- \( (\Gamma_0 \vdash p \ y_j : \tau_j \land \Gamma_0 \vdash q \ y_j : \tau'_j \land \tau_j = \tau'_j), \)
then \( P \cong Q \).

To prove soundness, we first prove the following lemmas.

Lemma 1 Assume that \( P \cong Q \) and
- \( \forall y_j \in CV(P) \exists \tau_j \exists \tau'_j \).
- \( (\Gamma_0 \vdash p \ y_j : \tau_j \land \Gamma_0 \vdash q \ y_j : \tau'_j \land \tau_j = \tau'_j). \)

Then for any computation prefix of \( P \):
\[
P \vdash M_0 \rightarrow \ldots \rightarrow \text{bins}_1 \rightarrow \ldots \rightarrow \text{bins}_n \rightarrow \ldots,
\]
where \( \text{bins}_1, \ldots, \text{bins}_n \) are branch instructions, there exists a computation prefix of \( Q \):
\[
Q \vdash M_0 \rightarrow \ldots \rightarrow \text{bins}_1 \rightarrow \ldots \rightarrow \text{bins}_n \rightarrow \ldots
\]
and conversely.

Lemma 2 Assume that \( P \cong Q \) and
- \( \forall y_j \in CV(P) \exists \tau_j \exists \tau'_j \).
- \( (\Gamma_0 \vdash p \ y_j : \tau_j \land \Gamma_0 \vdash q \ y_j : \tau'_j \land \tau_j = \tau'_j). \)

Moreover, assume that for any computation prefix of \( P \):
\[
P \vdash M_0 \rightarrow M_1 \ldots \rightarrow \text{assignment to } x \rightarrow M
\]
there exists a computation prefix of \( Q \):
\[
Q \vdash M_0 \rightarrow M'_1 \ldots \rightarrow \text{assignment to } x \rightarrow M',
\]
and conversely. If
- \( \exists \exists \exists \tau'_j, (\Gamma_0 \vdash p \ x : \tau \land \Gamma_0 \vdash q \ x : \tau' \land \tau = \tau') \),
then \( M[x] = M'[x] \).

We simultaneously prove Lemma 1 and Lemma 2.

Proof

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

Base Case

There are two cases of minimum length of computation prefix to the assignment to \( x \) (cases 1 and 2) and three cases of the least number of branches (cases 3, 4, and 5). Assume that the computation prefixes listed below are taken in \( P \).
1. \( P \vdash M_0^{\text{add}} \ y_0, \nu_0 \mapsto M_1 \)
2. \( P \vdash M_0^{\text{mov}} \ x, v \mapsto M_1 \)
3. \( P \vdash M_0^{\text{jmp}} l \mapsto M_1 \)
4. \( P \vdash M_0^{\text{beq}} \ y_0, \nu_0, l \mapsto M_1 \)
5. \( P \vdash M_0 \ \text{return} \ x_0 \mapsto \text{HALT} \)

In case 1, the type derivation for \( x \) is \( \Gamma_0 \vdash \text{int}(y_0), \text{int}(\nu_0)) \). 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 \( \nu_0 \) 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.

Induction Step

First we prove Lemma 1. Assume that for any computation prefix of \( P \):
\[
P \vdash M_0 \rightarrow \ldots \rightarrow \text{bins}_1 \rightarrow \ldots \rightarrow \text{bins}_n \rightarrow \ldots
\]
there is a computation prefix of \( Q \):
\[
Q \vdash M_0 \rightarrow \ldots \rightarrow \text{bins}_1 \rightarrow \ldots \rightarrow \text{bins}_n \rightarrow \ldots
\]
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. Otherwise \( z \) 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. Next, we prove Lemma 2. The case when \( \tau = \text{int}(i) \) is easy. We consider the other cases.

Case 1 \( (\tau \equiv (\tau_1, \tau_2)_+) \) (similar for other arithmetic types).

In this case a value is assigned to \( x \) by either an \text{add} instruction or a \text{mov} instruction. There are two cases for the last typing rule, (\text{type-mov}) and (\text{type-add}) rules. When (\text{type-mov}) is used, the type derivation is as follows.

\[
\begin{align*}
\text{mov} \ x, v \in P & \quad \Gamma_0 \vdash \text{add} \ x : (\tau_1, \tau_2)_+ \\
\end{align*}
\]

In this case the value of \( x \) depends on \( v \). Therefore we proceed to the proof for \( v \). When a value is assigned to \( x \) by \text{add} instructions in both \( P \) and \( Q \), the type derivation of \( x \) in \( P \) is:
add \(x, y, v \in P\) \(\Gamma_0 \vdash p \ y : \tau_1, v : \tau_2\) \(\Gamma_0 \vdash p \ x : (\tau_1, \tau_2)_+\) (type-add),
and the same as in \(Q\). In this case, though the variable used in \(\text{add}\) instruction is not necessarily the same, the types are equal. 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 \(x\). It is also the case that the type of \(y\) is the same in \(P\) and \(Q\), and that the value is assigned in a 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 \(x\). Next, we consider the case that \(v\) is a variable \(z\) (it is easy when \(v\) is an integer). By the same argument, the value of \(z\) is the same as that before the assignment to \(x\). Therefore the value of \(x\) is the same in \(P\) and \(Q\). Hence \(M[x] = M'[x]\).

**Case 2** \(\tau \equiv \{l_1 : \tau_1, l_2 : \tau_2\}\).

The type derivation for \(x\) in \(P\) (the same as in \(Q\)) is as follows:

\[
\text{ssa } x, (l_1 : x_1, l_2 : x_2) \in P \quad \Gamma_0 \vdash p \ x : (l_1 : \tau_1, l_2 : \tau_2)
\]

\(\text{(type-ssa)}\)

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

\(P \vdash M_0 \rightarrow \ldots \rightarrow \text{bins}_1 \ldots \text{bins}_k \ldots \text{ssa } x, (l_1 : x_1, l_2 : x_2) \ M,\)

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

\(Q \vdash M_0 \rightarrow \ldots \rightarrow \text{bins}_1 \ldots \text{bins}_k \ldots \text{ssa } x, (l_1 : x_1, l_2 : x_2) \ M'\)

and conversely. Because both \(CP\) and \(CP'\) pass the same sequence of branch instructions, \(CP\) and \(CP'\) take paths through the same \(l_i : B_i (i = 1\) or 2), and then reach the instruction \(\text{ssa } x, (l_1 : x_1, l_2 : x_2)\). Hence there must be assignments to the same \(x_i\) in both \(CP\) and \(CP'\). Because types of \(x_i\) in \(P\) and \(Q\) are equal and the value is assigned in a shorter prefix than that of the assignment to \(x\), by induction hypothesis, \(M[x] = M'[x]\) where \(M[x_i] = M'[x_i]\).

**Case 3** \(\tau \equiv \mu \alpha.\{l_1 : \tau_1, l_2 : \tau_2\}\).

In this case values may be assigned to \(x\) in more than once by the same instruction in a computation prefix. This case occurs when an assignment is done in a loop. Consider such a computation prefix of \(P\):

\(P \vdash M_0 \rightarrow \ldots \rightarrow \text{assignment to } x^1 \rightarrow \ldots \rightarrow \text{assignment to } x^i \ M,\)

where superscripts for \(x\) represent iteration times. Assume that there also exists the following computation prefix of \(Q\):

\(Q \vdash M_0 \rightarrow \ldots \rightarrow \text{assignment to } x^1 \rightarrow \ldots \rightarrow \text{assignment to } x^i \ M'.\)

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

\(P \vdash M_0 \rightarrow \ldots \rightarrow \text{bins}^1 \ldots \text{bins}^2 \ldots \text{bins}^l \rightarrow \ldots \rightarrow \text{HALT},\)

there exists a computation prefix of \(Q\),

\(Q \vdash M_0 \rightarrow \ldots \rightarrow \text{bins}^1 \ldots \text{bins}^2 \ldots \text{bins}^l \rightarrow \ldots \rightarrow \text{HALT},\)

and conversely. We prove this case by induction on the iteration times. Base cases for these two proofs are already proved in previous cases. By assumption, types of \(x\) and all variables in \(CV(P)\) have the same tree structures in \(P\) and \(Q\). Therefore, if \(\Gamma_0 \vdash p \ x^i : \tau_i \wedge \Gamma_0 \vdash Q x^i : \tau_i',\) then \(\tau_i = \tau_i'\). Moreover if \(\Gamma_0 \vdash p \ y^j : \tau_j'' \wedge \Gamma_0 \vdash Q y^j : \tau_j'''\), then \(\tau_j'' = \tau_j'''\) for all \(y \in CV(P)\). Therefore, by proving these cases simultaneously, we obtain the required result. \(\square\)

What remains is the proof of soundness.

**Proof of Soundness.** Assume that there exists a computation prefix of \(P\):

\(P \vdash M_0 \ldots \rightarrow \text{bins}_1 \ldots \rightarrow \text{bins}_n \ldots \rightarrow \text{HALT},\)

with \(\text{bins}_1, \ldots, \text{bins}_n\) branch instructions. By Lemma 1, there exists a computation prefix of \(Q\):

\(Q \vdash M_0 \ldots \rightarrow \text{bins}_1 \ldots \rightarrow \text{bins}_n \rightarrow \text{HALT},\)

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

Soundness ensures that many structure preserving optimizations are correct. For example, the instruction reordering transformation of Fig.13 is

\begin{align*}
x &= 1; & y &= 3; \\
y &= 3; & \rightarrow & x &= 1; \\
z &= x + y; & z &= x + y;
\end{align*}

Fig. 13 An Example of Instruction Reordering Optimization
proved correct because the type of z does not change as (int(1), int(3)). Though it seems trivial, it is not so easy to prove the correctness in previous works such as [2]. Furthermore, constant propagation, common subexpression elimination, loop invariant hoisting and value numbering are proved correct as corollaries of soundness. As another example, the common subexpression elimination of Fig.14 is proved correct because (τ₁, τ₂)ᵦ, the type of d (assuming b:τ₁ and c:τ₂), does not change after the transformation.

6 Several Optimizations

6.1 Constant Folding and Constant Propagation

Constant folding is a transformation that replaces a constant subexpression with the value evaluated at compile time. Constant propagation is a transformation that replaces a variable reference with a constant value. For example, in Fig.15, constant folding is applied first (1) and then constant propagation is applied (2). Although constant folding/propagation are simple optimizations, there needs a non-trivial setting in trying to give their formal definitions [10][2]. In this Section, we redefine optimizations by order relations between types of variables before and after the optimization are applied.

Definition 10 (Optimizing Order) We call an order relation as optimizing order, if it is transitive, reflexive, and satisfies the condition of Fig. 16.

In Fig.15 (int(3), int(2))ᵦ, the type of a, changes into int(6). Constant folding/propagation are redefined as program transformations which change types in this way.

Definition 11 Constant Folding/Propagation

Assume that P ≡ Q and OV(P) = OV(Q) = x.

iff

∃τ∃τ' (

Γ₀ ⊢ P x : τ ∧ Γ₀ ⊢ Q x : τ' ∧ τ ≥ₚ τ'),

where an optimizing order ≥ₚ is defined as the transitive closure of:

\{(int(i), int(j))ₚ | k = i aop j\}

The order \{l₁ : int(i), l₂ : int(i)\} ≥ₚ int(i) corresponds to a more aggressive constant propagation that “if all reaching definitions of a variable is the same constant, then all occurrences of the variable can be replaced with the constant” as in Fig.17. In Fig.17, \{l₁ : int(4), l₂ : int(4)\}, the type of x₁, changes into int(4).

It can be proved that constant folding/propagation preserve program semantics as follows.

Theorem 6 If P →ₚ Q, then P ≈ Q.

Proof.

To prove this theorem, we also exploit Lemma 1 and Lemma 2 where equality relation of types (=) are replaced with the optimizing order (¬ₚ). As for Soundness, we prove Lemma 2 by induction on the sum of length of computation prefix to the assignment to x in P and Q. As for Lemma 1, we
prove it by induction on the number of branch instructions.

**Base Case**

The shortest computation prefix satisfying

\[(\text{int}(i), \text{int}(j))_{\text{aop}} \geq_{\text{fp}} \text{int}(k)\]

(\(k = i \text{aop} j\)) is:

\[P \vdash M_0 \xrightarrow{\text{aop}} y_0 \quad \text{and} \quad Q \vdash M_0 \xrightarrow{\text{nov}} v_0 \quad M_1\]

Because \(\Gamma_0 \vdash p: \text{int}(i), v_0 : \text{int}(j)\) and \(\Gamma_0 \vdash q: v_0 : \text{int}(k)\), clearly \(M_1[x] = M_1'[x]\). As for \(\{l_1 : \text{int}(i), l_2 : \text{int}(j)\} \geq_{\text{fp}} \text{int}(i)\), because \(P \equiv Q\), the shortest prefixes for \(P\) and \(Q\) are:

\[P \vdash M_0 \xrightarrow{\text{bop}} y_0, l_1 \quad M_1 \xrightarrow{\text{nov}} v_0 \quad M_2 \xrightarrow{\text{ssa}} x (l_1; x_1; l_2; x_2) \quad M_3\]

\[Q \vdash M_0 \xrightarrow{\text{bop}} y_0, l_1 \quad M_1' \xrightarrow{\text{nov}} v_0 \quad M_2'\]

Because \(\Gamma_0 \vdash p: x : \{l_1 : \text{int}(i), l_2 : \text{int}(j)\}\) and \(\Gamma_0 \vdash q: x : \text{int}(i)\), clearly \(M_3[x] = M_3'[x] = i\) holds.

**Induction Step**

Lemma 1 in the case of the optimizing order (\(\rightarrow_{\text{fp}}\)) can be proved correct in the similar way to the equality relation. We prove Lemma 2 for the case of \(\rightarrow_{\text{fp}}\).

**Case 1** \((\tau \equiv (\text{int}(i), \text{int}(j))_{\text{aop}} \text{ and } \tau' \equiv \text{int}(k)\) where \(i \text{aop} j = k\).

The same as the base case.

**Case 2** \((\tau \equiv \{l_1 : \text{int}(i), l_2 : \text{int}(j)\} \text{ and } \tau' \equiv \text{int}(i)\).

The same as the base case.

**Case 3** \((\tau \equiv (\tau_1, \tau_2)_{\text{aop}} \text{ and } \tau' \equiv (\tau_3, \tau_4)_{\text{aop}}\) where \(\tau_1 \geq_{\text{fp}} \tau_3\) and \(\tau_2 \geq_{\text{fp}} \tau_4\).

The required result can be proved in the same way as in the case of the equality relation.

**Case 4** \((\tau \equiv \{l_1 : \tau_1, l_2 : \tau_2\} \text{ and } \tau' \equiv \{l_1 : \tau_3, l_2 : \tau_4\}\) where \(\tau_1 \geq_{\text{fp}} \tau_3\) and \(\tau_2 \geq_{\text{fp}} \tau_4\).

The same as Case 3.

**Case 5** \((\tau \text{ and } \tau' \text{ have infinite tree structures})\).

The required result can be proved in the similar way to the equality relation.

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

\[
a_1 = a_0 \ast 2; \quad a_1 = a_0 + b_0;\]

\[
b_1 = b_0 \ast 2; \quad \Rightarrow\]

\[
a_2 = a_1 + b_1; \quad a_2 = a_1 \ast 2;\]

Assuming that \(a_0 : \tau_1\) and \(b_0 : \tau_2\),

\[
((\tau_1, \text{int}(2))_{\text{x}}, (\tau_2, \text{int}(2))_{\text{x}})_{\text{+}},\]

the type of \(a_2\), changes into \(((\tau_1, \tau_2)_{\text{+}}, \text{int}(2))_{\text{x}}\).

### 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 \([8]\). For example, in Fig. 18, the computation of \(a_1 + b_1\) for the assignment to \(z_1\) 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. 18, \((\tau_1, \tau_2)_{\text{+}}\), the type of \(z_1\) changes into \(\{l_1 : (\tau_1, \tau_2)_{\text{+}}, l_2 : (\tau_1, \tau_2)_{\text{+}}\}\) (assuming \(a_1 : \tau_1\) and \(b_1 : \tau_2\)). We redefine code motion as a program transformation which changes types of variables as in Fig.18 (currently we assume that there are not any critical edges \([8][7]\) in source codes).

This definition would not subsume all code motions which have been proposed so far \([8][7]\). However, this definition captures the basic idea of code motion.

**Definition 12 (Code Motion)** Assume that \(P \equiv Q\) and \(\text{OV}(P) = \text{OV}(Q) = x\).

\[
P \rightarrow_m Q\]

iff

\[
\exists \tau \exists \tau'.
\]

\[
(\Gamma_0 \vdash p: x : \tau \land \Gamma_0 \vdash q: x : \tau' \land \tau \geq_m \tau),
\]

\[
\forall y_j \in \text{CV}(P) \exists \tau_j \exists \tau_j'.
\]

\[
(\Gamma_0 \vdash y_j : \tau_j \land \Gamma_0 \vdash q: y_j : \tau_j' \land \tau_j \geq_m \tau_j'),
\]

where an optimizing order \(\geq_m\) is defined as the transitive closure of:

\[
(\tau_1, \tau_2)_{\text{aop}} \geq_m \{l_1 : (\tau_1, \tau_2)_{\text{aop}}, l_2 : (\tau_1, \tau_2)_{\text{aop}}\}
\]

\((l_1 \text{ and } l_2 \text{ are the labels of predecessor blocks of a basic block in which the assignment of the variable of type } (\tau_1, \tau_2)_{\text{aop}} \text{ is in } P)\).

**Theorem 7** If \(P \rightarrow_m Q\), then \(P \approx Q\).

**Proof**

To prove this theorem, we also prove Lemma 1

![Fig. 18 An Example of Lazy Code Motion](image-url)
Fig. 19 An Example of Strength Reduction

\[
\mu a. \{ l_1 : \text{int}(i), l_2 : ( \alpha, \text{int}(j) )_+ \}, \text{int}(k) \} \geq_{sr} \mu a. \{ l_1 : \text{int}(i \cdot k), l_2 : ( \alpha, \text{int}(j \cdot k) )_+ \} \quad (1)
\]
\[
\mu a. \{ l_1 : \text{int}(i), l_2 : ( \alpha, \text{int}(j) )_+ \}, \text{int}(k) \} \geq_{sr} \mu a. \{ l_1 : \text{int}(i + k), l_2 : ( \alpha, \text{int}(j) )_+ \} \quad (2)
\]
\[
\{ l_1 : \text{int}(i), l_2 : \text{int}(j) \}_\text{aop} \geq_{fp} ( \text{int}(i), \text{int}(j) )_\text{aop} \geq_{fp} \text{int}(k) \quad (3)
\]

and Lemma 2 where the equality relation of types (=) are replaced with the optimizing order (\(\rightarrow_m\)).

As for Soundness, we prove Lemma 2 by induction on the sum of length of computation prefix to the assignment to \(x\) in \(P\) and \(Q\). As for Lemma 1, we prove it by induction on the number of branch instructions.

**Base Case**
The shortest computation prefix satisfying \(\tau \geq_m \tau'\) is the case that \(\tau = (\text{int}(i), \text{int}(j))_\text{aop}\) and \(\tau' = \{ l_1 : (\text{int}(i), \text{int}(j))_\text{aop}, l_2 : (\text{int}(i), \text{int}(j))_\text{aop} \}\). The proof of this case can be done in the same way as in the base case of Theorem 6.

**Induction Step**
Lemma 1 of the case of the optimizing order (\(\rightarrow_m\)) can be proved correct in the similar way to the equality relation. We prove Lemma 2 for the case of \(\rightarrow_m\).

**Case 1** \(\tau \equiv (\tau_1, \tau_2)_\text{aop}\) and \(\tau' \equiv \{ l_1 : (\tau_1, \tau_2)_\text{aop}, l_2 : (\tau_1, \tau_2)_\text{aop} \}\).
The same as the proof for soundness.

**Case 2** \(\tau \equiv (\tau_1, \tau_2)_\text{aop}\) and \(\tau' \equiv (\tau_3, \tau_4)_\text{aop}\) where \(\tau_1 \geq_m \tau_3\) and \(\tau_2 \geq_m \tau_4\).
The required result can be proved in the same way as in the proof of soundness.

**Case 3** \(\tau \equiv \{ l_1 : \tau_1, l_2 : \tau_2 \}\) and \(\tau' \equiv \{ l_1 : \tau_3, l_2 : \tau_4 \}\) where \(\tau_1 \geq_m \tau_3\) and \(\tau_2 \geq_m \tau_4\).
The same as Case 2.

**Case 4** \(\tau\) and \(\tau'\) have infinite tree structures. The required result can be proved in similar way to the proof of soundness. ☐

6.3 Strength Reduction and Induction Variables

Strength reduction is a transformation that a compiler uses to replace costly instructions with cheaper ones [4]. Fig.19 shows an example of strength reduction, in which the assignment to \(j_2\) by multiplication is replaced with the assignment to \(j_2^1\) by addition. This transformation can be represented by the optimizing order (denoted as \(\geq_{sr}\) (1) in Fig.20. Strength reduction is closely related to induction variable analysis [6]. A variable \(x\) whose value is assigned by \(x = x + c\) where \(c\) is a constant is called a basic induction variable [6]. The type of basic induction variable is of the form:

\[
\mu a. \{ l_1 : \text{int}(i), l_2 : (\alpha, \text{int}(j) )_+ \}\.
\]

A variable \(y\) to which a value is assigned by \(y = x + c\) where \(x\) is a basic induction variable and \(c\) is a constant is called a derived induction variable. It is known that a derived induction variable can be transformed to a basic induction variable. This fact is represented by the optimizing order (2) in Fig.20. Strength reduction is the case that the value of \(y\) is assigned by \(y = x \times c\) (\(x\) is a basic induction
variable and $c$ is a constant).

Combination of optimizations whose correctness are ensured by different optimizing orders can also be proved correct. Fig.21 shows an example of correct transformation, ensured by the optimizing order (3) in Fig.20. Actually this is conjunction of two order relations of constant folding/propagation where $k = i \text{ aop } j$.

7 Related Work

There have been many papers for verifying compiler optimizations, which are based on well known frameworks e.g., denotational semantics [2] and temporal logic [10][11]. Benton [2] proposed a denotational semantics based type system which can express dead code elimination and constant propagation on a simple while-language. Its problems are that the type system only tracks constancy and limited dependency (it can not express recursive calculation). Extending their system for unstructured programs represented by control flow graph 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. In their framework, dead code elimination, constant folding, and simple code motion are defined and analysed. However, defining other optimizations requires substantial amount of work [9], whereas our framework can define optimizations by just order relations on types.

Translation validation [12][16][14] is a technique for proving that original and optimized codes are equivalent. The strategy of translation validation is to check automatically whether two codes are bisimilar on simple operational semantics, irrespectively how the optimization is applied. Main difference between our approach and that of translation validation is that we are mainly concerned with formal definition of optimizations (and formal proof of the correctness).

8 Conclusion and Future Work

In this paper we have proposed a type-theoretical formalization for compiler optimizations and applied it for verifying 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 with cut elimination theorem. We have also proved that our type system is consistent in the meaning that if $\Gamma \vdash_t x : \tau$ and $\Gamma' \vdash_t x : \tau'$, $\tau = \tau'$. In Section 5, we have proved soundness of our type system. In Section 6, we have defined constant folding/propagation and code motion and stated that they preserve program semantics. We have also shown that many structure preserving optimizations can be handled in our framework. Our framework can be extended to several directions. We list two of those promising directions.

Toward Verifying Compiler Optimizations.

In our framework, verifying compiler optimizations is reduced to develop the following algorithms.
1. Type inference algorithm.
2. An algorithm for checking equality of recursive types.
3. An algorithm for checking whether types of variable in the source and optimized codes satisfy valid optimizing order.

The first task is easy, since our type system is essentially an abstraction of def-use analysis, which is the most basic analysis and has been applied by almost all optimizing compilers. The second task is itself a major research topic, and we can refer to many previous works [13]. The third task is worth studying, and seems not so easy. Verifying compiler optimizations is challenging. However, by introducing our type system, the task is organized as above three clear theoretical problems.

Type System not via SSA form. Our type system derives regular trees from programs via SSA form. It is better if regular trees are derivable not via SSA form. Furthermore, if regular trees are derivable without SSA form, since SSA form is mainly used for use-def analysis and regular trees represent use-def information (how the value of variables will be calculated), our type system will replace SSA form.

Acknowledgement

We would like to thank anonymous reviewers for their helpful and insightful comments.

References


