Regular Paper

A Type System for Verification of Compiler Optimizations

Yutaka Matsuno[†] and Hiroyuki Sato[††]

This paper presents a type theoretical framework for the verification of compiler optimizations. Though 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 framework, we introduce *assignment types* for variables, which represent how the value of variables will be calculated in a program. First we introduce our type system. Second we prove soundness of our type system meaning that if types of return values are equal in two programs, the programs are equivalent. Soundness ensures that many structure preserving optimizations preserve program semantics. Then by extending the notion of type equality to order relation, we redefine several optimizations and prove that they also preserve program semantics.

1. Introduction

Compiler optimizations have become essential parts of high performance computing, and fairly advanced optimizations have been applied. However, this makes the verification of compiler optimizations extremely difficult, and 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; the correctness proof for an optimization is based on ad-hoc combination of several proof techniques. To overcome this difficulty, some previous papers applied existing theoretical frameworks. However, none of them seems successful. Reasons are that their frameworks are so restrictive that only a few optimizations can be defined, or their frameworks require substantial amount of work even to define an optimization.

To establish a theoretical framework, first we back to basics of compiler optimizations. Allen and Kennedy [1)] stated *dependences* as a set of constrains which are sufficient to ensure that program transformations do not change the meaning of a programs as represented by the results it computes. These constraints are not precise; there are cases where dependences can be violated without changing the program meaning. However, they capture an important strategy for preserving correctness. Therefore conventional compilers verify that dependences among variables are preserved by 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. In modern compilers, SSA form has become a standard intermediate form.

In this paper, we formalize the problem of verifying compiler optimizations by a type system 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 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 which the assignments of variables of type τ_1 and τ_2 come from. $int(0)$ and $int(1)$ are singleton types [21)] of integer whose values are 0 and 1 respectively.

† Department of Frontier Informatics, The University of Tokyo
†† Information Technology Center, The University of Tokyo

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

Fig. 2 Syntax for intermediate language in SSA form.

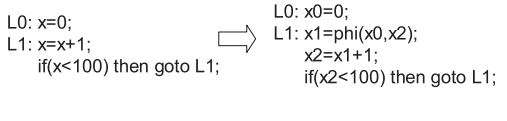

Fig. 1 Source and SSA form of a program P where $\texttt{x1} : \alpha$ and $\texttt{x2} : \beta$.

Type equations for α and β are:

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

Substituting (2) into (1) leads to $\alpha = \{L0 : int(0), L1 : (\alpha, int(1))_+\}$, in other words x1 has a recursive type of the form $\mu\alpha.\{L0 : int(0), L1 : (\alpha, int(1))_+\}$. By introducing recursive types for recursive calculation in a loop, we express clearly the difficulty of loop analysis which has been discussed in the literature of compiler optimization [1].

Our contributions are as follows.

- We devise a model of computations by a type system which captures the intuition behind the wide variety of compiler optimizations.
- We prove soundness of our type system that if types of return values of the source and target codes are equal, then they are equivalent programs. Soundness ensures that many structure preserving optimizations such as dead code elimination and constant propagation preserve program semantics.
- Furthermore, by extending the notion of type equality to order relation, we redefine several optimizations, and prove that they also preserve program semantics.

We believe that our type system is an appropriate theoretical framework for establishing *optimization verifying compilers*, which automatically verify that their optimization preserve program 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. 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.

In order to provide a simple framework this language has no memory operations and function calls. 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 informa-

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

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

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

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

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

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

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

Fig. 3 Dynamic semantics.

tion 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 $x_0, y_0, \ldots$. 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 $l \in L$. Also L includes l_{entry} and l_{exit} which are labels for the entry and exit of a program respectively. 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 [☆] of variables to their values, l_p is the label of the predecessor block the program execution has passed, and l_c is the label of the current block.

- Initial machine state: $M_0 = (0, V_0, (l_{entry}, l_0))$ where V_0 maps variables to initial values.
- 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.

Figure 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 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 next basic block, and returns the new pair of previous and current labels.

Definition 2 ($NL(pc, (l_p, l_c))$)

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

3. Program Equivalence

We define program equivalence based on a kind of *bisimulation* [12], which is also a basic notion of program equivalence defined in some previous papers [8),15)]. To verify the equivalence, we check all possible paths that the source and target programs may take, and verify that each path in the source program corre-

[☆] Let g be a map, $Dom(g)$ be the domain of g; for $x \in Dom(g)$, $g[x]$ 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 $y \in Dom(g)$: $(g[x \to v])[y] = if\ y \neq x\ then\ g[y]\ else\ v$.

sponds to a path in the target program 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)

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

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

To verify that the source and target programs are equivalent, we need to determine which variables need to be checked that they have the same values at the end of executions (it can be said that such variables are observable from outside of the program). We say such variables as live variables of program P and define *live variable set* $LV(P)$. If a program P ends with `return` x, then $LV(P)$ can be $\{x\}$.

Definition 4 (Live Variable Set $LV(P)$) The live variable set of a program P, $LV(P)$ is the set containing all live variables of P.

Note that an arbitrary variable can be live: it depends on the calling convention of the compiler.

Then we define program equivalence as follows.

Definition 5 (Program Equivalence) Assume that there are two programs P and Q such that $LV(P) = LV(Q)$ which share a label set L and a variable set Var.

$$P \approx Q$$

if and only if for each $x \in LV(P)$, for any finite computation prefix CP

$$CP = P \vdash M_0 \to M_1 \to M_2 \to \cdots \overset{ins}{\to} M,$$

where $ins \in P$ is an instruction of assignment to x, there exists a computation prefix CP' of Q

$$CP' = Q \vdash M_0 \to M'_1 \to M'_2 \to \cdots \overset{ins'}{\to} M'$$

such that $ins' \in Q$ is an assignment to x and $M[x] = M'[x]$ holds, and the same with P, Q interchanged.

Note that the length of CP and CP' are not necessarily equal; ins and ins' are not necessarily the same instruction.

A program is also represented as a network of basic blocks. In this paper we require that the structure of the network is preserved in the transformation. Optimizations preserving this property are known as *structure preserving optimizations* in which almost all global optimizations are included.

Definition 6 (Structural Equivalence) Two programs

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

which share a label set L 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.

- Both B_i and B'_i do 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 $l \in L$.
- Both B_i and B'_i end with the same `jmp` l where $l \in L$.
- Both B_i and B'_i end with the same `return` x where $x \in Var$.

This definition does not allow some simple optimizations e.g. replacing `bop` x, v, l by `bop` $3, v, l$ when x is proved to be 3 in all program executions. Extending the notion of structural equivalence for such optimizations is easy. For simplicity we state structural equivalence as in Definition 6.

To prove the correctness of structure modifying optimizations (loop unrolling, loop tiling, loop fusion, etc), this definition needs to be relaxed. However, because they also exploit use-def information essentially, our type system will also play the essential role for the proof of the correctness of structure modifying optimizations. This is left as future work.

As shown in Definition 6 variables used in branch instructions are also important. We define such variables as *Control Variables* as follows.

Definition 7 (Control Variable Set $CV(P)$) The control variable set for program P, $CV(P)$ is a set of variables which are used in branch instructions of P.

4. Type System

4.1 Assignment Types

We define assignment types as in **Fig. 4**. Type variables are represented by $\alpha, \beta, \ldots$. $\top$ is the super type of all other types. $int(i)$ is a singleton type of a variable or an integer whose value is i. Sometimes we use a notation $int(x)$ for $int(i)$ 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.

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

Fig. 4 Types.

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

$$\frac{}{\vdash \Gamma_0} \qquad \frac{\vdash \Gamma \quad x \notin \mathit{Dom}(\Gamma) \quad \alpha \text{ fresh}}{\vdash \Gamma, x : \alpha}$$

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

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

$$\frac{\texttt{aop}\ x, y, v \in P \quad \Gamma \vdash_P y : \tau_1, v : \tau_2}{\Gamma \vdash_P x : (\tau_1, \tau_2)_{\text{aop}}}\ (\textbf{type-aop}) \qquad \frac{\texttt{mov}\ x, v \in P \quad \Gamma \vdash_P v : \tau}{\Gamma \vdash_P x : \tau}\ (\textbf{type-mov})$$

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

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

Fig. 5 Typing rules.

Variables to which values are assigned by `mov` instructions may also have these types. Intuitively, an assignment type expresses how the value of the variable is calculated in a program. A type environment is defined as a set of type declaration of variables as usual. We denote $\mathit{Dom}(\Gamma)$ for the set of variables whose types are declared in Γ.

4.2 Type System

The type system is shown in **Fig. 5**. 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 $\mathit{int}(i), \mathit{int}(j), \ldots$ in Γ_0 if their initial values are $i, j, \ldots$ respectively.
- $\Gamma \vdash_P v : \tau$
  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**). There are two key typing rules: (**type-mu**) and (**cut**) rule. Intuitively (**type-mu**) means that if a variable x is given a type $\{l_1 : \tau_1, l_2 : \tau_2\}$ under the assumption that 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 corresponds to substitution for solving type equations as shown in Section 1. In (**cut**) rule, $\oplus$ operation is used for type environments.

Definition 8 ($\Gamma_1 \oplus \Gamma_2$) 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

$$\begin{array}{lll}
x''_i = x_j = x'_k \text{ and } \tau_j = \tau'_k & \text{then} & \tau''_i = \tau_j \\
x''_i = x_j = x'_k \text{ and } \tau_j \neq \tau'_k & \text{then} & \tau''_i = \top
\end{array}$$

else

$$\dfrac{\dfrac{\vdash \Gamma_0 \quad x_1 \notin dom(\Gamma_0) \quad \alpha \text{ fresh}}{\vdash \Gamma_0, x_1 : \alpha} \qquad x_1 : \alpha \in \Gamma_0, x_1 : \alpha}{\Gamma_0, x_1 : \alpha \vdash_P x_1 : \alpha, 1 : int(1)}\ (\textbf{type-var}) \text{ and } (\textbf{type-int})$$

$$\dfrac{\texttt{add}\ x_2, x_1, 1 \in P \qquad \Gamma_0, x_1 : \alpha \vdash_P x_1 : \alpha, 1 : int(1)}{\Gamma_0, x_1 : \alpha \vdash_P x_2 : (\alpha, int(1))_+ \quad (D_1)}\ (\textbf{type-add})$$

$$\dfrac{\dfrac{\vdash \Gamma_0 \quad x_2 \notin dom(\Gamma_0) \quad \beta \text{ fresh}}{\vdash \Gamma_0, x_2 : \beta} \qquad x_2 : \beta \in \Gamma_0, x_2 : \beta}{\Gamma_0, x_2 : \beta \vdash_P 0 : int(0), x_2 : \beta}\ (\textbf{type-int}) \text{ and } (\textbf{type-var})$$

$$\dfrac{\texttt{ssa}\ x_1, (L0 : x_0, L1 : x_2) \in P \qquad \Gamma_0, x_2 : \beta \vdash_P 0 : int(0), x_2 : \beta}{\Gamma_0, x_2 : \beta \vdash_P x_1 : \{L_0 : int(0), L_1 : \beta\} \quad (D_2)}\ (\textbf{type-ssa})$$

$$\dfrac{D_1 \quad D_2}{\Gamma_0, x_1 : \alpha \vdash_P x_1 : \{L0 : int(0), L1 : (\alpha, int(1))_+\}}\ (\textbf{cut})$$

$$\dfrac{\Gamma_0, x_1 : \alpha \vdash_P x_1 : \{L0 : int(0), L1 : (\alpha, int(1))_+\}}{\Gamma_0 \vdash_P x_1 : \mu\alpha.\{L0 : int(0), L1 : (\alpha, int(1))_+\}}\ (\textbf{type-mu})$$

Fig. 6 Type derivation of `x1` in Fig. 1.

$$\begin{array}{rl} T(int(i)) = & (int(i)) \\ T((\tau_1, \tau_2)_{aop}) = & (\text{aop}) \\ & \swarrow \quad \searrow \\ & T(\tau_1) \ \ T(\tau_2) \\ T(\{l_1 : \tau_1, l_2 : \tau_2\}) = & (\text{ssa}) \\ & \swarrow \quad \searrow \\ & l_1 : T(\tau_1) \ \ l_2 : T(\tau_2) \end{array}$$

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

Fig. 7 Tree structure of types.

$$\begin{array}{lll} x_i'' = x_j & \text{then} & \tau_i'' = \tau_j \\ x_i'' = x_k' & \text{then} & \tau_i'' = \tau_k' \end{array}$$

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 for variables in a program. As an example of typing, the type derivation of `x1` in Fig. 1 is shown in **Fig. 6** (we assume that `x0` is given as an argument, and $\Gamma_0 = x_0 : int(0)$).

4.3 Type Equality

Because our type system includes recursive types, type derivations for variables are not unique. We define type equality by the tree structure of types, as usual in theories of recursive types [3),16)].

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 9 (Regular Trees $T(\tau)$) Regular trees are defined as in **Fig. 7**. Nodes are of the form $(+)$,$(-)$, $(\times)$, (ssa), or denoted by leaves of the form $(int(i))$. If $T(\tau_1)$ and $T(\tau_2)$ have the same structure, we denote $T(\tau_1) = T(\tau_2)$.

Definition 10 (Type Equality) If $T(\tau_1) = T(\tau_2)$, then $\tau_1 = \tau_2$.

Note that this equality is obviously decidable.

Theorem 1 *For any derivations of* $\Gamma_0 \vdash_P x : \tau$ *and* $\Gamma_0 \vdash_P x : \tau'$, $\tau = \tau'$.

Proof

Consider if a type τ_1 of a variable changes to τ_2 by cut rule. Because (**cut**) rule only instantiates occurrences of a type variable in τ_1, $\tau_1 = \tau_2$. Also rule (**type-mu**) does not modify the tree structure of types. Hence we get the required result because Γ_0 is closed. □

The tree structure shows that our type system shares some ideas with several notions e.g., def-use analysis and program slicing [20)]. They are used in program analysis widely. It can be said that our type system with tree structure captures key aspects of computation, which have been discussed partially by such notions.

5. Soundness

The soundness of our type system says that if types of live variables and control variables of source and target programs are equivalent, the programs are equivalent.

Theorem 2 (Soundness) If $P \cong Q$, $LV(P) = LV(Q)$ and

- $\forall x_i \in LV(P)$. $(\Gamma_0 \vdash_P x_i : \tau_i \wedge \Gamma_0 \vdash_Q x_i : \tau_i' \wedge \tau_i = \tau_i')$
- $\forall y_j \in CV(P)$. $(\Gamma_0 \vdash_P y_j : \tau_j \wedge \Gamma_0 \vdash_Q y_j : \tau_j' \wedge \tau_j = \tau_j')$

then $P \approx Q$.

To prove soundness, we need to focus on variables used during the calculation. For proving soundness we sometimes slice programs for such variables. The conventional slicing algorithms involve finding the transitive closure of the data and control dependences of the appropriate node(s) in the program dependence graph (PDG) of the program. PDG of a program is obtained by merging its data and control dependence graphs, which depict the interstatement data and control dependences, respectively, in the program. There are several techniques for slicing [20]. We refer to previous works of backward slicing with respect to output variables, and assume that a program and its sliced program for a variable are equivalent (see Definition 5) for the variable.

Notation 1 *The sliced version of a program P for a variable x is denoted as P_x.*

To prove soundness, the following theorem is required.

Theorem 3 *If $P \cong Q$ and*

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

then for any computation prefix of P:

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

with $bins_i$ $(i = 1, 2, \ldots)$ branch instructions, there is a computation prefix of Q:

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

and conversely.

We prove soundness and Theorem 3 simultaneously.

Proof

Note that by $P \cong Q$, $CV(P) = CV(Q)$. We prove soundness by induction on the length of computation prefix for each $x \in LV(P)$ where $\Gamma_0 \vdash_P x : \tau$ and prove Theorem 3 by induction on the number of branch instructions.

Base Case

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

(1) $P \vdash M_0 \overset{\texttt{aop}\ y_0,z_0,v_0}{\rightarrow} M_1$

(2) $P \vdash M_0 \overset{\texttt{jmp}\ l}{\rightarrow} M_1$

(3) $P \vdash M_0 \overset{\texttt{bop}\ y_0,v_0,l}{\rightarrow} M_1$

(4) $P \vdash M_0 \overset{\texttt{return}\ x_0}{\rightarrow} \texttt{HALT}$

In (1), there are two possible type derivation for x in P: $\Gamma_0 \vdash_P x_0 : int(x_0)$ ($x \not\equiv y_0$, in this case x is an initial value) and $\Gamma_0 \vdash_P x : (int(z_0), int(v_0))_{\text{aop}}$ (otherwise). Because x has the same type in Q, by considering the type derivation of x, clearly x has the same value in both P and Q. In (2), because $P \cong Q$, P and Q are identical. In (3) because $P \cong Q$ and y_0 and v_0 are initial values, if P takes the true or the false branch then Q takes the same branch. In (4), because $P \cong Q$, P and Q are identical.

Induction Step

The case when $\tau = int(i)$ is easy. We consider other cases.

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 of the last typing rule: (**type-mov**) and (**type-add**) rules. When (**type-mov**) is used, the type derivation is as follows.

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

In this case the value of x depends on v. 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):

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

Let P_y and Q_y be sliced programs for y of P and Q respectively where $y \in LV(P_y) = LV(Q_y)$. By adding auxiliary basic blocks to either P_y or Q_y, $P_y \cong Q_y$. Also the type of y is the same in P_y and Q_y. Clearly the length of computation prefix of y is shorter than that of x. Therefore by IH, $P_y \approx Q_y$. We consider the case when v is a variable and denote such v as z (it is easy when v is an integer). By the same argument, we can construct $P_{y,z}$ and $Q_{y,z}$ which are sliced programs of P and Q for y and z respectively where $y, z \in LV(P_{y,z}) = LV(Q_{y,z})$

such that $P_{y,z} \approx Q_{y,z}$. By the property of SSA form the assignments to y and z dominate the assignment to x in both P and Q. Therefore all computation prefixes of P and Q pass the assignment to y and z. Therefore for any computation prefix of P:

$$P \vdash M_0 \to \ldots M_1 \to \ldots \overset{\text{assignment to } x}{\to} M_2$$

there exists a computation prefix of Q:

$$Q \vdash M_0 \to \ldots M_1' \to \ldots \overset{\text{assignment to } x}{\to} M_2'$$

such that $M_1[y] = M_1'[y]$ and $M_1[z] = M_1'[z]$ before the assignment to x; $M_2[x] = M_2'[x]$ because of $M_2[y] = M_1[y] = M_1'[y] = M_2'[y]$ and $M_2[z] = M_1[z] = M_1'[z] = M_2'[z]$ and conversely. Hence $P \approx Q$.

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):

$$\frac{\texttt{ssa}\ x, (l_1 : x_1, l_2 : x_2) \in P \quad \Gamma_0 \vdash_P x_1 : \tau_1, x_2 : \tau_2}{\Gamma_0 \vdash_P x : \{l_1 : \tau_1, l_2 : \tau_2\}}$$

(type-ssa)

where the types of x_i $(i = \{1, 2\})$ in P and Q are equal. Assume that for any computation prefix of P:

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

there is a computation prefix of Q:

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

and conversely. We slice P and Q for each variable z used in the $(k+1)$-th branch instruction. By IH from Theorem 2, the values of z in P_z and Q_z are equal (because $z \in CV(P)$, and calculated by shorter prefix than that of x). Hence both computation prefixes pass the same branch at the $(k+1)$-th branch instruction (this proves Theorem 3). Therefore, for any computation prefix of P

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

there is a computation prefix of Q,

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

and conversely. If CP and CP' pass $l_i : B_i$, there must be assignments of x_i in both CP and CP'. Because types of x_i in P and Q are equal and they are calculated by shorter prefix, by IH $M[x] = M'[x]$ where $M[x_i] = M'[x_i]$. Hence $P \approx Q$.

Case 3 $\tau = \mu\alpha.\{l_1 : \tau_1, l_2 : \tau_2\}$. In this case a value 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 represent iteration counts. We prove this case by induction on the iteration counts. We need to prove that for any computation prefix as above, there is a computation prefix of Q,

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

such that $M[x] = M'[x]$ and conversely. To prove this, we also need to consider the case a branch instruction is executed more than one time: 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 is 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. We prove this case by induction on the iteration counts. Base cases for these cases are already done in previous cases. By the assumptions of Theorem 2 all types of variables in $LV(P)$ and $CV(P)$ have the same tree structure. Therefore $\Gamma_0 \vdash_{P_{x^i}} x^i : \tau_i \wedge \Gamma_0 \vdash_{Q_{x^i}} x^i : \tau_i'$ for all $x \in LV(P)$ and $\Gamma_0 \vdash_{P_{y^j}} y^j : \tau_j'' \wedge \Gamma_0 \vdash_{Q_{y^j}} y^j : \tau_j'''$ for all $y \in CV(P)$ where τ_i and τ_i', and τ_j'' and τ_j''' have the same tree structure which are subtrees of the types of x and y respectively. By proving the theorem for these cases simultaneously as shown in **Case 2**, we get the required result. □

As a corollary of soundness, it can immediately be proved that dead code elimination defined in Ref. 8) (denoted as $\to_{dce}$ in this paper) preserves program semantics, because it only replaces an assignment instruction, the source and rewritten code are structurally equivalent, and this replacement is ensured not to affect the use-def chain of the live variables.

Corollary 1 *If $P \to_{dce} Q$, then $P \approx Q$.*

Also many optimizations which preserve use-def chains e.g., common subexpression elimination and value numbering satisfy soundness of our type system. Note that unlike Refs. 8), 9), our framework does not require specific definition of an algorithm to apply an optimization. All structure preserving optimizations which preserve types are correct.

6. Several Optimizations

6.1 Constant Folding and Constant Propagation

Constant folding rewrites an expression with constant operands by its evaluated value, thus improves run-time performance and reduces code size. Constant propagation propagates constants assigned to a variable through the flow graph and substitutes the value for a given expression at the use of the variable. For example, in the following program constant folding is applied first at (1) and then constant propagation is applied at (2).

```
a = 3 * 2;  (1) -> a = 6;
   ...
b = a + 4;  (2) -> b = 10;
```

Constant folding/propagation are simple optimizations. However, formally defining them is not easy because many different versions have been proposed. In this paper, we redefine an optimization by an order relation between types of live variables (and control variables) before and after the optimization is applied.

Definition 11 (Optimizing Order) *Optimizing order* on the set of types is defined as an order which is compatible with $()_+$,$()_-$,$()_\times$ and $\{\}$.

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/propagation is redefined as a program transformation which changes types in this way.

Definition 12 (Constant Folding/ Propagation)
Assume that $P \cong Q$ *and* $LV(P) = LV(Q)$.
$P \rightarrow_{fp} Q$
iff

- $\forall x_i \in LV(P)$.
  $(\Gamma_0 \vdash_P x_i : \tau_i \wedge \Gamma_0 \vdash_Q x_i : \tau'_i \wedge \tau_i \succeq_{fp} \tau'_i)$
- $\forall y_j \in CV(P)$.
  $(\Gamma_0 \vdash_P y_j : \tau_j \wedge \Gamma_0 \vdash_Q y_j : \tau'_j \wedge \tau_j \succeq_{fp} \tau'_j)$

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

$$\begin{cases} (int(i), int(j))_{\text{aop}} & \succeq_{fp} int(k) \\ & \quad (k = i \text{ aop } j) \\ \{l_1 : int(i), l_2 : int(i)\} & \succeq_{fp} int(i) \end{cases}$$

The order $\{l_1 : int(i), l_2 : int(i)\} \succeq_{fp} int(i)$ 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 by the constant" as in **Fig. 8**. In Fig. 8 $\{l_1 : int(4), l_2 : int(4)\}$, that is the type of `x1` before transformation changes into $int(4)$.

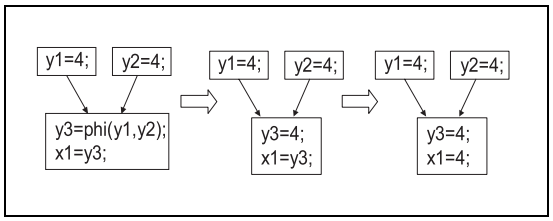

Fig. 8 An example of constant propagation.

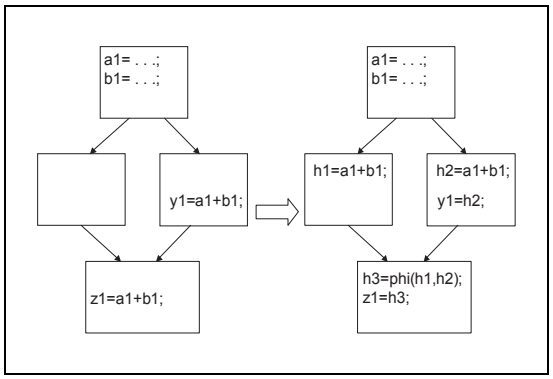

Fig. 9 An example of lazy code motion.

There are many versions of constant folding/propagation which use different algorithms. In this paper, if they are proved to preserve above order relations (and structural equivalence), we define them as "constant folding/propagation", irrespectively of which algorithms they use.

It can be proved that constant folding/propagation preserves program semantics by the same way of soundness.

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

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 [7]. For example, in **Fig. 9** 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. 9 $(\tau_1, \tau_2)_+$, 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 motion as a program transformation which changes types of variables as in Fig. 9 (currently we assume that there are not any *critical edges* [7] in source pro-

grams). Note that of course this definition does not subsume all code motions which have been proposed so far [6),7)]. To subsume all of them, more order relations are required.

Definition 13 (Code Motion) Assume that $P \cong Q$ and $LV(P) = LV(Q)$.

$P \rightarrow_m Q$

iff

- $\forall x_i \in LV(P)$.
  $(\Gamma_0 \vdash_P x_i : \tau_i \wedge \Gamma_0 \vdash_Q x_i : \tau'_i \wedge \tau_i \succeq_m \tau'_i)$
- $\forall y_j \in CV(P)$.
  $(\Gamma_0 \vdash_P y_j : \tau_j \wedge \Gamma_0 \vdash_Q y_j : \tau'_j \wedge \tau_j \succeq_m \tau'_j)$

where an optimizing order $\succeq_m$ is defined as the transitive closure of:

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

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

Theorem 5 *If $P \rightarrow_m Q$, then $P \approx Q$.*

In our framework, proving correctness of an algorithm for an optimization is reduced to proving that they preserve a valid optimizing order. Formally proving correctness of an algorithm is involved. Because our type system can deal with several optimizations uniformly by optimizing orders, our framework will give a clear view for the task.

7. Related Work

There have been many papers for the verification of compiler optimizations, which are based on well known theoretical frameworks e.g., abstract interpretation [4),19)], denotational semantics [2)], and temporal logic [8)]. Lacey, et al. [8)] showed that temporal logic is sufficient to express data dependence among variables and conditions for applying optimizations. One of the merits of 8) is that it defined the notion of program equivalence clearly, which our definition of equivalence is based on.

Translation validation [15),17),18),22)] is a practical solution for proving that source and target codes are equivalent. The strategy of translation validation is to check whether two programs are bisimilar on simple operational semantics.

Proving correctness of loop optimizations is still challenging. Previous works [10),22)] only considered structured loops (i.e., for-loops and do-loops) and exploited classical techniques. We believe that the notion of recursive types which we have introduced is essential for this problem.

Several type systems have been proposed for program analysis on low-level languages since the success of TAL [13)] and PCC [14)], and recognized as a promising approach for security issues for software systems. In our previous work [11)], we extend TAL for static array bound check elimination.

8. Conclusion and Future Work

In this paper we have proposed a type-theoretical formalization for the verification of optimizing compilers. 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/propagation and code motion and stated that they preserve program semantics.

There are a lot of things to do for future work e.g., dealing with arrays. Some of them could be handled in our current framework. However, in our study we see that there is a considerable algebraic structure in low level languages: not only usual arithmetic operations, but there exist additionally `ssa` operations (it can be regarded as "OR" operations) and μ. With the result of our study that optimizations can be defined by just order relations on types, for establishing optimization verifying compilers, we believe that more algebraic study for low level languages and compiler optimizations is essential.

References

1) Allen, R. and Kennedy, K.: *Optimizing Compilers for Modern Architectures*, Morgan Kaufmann Publishers (2001).
2) Benton, N.: Simple relational correctness proofs for static analyses and program transformations, *31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Venice, Italy (2004).
3) Brandt, M. and Henglein, F.: Coinductive axiomatization of recursive type equality and subtyping, Hindley, R. (ed.), *Proc. 3rd Int'l Conf. on Typed Lambda Calculi and Applications* (*TLCA*), Nancy, France, April 2–4, 1997, Vol.1210, pp.63–81, Springer-Verlag (1997).
4) Cousot, P. and Cousot, R.: Systematic design of program transformation frameworks by abstract interpretation, *29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp.178–190, Portland, Oregon, ACM Press, New York, NY (Jan.

2002).
5) Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N. and Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph, *ACM Trans. Prog. Lang. Syst.*, Vol.13, No.4, pp.451–490 (Oct. 1991).
6) Kennedy, R., Chan, S., Liu, S.-M., Lo, R., Tu, P. and Chow, F.: Partial redundancy elimination in SSA form, *ACM Trans. Prog. Lang. Syst.*, Vol.21, No.3, pp.627–676 (1999).
7) Knoop, J., Rüthing, O. and Steffen, B.: Optimal code motion: Theory and practice, *ACM Trans. Prog. Lang. Syst.*, Vol.16, No.4, pp.1117–1155 (July 1994).
8) Lacey, D., Jones, N., Van Wyk, E. and Frederikson, C.C.: Proving correctness of compiler optimizations by temporal logic, *Higher-Order and Symbolic Computation*, Vol.17, No.2 (2004).
9) Lerner, S., Millstein, T. and Chambers, C.: Automatically proving the correctness of compiler optimizations, *ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, San Diego, California, ACM Press, New York, NY (2003).
10) Mateev, N., Menon, V. and Pingali, K.: Fractal symbolic analysis, *ACM SIGARCH 15th International Conference on Supercomputing*, pp.38–49 (2001).
11) Matsuno, Y. and Sato, H.: Flow analytic type system for array bound checks, Harland, J. (ed.), *Electronic Notes in Theoretical Computer Science*, Vol.78, Elsevier (2003).
12) Milner, R.: *Communication and Concurrency*, Prentice-Hall (1989).
13) Morrisett, G., Walker, D., Crary, K. and Glew, N.: From System F to typed assembly language, *ACM Trans. Prog. Lang. Syst.*, Vol.21, No.3, pp.527–568 (1999).
14) Necula, G.: Proof-carrying code, *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges*, pp.106–119, Paris (Jan. 1997).
15) Necula, G.: Translation validation for an optimizing compiler, *ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, Vancouver, Canada (2000).
16) Pierce, B.C.: *Types and Programming Languages*, MIT Press (2002).
17) Pnueli, A., Siegel, M. and Singerman, E.: Translation validation, *Proc. TACAS'98, LNCS*, Vol.1384, pp.151–166, Springer (1998).
18) Rinard, M.: Credible compilers, Technical Report MIT/LCS/TR-776, MIT (1999).
19) Rival, X.: Symbolic transfer function-based approaches to certified compilation, *31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Venice, Italy (2004).
20) Tip, F.: A survey of program slicing techniques, *Journal of Programming Languages*, No.3, pp.121–189 (1995).
21) Xi, H. and Harper, R.: Dependently typed assembly language, *6th ACM SIGPLAN International Conference on Functional Programming*, pp.169–180, Florence (Sep. 2001).
22) Zuck, L., Pnueli, A., Fang, Y. and Goldberg, B.: Voc: A translation validator for optimizing compilers, Knoop, J. and Zimmermann, W. (eds.), *Electronic Notes in Theoretical Computer Science*, Vol.65, Elsevier (2002).

(Received December 20, 2003)
(Accepted February 24, 2004)

Yutaka Matsuno is a Ph.D. candidate in Department of Frontier Informatics, the University of Tokyo. He was born in 1977, and received his BEng and MSc degrees from the University of Tokyo in 2001 and 2003 respectively. His major research interests include lambda calculus, type system and compiler optimizations.

Hiroyuki Sato received BSc, MSc and DSc from the University of Tokyo in 1985, 1987 and 1990 respectively. He is currently an associate professor of the University of Tokyo (Information Technology Center/Department of Frontier Informatics). His major interests include compiler optimization and algebraic methods in program semantics.