PAPER

Loop Quasi-Invariance Code Motion

Litong SONG[†], **Yoshihiko FUTAMURA**[†], **Robert GLÜCK**[†], *and* **Zhenjiang HU**[††], ***Nonmembers***

SUMMARY Loop optimization plays an important role in compiler optimization and program transformation. Many sophisticated techniques such as loop-invariance code motion, loop restructuring and loop fusion have been developed. This paper introduces a novel technique called loop quasi-invariance code motion. It is a generalization of standard loop-invariance code motion, but based on loop quasi-invariance analysis. Loop quasi-invariance is similar to standard loop-invariance but allows for a finite number of iterations before computations in a loop become invariant. In this paper we define the notion of loop quasi-invariance, present an algorithm for statically computing the optimal unfolding length in While-programs and give a transformation method. Our method can increase the accuracy of program analyses and improve the efficiency of programs by making loops smaller and faster. Our technique is well-suited as supporting transformation in compilers, partial evaluators, and other program transformers.

key words: *loop quasi-invariance, code motion, program optimization, partial evaluation*

1. Introduction

Loop-invariance code motion is a well-known loop transformation technique that plays an important role in compiler optimization. When a computation in a loop does not change during the dynamic execution of the loop, we can hoist this computation out of the loop to improve execution time of the loop. For example, the evaluation of expression I×10 is loop-invariant in the following loop: While i<I×10 Do s:=s+i; i:=i+1; EndWhile. A more efficient program is: t:=I×10; While i<t Do s:=s+i; i:=i+1; EndWhile.

Traditional transformations move out of loops the following: (i) computations that are invariant during all loop iterations (loop invariance code motion [1]), (ii) computations that are invariant after the first iteration, (loop peeling, e.g., [23]), (iii) computations that are conditionally invariant (e.g., speculation motion [11]).

Techniques for code motion are basically limited to loop-invariant computations, this can be a problem in automatically produced programs. Consider the digital circuit in Fig. 1, which is simulated by the program in Fig. 2, where the simulation is carried out for T steps. We use f, g and g_i to denote functions simulating blocks.

With traditional code motion technique, we can only hoist $g_1(c_1)$ out of the loop, and save it using a fresh variable. In fact, we can obtain a more efficient code (Fig. 3) by unfolding the original loop three times:

In loop 1, the invariance of $g_1(c_1)$ results in the invariance of x_1, and the same assignment to x_1 in the following iterations can be removed safely. In loop 2, the invariance of $g_2(x_1,c_2)$ results in the invariance of x_2, and the assignments to x_2 afterwards can be safely removed too. For the same reason, the assignment to x_3 in loop 3 can be removed.

For the residual loop, after x_1, x_2 and x_3 have turned into loop-invariant variables, $g(x_1, x_2, x_3)$ becomes a loop-invariant computation. Applying traditional code motion technique to the remaining loop will yield the residual loop above (Fig. 2).

Table 1 shows the speedup where functions g_1, g_2, g_3, g, f, t and T are defined in Appendix 1.

To conclude, we have seen that a full optimization of the program in Fig. 2 is not possible by traditional code motion, because some of the computations in the loop are stabilize only after a finite number of itera-

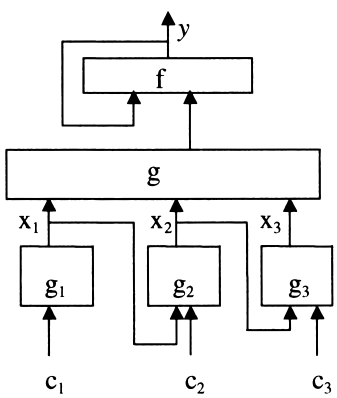


Fig. 1 A sequential digital circuit.

```
While t<T Do
    x3:=g3(x2, c3); x2:=g2(x1, c2); x1:=g1(c1);
    y:=f(g(x1, x2, x3), y);
    t:=t+1;
EndWhile.
```

Fig. 2 A simulation program for Fig. 1.

Manuscript received February 7, 2000.

[†]The authors are with the Department of Information and Computer Science, Graduate School of Science and Engineering, Waseda University, Tokyo, 169–0072 Japan.

[††]The author is with the Department of Information Engineering, the University of Tokyo, Tokyo, 113–0033 Japan.

```
If t<T Then     /* loop 1 */
    x3:=g3(x2, c3); x2:=g2(x1, c2); x1:=g1(c1);
    y:=f(g(x1, x2, x3), y);
    t:=t+1;
    If t<T Then     /* loop 2 */
        x3:=g3(x2, c3); x2:=g2(x1, c2);
        y:=f(g(x1, x2, x3), y);
        t:=t+1;
        If t<T Then     /* loop 3 */
            x3:=g3(x2, c3);
            y:=f(g(x1, x2, x3), y);
            t:=t+1;
            _g:= g(x1, x2, x3);
            While t<T Do       /* residual loop */
                y:=f(_g, y);
                t:=t+1;
            EndWhile;
        EndIf EndIf EndIf
```

Fig. 3 The program after loop quasi-invariance code motion of Fig. 2.

Table 1 The comparison of the runtime of the loop in Fig. 2 and the program in Fig. 3.

System	Gateway E5250 Pentium II xeon×2 Speed: 450MHZ Memory: 1GB, Visual C++ 6.0		Sun ULTRA 5 SunOS 5.6 Speed: 270MHZ Memory:192MB Gcc 2.7	
	Runtime	Speed up	Runtime	Speed up
Source	22sec	1	25sec	1
Result	5sec	4.4	5sec	5

tions. It is clear that a hand-optimization of program may be error-prone and tedious, and the initial specification of the circuit may change. This is why we are looking for an algorithmic solution.

The main contributions of this paper are:

- a formalization of loop quasi-invariance and optimal unfolding length for a loop,
- a static quasi-invariance analysis for computing the optimal unfolding length for any given loop in a program which allows to remove all quasi-invariant variables from a loop while avoiding over-unfolding,
- a loop transformation technique using the results of the analysis,
- an illustration of the effect of loop quasi-invariance code motion on program transformation, in particular partial evaluation.

2. Preliminaries

We define a source language and summarize the single static assignment form.

```
SS    ::=  S | S; SS
S     ::=  Ass | Cond | Loop | Call | Skip
Ass   ::=  V:=E
Cond  ::=  If E Then SS Else SS EndIf
           | If E Then SS EndIf
Loop  ::=  While E Do SS EndWhile
Call  ::=  F(E*)
E     ::=  Variable | Constant | Op(E*) | Call
Op    ::=  + | − |× | /
```

Fig. 4 The syntax of the While-language.

2.1 While-Language

We introduce an imperative While-language. The syntax is given in Fig. 4, the semantics is as in Pascal. A While-program is a sequence of statements. Each statement is either an assignment, a conditional, a while loop, a function call, or a skip statement. For simplicity of the technical presentation, we assume a call-by-value semantics for functions, and we treat all functions as primitive operations and assume they are free of side effects.

2.2 Single Static Assignment

This subsection summarizes the single static assignment form (SSA) [17]; readers familiar with SSA form may only want to look at the format of a while loop (Fig. 8). SSA form is a program representation in which there is only one assignment to each variable in the program and every use of a variable is defined by such an assignment. SSA form is important for program optimization.

Consider the following example:

```
While i<100 Do
    x:= 1; ...x...;
    x:=i+2; ...x...;
    i:=i+1;
EndWhile
```

There are two assignments to variable x inside the loop. After one iteration variable x in assignment "x:= 1" becomes invariant, but we can not move the assignment out of the loop because the value of x in the first use after the assignment would become equal to the value of x defined in assignment "x:=i+2" which is not correct.

The purpose of the SSA form is to represent data flow properties of a program in a normalized form. Many compilers use SSA or intermediate representation where a program is transformed into SSA form, optimized, and then transformed back to the original syntax. We follow the same approach.

Let us summarize the SSA form. First, the variable on the left-hand side of each assignment is given a unique name, and all of its uses are renamed correspondingly (Fig. 5). Second, if the use of a variable x can be reached by two or more definitions, which maybe

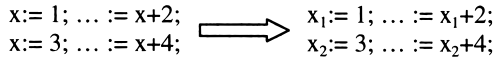


Fig. 5 Straight-line code and its SSA form.

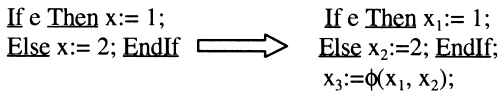


Fig. 6 An if-statement and its SSA form.

```
While i≤0 Do
  w:=w+10^3×x^3+10^2×x^2+10×x+1;
  x:=u+1;
  If even(i) Then y:=z+1; Else y:=v; EndIf;
  y:=x;
  If z>0 Then y:=1; If z>v Then y:=z+1; EndIf
  EndIf;
  If z>1000 Then y:=y+1; Else y:=i; EndIf
  u:=z–1;
  z:=v+1;
  i:=i+1;
EndWhile.
```

Fig. 7 An original loop.

```
While [ i1:=ϕ(i0, i2); x1:=ϕ(x0, x2); y1:=ϕ(y0, y12);
        u1:=ϕ(u0, u2); z1:=ϕ(z0, z2); w1:=ϕ(w0, w2);
      ] i1≤0 do
  w2:=w1+10^3×x1^3+10^2×x1^2+10×x1+1;
  x2:=u1+1;
  If even(i1) Then y2:=z1+1; Else y3:=v; EndIf;
  y4:=ϕ(y2, y3);
  y5:=x2;
  If z1>0 Then y6:=1;
              If z1>v Then y7:=z1+1; EndIf;
              y8:=ϕ(y6, y7);
  EndIf;
  y9:=ϕ(y5, y8);
  If z1>1000 Then y10:=y9+1; Else y11:=i1; EndIf;
  y12:=ϕ(y10, y11);
  u2:=z1–1;
  z2:=v+1;
  i2:=i1+1;
EndWhile.
```

Fig. 8 The SSA form of the loop in Fig. 7.

the case after a conditional, a special form of assignment called a ϕ-function, is added at the join point. The operands of the ϕ-function indicate which assignments to x reach the join point. Subsequent uses of x become uses of ϕ-function value. We call the values assigned by ϕ-function as virtual variables. This is illustrated in Fig. 6.

An efficient algorithm that converts a program into SSA form and works essential linear in the size of the original program has been proposed in [6]. The following example (Fig. 7) briefly explains the transformation. We are not interested in what the program does, only in illustrating the SSA form.

Using SSA technique, the loop is transformed into SSA form (Fig. 8). Note that we assume any loop in SSA form is in the form of While [ss_1] e Do ss_2 EndWhile, where ss_1 is a series of inserted assignments which should be executed before testing the entry condition and ss_2 is the body of the loop after SSA transformation.

3. Variable Dependency Graph

The loop transformation we want to perform depends on the loop quasi-invariant variables and the unfolding length of loops. Before we define these notions, we introduce two variable dependency relations and variable dependency graph to perform our quasi-invariance analysis. First, we assume a loop contains only assignment, then we extend our discussion to conditionals and nested loops. Here and in the remainder of this paper, we assume that all source programs are represented in SSA form.

3.1 Variable Dependency Graph

For any assignment v:=e, we assume that the value of v directly depends on all variables used in e. We can define a variable dependency relation as follows.

Definition 1 ($\angle$ *relation*): Let o be a loop, x, y be two variables assigned to inside o. If the value of x depends on that of y, then the dependency relation between x and y, is denoted by $x \angle y$ (called $\angle$ *relation*).

Within relation $\angle$, we can distinguish another variable dependency relation.

Definition 2 ($\lhd$ *relation*): Let o be a loop, x, y be two variables defined inside o, and $x \angle y$. If y is assigned to inside o after being used by x inside o, then the dependency relation between x and y, is denoted by $x \lhd y$ (called $\lhd$ *relation*).

To formalize loop quasi-invariance, we introduce a directed graph called *variable dependency graph*.

Definition 3 (*Variable dependency graph*): Let o be a loop. The *variable dependency graph* (VDG) of o is a directed graph where $Node(o)=\{x \mid$ variable x is defined in o$\}$ and $Edge(o)=\{x \rightarrow y \mid (y \angle x) \wedge \neg(y \lhd x)\} \cup \{x \twoheadrightarrow y \mid y \lhd x\}$.

For example, for the loop in Fig. 2, we have the $\angle$ relations: $\{x_2 \angle x_1, x_3 \angle x_2, y \angle x_1, y \angle x_2, y \angle x_3, y \angle y, t \angle t\}$, the $\lhd$ relations: $\{x_2 \lhd x_1, x_3 \lhd x_2, y \lhd y, t \lhd t\}$, and the VDG in Fig. 9, where relations $\angle$ and $\lhd$ are indicated by thin and thick edges, respectively. Note that there is only one assignment to one variable in Fig. 2, for concision we ignore the SSA form of Fig. 2 but only use the original form of Fig. 2.

3.2 Loop Quasi-Invariant Variable

Based on VDG, we now give a formal definition of loop

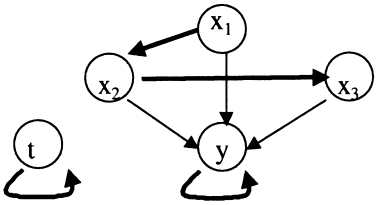


Fig. 9 The variable dependency graph of Fig. 2.

quasi-invariant (LQIV) variable.

Definition 4 (*LQIV*): Let o be a loop. For any node x on the VDG of o, variable x is loop quasi-invariant (*LQIV*) variable of loop o, if among all the paths ending in x, there is no path which contains a node that is a node on a circular path.

Furthermore, we can give a formal definition of invariant length of LQIV variable.

Definition 5 (*Invariant length*): Let o be a loop. For any LQIV node x (x is a LQIV variable of o) on the VDG of o, the *invariant length* of x $IL(x, o) =_{def}$ 1+max{n | n=the number of ➔ edges on a path ending in node x}.

For any LQIV variable x of a loop o, IL(x, o) means that x will be turned into loop-invariant variable after IL(x, o) iterations of o.

Proposition 1: Let o be a loop and x be LQIV of o. If $x^{(n)}$ is used to represent the value of x after n iterations of o, then $\forall n \geq IL(x, o).(x^{(n+1)} = x^{(n)})$.

The proposition is proven by induction over n; proof omitted. Based on the invariant lengths of the LQIV variables inside a loop, we can directly infer how many times the loop should be unfolded.

Definition 6 (*Unfolding length*): Let o be a loop. If there exists LQIV variable inside o, then the *unfolding length* of o $UL(o) =_{def} \max\{IL(x, o) \mid$ x is LQIV and not a virtual variable of o$\}$.

For Example, in the program of Fig. 2, we have LQIV variables x_1, x_2, x_3 and $IL(x_1, o)=1$, $IL(x_2, o)=2$, $IL(x_3, o)=3$, $UL(o)=3$.

Virtual variables are newly inserted variables and they will be removed at unfolding time. Therefore, the invariant lengths of virtual variables are ignored in the definition of the unfolding length.

According to Definition 4, there exists no circular path ending in a LQIV node x. Therefore, the number of edges in all paths ending in x is finite, and the invariant length of x is finite. This ensures the termination of the analysis and loop unfolding. After a loop o is unfolded UL(o) times, all LQIV variables in the residual loop of o become invariant and can be removed from the residual loop.

4. Conditionals in Loops

The variable dependency relation defined above is induced by assignment statements. As in other static analysis techniques, conditionals induce new variable dependencies. New dependencies are induced between the variables used in the test of a conditional and the variables defined in the branches of the conditional. We distinguish two cases:

CASE 1: If op($y_1, \ldots, y_n$) Then ... $x_i := \ldots$; ...
Else ... $x_j := \ldots$; ... EndIf;
$x_k := \phi(x_i, x_j)$;

Whether the assignment to x_i and x_j can be removed from the conditional depends on the variables used in expression op($y_1, \ldots, y_n$). If one of the variables $y_1, \ldots, y_n$ is not LQIV, which means that the value of op($y_1, \ldots, y_n$) may change at run time, none of the assignments to x_i and x_j can be removed even if they would otherwise be LQIV variables. This kind of dependency relation between x_i, x_j and $y_1, \ldots, y_n$ can be expressed by adding dependency relations between x_i, x_j and $y_1, \ldots, y_n$: $x_j \angle y_1, \ldots, x_j \angle y_n$; $x_i \angle y_1, \ldots, x_i \angle y_n$. We say, these relations are induced by the conditional.

CASE 2: $x_j := \ldots$;
If op($y_1, \ldots, y_n$) Then ... $x_i := \ldots$; ...
Else ... EndIf;
$x_k := \phi(x_i, x_j)$;

This is similar to CASE 1, except that the assignment to x_j is laid out of the conditional but not in one branch. For the same reason as in CASE 1, we need to establish the dependency relations between x_i, x_j and $y_1, \ldots, y_n$: $x_i \angle y_1, \ldots, x_i \angle y_n$; $x_j \angle y_1, \ldots, x_j \angle y_n$.

Before x_j becomes loop-invariant, assignment $x_i := \ldots$ can not be removed. (otherwise x_k will always be equal to x_j, which is not correct.) Therefore we add dependency relation $x_i \angle x_j$. (we also say, this relation is induced by the conditional.)

After we get the variable dependency relations induced by the conditionals in a loop, we must transfer the same relations from virtual variables to their operands via the corresponding ϕ-function. This is necessary because virtual variables will be removed afterwards, (as discussed in Sect. 6.) and the dependency relations (from virtual variables to other variables) induced by conditionals are actually the ones between their operands and other variables. Moreover, if the operands of virtual variables are also virtual variables, the dependency relations have to be transferred further. We now define how the dependency relations induced by conditionals in a loop o are transferred via virtual variables. For any a dependency relation $x \angle y$ directly induced by the conditionals like CASE 1 and CASE 2, we use a function called CS to evaluate the above-mentioned transitive closure of relation $x \angle y$.

$$CS_o(x, y) =_{def} \begin{cases} \{\ \} & \text{if x and y are identical} \\ CS_o(x_1, y) \cup CS_o(x_2, y) & \text{if x is a virtual variable of o} \\ & \quad \text{defined by } x := \phi(x_1, x_2) \\ \{x \angle y\} & \text{otherwise} \end{cases}$$

Let Cond_Rel(o) be the set of variable dependen-

cies directly induced by the conditionals in loop o, (as defined in CASE 1 and CASE 2.) and Vars(o) be the set of all variables defined in o. Then we define VS(o) (the set of all the relations derived from Cond_Rel(o), including Cond_Rel(o)) as follow:

$$VS(o) =_{def} \bigcup_{x\in Vars(o)} \left(\bigcup_{y\in Vars(o)\wedge x\angle y\in Cond\_Rel(o)} CS_o(x,y) \right)$$

For example, let a loop contain the conditional:

x_1:=1;
If i>j Then If k>5 Then x_2:=2; Else x_3:=3; EndIf;
x_4:= $\phi(x_2, x_3)$; EndIf;
x_5:= $\phi(x_1, x_4)$;

We can derive all the dependency relations induced by the conditional as follows:

$x_1\angle$i, $x_1\angle$j;
$x_2\angle$i, $x_2\angle$j, $x_2\angle$k, $x_2\angle x_1$
$x_3\angle$i, $x_3\angle$j, $x_3\angle$k, $x_3\angle x_1$;
$x_4\angle$i, $x_4\angle$j, $x_4\angle x_1$

We now present the VDG (Fig. 10) of the program in Fig. 8, where white nodes indicate variant variables, grey nodes indicate LQIV variables, and names of virtual variables are written italic. Relations $\angle$ and $\lhd$ are shown as thin and thick edges, respectively. According to Fig. 10, LQIV variables and their invariant lengths can be easily derived. LQIV variables: {z_1, z_2, x_1, x_2, y_5, y_6, y_7, y_8, y_9, y_{10}, u_1, u_2}; IL(z_1)= 2, IL(z_2)= 1, IL(x_1)= 4, IL(x_2)= 3, IL(y_5)= 3, IL(y_6)= 3, IL(y_7)= 3, IL(y_8)= 3, IL(y_9)= 3, IL(y_{10})= 3, IL(u_1)= 3, IL(u_2)= 2. Because x_1 is a virtual variable, unfolding length UL(o)= 3.

For any loop in the form of While [ss_1] e Do ss_2 EndWhile, only the variables defined in ss_1 depend on the variables that will be statically assigned afterwards. For example, in Fig. 8, y_1 depends on y_0 and y_{12}, where y_{12} will be assigned afterwards.

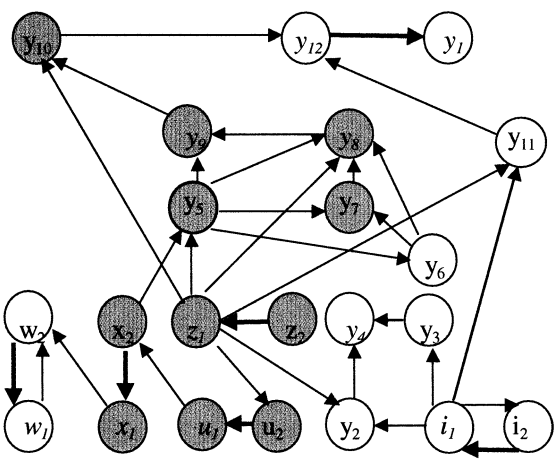


Fig. 10 The variable dependency graph of Fig. 8.

5. An Algorithm for LQIV Analysis

LQIV analysis consists of two phases. The first phase detects the dependency relations between the variables defined in a loop, the second phase finds all LQIV variables, computes their invariant lengths and the loop's unfolding length.

The second phase is the heart of the analysis. The algorithm for the second phase is given below. It is based on the classical algorithms by Warshall [22] and Floyd [8]. The time complexities of Warshall algorithm and Floyd algorithm are $O(n^3)$ in the worst case where n is the number of variables in the given loop. The first phase is not shown here; it can be done while parsing a program.

We assume that there are n variables (denoted with $v_1,\ldots,v_n$) in a loop o, the relations $\angle$ and $\lhd$ between $v_1,\ldots,v_n$ have been stored in a Boolean n×n matrix $R_\angle$ and an integer n×n matrix $R_\lhd$, respectively, where for any two variables v_i and v_j, if $v_i\angle v_j$ then $R_\angle[i,j]$= 1 else $R_\angle[i,j]$= 0 and if $v_i\lhd v_j$ then $R_\lhd[i,j]$= 1 else $R_\lhd[i,j]$= 0.

Algorithm (for LQIV analysis to singular loop):
Input: $R_\angle$, $R_\lhd$.
Output: the set (Lqivs) of all the LQIV variables, the invariant lengths of these LQIV variables, and the unfolding length (UL) of o.
Begin
{ Based on Warshall algorithm, computing all the LQIV variables assigned to inside loop o. }
$R1_\angle$:=$R_\angle$;
for i=1 to n do
for j=1 to n do
if $R1_\angle[j,i]$=1
then for k=1 to n do
$R1_\angle[j,k]$:=$R1_\angle[j,k]\vee R1_\angle[i,k]$;
endfor
endif
endfor endfor
Lqivs:={ i | $\forall i_{(1\le i\le n)}\forall j_{(1\le j\le n)}.(R1_\angle[i,j]$=1 $\rightarrow R1_\angle([j,j]\neq 1)$};
{ Based on Floyd algorithm, computing the invariant lengths of all LQIV variables and the unfolding length of o. Remark: all the variables in Lqivs and all dependency relations among these variables can be viewed as a directed graph, where any node represents a LQIV variable and any edge between two nodes represents $\angle$ or $\lhd$ relation between the two variables. Therefore, the invariant length of any LQIV variable is actually equivalent to the longest path ending in the variable, if the length of any $\angle$ edge is defined as 0 and that of any $\lhd$ edge is defined as 1. }
$R1_\lhd$:=$R_\lhd$;
for any i∈Lqivs do
for any j∈Lqivs do
for any k∈Lqivs do
if $R1_\angle[j,i]$=1 ∧ $R1_\angle[i,k]$=1 ∧ $R1_\angle[j,k]$=1

```
        then if R1◁[j, i]+R1◁[i, k]>R1◁[j, k]
             then R1◁[j, k]:=R1◁[j, i]+R1◁[i, k]; endif;
        endif;
    endfor endfor endfor
    for any i∈Lqivs do IL[i]:=0; endfor;
    for any i∈Lqivs do
        IL[i]:=1+max{ R1◁[j, i] | j∈Lqivs };
    endfor;
    UL:=max{ IL[i] | i∈Lqivs∧i∉virtual variables };
End.
```

6. Loop Transformation

The purpose of loop transformation is to remove all LQIV variables from a given loop. The unfolding length computed by the algorithm in the previous section tells us how many iterations are needed before all LQIV variables become invariant. The loop transformation is based on decomposing the loop into two loops where the first loop iterates UL(o) times to compute the values of the LQIV variables and the second loop is the remained loop after code motion. In the process of code motion, variable renaming is necessary.

6.1 A Note on Variable Renaming

When generating code for a loop represented in SSA form, we have to remove all assignments to virtual variables and to rename their uses correspondingly. Below we define how to rename variables.

For any virtual variable x, (assume it is defined as $x:=\phi(y,z)$.) all the uses of x are actually the uses of y or z. If $x:=\phi(y,z)$ is removed then all the uses of x will become undefined. Therefore, we must use same name for x, y and z so that all uses of x will naturally become the uses of y and z. We are going to discuss the problem in the following 2 cases.

CASE 1: y or z is also virtual variable.

When y or z is a virtual variable, we have the same situation as with x and its operands should also be renamed using the same name. The process continues recursively until no new virtual variables are met.

CASE 2: x is an operand of another virtual variable.

When x is an operand of another virtual variable (e.g., $w:=\phi(v,x)$), by the same reason, w, v and x should also be renamed using the same name. The process continues recursively until no new virtual variables are met.

We now define which variables should be renamed using the same name in a loop o. For this purpose, we define function RE which determines the set of variables that should be renamed using the same name.

$$\mathrm{RE_o(x)} =_{\mathrm{def}} \begin{cases} \{x\} & \text{if x is not a virtual variable} \\ \{\mathrm{x}\}\cup\mathrm{RE_o}(\mathrm{x_1})\cup\mathrm{RE_o}(\mathrm{x_2})\cup\mathrm{RE'_o}(\mathrm{x}) & \\ \qquad \text{if x is a virtual variable defined by } \mathrm{x}:=\phi(\mathrm{x_1},\mathrm{x_2}) & \end{cases}$$

$$\mathrm{RE'_o(x)} =_{\mathrm{def}} \begin{cases} \{\ \} & \text{if x is not an operand of another virtual variable} \\ \{\mathrm{y}\}\cup\mathrm{RE_o}(\mathrm{z})\cup\mathrm{RE'_o}(\mathrm{y}) & \\ \qquad \text{if x is an operand of another virtual variable y defined by } \mathrm{y}:=\phi(\mathrm{x},\mathrm{z}) \text{ or } \mathrm{y}:=\phi(\mathrm{z},\mathrm{x}) & \end{cases}$$

For example, in Fig. 8 there are five ϕ-function assignments to y: $y_1:=\phi(y_0,y_{12})$, $y_4:=\phi(y_2,y_3)$, $y_8:=\phi(y_6,y_7)$, $y_9:=\phi(y_5,y_8)$, $y_{12}:=\phi(y_{10},y_{11})$. The variables in set $\mathrm{RE_o}(y_1)=\mathrm{RE_o}(y_{12})=\{y_1, y_0, y_{12}, y_{10}, y_{11}\}$, $\mathrm{RE_o}(y_4)=\{y_4, y_2, y_3\}$, $\mathrm{RE_o}(y_8)=\mathrm{RE_o}(y_9)=\{y_8, y_6, y_7, y_9, y_5\}$ should be renamed using the same name respectively (e.g., y, Y_2, Y_3). Similarly, the variables in each of the sets $\{x_1, x_0, x_2\}$, $\{i_1, i_0, i_2\}$, $\{u_1, u_0, u_2\}$, $\{z_1, z_0, z_2\}$, and $\{w_1, w_0, w_2\}$ should be renamed using the same names (e.g., x, i, u, z, w).

```
If i≤0      /* Loop 1 */
Then
  w:=w+10³×x³+10²×x²+10×x+1;
  x:=u+1;
  If even(i) Then Y2:=z+1; Else Y2:=v; EndIf;
  Y3:=x;
  If z>0
  Then Y3:=1; If z>v Then Y3:=z+1; EndIf; EndIf
  If z>1000 Then y:=Y3+1; Else y:=i; EndIf;
  u:=z–1; z:=v+1; i:=i+1;
  If i≤0    /* Loop 2 */
  Then
    w:=w+10³×x³+10²×x²+10×x+1;
    x:=u+1;
    If even(i) Then Y2:=z+1; Else Y2:=v; EndIf;
    Y3:=x;
    If z>0 Then Y3:=1;
                If z>v Then Y3:=z+1; EndIf;
    EndIf;
    If z>1000 Then y:=Y3+1; Else y:=i; EndIf;
    u:=z–1; i:=i+1;
    If i≤0      /* Loop 3 */
    Then
      w:=w+10³×x³+10²×x²+10×x+1;
      x:=u+1;
      If even(i) Then Y2:=z+1; Else Y2:=v; EndIf;
      Y3:=x;
      If z>0 Then Y3:=1;
                  If z>v Then Y3:=z+1; EndIf;
      EndIf;
      If z>1000 Then y:=Y3+1; Else y:=i; EndIf;
      i:=i+1;
      While i≤0 Do   /* Residual loop */
        w:=w+10³×x³+10²×x²+10×x+1;
        If even(i) Then Y2:=z+1;
        Else Y2:=v; EndIf;
        If z>1000 Then Skip; Else y:=i; EndIf;
        i:=i+1;
      EndWhile
    EndIf EndIf EndIf
```

Fig. 11 The residual code (by unfolding) of the loop in Fig. 8.

Table 2 The comparison of the runtime of the original loop in Fig. 8 and the residual code in Fig. 11.

System	Gateway E5250 Pentium II xeon×2 Speed: 450MHZ Memory: 1GB Visual C++ 6.0		Sun ULTRA 5 SunOS 5.6 Speed: 270MHZ Memory: 192MB Gcc 2.7	
	Runtime	Speed up	runtime	speed up
Source	9sec	1	52sec	1
Result	3sec	3	30sec	1.73

6.2 Loop Transformation

Here, we use the loop in Fig. 8 to express how to unfold a loop, and a formalization using variable renaming is given in Appendix 2. By LQIV code motion using unfolding, we obtain the residual code in Fig. 11. In Fig. 11, all the LQIV variables have turned into loop-invariant variables and thus traditional loop-invariance code motion can be applied, e.g., expression $10^3 \times x^3 + 10^2 \times x^2 + 10 \times x + 1$ can be moved outside the residual loop.

Compared with the original loop, in the corresponding residual code, some new variables are introduced due to renaming and thus the other parts of the program containing the original loop must be modified for consistence. For any variable (e.g., x) defined in an original loop, there must be an assignment like $x_1 := \phi(x_0, x_n)$ in the corresponding SSA form and only x_1 is visible to the outside of the loop, where, x_0 refers to the x assigned to outside the loop and x_n refers to the x assigned to finally inside the loop. According to variable renaming, x_1, x_0, x_n will be renamed with same name. If we use x for the name, then the variable consistence problem above can be easily solved. However, the other new variables (e.g., Y_2 and Y_3) have to be declared in residual code.

To give an indication of the speedup, Table 2 shows the runtime of the original loop and the residual code, where $i = -10^8$ and $x = y = z = u = v = w = 10^3$.

7. About Nested Loops

In the previous section we considered the treatment of conditionals inside loops. Nested loops can be handled in a similar fashion. Here we give just a short description because the methods introduced so far can also handle the analysis of nested loops.

From the viewpoint of an outer loop o, only linear dependencies between the variables of an inner loop q are interesting. A variable which is variant in q may well be quasi-invariant in o (e.g., an inductive variable of q is variant in q, but can be quasi-invariant with respect to o). This means we are not interested in cyclic dependencies local to inner loops when analyzing o. Conceptually, this can be done by building a separate VDG for each loop o from a program where all inner loops of o are viewed as conditionals with empty else-branch (CASE 2 in Sect. 4.), thereby avoiding the representation of cycles local to the inner loops. In practice, more efficient construction methods may exist.

```
While t<TMAX Do
  x3:=g3(x2, true); x2:=g2(x1, true); x1:=true;
  y:=f(g(x1, x2, x3), y); t:=t+1;
EndWhile
```

(a) The result specializing the source program in Fig. 2

```
If t<TMAX
Then x3:=g3(x2, true); x2:=g2(x1, true);
     y:=f(g(true, x2, x3), y); t:=t+1;
     If t<TMAX
     Then x3:=g3(x2, true);
          y:=f(g(true, true, x3), y); t:=t+1;
          If t<TMAX
          Then y:=f(true, y); t:=t+1;
               While t<TMAX Do
                    y:=f(true, y); t:=t+1;
               EndWhile
          EndIf EndIf EndIf
```

(b) The result specializing the source program in Fig. 3

Fig. 12 Offline specialization with pointwise BTA.

8. Discussion of Applications

After LQIV code motion, all LQIV variables inside an original loop have been moved outside its residual loop; they are now invariant inside the loop. This means any static analysis can in principle benefit from the loop optimization. To illustrate the effect on a concrete program analysis, consider offline partial evaluation, which is controlled by a separate binding-time analysis (BTA). (We assume the readers are familiar with partial evaluation; for more details see the book [10])

Let us consider a pointwise BTA where the information computed by the analysis ("static," "dynamic") is merged at each program point. In addition, let the BTA dynamize static values under dynamic control as done in the TEMPO specializer for C [9]. To compare the effect of LQIV code motion on the analysis, we specialize the loop in Fig. 2 and its residual program in Fig. 3.

Consider specializing the programs with respect to the static values $c_1 = c_2 = c_3 = \text{true}$. The residual programs are shown in Figs. 12 (a) and (b) respectively. The difference should be obvious. In particular, observe that static value of g=true is inlined in the residual loop Fig. 12 (b) (possibly triggering further static computations in a loop). This is possible because all computations of LQIV variables have been moved out of the loop. Since all LQIV variables are invariant in the loop, their static values be treated as constants in-

side the loop even though the BTA strategy dynamizes static values under dynamic control.

9. Related Work

Code motion is a well-known technique in compiler optimization [1]. Code motion and loop-invariance have many uses in the optimization of source programs written in high-level languages as well as target programs written in assembly language (e.g., code generated for indexing multi-dimensional arrays). A comprehensive survey of different loop transformations and other dataflow based compiler optimizations can be found in [3]. However, none of them considers the combination of code motion, unfolding and loop quasi-invariance.

Many optimization techniques can be formalized conveniently using single static assignments, including the elimination of partial redundancies [14], constant propagation [5], [11], [17], and code motion [6]. We followed the same approach to express our loop optimization technique.

The notion of quasi-invariance grew out of work on automatic program transformation, in particular partial evaluation where the optimization of loops is of central importance (e.g., [2], [8], [9], [12]). Our technique statically determines a finite fixed point of computations induced by assignments, loops and conditionals and computes the optimal unfolding length (e.g., to make maximal use of known information while ensuring termination as shown in Sect. 8). LQIV code motion may support other static termination analyses, e.g., techniques for detecting static bounded variation [8], and other generalization techniques.

10. Conclusion and Future Work

In this paper we introduced LQIV code motion, which is based on a static analysis for loop quasi-invariance. The notion of loop quasi-invariance is similar to the traditional notion of loop-invariance but allows for a finite number of iterations before computations in a loop become invariant.

Our analysis determines the optimal unfolding length needed to remove all quasi-invariant computations from a loop while avoiding over-unfolding. The residual loop becomes smaller and faster. Our technique is well-suited as supporting transformation in compilers, partial evaluators, and other program transformers. It may not result in dramatic speedups in large practical applications, but has the potential to increase the accuracy of program analyses and to trigger stronger program optimizations which is of central importance in almost any kind of program transformation. The algorithms presented in this paper use the infrastructure already present in many compilers, such as control flow graphs and single static assignments. Thus they do not require fundamental changes to existing systems. To the best of our knowledge quasi-invariant code motion has not been considered earlier for program optimization.

It was shown [11] that standard constant folding can provide useful speed up for large hand-written programs, so loop quasi-invariance optimization may also be beneficial for practical applications written by hand. However, more should be known about the effect of quasi-invariance code motion on optimizing compilers. This and the application of our technique to larger practical programs will be a topic for future work.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley, 1986.
[2] L.O. Andersen, "Program analysis and specialization for the C programming language," Ph.D. Thesis, DIKU Report 94/19, Dept. of Computer Science, University of Copenhagen, 1994.
[3] D.F. Bacon and S.L. Graham, "Compiler transformations for high-performance computing," ACM Computing Surveys, vol.26, no.4, pp.345–420, Dec. 1994.
[4] R. Bodik, R. Gupta, and M.L. Soffa, "Complete removal of redundant expressions," Proc. ACM Conf. on Programming Language Design and Implementation, pp.1–14, ACM Press, 1998.
[5] M.A. Bulyonkov and D.V. Kochetov, "Practical aspects of specialization of Algol-like programs," eds. O. Danvy, R. Glück, and P. Thiemann, Partial Evaluation. Proceedings. LNCS, vol.1110, pp.17–32, Springer-Verlag, 1996.
[6] R. Cytron and J. Ferrante, "Efficiently computing static single assignment form and the control dependence graph," ACM TOPLAS, vol.13, no.4, pp.451–490, Oct. 1991.
[7] R. Cytron, A. Lowry, and F.K. Zadeck, "Code motion of control structures in high-level languages," Conference Record of the 13th ACM Symposium on Principle of Programming Languages, pp.70–85, ACM Press, 1986.
[8] R.W. Floyd, "Algorithm 97: shortest path," Comm. ACM 5:6, p.345, 1962.
[9] A.J. Glenstrup and N.D. Jones, "BTA algorithms to ensure termination of off-line partial evaluation," eds. D. Bjørner, M. Broy, and I.V. Pottosin, Perspectives of System Informatics. Proceedings. LNCS, vol.1181, pp.273–284, Springer-Verlag, 1996.
[10] A. Glenstrup, H. Makholm, and J.P. Secher, "C-Mix: specialization of C programs," eds. J. Hatcliff, T. Mogensen, and P. Thiemann, Partial Evaluation: Practice and Theory, LNCS, vol.1706, pp.108–153, Springer-Verlag, 1999.
[11] R. Gupta, D.A. Berson, and J.Z. Fang, "Path profile guided partial redundancy elimination using speculation," IEEE International Conf. Computer Languages, pp.230–239, IEEE Society Press, 1998.
[12] L. Hornof and J. Noyé, "Accurate binding-time analysis for imperative languages: Flow, context, and return sensitivity," Symposium on Partial Evaluation and Semantics-Based Program Manipulation, pp.63–73, ACM Press, 1997.
[13] N.D. Jones, C.K. Gomard, and P. Sestoft, Partial Evaluation and Automatic Program Generation, Prentice Hall, 1993.
[14] R. Metzger and S. Stroud, "Interprocedural constant propagation: An empirical study," ACM Letters on Programming Languages and Systems, 2(1-4), pp.213–232, 1993.
[15] U. Meyer, "Techniques for partial evaluation of imperative language," Symposium on Partial Evaluation and

Semantics-Based Program Manipulation, Sigplan Notices, vol. 26, no.9, pp.94–105, ACM Press, Sept. 1991.
[16] F. Nielson, H.R. Nielson, and C. Hankin, Principles of Program Analysis, Springer-Verlag, 1999.
[17] B.K. Rosen, M.N. Wegman, and F.K. Zadeck, "Global value numbers and redundant computations," Conference Record of the 15th ACM Symposium on Principles of Programming Languages, pp.12–27, ACM Press, 1988.
[18] L.T. Song and Y. Futamura, "Quasi-invariant variable and loop unfolding," The 15th National Conference of Software Society of Japan, B6-2, pp.221–224, Sept. 1998.
[19] B. Steffen, "Property oriented expansion," Symposium on Static Analysis, LNCS 1145, pp.22–41, Springer-Verlag, 1996.
[20] B. Steffen, J. Knoop, and O. Rüthing, "The value flow graph: A program representation for optimal program transformations," ed. N.D. Jones, ESOP'90, LNCS 432, pp.389–405, Springer-Verlag, 1990.
[21] S. Warshall, "A theorem on Boolean matrices," J. ACM, vol.9, no.1, pp.11–12, Jan. 1962.
[22] M.N. Wegman and F.K. Zadeck, "Constant propagation with conditional branches," ACM TOPLAS, vol.13, no.2, pp.181–210, April 1991.
[23] H. Zima and B. Chapman, Supercompiler for Parallel and Vector Computers, Frontier, Series, ACM Press, 1990.

Appendix 1: The definitions of g_1, g_2, g_3, g, f, t and T

$$g_1(c_1) = c_1,$$
$$g_2(x_1, c_2) = (\neg x_1 \wedge c_2) \vee (x_1 \wedge c_2),$$
$$g_3(x_1, x_2, c_3) = (\neg x_1 \wedge x_2 \wedge c_3) \vee (x_1 \wedge \neg x_2 \wedge c_3) \vee (x_1 \wedge x_2 \wedge c_3),$$
$$g(x_1, x_2, x_3) = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_3),$$
$$f(x, y) = x \wedge y,$$
$$t = 1,$$
$$T = 10^8.$$

Appendix 2: The algorithm of unfolding

In the following algorithm, we assume that there is a loop in the form of while [ss_1] e do ss_2 endwhile, *Rvs* indicates the set of all the related variable sets, *LQivs* indicates the set of all the LQIVs, *ILen* indicates the set of the invariant lengths of LQIVs, *ULen* indicates the unfolding length.

The algorithm is presented by some rules as follow and the initial call is $(0, LQivs, ILen, ULen, Rvs) \vdash_{\text{while}}$: while [$ss_1$] e do ss_2.

$$\frac{\begin{array}{c} t < ULen \qquad (Rvs) \vdash_{E}: e \Rightarrow e' \\ (t, LQivs, ILen, Rvs) \vdash_{ss}: ss_2 \Rightarrow ss_2' \\ (t{+}1, LQivs, ILen, ULen, Rvs) \vdash_{\text{while}}: \underline{\text{While}}\,[ss_1]\,e\,\underline{\text{Do}}\,ss_2 \Rightarrow ss' \end{array}}{(t, LQivs, ILen, ULen, Rvs) \vdash_{\text{while}}: \underline{\text{While}}\,[ss_1]\,e\,\underline{\text{Do}}\,ss_2 \Rightarrow \underline{\text{If}}\ e\ \underline{\text{Then}}\ ss_2'; ss'\ \underline{\text{EndIf}}}$$

$$\frac{\begin{array}{c} t = ULen \qquad (Rvs) \vdash_{E}: e \Rightarrow e' \\ (t, LQivs, ILen, Rvs) \vdash_{ss}: ss_2 \Rightarrow ss_2' \end{array}}{(t, LQivs, ILen, ULen, Rvs) \vdash_{\text{while}}: \underline{\text{While}}\,[ss_1]\,e\,\underline{\text{Do}}\,ss_2 \Rightarrow \underline{\text{While}}\ e'\ \underline{\text{Do}}\ ss_2'}$$

$$\frac{\begin{array}{c} (t, LQivs, ILen, Rvs) \vdash_{s}: s \Rightarrow s' \\ (t, LQivs, ILen, Rvs) \vdash_{ss}: ss \Rightarrow ss' \end{array}}{(t, LQivs, ILen, Rvs) \vdash_{ss}: s;\ ss \Rightarrow s';\ ss'}$$

$$\frac{}{(t, LQivs, ILen, Rvs) \vdash_{s}: v := \phi\text{-function} \Rightarrow \text{skip}}$$

$$\frac{e \neq \phi\text{-function} \qquad v \in LQivs \qquad t > ILen(v)}{(t, LQivs, ILen, Rvs) \vdash_{s}: v := e \Rightarrow \text{skip}}$$

$$\frac{\begin{array}{c} e \neq \phi\text{-function} \qquad v \in LQivs \wedge v \in \text{a RV of } Rvs \\ t \leq ILen(v) \qquad (Rvs) \vdash_{E}: e \Rightarrow e' \end{array}}{(t, LQivs, ILen, Rvs) \vdash_{s}: v := e \Rightarrow v_{NO} := e'}$$

/* where NO is a unique number assigned to the RV */

$$\frac{\begin{array}{c} e \neq \phi\text{-function} \qquad v \in LQivs \wedge v \notin \text{any a RV of } Rvs \\ t \leq ILen(v) \qquad (Rvs) \vdash_{E}: e \Rightarrow e' \end{array}}{(t, LQivs, ILen, Rvs) \vdash_{s}: v := e \Rightarrow v := e'}$$

$$\frac{\begin{array}{c} (Rvs) \vdash_{E}: e \Rightarrow e' \\ (t, LQivs, ILen, Rvs) \vdash_{ss}: ss_1 \Rightarrow ss_1' \\ (t, LQivs, ILen, Rvs) \vdash_{ss}: ss_2 \Rightarrow ss_2' \end{array}}{(t, LQivs, ILen, Rvs) \vdash_{s}: \underline{\text{If}}\ e\ \underline{\text{Then}}\ ss_1\ \underline{\text{Else}}\ ss_2\ \underline{\text{EndIf}} \Rightarrow \underline{\text{If}}\ e'\ \underline{\text{Then}}\ ss_1'\ \underline{\text{Else}}\ ss_2'\ \underline{\text{EndIf}}}$$

$$\frac{(Rvs) \vdash_{E}: e \Rightarrow e' \quad (t, LQivs, ILen, Rvs) \vdash_{ss}: ss \Rightarrow ss'}{(t, LQivs, ILen, Rvs) \vdash_{s}: \underline{\text{If}}\ e\ \underline{\text{Then}}\ ss\ \underline{\text{EndIf}} \Rightarrow \underline{\text{If}}\ e'\ \underline{\text{Then}}\ ss'\ \underline{\text{EndIf}}}$$

$$\frac{(Rvs) \vdash_{E}: e_i \Rightarrow e_i'}{(t, LQivs, ILen, Rvs) \vdash_{s}: p(e_1', \ldots, e_n') \Rightarrow p(e_1', \ldots, e_n')}$$

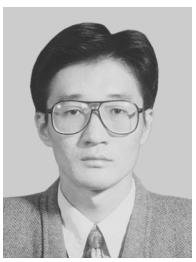
$$\frac{}{(Rvs) \vdash_{E}: \text{constant} \Rightarrow \text{constant}}$$

$$\frac{\mathrm{v} \in \text{a PV of } \mathit{Rvs}}{(\mathit{Rvs}) \vdash_{\mathrm{E}} \mathrm{v} \Rightarrow \mathrm{v}_{\mathrm{NO}}}$$

/* where NO is a unique number assigned to the RV */

$$\frac{\mathrm{v} \notin \text{any PV of } \mathit{Rvs}}{(\mathit{Rvs}) \vdash_{\mathrm{E}} \mathrm{v} \Rightarrow \mathrm{v}}$$

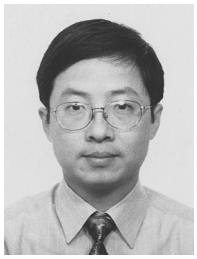
$$\frac{(\mathit{Rvs}) \vdash_{\mathrm{E}} \mathrm{e_i} \Rightarrow \mathrm{e'_i}}{(\mathrm{t}, \mathit{LQivs}, \mathit{ILen}, \mathit{Rvs}) \vdash_{\mathrm{s}} \mathrm{op}(\mathrm{e'_1}, \ldots, \mathrm{e'_n}) \Rightarrow \mathrm{op}(\mathrm{e'_1}, \ldots, \mathrm{e'_n})}$$



Litong Song is a Ph.D. student of Information and Computer Science at Waseda University. He received his BS in Computer Science from Nanjing University in 1987 and MS in Computer Science from Jilin University in 1990. His main research interests are partial evaluation and program transformation.

Yoshihiko Futamura is Professor of Department of Information and Computer Science at Waseda University. He is also the director of Software Production Technology Laboratory of the university. He obtained his diploma in mathematics from Hokkaido University in 1965, his MS from Harvard University in 1973 and his PhD from Hokkaido University in 1985. He entered Hitachi Central Research Laboratory in 1965 and moved to Waseda University in 1991. He was Visiting Professor at Uppsala University and Harvard University. He has worked on automatic generation of computer programs and programming methodology. He is the inventor of the Futamura Projections in partial evaluation and ISO8631 PAD.

Robert Glück is an associate professor of Computer Science at the University of Copenhagen. He received his M.Sc. and Ph.D. degrees in 1986 and 1991 from the University of Technology in Vienna, where he also worked as assistant professor. He received his Habiliation (venia docendi) in 1997. He was research assistant at the City University of New York and received twice the Erwin-Schrüdinger-Fellowship of the Austrian Science Foundation. Currently he is an Invited Fellow of the Japan Society for the Promotion of Science working at Waseda University in Tokyo. His main research interests are advanced programming languages, theory and practice of program transformation, and metaprogramming techniques.



Zhenjiang Hu is an assistant professor at the University of Tokyo. He received his BS and MS in computer science from Shanghai Jiao Tong University in 1988 and 1990 respectively, and his Ph.D. in information engineering from the University of Tokyo in 1996. His current research concerns functional programming, calculational program transformation systems and algorithm derivation.