Translation Validation of Optimizing Compilers

by

Yi Fang

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

Courant Institute of Mathematical Sciences

New York University

December, 2005

Advisor: Amir Pnueli

Abstract

There is a growing awareness, both in industry and academia, of the crucial role of formally verifying the translation from high-level source-code into low-level object code that is typically performed by an optimizing compiler. Formally verifying an optimizing compiler, as one would verify any other large program, is not feasible due to its size, ongoing evolution and modification, and possibly, proprietary considerations. *Translation validation* is a novel approach that offers an alternative to the verification of translator in general and compilers in particular: Rather than verifying the compiler itself, one constructs a validation tool which, after *every* run of the compiler, formally confirms that the target code produced in the run is a correct translation of the source program. This thesis work takes an important step towards ensuring an extremely high level of confidence in compilers targeted at EPIC architectures.

The dissertation focuses on the translation validation of *structure-preserving* optimizations, i.e., transformations that do not modify programs' structure in a major way, which include most of the global optimizations performed by compilers. The first part of the dissertation develops the theory of a correct translation, which provides a precise definition of the notion of a target program being a correct translation of a source program, and the method that formally establishes the correctness of structure preserving transformations based on computational induction. The second part of the dissertation describes a tool that applies the theory of the first part to the automatic validation of global optimizations performed by Intel's ORC compiler for IA-64 architecture. With minimal instrumentation from the compiler, the tool constructs "verification conditions" – formal theorems that, if valid, establish the correctness of a translation. This is achieved by performing own control-flow and data-flow analyses together with various heuristics. The verification condition are then transferred to an automatic theorem prover that checks their validity. Together with the theorem prover, the tool offers a fully automatic method to formally establish the correctness of each translation.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

There is a growing awareness, both in industry and academia, of the crucial role of formally proving the correctness of safety-critical portions of systems. Most verification methods focus on verification of specification with respect to requirements, and high-level code with respect to specification. However, if one is to prove that the high-level specification is correctly implemented in low-level code, one needs to verify the compiler which performs the translations. Verifying the correctness of modern optimizing compilers is challenging because of the complexity and reconfigurability of the target architectures, as well as the sophisticated analysis and optimization algorithms used in the compilers.

Formally verifying a full-fledged optimizing compiler, as one would verify any other large program, is not feasible, due to its size, evolution over time, and, possibly, proprietary considerations. *Translation Validation* is a novel approach that offers an alternative to the verification of translators in general and of compilers in particular. Using the translation validation approach, rather than verify the compiler itself one constructs a *validating tool* which, after every run of the compiler, formally confirms that the target code produced is a correct translation of the source program.

The introduction of new families of microprocessor architectures, such as the EPIC family exemplified by the Intel IA-64 architecture, places an even heavier responsibility on optimizing compilers. Compile-time dependence analysis and instruction scheduling is required to exploit instruction-level parallelism in order to compete with other architectures, such as the super-scalar class of machines where the hardware determines dependences and reorders instructions at run-time. As a result, a new family of sophisticated optimizations have been developed and incorporated into compilers targeted at EPIC architectures.

At first glance, the approach of translation validation is designed to ensure the correct functioning of compilers. But it also have impact on the testing process of compiler development. Since the output of compilers is unreadable, people can recognize a incorrect compilation only by observing an incorrect program execution. As compiler testers have observed, many times an “incorrect output” is caused not by the compiler itself, but by the errors in the program it compiles. As a result, much of the time and efforts in compiler testing is spent in isolating the bugs in the testing programs from those in the compiler. Since the translation validation approach checks the output of compilers directly, the error it detects is guaranteed to be a compilation error, and thus makes the compiler debugging process more efficient.

1.2 Related Works

This thesis research is an extension of the work on translation validation by Pnueli, Siegel and Singerman ([PSS98a]), which developed a tool for translation validation, CVT, that succeeded in automatically verifying translations involving approximately 10,000 lines of source code in about 10 minutes. The success of CVT critically depends on some simplifying assumptions that restrict the source and target to programs with a single external loop, and assume a very limited set of optimizations.

The work by Necula in [Nec00] covers some important aspects of my work. For one, it extends the source programs considered from single-loop programs to programs with arbitrarily nested loop structure. An additional important feature is that the method requires no compiler instrumentation at all, and applies various heuristics to recover and identify the optimizations performed and the associated refinement mappings. But the method used in Necula's work carries very limited information in between different check points, hence it cannot validate some common optimizations such as loop invariant code motion and strength reduction, which our approach can handle.

Another related work is credible compilation by Rinard and Marinov [RM00] which proposes a comparable approach to translation validation, where an important contribution is the ability to handle pointers in the source program. However, the method proposed there assumes *full* instrumentation of the compiler, which is not assumed here or in [Nec00].

The work in [LJWF04] presents a framework for describing global optimizations by rewrite rules with CTL formulae as side conditions, which allow for generation of correct optimizations, but not for verification of (possibly incorrect) optimizations. The work in [GGB02] proposes a method for deploying optimizing code generation while correct translation between input program and code. They focus on code selection and instruction scheduling for SIMD machines.

1.3 Overview of the Compiler Validation Project

The ultimate goal of our compiler validation project is to develop a methodology for the translation validation of advanced optimizing compilers, with an emphasis on EPIC-targeted compilers and the aggressive optimizations characteristic to such compilers. Our methods will handle an extensive set of optimizations and can be used to implement fully automatic certifiers for a wide range of compilers, ensuring an extremely high level of confidence in the compiler in areas, such as safety-critical systems and compilation into silicon, where correctness is of paramount concern.

The approach of translation validation is based on the theory of correct translation. In general terms, we first give common semantics to the source and target languages using the formalism of *Transition Systems* (TS's). The notion of a target code T being a correct implementation of a source code S is then defined in terms of *refinement*, stating that every computation of T corresponds to some computation of S with matching values of the corresponding variables. In Figure 1.1 we present the process of refinement as completion of a mapping diagram.

$$\begin{tikzcd}
S\text{: Source} \arrow[r, "\text{Semantics}", "\text{Mapping}"'] \arrow[d, "\text{Optimizing Compiler}"'] & \textit{Sem(S)} \\
T\text{: Target} \arrow[r, "\text{Semantics}", "\text{Mapping}"'] & \textit{Sem(T)} \arrow[u, dashed, "\text{Refinement}"']
\end{tikzcd}$$

Figure 1.1: Refinement Completes the Picture

To apply the general approach of translation validation to verifying compiler optimizations, we distinguish between *structure preserving* optimizations, which admit a clear mapping of control points in the target program to corresponding control points in the source program, and *structure modifying* optimizations that admit no such mapping.

Structure Preserving optimizations cover most high-level optimizations and many low-level control-flow optimizations. Our approach for dealing with these optimizations is to establish a correspondence between the target and source code, based on *refinement*, and to prove it by *simulation*. According to this approach, we establish a *refinement mapping* correlating the control points the source and target, and indicating how the relevant source variables correspond to the target variables or expressions at each control point. The proof is then broken into a set of *verification conditions* (also called *proof obligations*), each claiming that a segment of target execution corresponds to a segment of source execution.

A more challenging category of optimizations is that of structure modifying optimizations which includes, e.g., *loop distribution* and *fusion*, *loop tiling*, and *loop interchange*. For this class, it is often impossible to apply the refinement-based rule since there are often no control points where the states of the source and target programs can be compared. We identify a large class of these optimizations, namely the *reordering transformations*, and devise a set of *permutation rules* that allow for their effective translation validation[ZPG+02, ZPFG03].

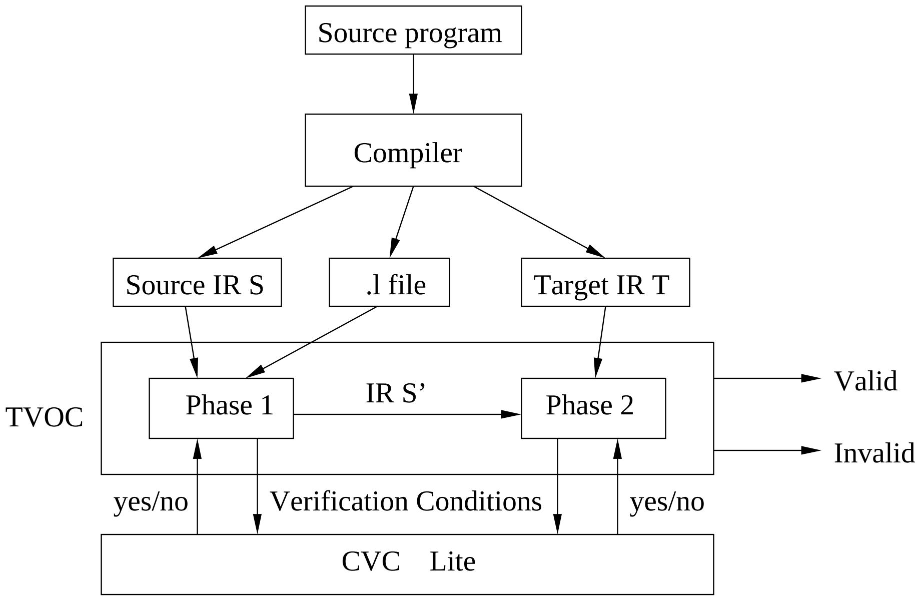


Figure 1.2: The architecture of TVOC.

A tool for translation validation of optimizing compilers, called TVOC, has been developed for Intel's Open Research Compiler (ORC) [CJW01]. Fig. 1.2 shows the overall design of TVOC. TVOC accepts as input a source program S

and target program T. These are provided in the WHIRL intermediate representation, a format used by the ORC compiler among others. Just as compilers perform optimizations in multiple passes, it is reasonable to break the validation into multiple phases, each using a different proof rule and focusing on a different set of optimizations. Currently, TVOC uses two phases to validate optimizations performed by the compiler, with two components TVOC-LOOP and TVOC-SP. In the first phase, TVOC-LOOP attempts to detect loop reordering transformations and generates an intermediate program S' which is obtained by transforming S so that it's loop structure corresponds to that found in T. The equivalence of S and S' is verified using rule PERMUTE [ZPG+02, ZPFG03]. Verification conditions are submitted to and checked by the automatic theorem prover CVC Lite [BB04]. In the second phase, TVOC-SP compares the program S' generated by phase 1 to the target program T. Since the transformation from S' to T is structure-preserving, the equivalence of these two programs is verified using rule VALIDATE (as presented in Chapter 3) Again, verification conditions are submitted to CVC Lite. If all verification conditions from both phases succeed, then the source and target are equivalent and TVOC outputs the result "VALID". Otherwise, TVOC outputs "INVALID" with a message indicating the stage at which the verification failed.

1.4 Contribution and Thesis Outline

The main goal of this thesis research is to develop theory and tools for validating structure preserving optimizations. The contribution of my thesis work includes:

- participating in desiging the current version of a proof rule, called VALIDATE, for validating structure preserving optimizations;
- developing TVOC-SP, a tool that automatically generate proofs of VALIDATE rule for structure preserving optimizations of an EPIC compiler;
- modeling memory operations and aliasing information to accomodate programs with arrays and pointers.

Chapter 2 describes a formal model and theory of correct translation. It also presents definitions and terms commonly used in the research of compiler optimizations that is later refered in this thesis.

Chapter 3 first describes a general proof rule, called VALIDATE, for translation validation of structure preserving optimizations. The proof rule presented in this thesis succeeds a series of proof rules that were sound but failed to handle some common optimizations. The current version has the versitility to handle minor structure changes in transformation, as long as the structure of loops is preserved. Then I define the well-formedness of a VALIDATE proof, and show that, given well-formed proofs $\mathbb{P}_1$ and $\mathbb{P}_2$ that establish the refinement between S and T_1, and the refinement between T_1 and T_2, respectively, we can compose a VALIDATE proof from $\mathbb{P}_1$ and $\mathbb{P}_2$ that establishes the refinement between S and T_2 directly.

Chapter 4 studies a series of individual compiler optimizations that are candidates for validation with rule VALIDATE. The study of individual optimizations not only shows the scope of rule VALIDATE, but also gives valuable insights of what type of the refinement mapping and auxiliary invariants are required to construct a VALIDATE proof, which lead to the various heuristics used in the validation tool TVOC-SP.

Chapter 5 presents the heuristics and algorithms in the tool TVOC-SP which automatically generates VALIDATE proofs for global optimizations performed by the Intel's ORC compiler. When I started building the tool, I had expected that compiler would provide us with the information of what kind of optimizaitons are performed and in which order they are performed, and the intermediate code produced by the compiler would contain program annotations that could be used as auxiliary invariants. However, it turned out that none of these information is available from ORC, and TVOC-SP ended up performing its own static analyses to construct VALIDATE proofs. One major challenge in the generation of a VALIDATE proof is to produce sufficient auxiliary invariants, hence this topic is explored extensively in Chapter 5.

Chapter 6 discusses the issues in validating programs with aliasing with rule VALIDATE. The challenge includes how to model program identities that can be accessed in more than one way and how to represent aliasing information as program invariants.

Chapter 7 summarizes the results and discusses possible future works.

Chapter 2

Preliminaries

2.1 Program Representation

The programs we consider are written in WHIRL [CJW01], the intermediate language (IR) for the ORC compiler. As compilation can be viewed as a process of gradual transition from the high level language constructs to the low level machine instructions, there are different levels of IR possible. The closer an IR is to the source language, the higher is its level. The more an IR resembles the machine instructions, the lower is its level. The WHIRL IR was designed to be capable of representing any level of IR from Very High to Very Low except the level that corresponds to the machine instructions. The compiler optimizations we focus on are the global optimizations, which are performed on High and Mid WHIRL, and the control-flow optimziations, which are perfomed on Low WHIRL.

During the compilation from High WHIRL to Low WHIRL, high level control flow constructs such as the operator DO_LOOP, DO_WHILE, WHILD_DO, IF ,etc. are translated to be uniformly represented via TRUEBR, FALSEBR or GOTO in Low WHIRL. In addition, the form of array subscription preserved in High WHIRL via ARRAY is lowered to address expressions in Mid and Low WHIRL.

For exposition purposes we assume that programs are written in the IR intermediate language, which is a subset of Mid and Low WHIRL, whose syntax is shown in Fig. 2.1. A function body is a sequence of IR statements. Among statements, we have direct assignments to symbols, denoted by $id \leftarrow expr$; function entries, denoted by FUNC_ENTRY $f, id_1, \ldots, id_m$, which represents a function f with an array of formal parameters $id_1, \ldots, id_m$; function returns RETURN id; labels LABEL L :; unconditional jumps GOTO; and conditional jumps TRUEBR and FALSEBR. The expressions in the IR language contain symbol identities id, constants c, and composite expressions using a variety of operators $expr_1$ `op` $expr_2$. We assume that the evaluation of expression does not have side-effect, i.e. none of the program variables have their values modified in the process of evaluation. With the set of statements in IR as represented in Fig. 2.1, the high level control constructs such as loops and conditional branches preserved by the High WHIRL operators DO_LOOP, DO_WHILE, WHILD_DO, IF, etc. are translated to be uniformly represented via TRUEBR, FALSEBR or GOTO in Low WHIRL.

Example 1 *Consider the function foo written in C language in Fig. 2.2 and its translation, by Intel ORC compiler into the intermediate code* IR.

Statements	$stat$	$::= id \leftarrow expr$ $\mid$ LABEL $L:$ $\mid$ FUNC_ENTRY $f, id_1, \ldots, id_m$ $\mid$ RETURN id $\mid$ GOTO L $\mid$ TRUEBR $expr, L$ $\mid$ FALSEBR $expr, L$
Expressions	$expr$	$::= id \mid c \mid expr_1 \; op \; expr_2$
Operators	op	$::= + \mid - \mid * \mid / \mid = \mid < \mid \leq \mid \ldots$

Figure 2.1: The syntax of the IR intermediate language

```
int k;                            FUNC_ENTRY foo,n
int foo(int n) {                    k <- 0
  int i,j;                          i <- 1
  k = 0;                            j <- 2
  i = 1;                            FALSEBR (n>=i),L1
  j = 2;                            j <- j * 2
  while (i<=n) {                    k <- 1
    j = j*2;                        i <- i + 1
    k = 1;                        L1:
    i = i+1;                        FALSEBR (k!=0),L2
  }                                 i <- j
  if (k)                            GOTO L3
    i = j;                        L2:
  else                              i <- i + 1
    i = i + 1;                    L3:
  return i;                         RETURN i
}
```

Figure 2.2: A C Program and its Intermediate Code

2.2 Control-Flow Analyses

Given a program in its intermediate representation, it is seen as a sequence of statements with few hints about what the program does or how it does it. It remains for control-flow analyses to discover the hierarchical flow of control within each procedure and for data-flow analysis to determine global (i.e., procedural-wide) information about the manipulation of data. Understanding the control-flow structure of a program is important not only to transforming the program to a more efficient one, but also essential to validating the correctness of the transformation.

Given a program in IR, the program's control structure is discovered with the following steps:

1. Identify basic blocks and construct control flow graphs (CFG). *Basic block* is, informally, a straight-line sequence of code that can be entered only at the beginning and exited only at the end. A *control flow graph* is an abstract representation of a procedure or a program. Each node in the graph represents a basic block. Directed edges are used to represent jumps in the control flow.

2. Identify the relation of dominance for each node in the control flow graph. We say that node d *dominates* node i, if every possible execution path from entry to i includes d. We say that d is an *immediate dominator* of i, if every dominator of i that is not d dominates d. Identifying dominance relation is important for performing further control-flow and data-flow analyses.

3. Identify loops. Without loss of generality, we assume that the programs we consider only have *natural loops*, where a natural loop is, informally, a strongly connected components with a unique entry node [1]. To determine the natural loops in a CFG, we first define a *back edges* as an edge in CFG whose head dominate its tail. Each back edge $m \rightarrow n$ characterizes a *natual loop* as a subgraph consisting of the set of nodes containing n and all the nodes from which m can be reached in the flowgraph without passing

[1]A technique called *node splitting* can transform loops that are not natural into natural ones[JC97]

through n and the edge set connecting all the nodes in its node set. For a natual loop, we define

- *loop header* to be the node that dominates every node in the loop.
- *loop preheader* to be a new (initially empty) block placed just before the header of the loop, such that all the edges that previously went to the header from outside the loop now go to the preheader, and there is a single new edge from the preheader to the header.

Fig. 2.3 presents the control flow graph corresponding to the IR program in Fig. 2.2. In the flowgraph, the set of blocks $\{\texttt{B2}, \texttt{B3}\}$ forms a natual loop characterized by the back edge $\texttt{B3} \rightarrow \texttt{B2}$. B2 is the loop's header, and B1 is the preheader.

2.3 Static Single-Assignment (SSA) Form

Static single-assignment (SSA) form [CFR+89, CRF+91] is a relatively new intermediate representation that effectively separates the value operated on in a program from the locations they are stored in, making possible more effective version of several optimizations.

A procedure is in *static single-assignment form* if every variable assigned a value in it occurs as the target of only one assignment. In SSA form du-chains are explicit in the representation of a procedure: a use of a variable may use the value produced by a particular definition if and only if the definition and use have exactly the same name for the variable in the SSA form of the procedure. This

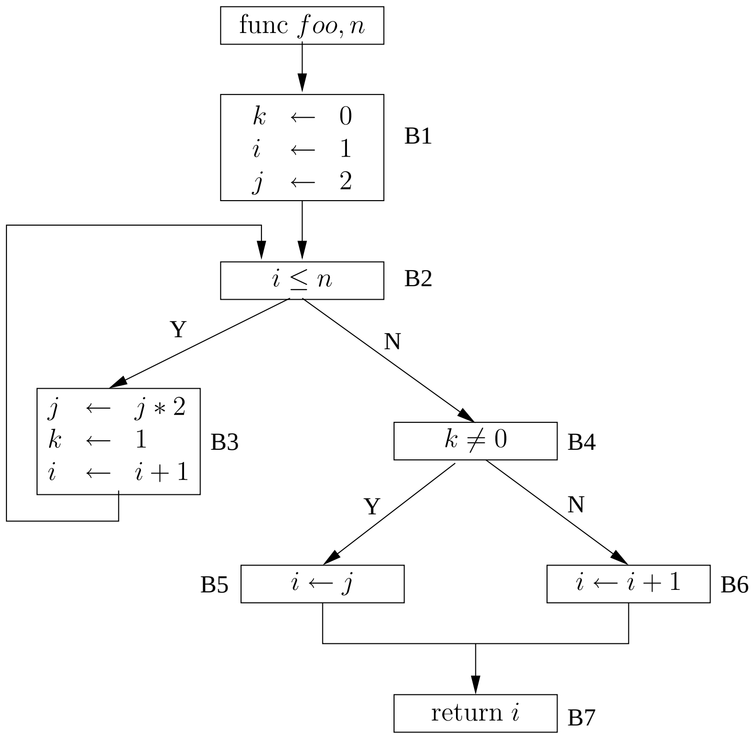


Figure 2.3: Example 1 represented as Control-Flow Graph

simplifies and makes more effective several kinds of optimizing transformations, including constant propagation, value numbering, invariant code motion and removal, strength reduction, and partial-redundancy elimination. Thus, it is valuable to be able to translate a given representation of a procedure into SSA form, to operate on it and, when appropriate, to translate it back into the original form.

In translation to SSA form, the standard mechanism is to subscript each of the variables and to use so-called ϕ-functions at join points, i.e. location where two or more control flows merge, to sort out the multiple assignments to a variable. Each ϕ- *function* has as many argument positions as there are versions of the variable coming together at that point, and each argument position corresponds to a particular control-flow predecessor of the point. Thus, the standard SSA-form representation of our example in Fig. 2.2 is shown in Fig. 2.4.

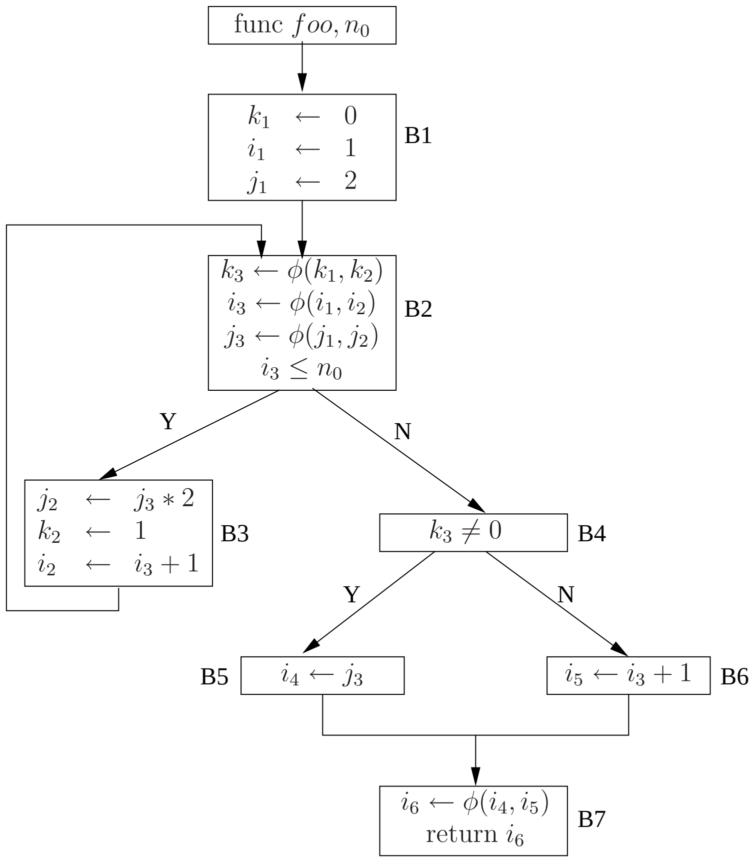


Figure 2.4: Example 1 represented in SSA Form

2.4 Transition Systems

In order to present the formal semantics of source and intermediate code we introduce *transition systems* (TS's), a variant of the *transition systems* of [PSS98b].

A *Transition System* $S = \langle V, \mathcal{O}, \Theta, \rho \rangle$ is a state machine consisting of:

- V a set of *state variables,*
- $\mathcal{O} \subseteq V$ a set of *observable variables*,
- Θ an *initial condition* characterizing the initial states of the system, and
- ρ a *transition relation*, relating a state to its possible successors.

The variables are typed, and a *state* of a TS is a type-consistent interpretation of the variables. For a state s and a variable $x \in V$, we denote by $s[x]$ the value that s assigns to x. The transition relation refers to both unprimed and primed versions of the variables, where the primed versions refer to the values of the variables in the successor states, while unprimed versions of variables refer to their value in the pre-transition state. Thus, e.g., the transition relation may include "$y' = y + 1$" to denote that the value of the variable y in the successor state is greater by one than its value in the old (pre-transition) state.

The observable variables are the variables we care about, where we treat each I/O device as a variable, and each I/O operation removes/appends elements to corresponding variable. If desired, we can also include among the observables the history of external procedure calls for a selected set of procedures. When comparing two systems, we require that the observable variables in the two system match.

A computation of a TS is a maximal finite or infinite sequence of states $\sigma :$ $s_0, s_1, \ldots$, starting with a state that satisfies the initial condition such that every two consecutive states are related by the transition relation. I.e., $s_0 \models \Theta$ and $\langle s_i, s_{i+1}\rangle \models \rho$ for every i, $0 \le i+1 < |\sigma|$[2].

Example 2 We translate the intermediate code in Fig. 2.3 into a TS. The set of state variables V includes `i`,`j`,`k`,`n`, among which the observable varaibles are global variable `k` and local variable `i` whose value is returned by function `foo`. We also include in V the control variable (program counter) `pc` that points to the next statement to be executed. Because the transitions connecting two consecutive states can represent either a single statement or a sequence of statements, depending on the intermediate states users care about in the computation, the range of `pc` can be the set of all program locations or selective locations. Here we choose the range of `pc` to be $\{\mathtt{B1}, \mathtt{B2}, \mathtt{B4}, \mathtt{B7}\}$, where `Bi` denotes the location right before basic block i. The initial condition, given by $\Theta\colon \pi = \mathtt{B1}$, states that the program starts at location `B1`.

The transition relation ρ can be presented as the disjunction of four disjuncts $\rho = \rho_{12} \vee \rho_{22} \vee \rho_{24} \vee \rho_{47}$, where ρ_{ij} describes all possible moves from `Bi` to `Bj` without passing through locations in the range of π.

E.g., ρ_{47} is:

$$(k \neq 0 \ \wedge\ i' = j \ \vee\ k = 0 \ \wedge\ i' = i+1) \ \wedge\ pres(\{j, k, n\})$$

where $pres(V)$ is an abbreviation of formula $\bigwedge_{v\in V}(v' = v)$.

[2] $|\sigma|$, the *length of* σ, is the number of states in σ. When σ is infinite, its length is ω.

A computation of the program starts with `B1`, continues to `B2`, cycles to `B3` and back to `B2` several times, then exits the cycle and branches to `B5` or `B6`, depending on the value of k, and finally terminates at `B7`. The state reached at each block is described by the values assigned to the variables.

A transition system $\mathcal{T}$ is called *deterministic* if the observable part of the initial condition uniquely determines the rest of the computation. That is, if $\mathcal{T}$ has two computations $s_0, s_1, \ldots$ and $t_0, t_1, \ldots$ such that the observable part (values of the observable variables) of s_0 agrees with the observable part of t_0, then the two computations are identical. We restrict our attention to deterministic transition systems and the programs which generate such systems. Thus, to simplify the presentation, we do not consider here programs whose behavior may depend on additional inputs which the program reads throughout the computation. It is straightforward to extend the theory and methods to such intermediate input-driven programs.

The translation of an intermediate code into a TS is straightforward; we therefore assume that all code we are dealing with here is described by a TS.

2.4.1 Comparison and Refinement between TSs

Let $P_S = \langle V_S, \mathcal{O}_S, \Theta_S, \rho_S \rangle$ and $P_T = \langle V_T, \mathcal{O}_T, \Theta_T, \rho_T \rangle$ be two TS's, to which we refer as the *source* and *target* TS's, respectively. Such two systems are called *comparable* if there exists a one-to-one correspondence between the observables of P_S and those of P_T. To simplify the notation, we denote by $X \in \mathcal{O}_S$ and $x \in \mathcal{O}_T$ the corresponding observables in the two systems. A source state s is defined to be *compatible* with the target state t, if s and t agree on their observable parts. That is, $s[X] = t[x]$ for every $x \in \mathcal{O}_{T}$. We say that P_{T} is a *correct translation* (*refinement*) of P_{S} if they are comparable and, for every $\sigma_{T} : t_0, t_1, \ldots$ a computation of P_{T} and every $\sigma_{S} : s_0, s_1, \ldots$ a computation of P_{S} such that s_0 is compatible with t_0, σ_{T} is terminating (finite) iff σ_{S} is and, in the case of termination, their final states are compatible.

Our goal is to provide an automated method that will establish (or refute) that a given target code correctly implements a given source code, where both are expressed as TSs.

Chapter 3

Validating Structure-Preserving Optimizations

3.1 Rule VALIDATE

Let $P_S = \langle V_S, \mathcal{O}_S, \Theta_S, \rho_S \rangle$ and $P_T = \langle V_T, \mathcal{O}_T, \Theta_T, \rho_T \rangle$ be comparable TSs, where P_S is the *source* and P_T is the *target*. In order to establish that P_T is a correct translation of P_S for the cases that P_T does not alter the structure of P_S in a major way, we introduce a proof rule that is an elaboration of the computational induction approach ([Flo67]) which offers a proof methodology to validate that one program *refines* another. This is achieved by establishing a *control mapping* from target locations to source locations, a *data abstraction* mapping from source variables to target variables, and proving that they are preserved with each step of the target program.

Some caution is needed in order to guarantee that VALIDATE can handle minor structure changes, for example, as in loop invariant code motion when assignments in the target occur before their counterparts in the source. It is actually a difficult problem to devise such a proof rule – the current version of the proof rule succeeds a series of proof rules that were sound but failed to handle some common optimizations. The version of the proof rule presented here has been successful in handling numerous examples, and we believe it is capable of handling all the optimizations that do not involve major structure modifications (as in the various loop optimizations.)

The proof rule is presented in Fig. 3.1. There, each TS is assumed to have a cutpoint set CP (i.e., a set that includes the initial and terminal block, as well as at least one block from each of the cycles; note that the cut-point set is a subset of the data domain of the control variable.) A *simple path* in between nodes in the cutpoint set refers to a path, in the flow graph of system, that leads in between two blocks of the cutpoint set and does not include, as an internal node, any other block of the set. For each such simple path leading from cut point i to cut point j, ρ_{ij} describes the transition relation between i and j. Note that, when the path from i to j passes through program locationss that are not in the cutpoint set, ρ_{ij} is a compressed transition relation that can be computed by the composition the intermediate transition relation on the path from i to j.

The control abstraction κ in part (1) of VALIDATE is the standard Floyd control mapping. The target invariants $\varphi(i)$ in part (2) and source invariants $\psi(i)$ in part (3) are program annotations that are expected to be provided by compiler from its data flow analysis. Intuitively, their role is to carry information in between cut points. The data abstraction α in part (4) is a variant of the standard Floyd-like data abstraction. The two differences are that we allow for α to be partial, and to be different at each target cut point. The motivation for allowing α to be partial is to accommodate situations, that occur for example in dead code elimination, where source variables have no correspondence in the target. The motivation for allowing α to be different at each target cut point is to accommodate situations, that occur for example in loop invariant code motion, where at some points of the execution, source variables have no correspondence in the target, while at other points they do.

The verification conditions assert that at each (target) transition from cutpoint i to cut point j[1], if the assertion $\varphi(i)$, $\psi(\kappa(i))$ and the data abstraction hold before the transition, and the transition takes place, then after the transition there exist new source variables that reflect the corresponding transition in the source, while the data abstraction and the assertions $\varphi(j)$ and $\psi(\kappa(j))$ hold in the new state. Hence, φ_i and $\psi(\kappa(i))$ are used as a hypothesis at the antecedent of the implication C_{ij}. In return, the validator also has to establish that φ_j and $\psi(\kappa(j))$ hold after the transition. Thus, we do not trust the annotation provided by the instrumented compiler but, as part of the verification effort, we confirm that the proposed assertions are indeed inductive and hold whenever visited. Since the assertion $\varphi(i)$ mentions only target variables, their validity should depend solely on the target code. Similarly, the validity of $\psi(i)$ should depend solely on the source code. In

[1]Recall that we assume path described by the transition is simple.

most cases, the primed source variables can be easily realized from the code, and the existential quantification in verification condition (5) can be eliminated, since the implication $q \rightarrow \exists x' : (x' = E) \wedge r$ is validity-equivalent to the implication $q \wedge (x' = E) \rightarrow r$. However, this may not always be the case and we are forced to leave the existential quantifier in (4). In Section 3.3, we will discuss in which circumstance we can eliminate the quantifier in verification conditions.

Example 3 Consider the program of Fig. 3.2 after a series of optimizations: Constant folding, copy propagation, dead code elimination, control flow graph optimization (loop inversion), and strength reduction. The annotation (φ_1, denoted `phi1`) is supplied by the translation validation tool (see Chapter 5).
To validate the program, we use the control mapping $\kappa = \{0 \mapsto 0, 1 \mapsto 2, 2 \mapsto 4\}$, and the data abstraction

$$\alpha : \left(\begin{array}{c} (\mathtt{PC} = \kappa(\mathtt{pc})) \wedge (\mathtt{pc} \neq 0 \rightarrow \mathtt{Y} = \mathtt{y}) \\ \wedge\, (\mathtt{pc} \neq 0 \rightarrow \mathtt{W} = \mathtt{w}) \wedge (\mathtt{pc} \neq 0 \rightarrow \mathtt{N} = 500) \end{array} \right)$$

Note that we always include in α the control mapping $\mathtt{PC} = \kappa(\mathtt{pc})$.

The verification condition C_{01} obtained for the simple path from `B0` to `B1`, after simplification (including the removal of the existential quantifier), is:

$$C_{01} : \quad \overset{T}{\rho}_{01} \;\wedge\; \alpha' \;\rightarrow\; \overset{S}{\rho}_{02} \;\wedge\; \varphi_1'$$

where ρ_{01}^{T} is defined by:

$$\left(\; (\mathtt{pc} = 0) \;\wedge\; (.t264' = 0) \;\wedge\; (\mathtt{y}' = 0) \;\wedge\; (\mathtt{w}' = 1) \;\wedge\; (\mathtt{pc}' = 1) \; \right)$$

and ρ_{02}^{S} is defined by:

$$\left(\; (\mathtt{PC} = 0) \;\wedge\; (\mathtt{Y}' = 0) \;\wedge\; (\mathtt{W}' = 1) \;\wedge\; (\mathtt{N}' = 500) \;\wedge\; (\mathtt{N}' \geq \mathtt{W}') \;\wedge\; (\mathtt{PC}' = 2) \; \right)$$

We also have:

$$\alpha' : \left(\; (\mathtt{Y}' = \mathtt{y}') \;\wedge\; (\mathtt{W}' = \mathtt{w}') \;\wedge\; (\mathtt{N}' = 500) \; \right)$$

and $\varphi_1' : (.t264' = 2 * \mathtt{y}')$ The other verification conditions are constructed similarly. They are all trivial to verify.

3.2 Soundness of VALIDATE

Let $P_S = \langle V_S, \mathcal{O}_S, \Theta_S, \rho_S \rangle$ and $P_T = \langle V_T, \mathcal{O}_T, \Theta_T, \rho_T \rangle$ be two comparable deterministic TSs. Let CP_S and CP_T be the sets of cut-points of P_S and P_T respectively. Assume that the control mapping κ, the data abstraction α, and the invariants $\varphi(i)$s and $\psi(i)$s are all defined. Assume further that all verification conditions C_{ij} (for every i and j such that there is a simple path in P_T leading from `Bi` to `Bj`) have been established. We proceed to show that P_T is a correct translation of P_S.

Let

$$\sigma^{T} : \quad s_1, s_2, \ldots$$

be a finite or infinite computation of P_{T}, which visits blocks $\mathtt{Bi}_1, \mathtt{Bi}_2, \ldots$, respectively. Obviously $\mathtt{Bi}_1$ is in CP_{T} and if the computation is finite (terminating) then its last block is also in CP_{T}. According to an observation made in [Flo67], σ^{T} can be decomposed into a fusion[2] of simple paths

$$\beta^{T} : \quad \underbrace{(\mathtt{Bj}_1, \ldots, \mathtt{Bj}_2)}_{\pi_1^{T}} \circ \underbrace{(\mathtt{Bj}_2, \ldots, \mathtt{Bj}_3)}_{\pi_2^{T}} \circ \cdots$$

such that $\mathtt{Bj}_1 = \mathtt{Bi}_1$, every $\mathtt{Bj}_\mathtt{k}$ is in the cut-point set CP^{T}, and $\pi_m^{T} = \mathtt{Bj}_\mathtt{m}, \ldots, \mathtt{Bj}_{\mathtt{m}+1}$ is a simple path. Since all VCs are assumed to hold, we have that

$$C_{j_k j_{k+1}} \quad : \quad \begin{array}{l} \varphi(j_k) \;\wedge\; \psi(\kappa(j_k)) \;\wedge\; \alpha \;\wedge\; \rho^{T}_{j_k j_k+1} \quad \rightarrow \\ \exists {V_{S}}' : (\bigvee_{\Pi \in Paths(\kappa(j_k), \kappa(j_{k+1}))} \rho^{S}_{\Pi}) \;\wedge\; \alpha' \;\wedge\; \varphi'(j_{k+1}) \;\wedge\; \psi'(\kappa(j_{k+1})) \end{array}$$

holds for every $k = 1, 2, \ldots$.

We can show that there exists a computation of P_{S}:

$$\sigma^{S} : \quad S_1, \ldots, S_2, \ldots,$$

such that S_1 visits cut-point $\mathtt{B}\kappa(\mathtt{j}_1)$, S_2 visits cut-point $\mathtt{B}\kappa(\mathtt{j}_2)$, and so on, and such that the source state visiting cut-point $\mathtt{B}\kappa(\mathtt{j}_\mathtt{r})$ is compatible with the target

[2]concatenation which does not duplicate the node which appears at the end of the first path and the beginning of the second path

state visiting cut-point $\mathtt{Bj_r}$, for every $r = 0, 1, \ldots$.

Consider now the case that the target computation is terminating. In this case, the last state s_r of σ^T visits some terminal cut-point $\mathtt{Bj_r}$. It follows that the computation σ^S is also finite, and its last state S_m (σ^T and σ^S are often of different lengths) visits cut-point $\mathtt{B}(\kappa(j_r))$ and is compatible with s_r. Thus, every terminating target computation corresponds to a terminating source computation with compatible final states.

In the other direction, let $\sigma^S : S_0, \ldots, S_n$ be a terminating source computation. Let $\sigma^T : s_0, s_1, \ldots$ be the unique (due to determinism) target computation evolving from the initial state s_0 which is compatible with S_0. If σ^T is terminating then, by the previous line of arguments, its final state must be compatible with the last state of σ^S. If σ^T is infinite, we can follow the previously sketched construction and obtain another source computation $\widetilde{\sigma}^S : \widetilde{S}_0, \widetilde{S}_1, \ldots$ which is infinite and compatible with σ^S. Since both S_0 and $\widetilde{S}_0$ are compatible with s_0 they have an identical observable part. This contradicts the assumption that P^S is deterministic and can have at most a single computation with a given observable part of its initial state.

It follows that every terminating source computation has a compatible terminating target computation.

3.3 Quantifier Elimination in VALIDATE

As mentioned in Section 3.1, the verification condition in rule VALIDATE (4) consists of existential quantifiers that cannot always be eliminated. Elimination of the existential quantifier is desirable because makes it more likely that the generated verification conditions can be checked automatically. The following theorem shows that the rules are equivalent under the assumption that the transition systems are deterministic.

Theorem 1 *The following verification conditions are equivalent:*

$$\varphi_i \wedge \psi_{\kappa(i)} \wedge \alpha \wedge \overset{T}{\rho}_{ij} \quad \rightarrow \quad \exists {V_S}' : (\bigvee_{\pi_1 \in Paths(\kappa(i), \kappa(j))} \overset{S}{\rho}_{\pi_1}) \wedge \alpha' \wedge \varphi'_j \wedge \psi'_{\kappa(j)}, \tag{3.1}$$

and

$$\varphi_i \ \wedge \ \psi_{\kappa(i)} \ \wedge \ \alpha \ \wedge \ \overset{T}{\rho}_{ij} \ \wedge \ (\bigvee_{\pi_2 \in Paths(\kappa(i))} \overset{S}{\rho}_{\pi_2}) \quad \rightarrow \quad \alpha' \ \wedge \ \varphi'_j \ \wedge \ \psi'_{\kappa(j)}. \tag{3.2}$$

Proof:

In one direction, suppose (3.2) holds and suppose that we have $\varphi_i \wedge \alpha \wedge \overset{T}{\rho}_{ij}$. By definition of ρ and α, it follows that $\texttt{PC} = \kappa(i)$. Without loss of generality, we can assume that at every non-terminal source cut-point, some transition must be taken and that at terminal source cut-points, no transitions are enabled. It then follows that $(\bigvee_{\pi_2 \in Paths(\kappa(i))} \rho^{S}_{\pi_2})$ holds. Thus, by (3.2), we have $\alpha' \wedge \varphi'_j$. But from $\rho^{T}_{ij} \wedge \alpha'$, $\mathtt{PC}' = \kappa(j)$ follows. We thus have $(\bigvee_{\pi_1 \in Paths(\kappa(i),\kappa(j))} \rho^{S}_{\pi_1}) \wedge \alpha' \wedge \varphi'_j$, and so clearly we also have $\exists {V_S}' \colon (\bigvee_{\pi_1 \in Paths(\kappa(i),\kappa(j))} \rho^{S}_{\pi_1}) \wedge \alpha' \wedge \varphi'_j$.

In the other direction, suppose that (3.1) holds and suppose that we have $\varphi_i \wedge \alpha \wedge \rho^{T}_{ij} \wedge (\bigvee_{\pi_2 \in Paths(\kappa(i))} \rho^{S}_{\pi_2})$. In particular, one of the transitions $\rho^{S}_{\pi_2}$ is true. By (3.1), we have $\exists {V_S}' \colon (\bigvee_{\pi_1 \in Paths(\kappa(i),\kappa(j))} \rho^{S}_{\pi_1}) \wedge \alpha' \wedge \varphi'_j$. Thus, there exists some successor state of the current-state named by V_S such that one of the transitions $\rho^{S}_{\pi_1}$ is true together with α' and φ'_j. But because the transition system is deterministic, the next-state variables ${V_S}'$ are uniquely determined by the present-state variables V_S. In other words, the existence of a successor state which satisfies $\alpha' \wedge \varphi'_j$ implies that every successor state satisfies α'. Since $\rho^{S}_{\pi_2}$ names a specific successor state, this successor state satisfies $\alpha' \wedge \varphi'_j$. ■

1. Establish a *control abstraction* $\kappa\colon \mathsf{CP}_{T} \to \mathsf{CP}_{S}$ that maps initial and terminal points of the target into the initial and terminal points of the source.

2. For each target cut point i in CP_{T}, form an *target invariant* $\varphi(i)$ that may refer only to concrete (target) variables.

3. For each source cut point i in CP_{S}, form a *source invariant* $\psi(i)$ that may refer only to abstract (source) variables.

4. Establish, for each target cut point i in CP_{T}, a *data abstraction*

$$\alpha(i) : (v_1 = E_1) \ \wedge \ \cdots \ \wedge \ (v_n = E_n)$$

   assigning to *some* non-control source state variables $v_k \in V_{S}$ an expression E_k over the target state variables. Note that $\alpha(i)$ is allowed to be partial, i.e., it may contain no clause for some variables. It is required that, for every *observable* variable $V \in \mathcal{O}_{S}$ (whose target counterpart is v) and every terminal point t, $\alpha(t)$ has a clause $(V = v)$.

5. For each cut points i and j such that there is a simple path from i to j in the control graph of P_{T}, form the verification condition

$$C_{ij}\colon \qquad \left(\begin{array}{l} \varphi(i) \wedge \psi(\kappa(i)) \ \wedge \ \alpha(i) \ \wedge \ \overset{T}{\rho}_{ij} \quad \to \\ \qquad \exists {V_{S}}'\colon \overset{S}{\rho}_{\kappa(i),\kappa(j)} \ \wedge \ \alpha'(j) \ \wedge \ \psi'(\kappa(j)) \ \wedge \ \varphi'(j). \end{array} \right)$$

6. Establish the validity of all the generated verification conditions.

Figure 3.1: The Proof Rule VALIDATE

```
B0  N <- 500
    Y <- 0
    W <- 1
B1  WHILE (W <= N)
B2  BLOCK
    W <- W + 2 * Y + 3
    Y <- Y + 1
    END_BLOCK
B4
```

(a) Input Program

```
B0  .t264 <- 0
    y <- 0
    w <- 1
B1  {phi1:  .t264 = 2 * y}
    w <- .t264 + w + 3
    y <- y + 1
    .t264 <- .t264 + 2
    IF (w <= 500) GOTO B1
B2
```

(b) Optimized Code

Figure 3.2: Example : Source and Annotated Target Programs

3.4 Transitivity of VALIDATE Proofs

In the previous section, we present rule VALIDATE and prove its soundness in establishing the equivalence between two transition systems. One question people often ask is the completeness of this proof rule. We claim that this rule can be applied to validate transformations that are *structure-preserving*, i.e., transformations that do not modify the structure of the source program in a major way. A formal description of structure preserving transformations is that, for some cut-point sets of source and target programs, there exists Floyd-like data mapping between corresponding cut points in source and target. But this definition is exactly what rule VALIDATE attempt to establish, hence is not helpful in identifying the scope of rule VALIDATE.

Although we are cannot precisely characterize the set of transformations rule VALIDATE is capable of verifying, we can still check whether the rule can be applied to verifying individual transformations. Our efforts to find out the completeness of rule VALIDATE can start with finding out the set of individual optimizations performed by compilers that rule VALIDATE can handle. For instance, copy propagation and constant folding do not modify control flow graph at all, thus can be verified by rule VALIDATE; if simplifications eliminate a branch in IF conditional, with a careful control abstraction, they can still be verified by rule VALIDATE. A list of individual compiler optimizations that can be handled by rule VALIDATE will be described in detail in the next chapter. Transformations performed by compilers consist of a sequence of individual optimizations, not necessarily in any particular order, hence it is desirable for rule VALIDATE to be transitive, i.e. if each individual optimizations can be handled by rule VALIDATE, so should their composition. Having rule VALIDATE transitive will also meet with our intuition, because the composition of two structure-preserving optimizations should also be structure-preserving.

In order to formally define the transitivity of VALIDATE rule, we first define what a VALIDATE proof is. A VALIDATE proof $\mathbb{P} = \langle \kappa, \alpha, \varphi, \psi \rangle$ that establishes the refinement between a source program $P_{_S}$ and a target program $P_{_T}$ consists of

- a control mapping $\kappa : \mathsf{CP}_{_T} \to \mathsf{CP}_{_S}$,
- a data abstraction $\alpha : \bigwedge\limits_{i \in \mathsf{CP}_{_T}} (\mathtt{pc} = i \to \alpha(i))$,
- a target invariant $\varphi : \bigwedge\limits_{i \in \mathsf{CP}_{_T}} (\mathtt{pc} = i \to \varphi(i))$,
- a source invariant $\psi : \bigwedge\limits_{i \in \mathsf{CP}_{_S}} (\mathtt{PC} = i \to \psi(i))$.

The transitivity of a VALIDATE proof can be described as follows:

> if there exist proofs of rule VALIDATE for transformations $\mathcal{F}_1$ and $\mathcal{F}_2$, there also exist a proof of rule VALIDATE for the transformation obtained by performing $\mathcal{F}_1$ followed by $\mathcal{F}_2$.

In general, a VALIDATE proof is not transitive. However, we show, some “well formed” applications of rule VALIDATE are. Before we formally define “well formedness”, we make the assumption that the programs we consider are represented in SSA form, and we define *live variables* and *visible variables* of programs in SSA form. In the rest of this section, We define a variable to be $live$ if it contributes to the computation of observable variables. To be presice, the set of *live variables* can be determined with a recursive approach as follows:

1. mark a variable to be live if it is *observable*, where a value is *observable* means that it either is definitely returned or output by the procedure or it affects a storage location that may be accessible from outside the procedure;

2. mark a variable to be live at location i if its value is used to compute the marked variable.

3. repeat the previous step until the process stabilized.

We define variable v to be *visible at location* i if the location where v is defined dominates location i. We define a VALIDATE proof $\mathbb{P} = \langle \kappa, \alpha, \varphi, \psi \rangle$ to be *well formed* if

1. data abstraction $\alpha(i)$ at each target cut point i has the form

$$\alpha(i) : \bigwedge_{u \in U} (u = E_u)$$

   where $U \subset V_S$ is the set of *live* variables that are *visible at location* $\kappa(i)$, and E_u is an expression over target *live* variables that are *visible* at location i;

2. target invariant $\varphi(i)$ at a target cut point i is a boolean expression over the set of target *live* variables that are *visible at location* i in the target;

3. source invariant ψ is T.

In the above definition, we require that the data abstraction correlates every live source variable to a target expression, since, without having the value of a live source variable encoded as a target expression at a check point, we may lose track of its value in the computation after that check point, hence we will not be able to establish the data abstraction at the next check point. We also require that dead variables appears neither in the data mapping nor in the target invariant, since dead variables do not contribute to the computation of "useful" information in the programs. Finally, we require that the source invariant be trivial because it can always be encoded as part of the target invariant. That is, for a source invariant $\psi(\kappa(i))$ at source cut point $\kappa(i)$ over the set of live variable $\{u_1, \ldots, u_m\}$ that are visible at $\kappa(i)$, and a data abstraction

$$\alpha(i) : (u_1 = E_{u_1}) \ \wedge \ \ldots \ \wedge \ (u_k = E_{u_m})$$

that maps u_l at source location $\kappa(i)$ to E_{u_l} at target location i, we can substitute each u_l in $\psi(\kappa(i))$ with E_{u_l} and obtain a target invariant

$$\psi(\kappa(i))[E_{u_1}/u_1, \ldots, E_{u_m}/u_m].$$

Theorem 2 *Let $P_S = \langle V_S, \mathcal{O}_S, \Theta_S, \rho_S \rangle$, $P_{T_1} = \langle V_{T_1}, \mathcal{O}_{T_1}, \Theta_{T_1}, \rho_{T_1} \rangle$ and $P_{T_2} = \langle V_{T_2}, \mathcal{O}_{T_2}, \Theta_{T_2}, \rho_{T_2} \rangle$ be comparable* TS*'s, where P_S is the source, P_{T_1} is a refinement of P_S, and P_{T_2} is a refinement of P_{T_1}. If there exist well-formed* VALIDATE *proofs that establish the refinement from P_S to P_{T_1}, and from P_{T_1} to P_{T_2}, then there also exists a well-formed* VALIDATE *proof that validates the refinement directly from P_S to P_{T_2}.*

Proof:

Let $\mathbb{P}_1 = \langle \kappa_1, \alpha_1, \varphi_1, \mathrm{T}\rangle$ be a well-formed proof for the refinement from P_S to P_{T_1}, Let $\mathbb{P}_2 = \langle \kappa_2, \alpha_2, \varphi_2, \mathrm{T}\rangle$ be a well-formed proof for the refinement from P_{T_1} to P_{T_2}. Let (i_2, j_2) be a pair of cut points in P_{T_2}. Let (i_1, j_1) be the corresponding cut points in P_{T_1}, and (i, j) be the corresponding cut points in P_S. That is,

$$\begin{aligned} & i_1 = \kappa_2(i), && j_1 = \kappa_2(j) \\ \text{and} \quad & i_S = \kappa_1 \circ \kappa_2(i), && j_S = \kappa_1 \circ \kappa_2(j). \end{aligned}$$

In proof $\mathbb{P}_2$, if there exists a simple path from i_2 to j_2 in the control graph of P_{T_2}, there exists a valid verification condition

$$C^{T_1T_2}_{i_2j_2}: \quad \begin{pmatrix} \varphi_2(i_2) \wedge \alpha_2 \;\wedge\; \rho^{T_2}_{i_2j_2} \\ \qquad \rightarrow \quad \exists V_{T_1}{}' \colon \rho^{S}_{i_1j_1} \;\wedge\; \alpha_2'(j_2) \;\wedge\; \varphi_2'(j_2) \end{pmatrix} \tag{3.3}$$

Similarly, in proof $\mathbb{P}_1$, for the pair of cut points $\kappa_2(i), \kappa_2(j)$ in P_{T_1}, there exists valid verification condtion

$$C^{ST_1}_{i_1j_1}: \quad \begin{pmatrix} \varphi_1(i_1) \wedge \alpha_1(i_1) \;\wedge\; \rho^{T_1}_{i_1j_1} \\ \qquad \rightarrow \quad \exists V_S{}' \colon \rho^{S}_{ij} \;\wedge\; \alpha_1'(j_1) \;\wedge\; \varphi_1'(j_1) \end{pmatrix} \tag{3.4}$$

Combining formula 3.3 and formula 3.4, we obtain a valid formula

$$\exists V'_S : \left(\begin{array}{l} \wedge\, \varphi_2(i_2) \;\wedge\; \rho^{T_2}_{i_2 j_2} \\ \wedge\, \exists V_{T_1} : \big(\alpha_2(i_2) \;\wedge\; \varphi_1(i_1) \;\wedge\; \alpha_1(i_1)\big) \end{array} \right) \rightarrow \left(\begin{array}{l} \varphi'_2(j_2) \;\wedge\; \rho^{S}_{ij} \\ \wedge\, \exists V'_{T_1} : \big(\alpha'_2(j_2) \;\wedge\; \varphi'_1(j_1) \;\wedge\; \alpha'_1(j_1)\big) \end{array} \right). \tag{3.5}$$

Formula 3.5 is similar to our desired form, however, it has two clauses of the form

$$\exists V_{T_1} : \big(\alpha_1(i_1) \;\wedge\; \alpha_2(i) \;\wedge\; \varphi_1(i_1)\big) \tag{3.6}$$

(first in a non-primed, and second in a primed form) that violate the desired form. Thus, it is suffices to show that formula 3.6 can be separated to a well-formed data abstraction from V_S to V_{T_2} and an invariant of P_{T_2}.

Let $U_1 = \{u_1, \ldots, u_m\} \subseteq V_{T_1}$ be the set of live variables that are visible at location i_1 in program T_1. Proof $\mathbb{P}_1$ is well formed, hence the V_{T_1} variables that appear free in $\alpha_1(i_1)$ and $\varphi_1(i_1)$ are in U_1. Proof $\mathbb{P}_2$ is also well formed, hence we have the data mapping

$$\alpha_2(i_2) : \quad (u_1 = E_{u_1}) \;\wedge\; \ldots (u_m = E_{u_m}) \tag{3.7}$$

where $E_{u_1}, \ldots, E_{u_m}$ are expressions over live variables in V_{T_2} that are visible at location i_2.

By substituting, for each $k = 1..m$, u_k in $\alpha_1(i_1)$ and $\varphi_1(i_1)$, with E_{u_k} we can eliminate the quantifiers in formula 3.6 and obtain an equivalent formula

$$\alpha_1(i_1)[E_{u_1}/u_1, \ldots, E_{u_m}/u_m] \ \wedge \ \varphi_1(i_1)[E_{u_1}/u_1, \ldots, E_{u_m}/u_m]$$

where the first conjunct is a well-formed data mapping from P_S to P_{T_2} and the second conjunct is an invariant of P_{T_2}.

Noticing that

$$\begin{array}{rlcl} & \exists V_{T_1} : \big(\alpha_1(i_1) \ \wedge \ \alpha_2(i_2)\big) & \Longleftrightarrow & \alpha_1(i_1)[E_{u_1}/u_1, \ldots, E_{u_m}/u_m] \\ \text{and} & \exists V_{T_1} : \big(\varphi_1(i_1) \ \wedge \ \alpha_2(i_2)\big) & \Longleftrightarrow & \varphi_1(i_1)[E_{u_1}/u_1, \ldots, E_{u_m}/u_m], \end{array}$$

we can construct a VALIDATE proof $\mathbb{P} = \langle \kappa, \alpha, \varphi, \psi \rangle$ that establishes the refinement from P_S to P_{T_2} with

- contorl abstraction $\kappa = \kappa_1 \circ \kappa_2$,
- data abstraction $\alpha : \bigwedge\limits_{i \in \mathsf{CP}_{T_2}} (\mathtt{pc} = i \rightarrow \alpha(i))$ where

$$\alpha(i) : \exists V_{T_1} : (\alpha_1(\kappa(i)) \ \wedge \ \alpha_2(i)) \text{ for each } i \in \mathsf{CP}_{T_2},$$

- target invariant $\varphi : \bigwedge\limits_{i \in \mathsf{CP}_{T_2}} (\mathtt{pc} = i \rightarrow \varphi(i))$ where

$$\varphi(i) : \varphi_2(i) \ \wedge \ \exists V_{T_1} : (\varphi_1(\kappa(i)) \ \wedge \ \alpha_2(i)) \text{ for each } i \in \mathsf{CP}_{T_2},$$

- source invariant $\psi = \mathtt{T}$,

and $\mathbb{P}$ is well formed.

$\lrcorner$

Chapter 4

Proving Various Optimizations with Rule VALIDATE

In the previous chapter, we present a proof rule called VALIDATE for validating *structure preserving* optimizations. Essentially, to be able to apply the rule, it requires that the loop structure be the same in the source and target programs. In this chapter, we show how to apply this rule to validating a set of individual optimizations that are commonly implemented in compilers. For each optimization that can be validated by rule VALIDATE, we give a well-formed VALIDATE proof. Section 4.1 discuss a series of data-flow analyses optimizations that make no changes to control flow, while Section 4.2 covers a set of control-flow optimizations that modify control flow but not in a major way. Still, there exist control-flow optimizations that are beyond the capability of rule VALIDATE. In Section 4.3, we give two examples of optimizations that rule VALIDATE cannot handle.

4.1 Global Optimizations

In this section, we discuss a series of optimizations that add/remove/modify instructions without changing the data flow of a program.

For transformations that do not change data flow, we can assume that there exists one-to-one correspondence between the program locations in source and target programs. This can be achieved by adding SKIP at the location in source where an instruction is inserted by the transformation and at the location in target where an instruction is removed. For such source and target programs, we can use a trivial control mapping, i.e. we take the set of program locations as cut-point set for both programs, and have $\kappa(i) = i$ for each cut point. For presentation's purpose, we only present the data mapping and the target invariant at the cut points at which changes happen. The data mapping and the invariant at other cut points are either trivial, or can be easily obtained by propagating the information from the point where changes happen. Besides, we assume source and target programs are written in SSA form. Let $\mathcal{V}$ be the set of variables that appear in both source and target. To simplify the notation, for a variable $x \in \mathcal{V}$, we denote by X and x the corresponding variables in source and target systems, respectively.

4.1.1 Algebraic Simplification and Reassociation

Algebraic simplifications use algebraic properties of operators or particular operator-operand combinations to simplify expressions. *Reassociation* refers to using specific algebraic properties – namely, associativity, commutativity, and distributivity

– to divide an expression into parts that are constant, loop-invariant(i.e., have the same value for each iteration of a loop), and variables. In general, this class of optimizatons can be described as below, where the source and target programs are the same except that the expressions assgined to x at location i are different.

$$\begin{array}{lcl} & & \ldots \\ \texttt{Li} & : & X \leftarrow expr_1 \\ \texttt{Li}+\texttt{1} & : & \ldots \end{array} \qquad\qquad \begin{array}{lcl} & & \ldots \\ \texttt{Li} & : & x \leftarrow expr_2 \\ \texttt{Li}+\texttt{1} & : & \ldots \end{array}$$

Source Target

To apply rule VALIDATE to this case, we define

$$\begin{array}{lcl} \alpha(i) & : & \bigwedge_{v\in\mathcal{V}}(V = v) \\ \varphi(i) & : & \mathtt{T} \end{array}$$

and the non-trivial verification condition is

$$C_{i,i+1} : \left(\begin{array}{l} \alpha(i) \wedge (x' = expr_1) \ \wedge \ pres(V_T - \{x\}) \\ \qquad \wedge (X' = expr_2) \ \wedge \ pres(V_S - \{X\}) \end{array} \right) \rightarrow \alpha'(i+1)$$

If the transformation is correct, $expr_1$ and $expr_2$ evaluate to the same value, which means $(X = x)$ holds after the transition and this condition is valid.

4.1.2 Constant Propagation

Constant propagation is a transformation that, given an assignment `x <- c` for a variable x and a constant c, replaces later uses of x with uses of c as long as intervening assignments have not changed the value of x. *Sparse conditional constant propagation* detects variables that are constant by deriving information from conditionals. The following programs represents a typical scenario of constant propagation: variable y has a constant value c when the execution gets to `Li`, hence the use of y at `Li` is replaced by c in the target.

$$\begin{array}{lll}
 & & \ldots \\
\texttt{Li} & : & X \leftarrow Y \ \texttt{op} \ Z \\
\texttt{Li}+1 & : & \ldots
\end{array}
\qquad\qquad
\begin{array}{lll}
 & & \ldots \\
\texttt{Li} & : & x \leftarrow c \ \texttt{op} \ z \\
\texttt{Li}+1 & : & \ldots
\end{array}$$

Source Target

To apply rule VALIDATE to this case, we define

$$\begin{array}{lll}
\alpha(i) & : & (Y = c) \ \wedge \bigwedge_{v \in \mathcal{V}-\{y\}} (V = v) \\
\varphi(i) & : & \texttt{T}
\end{array}$$

4.1.3 Value Numbering

Value numbering is one of the several methods for determining that two computations are equivalent and eliminating one of them. It associates a symbolic value with each computation without interpreting the operation performed by the computation, but in such a way that two computations with the same symbolic value always compute the same value.

Two varaibles are congruent to each other if the computation that define them have identical operators (or constant values) and their corresponding operands are congruent.

$$
\begin{array}{llllll}
 & & \ldots & & & \ldots \\
\texttt{Li} & : & A \leftarrow \texttt{op}(Y^1, \ldots, Y^m) & \texttt{Li} & : & a \leftarrow \texttt{op}(y^1, \ldots, y^m) \\
 & & \ldots & & & \ldots \\
\texttt{Lj} & : & B \leftarrow \texttt{op}(Z^1, \ldots, Z^m) & \texttt{Lj} & : & b \leftarrow a \\
\texttt{Lj}+\texttt{1} & : & & \texttt{Lj}+\texttt{1} & : &
\end{array}
$$

Source Target

A typical optimization of value numbering is described by the programs below: in source program, variables a and b are found to be congruent, so in target program b gets the value of a instead of the expression $\texttt{op}(z^1, \ldots, z^m)$. Notice that a and b being congruent means that the corresponding operands of operator op in source and target are congruent, which entails that, for each $k = 1 \ldots m$, $(y^k = z^k)$.

To apply rule VALIDATE to this case, we define

$$
\begin{array}{lll}
\alpha(j) & : & \bigwedge_{k=1..m} (Z^k = y^k) \;\wedge \bigwedge_{v \in \mathcal{V} - \{z^1, \ldots, z^m\}} (V = v) \\
\varphi(j) & : & a = \texttt{op}(y^1, \ldots, y^m)
\end{array}
$$

with which we can show that $B = b$ holds after the transitions at Lj in source and target are taken.

4.1.4 Copy Propagation

Copy propagation is a transformation that, given an assignment $x \leftarrow y$ for some variables x and y, replaces later uses of x with uses of y, as long as intervening instructions have not changed the value of either x or y. Following programs describe the scenario of copy propagation.

$$\begin{array}{lcl} & & \ldots \\ \texttt{Li} & : & A \leftarrow X \texttt{ op } Z \\ \texttt{Li}+\texttt{1} & : & \ldots \\ \end{array} \qquad\qquad \begin{array}{lcl} & & \ldots \\ \texttt{Li} & : & a \leftarrow y \texttt{ op } z \\ \texttt{Li}+\texttt{1} & : & \ldots \\ \end{array}$$

Source Target

To apply rule VALIDATE to such a transformation, we define

$$\begin{array}{lcl} \alpha(i) & : & (X = y) \ \wedge \bigwedge_{v \in \mathcal{V}-\{x\}} (V = v) \\ \varphi(i) & : & \texttt{T} \end{array}$$

The data abstraction at `Li` maps X to y and every other source variables V to the corresponding v in target. Thus, the right-hand-side expressions at `Li` in source and target evalute to the same value, and we obtain $A = a$ at $\texttt{Li}+\texttt{1}$.

4.1.5 Redundancy Elimination

The following three optimizations that deal with elimination of redundant computations are discussed here:

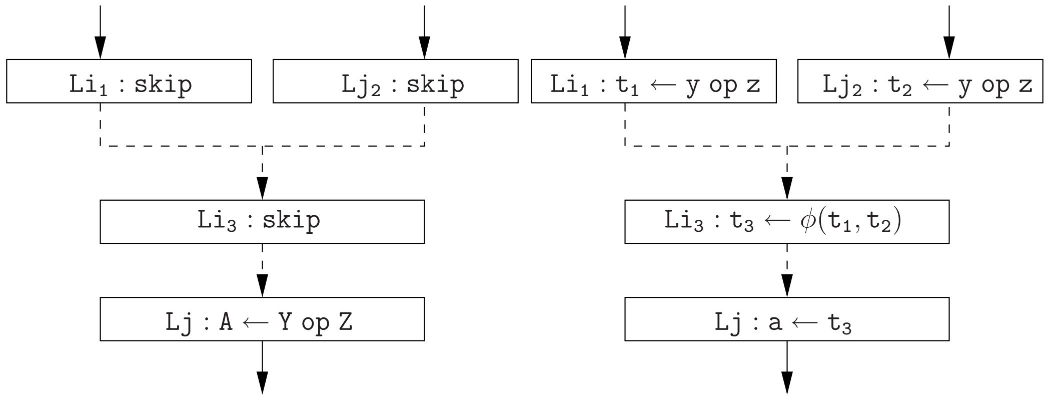
- The first one, *common-subexpression elimination*, finds computations that are always performed at least twice on a given execution path and eliminate the second and later occurences of them. This optimization requires data-flow analysis to locate redundant compuations.

- The second, *loop-invariant code motion*, finds computations that produce the same result every time a loop is iterated and moves them out of the loop. While this can be determined by an independent data-flow analysis, it is usually based on using ud-chains.

- The third, *partial-redundancy elimination*, moves computations that are at least partially redundant (i.e., those that are computed more than once on some path through the flowgraph) to their optimal computation points and eliminates totally redundant ones. It encompasses common-subexpression elmination, loop invariant code motion, and more. The data-flow analyses for this optimization is more complex than any other case we consider, and it requires the computation a series of local and global data-flow properties of expressions.

As readers may have noticed, to validate an optimiztion, what matters is not *how* the optimization is implemented, but *what* it produces. In the case of redundancy elimination, the first two optimizations mentioned above are both subcases of the third one. Therefore, even though these the first two optimizations are computed by totally different approaches from the third, for the purpose of validation, a VALIDATE proof for partial redundancy elimination can also be applied to the other two optimizations.

We consider the following programs which is a representative case of redundancy elimination.



$\mathtt{Li_1 : skip}$
$\mathtt{Lj_2 : skip}$
$\mathtt{Li_3 : skip}$
$\mathtt{Lj : A \leftarrow Y\ op\ Z}$
$\mathtt{Li_1 : t_1 \leftarrow y\ op\ z}$
$\mathtt{Lj_2 : t_2 \leftarrow y\ op\ z}$
$\mathtt{Li_3 : t_3 \leftarrow \phi(t_1, t_2)}$
$\mathtt{Lj : a \leftarrow t_3}$

Source Target

In the above control-flow graphs, we assume that execution reaches $\mathtt{Li_3}$ from either $\mathtt{Li_1}$ or $\mathtt{Li_2}$, and $\mathtt{Li_3}$ dominates $\mathtt{Li}$. Variables t_1, t_2 and t_3 are temporary variables introduced during optimization. To apply rule VALIDATE to this case, we define

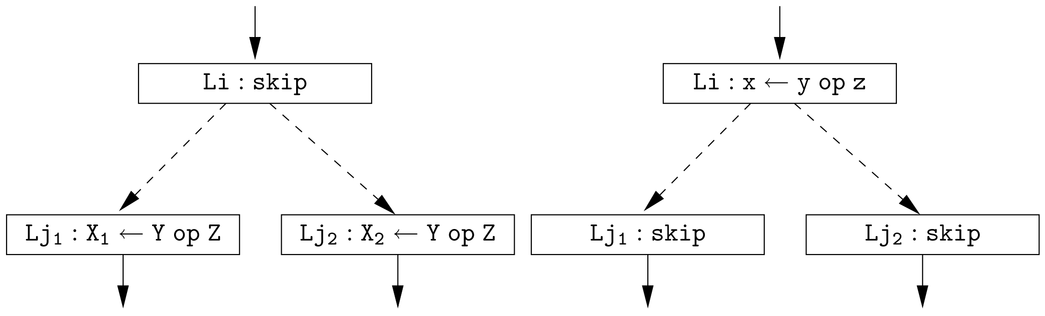
$$\begin{array}{lcl}\alpha(j) & : & \bigwedge_{v\in\mathcal{V}}(V = v) \\ \varphi(j) & : & (t_1 = y\ \mathtt{op}\ z)\ \wedge\ (t_2 = y\ \mathtt{op}\ z)\ \wedge\ (t_3 = t_1\ \vee\ t_3 = t_2)\end{array}$$

With the invariant $\varphi(j)$ which implies $(t_3 = y\ \mathtt{op}\ z)$, we can show that $A = a$ holds after taking the transitions at $\mathtt{Lj}$ in source and target.

4.1.6 Code Motion

Code motion is to move identical instructions from basic blocks to their common ancestor (known as code hoisting) or descendent in flow graphs (known as code sinking).

The transformation of code hoisting is represented in the flowgraphs below, where `Li` is a common ancestor of $\mathtt{Lj}_1$ and $\mathtt{Lj}_2$.



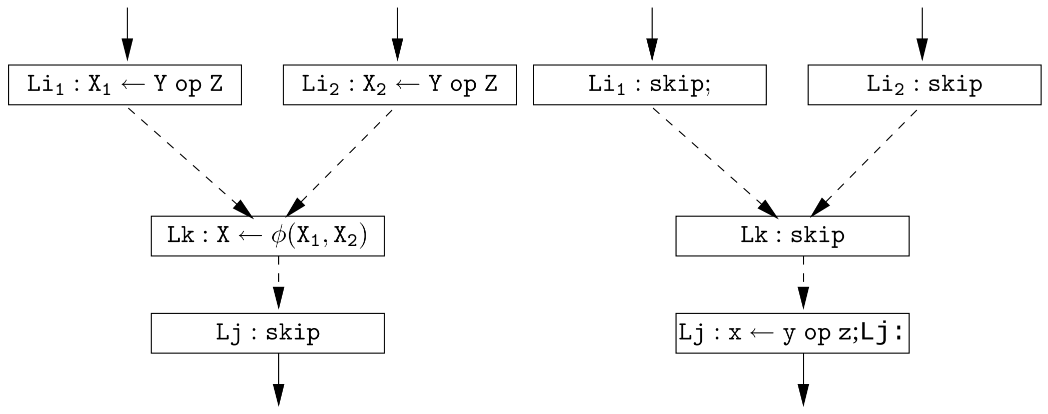
Source Target

To apply rule VALIDATE, we define

$$\begin{aligned} \alpha(j_1) &\ :\ \textstyle\bigwedge_{v \in \mathcal{V} - \{X_1\}} (V = v) \\ \varphi(j_1) &\ :\ (x = y \ \mathtt{op}\ z) \end{aligned}$$

It is straightforward to show that $X_1 = x$ holds after the instructions at $\mathtt{Lj}_1 + 1$ are executed. Besides, $X_1 = x$ is not guaranteed along the path from `Li` to $\mathtt{Lj}_1$, but when the code hoisting is performed safely, there should not exist use of x along the paths, hence not having $X_1 = x$ in the data abstraction does not affect the validity of the verification conditions. The motion of the instruction at $\mathtt{Lj}_2$ can be validated analogously.

Next, we show the transformation of code sinking in the following flowgraphs, where `Lk` is the location where the paths from $\mathtt{Li}_1$ and from $\mathtt{Li}_2$ join, and `Lk` dominates `Lj`.


$\mathtt{Li_1 : X_1 \leftarrow Y\ op\ Z}$
$\mathtt{Li_2 : X_2 \leftarrow Y\ op\ Z}$
$\mathtt{Lk : X \leftarrow \phi(X_1, X_2)}$
$\mathtt{Lj : skip}$
$\mathtt{Li_1 : skip;}$
$\mathtt{Li_2 : skip}$
$\mathtt{Lk : skip}$
$\mathtt{Lj : x \leftarrow y\ op\ z; Lj :}$

Source Target

To apply rule VALIDATE, we define

$$\begin{aligned} \alpha(j) &\ :\ (X = y\ \mathtt{op}\ z) \ \wedge\ \textstyle\bigwedge_{v \in \mathcal{V} - \{X\}} (V = v) \\ \varphi(j) &\ :\ \mathtt{T} \end{aligned}$$

It is straightforward to show that $X = x$ holds at $\mathtt{Lj} + \mathtt{1}$. The invariant $\varphi(j)$ is derived from the assertion $(X_1 = y\ \mathtt{op}\ z) \wedge (X = X_1) \vee (X_2 = y\ \mathtt{op}\ z) \wedge (X = X_2)$ propagated from $\mathtt{Lk}$.

4.1.7 Loop Optimizations

The optimizations covered in this subsection apply directly to the class of *well-behaved loops* [Muc97] where the loop's iterations are counted by an integer-valued variable that proceeds upward (or downward) by a constant amount with each iteration. All of the following three optimizations involve manipulation on *induction variables*, i.e. those whose value is incremented (or decremented) by a constant amount with each iteration.

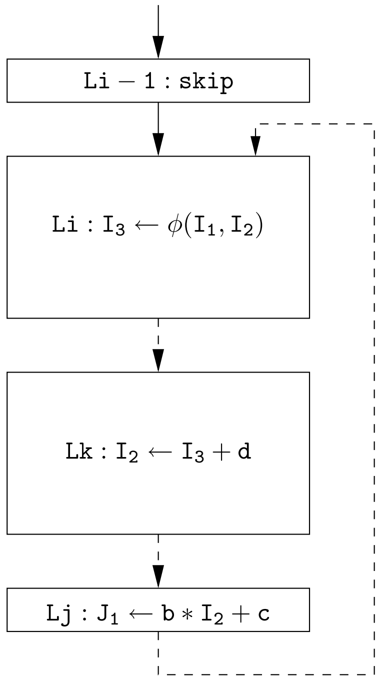
Strength Reduction replaces expensive operations, such as multiplications , by less expensive ones, such as additions. To perform strength reduction for a loop, one need to devide the loop's induction variables into *basic induction variables*, which are explicitly modified by the same constant amount during each iteration of a loop, and *dependent induction variables* in the class of a basic induction variable, which are computed by adding or multiplying that basic induction variable by a constant. Assume that i is a basic induction variable incremented by a constant d with each iteration, and j is in the class of i with linear equation $j = b*i+c$. Let db be a value of the expression $d*b$. The following programs demonstrate strength reduction applied to induction variable j.

To apply rule VALIDATE to this case, we define

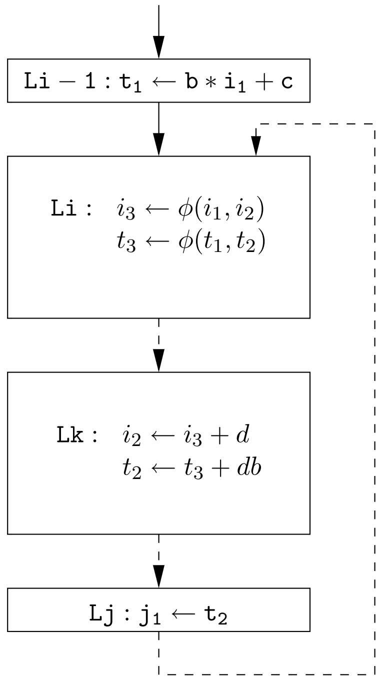
$$\begin{aligned}
\alpha(j) &: \bigwedge_{v \in V} (V = v) \\
\varphi(j) &: (t_1 = b * i_1 + c) \wedge (t_2 = b * i_2 + c) \wedge (t_3 = b * i_3 + c)
\end{aligned}$$

The three clauses in the invariant $\varphi(j)$ is propagated from the instructions at `Li`$-$`1`, `Lk` and `Li`, respectively. With the clause $(t_2 = b * i_3 + c)$ in $\varphi(j)$, it is straightforward to see that $J_1 = j_2$ holds after the instructions at `Lj` are executed.

Linear-Function Test Replacement replaces the loop-closing test by another induction variable in that context. It is beneficial when an induction variable is


$\mathtt{Li-1:skip}$
$\mathtt{Li:I_3} \leftarrow \phi(\mathtt{I_1,I_2})$
$\mathtt{Lk:I_2} \leftarrow \mathtt{I_3+d}$
$\mathtt{Lj:J_1} \leftarrow \mathtt{b*I_2+c}$

Source

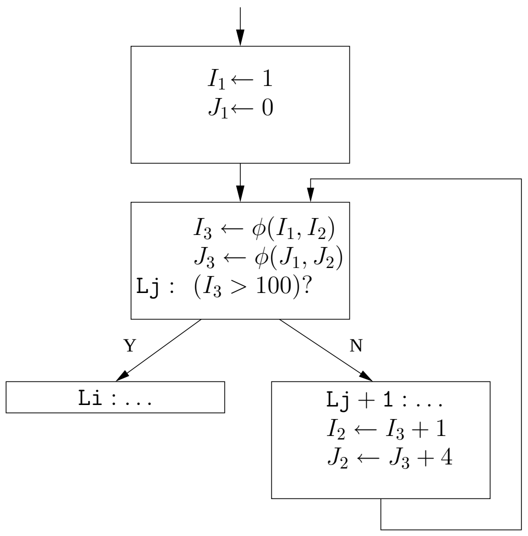

$\mathtt{Li-1:t_1} \leftarrow \mathtt{b*i_1+c}$
$\mathtt{Li:}$ $i_3 \leftarrow \phi(i_1, i_2)$
$t_3 \leftarrow \phi(t_1, t_2)$
$\mathtt{Lk:}$ $i_2 \leftarrow i_3 + d$
$t_2 \leftarrow t_3 + db$
$\mathtt{Lj:j_1} \leftarrow \mathtt{t_2}$

Target

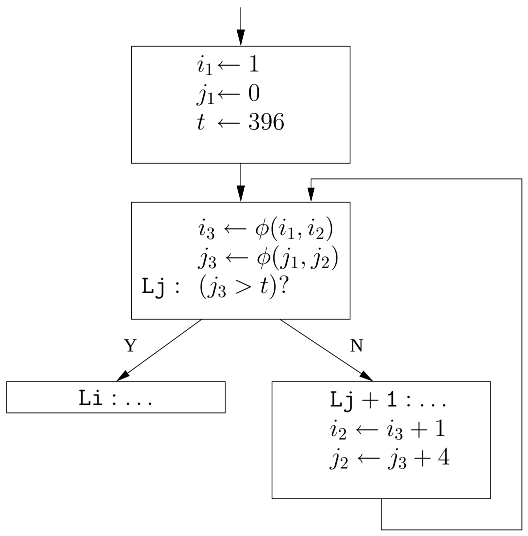
used only in the loop-closing test, and can be removed after being replaced.

To verify a linear-fucntion test replacement that replacing a loop-closing test $f(i)$ by $g(j)$ where i and j are induction variables and f and g are linear functions, essentially we need to show $f(i) \leftrightarrow g(j)$ for the values i and j take in the same iteration. Having i and j as induction varialbes, we can always find a linear fucntion h such that $j = h(i)$ at the point of loop-closing test. If the test replacement is correct, we should have $f(i) \leftrightarrow g(j)$ valid under the constraint $j = h(i)$.

Here is an example of linear-function test replacement.



Source



Target

To apply rule VALIDATE to this example, we define

$$\begin{aligned} \alpha(j) &: (I_1 = 1) \wedge (I_2 = j_2/4 + 1) \wedge (I_3 = j_3/4 + 1) \wedge \bigwedge_{v \in \mathcal{V} - \{i_1, i_2, i_3\}} (V = v) \\ \varphi(j) &: (j_1 = 0) \wedge (t = 396) \end{aligned}$$

The clause $(I_3 = j_3/4 + 1)$ in $\alpha(j)$ implies that the test conditions $(I_3 > 100)$ and $(j_3 > 396)$ are equivalent, hence the two verification conditions $C_{j,j+1}$ and $C_{j,i}$ are valid. Notice that the data abstraction at Lj maps I_3 to an expression over j_3 instead of i_3. This is because, after the test replacement, i_3 can be dead if it is not used after the loop, while a well-formed data abstraction requires to map source variables to expressions over live target variables.

Removal of Induction Variables In addition to strength-reducing induction variables, we can often remove them entirely when they serve no useful purpose in the program. There are several situations where induction variables can be removed:

1. The variable may have contributed nothing to the computation to begin with.
2. The variable may become useless as a result of another transformation, such as strength reduction.
3. The variable may have been created in the process of performing a strength reduction and then become useless as a result of another one.
4. The variable may be used only in the loop-closing test and may be replaceable by another induction variable in that context.

Removal of induction variables is a subcase of dead code elimination that will be discussed next.

4.1.8 Dead Code Elimination

A variable is *dead* if it is not used on any path from the location in the code where it is defined to the exit point of the routine in question. An instruction is *dead* if it computes only values that are not used on any executable path leading from the instruction. Programs may include dead code beofre optimization, but such code is much more likely to arise from optimization; removal of induction variables mentioned in Subsection 4.1.7 is an example that produces dead code; and there are many other. Many optimizations creates dead code as part of a division of labor priciple: keep each optimization phase as simple as possible so as make it easy to implement and maintain, leaving it to other phase to clean up after it.

The maximal set of dead instructions can be determined with an optmistice recursive approach as follows:

1. mark all instructions that compute observable values (also known as *essential values*, where a value is *observable* if it either is definitely returned or output by the procedure or it affects a storage location that may be accessible from outside the procedure;

2. mark instructions that contribute to the computation of the marked instructions;

3. repeat the previous step until the process stabilized. All unmarked instructions are dead and may be deleted.

Dead code elimination causes the data mapping to be partial: if a variable x has dead value at a set of locations $\mathcal{L}$ and the instruction that assigns that dead value to x is deleted in the target program, the mapping from x's abstract version to its concrete version $X = x$ does not hold at any location in $\mathcal{L}$. However, since the dead values does not contribute to the compuation of observable values, the data abstraction can still establish the equality between observable values in the source and their counterpart in the target and all verification conditions are still valid.

4.2 Control Flow Optimizations

Here we discuss optimizations that apply to the control flow of a procedure that are usually carried out on programs in a low-level intermediate code. Such optimizations often produce longer basic blocks, thus have the potential to increase available instruction-level parallelism.

In Section 3.1, we claim that rule VALIDATE is capable of handling all the optimizations that do not involve major structure modification. Control flow optimizations are a potentially problematic category to apply rule VALIDATE, due to changes of control flows in the target programs they produce. Surprising, most of the commonly used control flow optimizations do fall into the category that rule VALIDATE is capable to validate, which demonstrates the versatility of rule VALIDATE. In the rest of this subsection, we discuss a list of control flow optimizations that rule VALIDATE can be applied to.

In order to apply rule VALIDATE to control flow optimizations that modify the control flow in the transformation while preserve the data flow, we need an appropriate control mapping over carefully chosen sets of cut points in source and target programs, and leave the data mapping to be trivial, i.e., for a program variable $v \in \mathcal{V}$, its abstract version and concrete version are equivalent at every cut points, and the control variables $\texttt{PC} = \kappa(\texttt{pc})$.

4.2.1 Unreachable-code elimination

Unreachable code is code that cannot possibly be executed, regardless of the input data. In general, rule VALIDATE is not able to validate unreachable-code elimiantion. As a matter of fact, the refinement relation established by rule VALIDATE allows computations in the target program to be a subset of those in the source, hence will not detect the elimination of reachable code. However, under the assumption that programs are deterministic, rule VALIDATE establish the one-to-one correspondence between computations in source and target. Thus, with a control abstraction maps each location in the target to its corresponding location in the source, and a trivial data mapping, we can establish the correctness of unreachable-code elimination.

4.2.2 Straightening

Straightening is an optimization that applies, in its most basic form, to pairs of basic blocks such that the first has no successors other than the second and the second has no predecessors other than the first. Since they are both basic blocks, either the second one immediately follows the first one or the first of them must end with an unconditional brach to the second. In both case, the transformation fuses the two basic blocks into a single new block. Straightening does not modify the "shape" of control graph. To be specific, if we represent source and target as control flow graphs of *maximal basic blocks*, where each block is the longest sequence of instruction such that only the first instruction may have more than one entry and only the last instruction may have more than one exit, the two graph are identical. Thus, we can choose the point right before each maximal basic blocks as well as the terminating point in source and target as the set of cut-

```
if (a>d) {
  b = a;
  if (a>d) || bool
    d = b
  else
    d = c
}
```

Figure 4.1: An example with a condition that is a common subexpression

points in source and target, respectively, and establish a one-to-one mapping to each corresponding pair of cut-points. Together with a trival data mapping and no invariants, the validity of verification conditions is straightforward.

4.2.3 If simplifications

If simplifications apply to conditional constructs; these optimizations can reduce the number of conditional branches in a program. In its simplest form, if simplication apply to IF constructs with constant-valued conditions and remove the arm of the IF which is unexecutable. A more complicated form of if simplification is the occurrence of common subexpressions as conditions in an IF and in a subsequent dependent IF; of course, the value of the variables involved must not have been changed between the test. Such a case is illustrated in Fig. 4.1. The second IF test $(a > d) \lor bool$ is guaranteed to be satisfied because $a > d$ was satisfied in the first IF test, and neither a nor d has been changed in between.

In general, if simplifications apply to conditional jump if (i) the condition is constant-valued (ii) a static analysis concludes that the condition evaluates to a constant every time an execution reaches that conditional jump. Such simplifications can be viewed as a two-step process: in step one, the conditional jump is either removed or replaced by an unconditional jump; in step two, the code made unreachable becaue of step one is removed. Here, we only give the proof of step one. The proof of step two can be found above in *unreachable-code elimination*, and the synthesis of two VALIDATE proofs can be found in Section 3.4.

Below are the programs describing a transformation that removes a conditional jump when the condition always evaluates to be false, and proof of the transformation. The cases where the conditional jump always take the false branch can be validated analogically.

	Source		Target
	$\ldots$		$\ldots$
$\texttt{Li}:$	$\texttt{truebr}\ cond, \texttt{Lj}$	$\texttt{Li}:$	$\texttt{skip}$
$\texttt{Li}+1:$	$\ldots$	$\texttt{Li}+1:$	$\ldots$

To validate this transformation, we choose data abstraction and control abstraction to be trivial. We know that the condition $cond$ at `Li` is statically decided to be true, which means there exists an boolean assertion γ being an invariant at

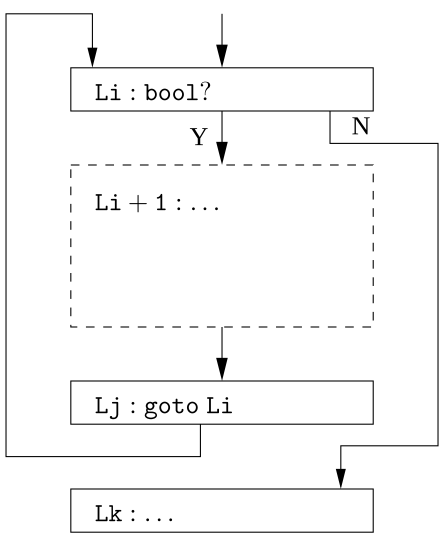
Li, such that $\gamma \rightarrow cond$. The non-trivial verification condition, in this cases, is

$$C_{i,i+1} : \left(\begin{array}{l} \quad \alpha(i) \ \wedge \ \gamma \\ \wedge \ (\mathtt{pc} = i \ \wedge \ \mathtt{pc}' = i+1 \ \wedge \ pres(V_T)) \\ \wedge \ ((\mathtt{PC} = i \ \wedge \ \mathtt{PC}' = i+1 \ \wedge \ cond \ \wedge \ pres(V_S)) \vee \\ \quad (\mathtt{PC} = i \ \wedge \ \mathtt{PC}' = j \ \wedge \ \neg cond \ \wedge \ pres(V_S))) \end{array} \right) \rightarrow \alpha'(i+1)$$

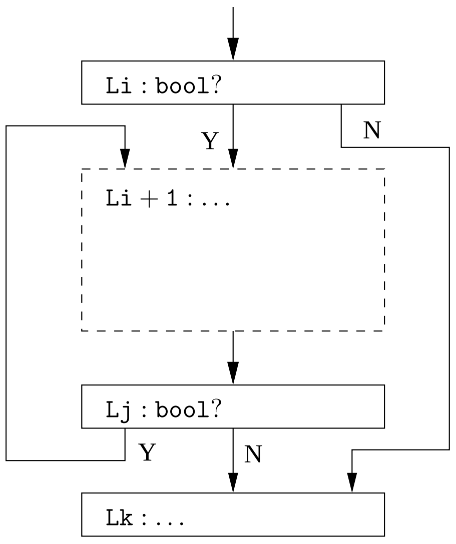
Due to the fact $\gamma \rightarrow cond$, the second disjunct of source transition relation cannot be satisfied, and $\mathtt{PC}'$ can only gets $(i+1)$, hence we have $\mathtt{PC}' = \mathtt{pc}'$ holds in the right hand side of the implication. This verification condition is valid.

4.2.4 Loop inversion

Loop inversion, in source-language terms, transforms a WHILEloop into a REPEATloop. In other words, it moves the loop-closing test from before the body of the loop to after it. This has the advantage that only one branch instruction need be executed to close the loop, rather than one to get from the end back to the beginning and another at the beginning to perform the test. The following programs demonstrate the transformation fo loop inversion.



Source



Target

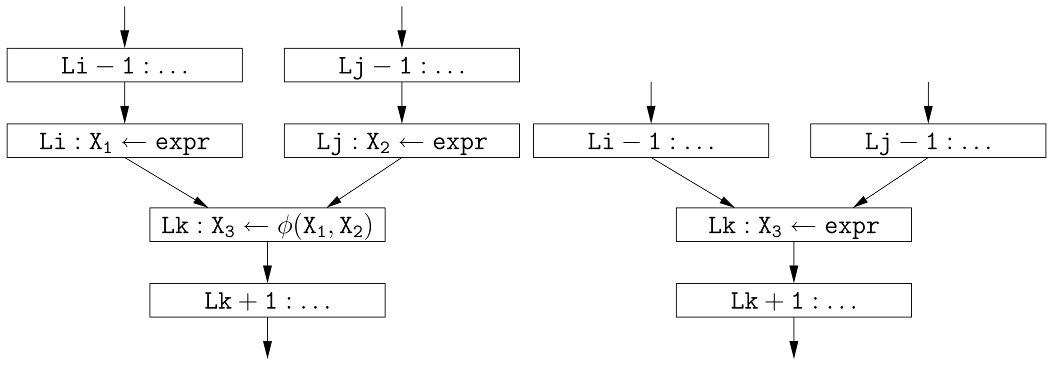
To apply rule VALIDATE to verifying this example, we choose the cut-point sets to consist of every locations for both source and target, and define the control abstraction as

$$\kappa(loc) = \begin{cases} i & \text{if } loc = j, \\ l & \text{otherwise.} \end{cases}$$

With a trivial data mapping and no invariant, we can obtain a set of verifiaiton conditions that are valid.

4.2.5 Tail merging

Tail merging, also known as *cross jumping*, is an optimization that always saves code space and may also save time. It searches for basic blocks in which the last few instructions are identical and that continue execution at the same location, either by one branching to the instruction following the other or by both branching to the same location. What the optimization does is to replace the matching instructions of one of the blocks by a branch to the corresponding point in the other. The transformation is demonstrated as follows:



Source Target

To apply rule VALIDATE to verifying this example, it is crucial to selecting the right cut points in the source and target programs. Let $\mathcal{L}_S$ and $\mathcal{L}_T$ be the set of program locations in source and target, respectively. We take

$$\mathsf{CP}_S = \mathcal{L}_S - \{k\}$$
$$\mathsf{CP}_T = \mathcal{L}_T - \{k\}$$

With both the control mapping and the data mapping being trivial and no invariants, we can obtain a set of verification conditions that are all valid.

4.3 Beyond Rule VALIDATE

Although we have already shown that transformations performed by many control-flow optimizations are considered structure-preserving by rule VALIDATE, there still exist some transformations that modify a program's structure in one way or another such that the rule fails to apply. Next, we give two examples of such transformations.

4.3.1 Loop Simplification

A loop whose body is empty can be eliminated, as long as the iteration-control code has no live side effects. If the iteration-control code has the side effects, they may be simple enough that they can be replaced by non-looping code. Following is an example of such optimization:

Source	Target
L0 : falsebr $(N \geq 0)$, L1	L0 : falsebr $(N \geq 0)$, L1
$i \leftarrow 0$	$i \leftarrow N+1$
L2 : truebr $(i > N)$goto L1	
$i \leftarrow i+1$	
goto L2	
L1 :	L1 :

Since, in the source program, the only variable modified in the loop is i, and value of i at the point of loop termination must be $(N+1)$, in the target program, the loop is replaced by $i \leftarrow N+1$. Naturally, the control points `L0` and `L1` in target corresponds to `L0` and `L1` in source, respectively. However, there is no control point in target that can be related to `L2`, and it is not easy task to automatically computing the transition relation from `L0` to `L1` in source because we do not have an algorithmic way to compute the transition relation for a loop. Rule VALIDATE fails to apply to this example.

4.3.2 Unswitching

Unswitching is a control-flow transformation that moves loop-invariant conditional branches out of loops. For example, consider the C programs below. The IF statement in the source has a loop invariant condition $k = 2$, thus is moved out of the loop in the target. Because the IF conditional has no else part in the source, we need to supply one in the target that sets the loop-control variable i to its final value in case that i is live after the loop. Similar to what happens in loop simplification, the transformation eliminates the loop in the source when $k = 2$ is not satisfied. Again, this is a transformation that rule VALIDATE cannot handle.

```
for (i=0;i<100;i++)
   if (k=2)
      a[i] = a[i] + 1;
```

Source

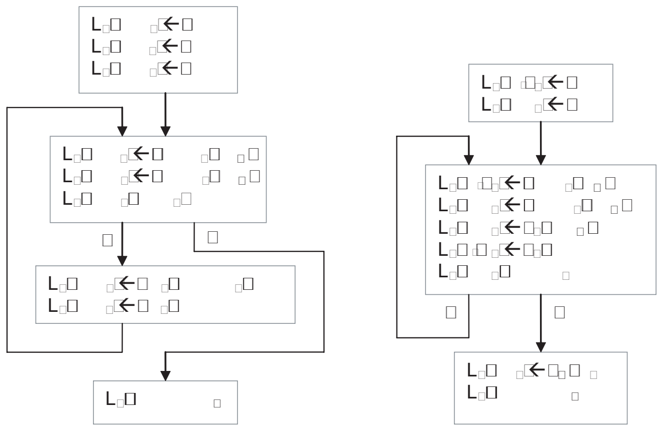
```
if (k=2)
   for (i=0;i<100;i++)
      a[i] = a[i] + 1;
else
   i = 100;
```

Target

Chapter 5

TVOC-SP: a Tool for Validating Structure Preserving Optimizations

In Chapter 4, we described how to apply rule VALIDATE to various structure-preserving optimizations, given that each of such optimizations is performed individually. In practice, it is not always possible to have compiler output a transformed program after each individual optimizations is performed. As a matter of fact, even though compilers perform optimizations in multiple passes, they tend to perform a group of optimizations in one single pass and leave the users no clue of what optimiztions are performed and in which order they are applied. For example, when working with ORC compiler, the source and target programs we take are the intermediate code the compiler produces right before and immediately after the global-optimization phase. In addition, we obtain a symbol table from the compiler, which gives type information of program variables. In such a case, it is not possible for us to “compose” a VALIDATE proof as suggested in Section 3.4. Instead, we build a tool called TVOC-SP that attempts to use various heuristics to generate proofs for transformations that are concatenation of multiple global optimizations. In this chapter, we describe how TVOC-SP performs its own control-flow and data-flow analyses to generate the ingredients including control abstraction, data abstraction and invariants of source and target programs to establish a VALIDATE proof. We use the source and target programs in Fig. 5.1 as a running example to demonstrate the generation of each ingredient.



(a) Source (b) Target

Figure 5.1: Example: a source program and its target transformed by a series of optimizations: Constant folding, copy propagation, dead code elimination, control flow graph optimization (loop inversion), and strength reduction, both in SSA form.

5.1 Control Abstraction

Cutpoint set Because global optimizations may add and delete instructions, or eliminate whole branches, it is often the case that there exists no bijection between locations in source and target programs. However, for our purposes, it suffices to find a subset of program locations for the cutpoint set, as long as the symbolic execution of simple paths between two cut points (i.e., paths that contain no intermediate cutpoints) can be represented as transition relations.

Since all, or almost all, the global optimizations we are dealing with do not add or eliminate loops, hence, there always is a bijection between the loops in source and target programs. Besides, most of the global optimizations preserve the control structure of loops, except for loop inversion where loop-closing test is moved from the beginning of the loop body to the end. Hence, we chose the cutpoint sets to consist of the initial and terminal locations of each program, as well as the beginning of each loop's body (since we are assuming SSA form, we stipulate that the latter are chosen to be *after* assignments of φ- functions).

For our example, the source cut-point set consists of locations $\{1, 7, 9\}$ (the initial location, terminal location, and the first location of the loop), while the target cutpoint set consists of locations $\{1, 5, 9\}$ (the initial location, terminal location, and the first non-φ statement in the loop).

Control abstraction Control abstraction establishes one-to-one correspondence between cutpoints in source and target. Since every program unit can be represented as a single-entry, single-exit control flow graph, control abstraction maps initial and terminal locations of the target to initial and terminal locations of the source, respectively.

Since we choose a single cutpoint for each loop, the problem of mapping cutpoints of loops is reduced to finding a mapping between the loops in source and target. Given two control graphs, the problem of automatically identifying corresponding loops in the two may be cost prohibitive. However, we assume that compilers can provide annotations to programs in IR that will indicate the statement numbers in both codes of loops that are expected to match.

The control mapping κ maps target cutpoints into source cutpoints. In our example, we have $\kappa : \{\texttt{1} \mapsto \texttt{1}, \texttt{5} \mapsto \texttt{7}, \texttt{9} \mapsto \texttt{9}\}$.

Paths and transition relation After choosing the cutpoint sets for source and target programs, we compute, for both programs, the sets of *simple paths*, i.e., paths of locations that start and end at cutpoints and do not contain as intermediate cutpoints. For our example, the set of simple paths in the source is:

$$\texttt{CP\_PATHS} = \left\{ \begin{array}{l} 1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 7, \\ 1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 9, \\ 7 \to 8 \to 4 \to 5 \to 6 \to 7, \\ 7 \to 8 \to 4 \to 5 \to 6 \to 9 \end{array} \right\}$$

For each simple path, we compute its transition relation using the symbolic simulator. For example, for the path $[7 \to 8 \to 4 \to 5 \to 6 \to 7]$ in the source, we have the transition relation:

$$
\begin{aligned}
(W_3' = W_2 + Y_2 + 3) \wedge (Y_3' = Y_2 + 1) \ \wedge \ (Y_2' = Y_3') \ \wedge \ (W_2' = W_3') \\
\wedge \, (W_2' \leq N_1) \wedge (N = N_1) \ \wedge \ (Y = Y_2) \ \wedge \ (W = W_2) \\
\wedge \, (N' = N_1) \ \wedge \ (Y' = Y_2') \ \wedge \ (W' = W_2')
\end{aligned}
$$

5.2 Data Abstraction

To construct data abstraction, we need to find, for each target cut point i, a set of equality relations $V = E_v$ that relates the value of a source variable V at source location $\kappa(i)$ to the value of an target expression E_v at target location i. While this is, in general, a hard problem, we rely again on information supplied by compilers, mostly in the form of existing annotation that are expressed in terms of predicates of the form $(U = v)$. To be specific, we noticed that, during compiler optimizations, the names of program variables preserves in IR, and temporary varaibles introduced during optimizations are often annotated with variable names in the source program it corresponds to.

Thus, we can assume some base set of candidate data X mapping of the form $(U = v)$, from which we choose the maximal subset that holds at target locations, and take $\alpha(i)$ to be their conjunction. The computation of $\alpha(i)$ is described by the following pseudo code:

```
{INITIALIZATION}
for each target cutpoint i
  $\gamma$(i) := $X$
```

```
    $\alpha(i) = \bigwedge_{E \in X}\{E\}$
    paths(i) = {set of all simple paths leading into i}

{FIX-POINT COMPUTATION}
repeat
   for every path $\pi$ in path(i)
       j := start point of $\pi$
       $\gamma(i)$ := $\gamma$(i) $\cap$ {$E \in \gamma$(i) : $\varphi_T(j) \wedge \varphi_S(\kappa(j)) \wedge \alpha(j) \wedge$
                                   $\rho^T_\pi \wedge \rho^S_{\kappa(j)\kappa(i)} \rightarrow prime(E)$ }
       $\alpha(i) = \bigwedge_{E \in \gamma(i)} E$
until sets stabilize
```

Here, $prime(E)$ is the "primed" version of E, that is, when every variable is replaced by its primed version. $\varphi_T(j)$ and $\varphi_S(\kappa(j))$ represent the source invariant at location j and the target invariant at location $\kappa(j)$, respectively. The generation of invariants will be discussed in the next section. Since invariants are generated for each program separately, they do not require the control or data mapping that are inter-programs. Thus, for now we assume that we have $\varphi^T(i)$ and $\varphi^S(j)$ for every $i \in \mathsf{CP}_T$ and $j \in \mathsf{CP}_S$. Note that while we chose the target transition function that corresponds to the target path π, we take the matching source path to be any path in between the two matching π-endpoints of the source. In fact, if there is more then one matching source path, we should take the disjunction of the transition relations of the matching source paths.

The procedure described above can be viewed as an iterative forward data-flow analysis operated on a lattice that consists of the powerset of X ordered by $\subseteq$. The flow functions of γ can be solved by starting with X and descending values in the lattice in each iteration until a fixpoint is reached. The validity of the logic formula in the flow equation of γ is decided by using a theorem prover. Unlike iterative data flow analyses used in compiler optimizations, this procedure applies a joint analysis on source and target programs.

In our example, we obtain in the data abstraction, e.g., e.g.,

$$\begin{array}{rcl} \alpha(1) & : & (W = w) \wedge (Y = y) \\ \alpha(7) & : & (W = w) \\ \alpha(9) & : & (W = w) \wedge (Y = y) \end{array}$$

The data mappings we construct with the above algorithm is in a very restricted form, i.e., it always maps a source variable to a target variable instead of a target expression, and it gives rise to the question of whether such a data mappings is sufficient to establish a VALIDATE proof. The answer is that, the part of data abstraction that maps a source variable to a target expression (instead of a target variable) can be encoded as part of the source invariant. For instance, a well-formed data mapping

$$\alpha(i) : (X = x) \wedge (Y = y) \wedge (Z = x + y)$$

can also be expressed as the conjunction of a data mapping together and a source invariant as follows:

$$\begin{aligned} \alpha(i) &\ : \ (X = x) \ \wedge \ (Y = y), \\ \psi(\kappa(i)) &\ : \ (Z = X + Y). \end{aligned}$$

5.3 Generating Invariants

As mentioned above, invariant generation is the most challenging tasks that are required in order to apply VALIDATE. In this section, we represent two methods that both generates invariants: The first one is based on iterative data-flow analysis, which is commonly performed by compilers; while the second one operates directly on programs in SSA form. We believe it is both more efficient and generates more information than that data-flow based method. This is due to the fact that the data-flow based method is purely syntactic, while the SSA-form based method is also semantic.

5.3.1 Generating Invariants from Data-flow Analysis

Invariants that are generated by data flow analysis are of definitions that are carried out into a basic blocks by all its predecessors. We outline here the procedure that generates these invariants.

For a basic block `B`, define:

`kill`(`B`)	:	$setof$`B`'s assignments $y = e(\bar{x})$ some of whose terms are redefined later in `B`
`gen`(`B`)	:	set of `B`'s assignments none of whose terms are redefined later in `B`

Both `kill`(`B`) and `gen`(`B`) are easy to construct. TVOC-SP constructs two other sets, `in`(`B`), that is the set of invariants upon entry to B, and `out`(`B`), that is the set of invariants upon exit from B. These two sets are computed by the procedure described in Fig. 5.2, where $pred(\texttt{B})$ is the set of all basic blocks leading into B.

```
For every block B
    in(B) init {if BB is initial
                   then set of $u = u_0$ for every variable $u$
                   else emptyset}
    out(B) init emptyset

repeat for every block B
   out(B) := (in(B) $\setminus$ kill(B)) $\cup$ gen(B)
   in(B)  := $\bigcap_{p \in pred\texttt{B}}$ out(p)
until all sets stabilize
```

Figure 5.2: Procedure to Compute Invariants

Set operations and comparisons are performed syntactically. An obvious enhancement to our tool is perform those operations semantically. Another possible enhancement is to add inequalities to the computed invariants, which are readily available from the data flow analysis.

Even with the two enhancement mention above, the algorithm in Fig. 5.2 is still not able to produce sufficient invariants for some compiler optimizations. Consider the C programs in Fig. 5.3. Sparse conditional constant propagation detects that $X = 1$ is a constant in the source, hence replaces the use of X at L2 by 1 and eliminates X in the target. In order to establish the equivalence between the source and target, we need to compute $(X = 1)$ as an invariant of the source program at L2. However, this is not an invariant that can be computed without trying to evaluate conditionals.

```
L0 :  I = 1;            L0 :  i = 1;
      if(I==1)
        X = I;
      else
        X = I + 1;
L1 :  while (I<100)     L1 :  while (i<100)
        I = I * 2;              i = i * 2;
L2 :  Y = I + X;        L2 :  y = j + 1;
```

(a) Source (b) Target

Figure 5.3: An example of sparse conditional constant propagation

5.3.2 Generating Invariants Based on SSA Form

Intuitively, the role of invariants in VALIDATE is to carry information in between basic blocks. For a program in SSA form, such a task can be performed simply by collecting all the definitions as well as the branching conditions that reach certain program points, which can later be fed into theorem provers which are capable of semantic reasoning to obtain more "hidden" information.

The following is an example that illustrate how invariant generation works on programs in SSA form.

```
B0:  if ¬$c_0$ goto B2
B1:  $x_1 \leftarrow 3$
     $y_1 \leftarrow 5$
     goto B3
B2:  $x_2 \leftarrow 5$
     $y_2 \leftarrow 3$
B3:  $x_3 \leftarrow \phi(x_1, x_2)$
     $y_3 \leftarrow \phi(y_1, y_2)$
     $z_1 \leftarrow x_3 + y_3$
B4:
```

Figure 5.4: An example program

Observe that, no matter how c_0 evaluates at B0, $(z1 = 8)$ at B4. Rather than attempting to directly detect $(z_1 = 8)$ as an invariant at B4, we first check the assignments appearing in the locations that dominates B4, i.e., the definitions that hold at B4. This leads us to computing $(z_1 = x_3 + y_3)$ as an invariant at B4, but not to they desired $(z_1 = 8)$, which requires information about the values of x_3 and y_3, which, in turn, depends on the branch condition c_0. By backtracking from B4 to B0, we obtain:

$$\begin{aligned}&((x_3 = x_1 \ \wedge \ y_3 = y_1 \ \wedge \ x_1 = 3 \ \wedge \ y_1 = 5 \ \wedge \ c_0) \vee \\ &(x_3 = x_2 \ \wedge \ y_3 = y_2 \ \wedge \ y_2 = 3 \ \wedge \ x_2 = 5 \ \wedge \ \neg c_0)) \\ \wedge \ &(z_1 = x_3 + y_3)\end{aligned} \tag{5.1}$$

as an invariant at B4, which implies that $(z_1 = x_3 + y_3 = 8)$.

Here we present a methodology that computes invariant for programs in SSA representation. Assume we have a control flow graph G of a program in SSA form whose loops are all *natural*, i.e., strongly connected components with single entry locations. (If some of G's loops are not natural, *node splitting* can be used to transforms the offensive loops.) We denote the entry node of a loop as a loop header. We assume that each loop header has only one incoming edge from outside of the loop. (When this is not the case, we introduce *preheader*s – new (initially empty) blocks placed just before the header of a loop, such that all the edges that previously lead into the header from outside the loop lead into the preheader, and there is a single new edge from the preheader to the header.)

Thus, we assume a CFG G where each node is a basic block. Since all loops in

G are assumed to be natural, G's edges can be partitioned into the set of backward edges (or back edges) and the set of forward edges. The nodes of G and the forward edges define a dag that induces a partial order among the nodes. For a program in SSA form, a variable's definition always reaches descendents of the node where it is defined. Hence, the solution to the equations in Fig. 5.5 results in a safe propagation of invariants in the ancestor of the node into the node's invariant.

We follow the standard notation, and two node $x, y \in G$, we say that x ***dominates*** y if every path from the entry of G into y passes through x. We say that x is an ***immediate dominator*** of y if every dominator of y is either x or else is strictly dominated by x. The immediate dominator of a node y is denoted by $idom(x)$. For a node $y \in G$, we denote by $G_{idom(x)}$ the graph obtained from G by removing all edges that lead into $idom(x)$, and then removing all the nodes and edges that do no reach x.

Given a CFG G and a node $x \in G$, let $\texttt{assign}(x)$ be the set of non ϕ-assignments in x. If x is not a loop header, we define:

$$\texttt{gen}(x) \;=\; \bigwedge_{(v:=exp)\in\texttt{assign}(x)} (v = exp).$$

The expression $\texttt{gen}(x)$ describes the invariants generated by x regardless of its environment.

For a node x with an immediate successor y, we denote by $\texttt{cond}(x, y)$ the condition under which the control transfers from x into y.

Let $x \in G$ be a node that is *not* a loop header. Assume that x has m_x predecessors, $x'_1, \ldots, x'_{m_x}$. Let V_x denote the set of variables defined by ϕ-functions in x. Assume that for every $v \in V_x$, the definition of v in x by the ϕ-function, is

$$v_{t_0^x(v)} \leftarrow \phi(v_{t_1^x(v)}, \ldots, v_{t_{m_x}^x}(v)).$$

Let $x \in G$ be now a node which is a loop header. Obviously, if x is reached through a back edge, we cannot simply take the definitions of the induction variables as expressed after the ϕ-functions, since, together with their entry value, we may get wrong information. E.g., consider the source program in Fig. 5.1. When computing the invariant for Y_2 at the the loop header (the block that starts with location 4), we have, from the previous iteration, $Y_3 = Y_2 + 1$, and from the ϕ-function, $Y_2 = Y_3$, thus, without further ado, we'll obtain an invariant $Y_2 = Y_2+1$, which is obviously wrong. We remedy this by including the correct information about the induction variables. That is, if $v_i^x, \ldots, v_K^x$ be the basic induction variables of the loop whose header is x, where each induction variable v_i^x is initialized to b_i before entering the loop, and is incremented by c_i at each iteration, we define:

$$\texttt{induc}(x) \; = \; \exists \hat{v}_x \geq 0. \bigwedge_{i=1}^{K} (v_i = b_i + c_i \times \hat{v}_x) \tag{5.2}$$

where $\hat{v}_x$ is a new variable. I.e., $\hat{v}_x$ is a loop iteration count, and $\texttt{induc}(x)$ captures the values of the induction variables at *some* (possibly the 0^{th}) iteration of the loop. We shall return the issue of dealing with the existential variables.

In Fig. 5.5 we describe data flow equations that allow to compute the assertions $\texttt{in}(x)$ and $\texttt{out}(x)$ for every node $x \in G$. The former is an invariant at the beginning of x, after all the ϕ-functions, and the latter is an invariant at the end of x. The invariants $\texttt{in}(x)$ and $\texttt{out}(x)$ can be computed for every $x \in G$ simultaneously by forward traversal of the dag induced by the forward edges of G.

$$\texttt{in}(x,G) \quad = \begin{cases} \texttt{out}(idom(x),G) \wedge \\ \qquad \texttt{cond}(idom(x),x) \ \wedge \ \texttt{induc}(x) & \text{if } x \text{ is a loop header,} \\ \texttt{out}(idom(x),G) \wedge \\ \qquad \bigvee_{i=1}^{m_x} \left(\begin{array}{c} \texttt{out}(x'_i, G_{idom(x)} \ \wedge \ \texttt{cond}(x'_i, x) \\ \wedge \bigwedge_{v \in V_x} (v_{t_0^x(v)} = v_{t_i^x(v)}) \end{array} \right) & \\ & \text{otherwise} \end{cases}$$

$$\texttt{out}(x,G) \ = \begin{cases} \texttt{in}(x,G) \ \wedge \ \texttt{gen}(x) & \text{if } x \in G, \\ true & \text{otherwise.} \end{cases}$$

Figure 5.5: Data-flow equations for $\texttt{in}(x,G)$ and $\texttt{out}(x,G)$

Theorem 3 *The data-flow equations computer in Fig. 5.5 are sound, i.e., during an execution of a program, for every basic block* B *represented by node x in the* CFG*G, when the program reaches* B *(after the ϕ-functions),* $\texttt{in}(x,G)$ *holds, and whenever the program exits* B, $\texttt{out}(x,G)$ *holds.*

Proof Outline: The proof is by induction on the BFS of the G. The base case is for the entry node(s). Then, we have the data-flow equation for `out`, which is trivially true. For the inductive step, we distinguish between loop headers and non-loop headers. Suppose that x is a loop header. Since we assume that all loops are natural, the control reaches a loop header either from its immediate dominator or from a back edge. Since we assume SSA form, we have that all invariants at the end of the immediate dominator, as well as the condition leading to the loop, hold. If this is the first entry to the loop, then `induct`(x) true with the trivial $\hat{v_x} = 0$. If the loop header is reached by a back edge, then `induct`(x) true with the trivial $\hat{v_x} > 0$. By the induction hypothesis, the `out`$(idom(x), G)$ is sound. We can therefore conclude that `in`(x, G) is sound.

If x is not a loop header, then x is reached through one of its predecessor, x'_i, with `cond`(x'_i, x) holding. Thus, the soundness of `in`(x, G) follows immediately from the induction hypothesis.

Finally, whether x is or is not a loop header, `out`(x, G) is a conjunction of `in`(x, G) with `gen`(x), the effect that `B`. Since we assume that `in`(x, G) is sound, the soundness of `out`(x, G) follows. ▟

Finally, for the invariant $\varphi(i)$ we take `in`(x, G) where x is the basic block whose initial location is i in the CFG G corresponding to the program.

Note that `in`(x, G) and `out`(x, G) are mutually recursive function over the structure of the dag induced by G. Hence, no fix-point computation is necessary to solve the equations. As a matter of fact, each node and edge of x's ancestor in G is visited only once in the computation of `in`(x, G) and `out`(x, G), thus the complexity of the computation is linear to the size of G.

We now return to the issue of existential quantifiers in the invariants generated by the above computation. Since VALIDATE requires to have the invariant as both left and right side of the implication, and since most theorem provers do not accept existential formulae as consequents. We can, however, instantiate the offensive existential quantifier when constructing VCs according to VALIDATE. Suppose that we are dealing with target invariants. (The case of source invariants is similar.) Consider a VC C_{ij} that has a $\exists\hat{v}_x$ on the r-h-s, thus x is a loop header. Let "the loop" mean "the loop whose header is x" for this discussion. Assume that on the l-h-s of the VCs we have the invariant $\varphi_{_T}(i)$. We distinguish between the following cases:

j **is the loop's header, and** i **is outside the loop.** Thus, the simple path between i and j is one the corresponds to the first entry into the loop. In this case, we can instantiate $\hat{v}_x$ to 0, and replace the existential part of $\varphi'_{_T}(j)$ with

$$\bigwedge_{i=1}^{K}(v_i = b_i).$$

j **is the loop header and** i **is in the loop.** Thus, the simple path between i and j corresponds to a back edge of G. We can then "re-use" the value of $\hat{v}_x$ from the antecedent and replace the existential part of $\varphi'_{_T}(j)$ with

$$\bigwedge_{i=1}^{K}(v_i = b_i + c_i \times (\hat{v}_x + 1))$$

Neither of the previous cases. Thus, simple path between i and j does not alter the values of the induction variables, and we can "re-use" the value of $\hat{v}_x$

from the antecedent, thus replacing the existential part of $\varphi'_{_T}(j)$ with

$$\bigwedge_{i=1}^{K}(v_i = b_i + c_i \times \hat{v}_x)$$

We note that the method outlined above may not produce sufficient invariants under all conditions. We are aware of its shortcomings in the cases when optimizations depend on data-flow information. Yet, to the best of our knowledge, such cases do not exist in IR programs that are derived with the goal of preserving backward compatibility.

Example Consider the program in Fig. 5.3.2.

```
B1:  $n_1 \leftarrow 100$
     $i_1 \leftarrow 0$
     $j_1 \leftarrow 0$
     $s_1 \leftarrow 1$
B2:  $i_3 \leftarrow \phi(i_1, i_2)$
     $j_3 \leftarrow \phi(j_1, j_2)$
     $s_3 \leftarrow \phi(s_1, s_2)$
     $i_2 \leftarrow i_3 + 1$
     $j_2 \leftarrow j_3 + 2$
     $s_2 \leftarrow s_3 * j_2$
     if $(i_2 < n_1)$ goto B2
B3:
```

For which we have:

$$\begin{array}{lll} \texttt{gen}(\texttt{B1}) & : & (n_1 = 100) \ \wedge \ (i_1 = 0) \ \wedge \ (j_1 = 0) \ \wedge \ (s_1 = 1) \\ \texttt{gen}(\texttt{B2}) & : & (i_2 = i_3 + 1) \ \wedge \ (j_2 = j_3 + 2) \ \wedge \ (s_2 = s_3 * j_2) \\ \texttt{cond}(\texttt{B1}, \texttt{B2}) & : & true \\ \texttt{cond}(\texttt{B2}, \texttt{B3}) & : & \neg(i_2 < n_1) \end{array}$$

The program has a loop $\{\texttt{B2}\}$ in which there are two induction variables, i_3, which is initialized to i_1 before the loop and is incremented by 1 at each iteration, and j_3, which is initialized to j_1 before the loop and is incremented by 2 at each iteration.

We therefore have:

$$\texttt{induct}(\texttt{B2}) \ = \ \exists \hat{v}.(i_3 = i_1 + \hat{v}) \ \wedge \ (J_3 = j_1 + 2\hat{v})$$

Solving the equations of Fig. 5.5, we obtain:

$$
\begin{aligned}
\texttt{in}(\texttt{B1}) &: \ \mathit{true} \\
\texttt{out}(\texttt{B1}) &: \ (n_1 = 100) \ \wedge \ (i_1 = 0) \ \wedge \ (j_1 = 0) \ \wedge \ (s_1 = 1)
\end{aligned}
$$

$$
\begin{aligned}
\texttt{in}(\texttt{B2}) &: \ (n_1 = 100) \ \wedge \ (i_1 = 0) \ \wedge \ (j_1 = 0) \ \wedge \ (s_1 = 1) \\
&\quad \wedge \exists \hat{v} : ((i_3 = i_1 + \hat{v}) \ \wedge \ (j_3 = j_1 + 2\hat{v})) \\
\texttt{out}(\texttt{B2}) &: \ (n_1 = 100) \ \wedge \ (i_1 = 0) \ \wedge \ (j_1 = 0) \ \wedge \ (s_1 = 1) \\
&\quad \wedge \exists \hat{v} : ((i_3 = i_1 + \hat{v}) \ \wedge \ (j_3 = j_1 + 2\hat{v})) \\
&\quad \wedge (i_2 = i_3 + 1) \ \wedge \ (j_2 = j_3 + 2) \ \wedge \ (s_2 = s_3 \cdot j_2)
\end{aligned}
$$

$$
\begin{aligned}
\texttt{in}(\texttt{B3}) &: \ (n_1 = 100) \ \wedge \ (i_1 = 0) \ \wedge \ (j_1 = 0) \ \wedge \ (s_1 = 1) \\
&\quad \wedge \exists \hat{v} : ((i_3 = i_1 + \hat{v}) \ \wedge \ (j_3 = j_1 + 2\hat{v})) \\
&\quad \wedge (i_2 = i_3 + 1) \ \wedge \ (j_2 = j_3 + 2) \ \wedge \ (s_2 = s_3 \cdot j_2) \\
&\quad \wedge \neg (i_2 < n_1)
\end{aligned}
$$

We can therefore conclude that:

$$\begin{aligned}
\varphi(\texttt{B1}) &= \mathit{true} \\
\varphi(\texttt{B2}) &= (n_1 = 100) \;\wedge\; (i_1 = 0) \;\wedge\; (j_1 = 0) \;\wedge\; (s_1 = 1) \\
&\quad \wedge \exists \hat{v} : ((i_3 = i_1 + \hat{v}) \;\wedge\; (j_3 = j_1 + 2\hat{v})) \\
\varphi(\texttt{B3}) &: (n_1 = 100) \;\wedge\; (i_1 = 0) \;\wedge\; (j_1 = 0) \;\wedge\; (s_1 = 1) \\
&\quad \wedge \exists \hat{v} : ((i_3 = i_1 + \hat{v}) \;\wedge\; (j_3 = j_1 + 2\hat{v})) \\
&\quad \wedge (i_2 = i_3 + 1) \;\wedge\; (j_2 = j_3 + 2) \;\wedge\; (s_2 = s_3 \cdot j_2) \\
&\quad \wedge \neg(i_2 < n_1)
\end{aligned}$$

Chapter 6

Program with Aliases

Aliases refer to the phenomenon that, in many program languages, storage locations can be accessed in two or more ways. Alias information is central to determining what memory locations are modified or referenced, which can affect the precision and efficiency of data-flow analyses required for performing optimizations. Similarly, choosing a strorage model on which aliasing information can be represented effectively is essential to the soundness and completeness for validating optimizations.

6.1 Modeling Aliases in Transition Systems

Consider the following segment of code that involves access to integer variables a_1, a_2 and b:

At the first glance, the value of b at L2 should be 3. However, if the storage

```
    ...
L1: a1 <- 1
    a2 <- 2
    b  <- a1 + 2
L2: ...
```

Figure 6.1: Example: a program that may have aliases

location of a_1 overlaps with that of a_2, the assignment to a_2 simultaneously modifies a_1 to be 2 and, subsequently, the value of b at L2 is 4. So the value of b at L2 can be either 4 or 3, depending on whether a_1 and a_2 are aliased with each other or not.

Up till this point, every time we translate a program into a transition system that represents the program's formal semantics, we always take the set of program variables as state variables and represent reference of a program variable as accesses to its corresponding state variable. For instance, the transition system corresponding to the program in Fig. 6.1 has a set of state variables $\{a_1, a_2, b, pc\}$ where pc is the control variable, and the transition from L1 to L2 is written as

$$(pc = 1) \ \wedge \ (pc' = 2) \ \wedge \ (a_1' = 1) \ \wedge \ (a_2' = 2) \ \wedge \ (b' = a_1' + 2)$$

Such a translation makes an implicit assumption that none of the program variables aliases with each other and each variable can only be accessed explicitly by its name. As a result, the transition system above only allows the value of b at L2 to be 3, which is not exactly the case in the original program.

Generally speaking, a program's state is the state of its memory as well as the set of registers. If we assume that no inter-procedural optimizations are performed, the program state that we are concerned with is the value of control variable, the state of memory locations corresponding to global, local variables and the state of registers as well. If the program has dynamically allocated storage areas, we also have to consider the state of heaps. For the sake of simplicity, our intermediate language IR does not allow operator of dynamic allocation. Natually, we can choose a control variable, a memory array and program registers as the set of state variables of the transition system. Although, in the intermediate language level, we may not know the architecture of the machine that the compiler targets to, nor do we know the physical location in which each variable resides, we can still associate each program object with an *abstract memory location*, e.g., the abstract memory location of a variable x is denoted as $addr_x$. By introducing a virtual memory array mem whose indices are abstract memory locations, we introduce, in the first order logic we use, the following array expressions to represent accesses to variables:

- $\mathsf{sel}(\mathsf{mem}, loc)$ returns the value of mem at location loc;
- $\mathsf{upd}(\mathsf{mem}, loc, val)$ returns a new memory array which is the same as mem except that content of mem at location loc is replaced by val.

Furthermore, we augment IR with the following language elements to allow for variable of array type and indirect access with pointers:

- expression $\&id$ that denotes the address of variable id;
- expression $[e]$ that denotes indirect read from memory location computed in expression e;

- statement $[e_1] \leftarrow e_2$ that denotes indirect write to memory location e_1 with value of e_2.

For the sake of simplicity, we only allow variables of type integer and array of integers, and we assume the size of type integer is 1. Accesses to an array element $a[i_1] \ldots [i_m]$ of an m-dimensional array a can be represented, in IR language, as $[e]$ where e is an expression that computes the address of $a[i_1] \ldots [i_m]$ using the array addressing rule. Notice that control variable and program registers do not have aliases, hence can be accessed directly by their name.

Fig. 6.1 describes the rules to translate from expressions and instructions of IR language to terms and predicates in transition relation. We use $\mathbb{T}(e)$ to denote the term corresponding to expression e in transition relation. We use id to represent a program variable's name, and reg to represent a program register's name. The difference between a program variable and a register is that the former may be aliased, hence has to be accessed by its address, while the latter has no aliases, hence can always be accessed by its name. In general, we assume that any variable whose address is never computed in the program does not have aliases, hence can be accessed directly by its name.

For the example in Fig. 6.1, a correct definition of the corresponding transition system has mem and pc as its state variables, and the transition from L1 to L2 is

$$\begin{aligned}
&(pc = 1) \ \wedge \ (pc' = 2) \\
\wedge\, &(\mathsf{mem}_1 = \mathsf{upd}(\mathsf{mem}, addr_{a_1}, 1)) \ \wedge \ (\mathsf{mem}_2 = \mathsf{upd}(\mathsf{mem}_1, addr_{a_2}, 2)) \\
\wedge\, &(\mathsf{mem}_3 = \mathsf{upd}(\mathsf{mem}_2, addr_b, \mathsf{sel}(\mathsf{mem}_2, addr_{a_1}) + 2)) \ \wedge \ (\mathsf{mem}' = \mathsf{mem}_3)
\end{aligned}$$

Expression e in IR	Term $\mathbb{T}(e)$ in Transition Relation
id	$\mathsf{sel}(\mathsf{mem}, addr_{id})$
reg	reg
$\&id$	$addr_{id}$
$[e]$	$\mathsf{sel}(\mathsf{mem}, \mathbb{T}(e))$
e_1 op e_2	$\mathbb{T}(e_1)$ op $\mathbb{T}(e_2)$

Statement in IR	Transition Relation
$id \leftarrow e$	$\mathsf{mem}' = \mathsf{upd}(\mathsf{mem}, addr_{id}, \mathbb{T}(e)) \ \wedge \ pres(V - \{\mathsf{mem}\})$
$reg \leftarrow e$	$reg' = \mathbb{T}(e) \ \wedge \ pres(V - \{reg\})$
$[e_1] \leftarrow e_2$	$\mathsf{mem}' = \mathsf{upd}(\mathsf{mem}, \mathbb{T}(e_1), \mathbb{T}(e_2)) \ \wedge \ pres(V - \{\mathsf{mem}\})$

Table 6.1: Translation from IR to Transition Relation

6.2 Data Mapping with Aliasing

For programs with aliased variables, the data mapping takes a more complicated form. Since we do not allow arithmetic on pointers, the layout of variables in memory does not affect the correctness of program transformation. We can therefore safely assume that, for an potentially aliased variable v, it has the same abstract storage location in source and target programs. That is,

$$addr_{v_S} = addr_{v_T} = addr_v$$

. Under this assumption, if every variable located in memory is equivalent in source and target, we should have $\mathsf{mem}_S = \mathsf{mem}_T$. However, due to transformations such as code motion and dead code elimination, mem_S and mem_T may only have some of their elements identical. Consequently, we require a data mapping that can express the equality between every object in source memory array and its counterpart in target. That is, for a variable v stored in virtual memory array mem, we define the data mapping of v as follows:

- $\mathsf{sel}(\mathsf{mem}_S, addr_v) = \mathsf{sel}(\mathsf{mem}_T, addr_v)$ if v is a scalar variable;
- $\forall i \in [0..sz_v - 1] : \mathsf{sel}(\mathsf{mem}_S, addr_v + i) = \mathsf{sel}(\mathsf{mem}_T, addr_v + i)$ if v is an array of size sz_v.

As described in Section 3.1, we allow the data mapping of v be guarded by a condition $\texttt{pc} \in \mathcal{L}$, where $\texttt{pc}$ is a target program counter and $\mathcal{L}$ is a set of target program locations.

6.3 Representing Aliasing Information

6.3.1 Aliasing Analysis

Aliasing analysis refers to the determination of storage locations that may be accessed in two or more ways. High-quality aliaisng information produced by aliasing analysis is essential to correct and aggressive optimizations. Taking the program in Fig. 6.1 as an example, constant folding can replace the assignment to b with either $b \leftarrow 4$ or $b \leftarrow 3$ if the aliasing relation between a_1 and a_2 is provided. Without such information, however, neither of the above assignments can replace the assginment to b. Determining the range of possible aliases in a program is crucial to optimizing it correctly, while minimizing the sets of aliases found is important to aggressive optimizations.

The information computed by aliasing analysis ([And94, Ste96, Cou86, LH88, Hin01], etc.) can be classified by the following dimensions:

- **"May" versus "Must"** It is useful to distinguish *may* alias information from *must* alias information. The former indicates what *may* occur on all paths through the flowgraph, while the latter indicates what *must* occur on all paths through a flow graph. Consider the C programs in Fig. 6.2. The program in (a) has p = &x on both branches of an IF statement, thus "p points to x" is must alias information after the IF statement at `Li`. On the other hand, the program in (b) has q = &y on one branch of an IF statement and q = &z on the other, then "q may point to y or z" is may alias information at `Li`. Must alias information tells us properties that must hold, and is desirable for aggressive optimizations; may alias information provides range of possible aliases, and so is important for safe optimizations.

```
    . . .
    if (bool) {
      . . .
      p = &x;
    } else {
      . . .
      p = &x;
    }
Li: . . .
```

(a)

```
    . . .
    if (bool) {
      . . .
      q = &y;
    } else {
      . . .
      q = &z;
    }
Li: . . .
```

(b)

Figure 6.2: Examples of May and Must Alias Information. In both programs, variables p,q and $bool$ are not aliased, and $bool$ is not redefined in either branch of the IF statement.

- **Flow-insensitive versus Flow-sensitive** *Flow-insensitive* alias information is independent of the control flow encountered in a

procedure, while *flow-sensitive* alias information depends on control flow. An example of approahces that produces flow-insensitive information is the minimal pointer analysis in C program, which assumes that only variables whose address are computed are aliased and that any pointer-valued variable may point to any of them. On the other hand, the above mentioned "p must point to x at `Li`" and "q may point to y or z at `Li`" are both flow-sensitive information. By disregarding the control flow information, flow-insensitive analyses compute a single conservative summary either for the whole program or for each procedure, whereas flow-sensitive analyses requires that one follow the control-flow paths through the flow graph, and computes a solution for each program point. Flow-insensitive analyses can be more efficient, but less precise than a flow-sensitive analysis.

The sources of aliases vary from language to language. Generally speaking, aliases are present because of

1. overlapping of the memory allocated for two objects;
2. references through pointers;
3. references to arrays, array sections, or array elements;
4. parameter passing.

Although the sources of aliases depends on language-specific rules, there is also a component of alias analyses that is common to every languages. For example, a language may allow two variables to overlay each other or may allow one to be a pointer to the other or not, but regardless of these language-specific rules, if a variable a is pointed to by variable b and b is pointed to by c at the same time, then a is reachable by following pointers from c. Thus, the alias compuation can be devided into two parts [Cou86, Muc97]:

1. a language-specific component, called the *alias gatherer*, that is expected to provided by the compiler front end;

2. a single component in the optimizer, called the *alias propagator*, that performs data-flow analysis using the aliasing relations discovered by the front end to combine aliasing information and transfer it to the points where it is needed.

6.3.2 Aliasing Information as Program Invariants

As aliasing analysis is essential to performing most optimizations correctly, being able to encode aliasing information as program invariant is crucial for establishing the correctness of translations.

For program variables x and y whose sizes are sz_x and sz_y, respectively, there are the following basic aliasing relations between x and y that we care about:

- The memory locations of x and y overlap. This scenario can be captured by the predicate $addr_x + c = addr_y$ with some integer constant $0 \leq c < sz_x$. Notice that x is a composite variable and $c > 0$ represents the senario of partial overlapping of x and y.

- x points to the memory location of y. This can be expressed by the predicate $x = addr_y$.

- y is an array, and x points to some element of y. Because we do not know the exact position in y that x points to, we use a quantified boolean expression $\exists i \in [0..sz_y - 1].\mathbb{T}(x) = addr_y + i$ to represent such situation.

- x is an array of pointers, and elements of x must point to the set of memory location $\mathcal{L}$. Again, this is expressed by a quantified boolean expression $\forall i \in [0..sz_x - 1]. \bigvee_{loc \in \mathcal{L}} \big(\mathsf{sel}(\mathsf{mem}, addr_x + i) = loc\big)$.

- The memory location of y is reachable through arbitrarily many dereferences from x. This scenario can not be expressed directly in first order logic. However, due to the fact that we only have finite memory locations corresponding to global and local variables in a procedure, the "reachability" information can be expressed as a conjunction of a set of points-to predicates, whose number is bounded by the number of different memory locations.

Any type of aliasing information is based on the above basic relations and can be expressed as program invariants that are boolean combination of the predicates mentioned above as well as other types of predicates. For instance, a single predicate $(p = addr_x)$ indicates that p *must* point to x, while a disjunction $(q = addr_y) \vee (q = addr_z)$ indicates that q may point to y or z. We can express flow-sensitive information by taking constraints over the program counter and path conditions into account. E.g., in the program in Fig. 6.2 (a), the fact that p points to x at location `Li` can be expressed as

$$(\mathtt{pc} = \mathtt{Li}) \rightarrow (\mathtt{p} = \mathtt{addr_x})$$

and the fact that, at location L_i in program (b), q either points to y when $bool$ holds or points to z otherwise can be expressed as

$$(pc = \mathtt{Li}) \rightarrow \big((\mathtt{p} = \mathtt{addr_y})\ \wedge\ \mathtt{bool}\ \vee\ (\mathtt{p} = \mathtt{addr_z})\ \wedge\ \neg\mathtt{bool}\big)$$

As described in Subsection 6.3.1, an aliasing analyzer consists of two components: a language-specific *alias gatherer* in the compiler front-end and a language independent *alias propagator* in the optimizer. Analogously, part of the invariants related to aliasing information comes from language-specific rules, and the rest are generated by propagating language-specific aliasing information along pahts in the flow graph.

Generating Language-specific Aliasing Invariants For the IR language we are concerned, the following aliasing rules apply:

1. *Aliasing between different objects*: Memory locations allocated for different objects never overlap. For the sake of simplicity, we assume that IR language allows either scalar objects or composite objects that are arrays. Then, for variables x and y, one of the following constraints holds at every location in the program, depending on the type of x and y:

- $addr_x \neq addr_y$, if x and y are scalars;
- $(addr_x < addr_y) \lor (addr_x > addr_y + sz_y)$, if x is a scalar and y is an array;
- $addr_x + sz_x < addr_y \lor addr_x > addr_y + sz_y$, if both x and y are arrays.

2. *References through pointers*: In order to perform sound and effective pointer analysis, we have to restrict the operations allowed over pointers. In ANSI C standard, the behavior of code that has arithmetic on pointers is considered to be undefined [ANS89]. Since it is meaningless to apply the validation efforts to the code whose semantics is undefined, we assume that there is no arithmetics over pointers in programs in the source language. Thus, for a program variable p that is a pointer, we have

$$p \in \{addr_v | v \in V\} \cup \{\texttt{nil}\}$$

where V is the set of variables in the program and `nil` represents a null pointer value which an uninitialized pointer takes. We also assume that a read through a `nil` pointer returns a null pointer, and a write through `nil` point does not change the memory. That is,

$$\begin{aligned} &\mathsf{sel}(\mathsf{mem}, \texttt{nil}) = \texttt{nil}, \\ &\mathsf{upd}(\mathsf{mem}, \texttt{nil}, \texttt{val}) = \mathsf{mem}. \end{aligned}$$

Besides, since we do not assume any particular layout of the memory, the only relational operations we allow over pointers the comparison for equality (and inequality) between two pointers.

3. *References to array elements*: In the IR language, array elements are accessed through the virtual memory mem with the appropriate address computed using array addressing rules. In a program that is well behaved, references to array elements should have all the array indices within specified bounds. When a reference to an element of array a is translated to IR language as $[e]$ where e computes the address of the array element, the value of e should always within the array bounds. That is, we constrain e by

$$addr_x \leq e < addr_x + sz_x$$

   Such constraints are indispensible to formally establishing alising invariants since it is not always possible to statically determine whether the indices of array elements are within specified range and, without such constraints, accesses to array elements can modify any part of the memory, making aliasing analysis impossible.

4. *Parameter passing*: In the IR language, we assume that parameters are always passed by value and we do not allow pointers to be passed as parameters. Since we do not have inter-procedural analysis at this moment, we also assume that function calls do not have side-effect. That is, function calls do not modify variables in the scope of the calling procedure.

Propagating aliasing invariants through Flow Graph The algorithm GenInv presented in Section 5.3 can propagate aliasing infomation as a data flow through paths in flow graph, being able to generate flow-sensitive aliasing invariants in most of the cases.

```
int arith(n)
    int n;
{   int i, j, k, *p, *q;
     . . .
    p = &i;
    i = n + 1;
    q = &j;
    j = n * 2;
Li:
     . . .
Lj: k = *p + *q;
     . . .
}
```

Figure 6.3: An example of aliasing invariants generation

$$
\begin{array}{llll}
 & \dots & & \\
 & p_1 & \leftarrow & \&i \\
 & \mathsf{mem}_1 & \leftarrow & \mathsf{upd}(\mathsf{mem}_0, \&i, n+1) \\
 & q_1 & \leftarrow & \&j \\
 & \mathsf{mem}_2 & \leftarrow & \mathsf{upd}(\mathsf{mem}_1, \&j, n*2) \\
Li: & & & \\
 & \dots & & \\
Lj: & k & \leftarrow & \mathsf{sel}(\mathsf{mem}_2, p) + \mathsf{sel}(\mathsf{mem}_2, q) \\
 & \dots & &
\end{array}
$$

Figure 6.4: IR code in SSA form for the C code in Fig. 6.3

Here, we present an example of aliasing invariants generated by algorithm

GenInv. Consider the C code shown in Fig. 6.3 and the corresponding IR code in SSA form in Fig. 6.4. In the program, only variables i and j have their addresses computed, hence they are the variables potentially aliased, while the other variables can be accessed directly. Applying alogrithm GenInv to the IR code, we obtain, at location `Li`, the invariant

$$\begin{aligned} p_1 = addr_i \wedge \mathsf{mem}_1 = \mathsf{upd}(\mathsf{mem}_0, addr_i, n+1) \\ \wedge\, p_2 = addr_j \wedge \mathsf{mem}_2 = \mathsf{upd}(\mathsf{mem}_1, addr_j, n*2) \end{aligned}$$

If there exists no definition of p and q in the paths between `Li` and `Lj`, optimizations will replace `k=*p+*q` by `k=i+j` and remove the assignments to p and q completely. This transformation is possible due to the fact that, at `Lj`, p and q points to i and j, respectively, which is caught by the invariant $(p = addr_i \wedge q = addr_j)$ propagated from `Li` to `Lj`.

As discussed in Subsection 5.3.2, algorithm GenInv does not propagate data-flow information iteratively, thus it cannot proceed around cycles in flow graphs. For example, consider the following fragment of a program in C:

```
int *p, *q;
. . .
for (q = p; q == NIL; q = *q) {
 . . .
}
```

We need to perform iterative data-flow analysis to discover that, when executing the loop, pointer q may point to any memory location reachable from p before the loop. Thus, any memory location reachable from p may be modified through q in the loop. An algorithm that is able to deal with this type of scenario is described in [Muc97], and we can use the same algorithm to generate the invariants of aliasing information generated by loops.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In the previous chapters, we describe a general rule VALIDATE and its application towards the automatic validation of compiler optimizations that are structure-preserving. As part of the *Translation Validation of Optimizing Compiler* project that is being carried on at New York University, the prototype tool TVOC-SP is successful in verifying various optimizations performed by the global optimizer of Intel's ORC compiler with very little instrumentation from the compiler. The success is attributed to the following aspects:

A versatile proof rule To compare two programs that terminate, what we care about is essentially the matching of their final states, which is established through a set of inductive steps along the execution. Refinement mapping that correlates the cut points in the middle of programs is only the result of an inductive step towrads the end of programs. Therefore, the refinement mapping established between source and target programs can be flexible in several dimensions. First, the control mapping is allowed to correlates particular locations instead of every location in the programs, and the criteria is that at least one control point is required for each loop. Second, data abstraction is allowed to be partial, because after all some of the variables are dead and others that are not mentioned in the data abstraction can have their data information carried to the final states as program invariants. That is, The invariants of source and target programs serve as a supplement of data abstraction: it captures and propagates the information that is indispensible to the matching of final states, but is missing in data abstraction. Still, since each inductive step can only be taken within one iteration of a loop, this method requires that each iteration of the corresponding loops in the source and target have more or less the same effect, and that is why rule VALIDATE cannot apply to optimizations that transform loops' structure.

Various program analysis techniques Studies of compilers' individual optimizations indicate that the auxiliary invariant necessary in the validation a particular optimizationc can also be detected by the same static analysis performed for the optimization. However, different optimizations call for different static analysis techniques, each of which exploring some aspect of the program's semantics [SS98], and it will be costly to repeat every single data-flow analysis performed for global optimizations, considering that many of them involve iterative fix-point computation. As a matter of fact, the first invariant generation method we designed (see Subsection 5.3.1 for details) attempts to mimic the iterative data flow analyses performed by compilers and managed to produce necessary invariants for some common optimizations such as constant folding, common expression elimination, etc. But there is other data flow information this first method fails to capture, among which there are constant variables recognized by sparse conditional constant folding [WZ91], and equality between variables detected by global value numbering [AWZ88]. Then we designed the second invariant generation method GenInv (see Subsection 5.3.2) based on the following observation: although it requires basically the same amount of data flow information to perform an optimization as to validate it, the compiler optimizer requires the information to be *explicitly* while the validator allows for the same information hidden *implicitly* in the auxiliary invariants, because later a theorem prover will "rediscover" the information when it checks the validity of verification conditions. Thus, since the theorem prover serves as an information interpreter and the GenInv procedure merely as an information collector, the procedure to generate invariants becomes much more efficient.

In addition to auxiliary invariants over the variables of a single program, we also construct data abstraction that correlates variables in source and target programs, by performing an iterative data flow analysis operated on both programs at the same time (see Section 5.2 for details). We made such an analysis possible by choosing a fairly simple lattice (where the elements are a set equality between source and target variables) and computing the flow functions using a theorem prover.

Limitation The current invariant generation algorithm GenInv still has two drawbacks: first, it fails to carry information along backward edges in CFG, thus is not as powerful as the iterative data flow analyses which can proceed information around cycles in control flow graphs; second, the invariants it produces for a cut point includes definitions of almost every SSA variables that appear in the program before that cut point, sometimes carrying too much unnecessary data information, which will slow down the theorem prover. The possible improvement for the first drawback can be a GenInv algorithm augmented with iterative analysis, and the improvement for the second one is to apply "program slicing" techniques [Tip95] to eliminate irrelavent SSA definitions.

Because of the impotence of the static analysis techniques, a translation validator can produce "false alarms", which means, for a source and its correct translation, the validator may construct a set of verification conditions, some of which are invalid. A false alarm is caused either by the incompetence of the validator, i.e. it does not find the right refinement abstraction, or does not produce sufficient auxiliary invariants; or by the incompleteness of the proof rule VALIDATE, i.e., it cannot be applied to establish the correctness of structure modifying transformations.

7.2 Future Work

There are many directions for future work. We list a few as follows:

- *Procedures and inter-procedural optimizations* To deal with programs with procedures, we need to extend the current proof rule. If inter-procedural optimizaitons are involved, extra auxiliary invariants will be required to serve as a “summary” of the procedure, under which the inter-procedural optimizaitons are possible.

- *Exception handling* Our current notion of a correct translation does not consider programs with exceptions. There are surely issues of exceptions that affect the correctness of a translation. E.g., translation that introduces exceptions is considered to be incorrect. The current theory need to be extended to deal with exceptions.

- *Hardware related optimizations* This is a category that is of particularly importance to the performance of EPIC architecture, where instruction scheduling is performed by compilers. The validation of hardware related optimizations is still a big challenge.

- *Counter examples* The counter-examples obtained from tool TVOC-SP are the states of source and target programs that invalidate a verification conditions. When a translation cannot be established to be correct, we hope the validator to provide a “witness” that are unmatching program executions in source and target.

- *Sefl-certified compiler* One of the side-products we anticipate from this work is the formulation of validation-oriented instrumentation, which will instruct writers of future compilers how to incorporate into the optimization modules appropriate additional outputs which will facilitate validation. This will lead to a theory of construction of *self-certifying* compilers. Some work in this area has been done in this thesis research, including how to construct well formed VALIDATE proofs for a set of individual optimziaitons, and how to compose two well formed proofs. Still, there is more work to do in this area.

Bibliography

[And94] L. O. Andersen. *Program Analysis and Sepcialization for the C Programming Languag*. Phd thesis, DIKU, University of Copenhagen, 1994.

[ANS89] *American National Standard for Information Systems: Programming Language C, ANSI X3.159-1989*. American National Standards Institute, New York, NY, 1989.

[AWZ88] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–11, New York, NY, USA, 1988. ACM Press.

[BB04] Clark Barrett and Sergey Berezin. CVC Lite: A new implementation of the cooperating validity checker. In Rajeev Alur and Doron A. Peled, editors, *Proceedings of the 16^{th} International Conference on Computer Aided Verification (CAV '04)*, volume 3114 of *Lecture Notes in Computer Science*, pages 515–518. Springer-Verlag, July 2004. Boston, Massachusetts.

[CFR$^+$89] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–35, New York, NY, USA, 1989. ACM Press.

[CJW01] S. Chan, R. Ju, and C. Wu. Tutorial: Open research compiler (orc) for the itanium processor family. In Micro 34, 2001.

[Cou86] D. S. Coutant. Retargetable high-level alias analysis. In *POPL '86: Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Prin-*

ciples of programming languages, pages 110–118, New York, NY, USA, 1986. ACM Press.

[CRF$^+$91] R. Cytron, Ronald, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the program dependence graph. In *ACM TOPLAS*, volume 13, pages 451–490, New York, NY, USA, 1991. ACM Press.

[Flo67] R.W. Floyd. Assigning meanings to programs. *Proc. Symposia in Applied Mathematics*, 19:19–32, 1967.

[GGB02] S. Glesner, R. Geiß and B. Boesler. Verified code generation for embedded systems. In Proc. of Compiler Optimization meets Compiler Verificaiton (COCV) 2002, *Electronic Notes in Theoretical Computer Science (ENTCS)*, volume 65, 2002.

[Hin01] M. Hind. Pointer analysis: haven't we solved this problem yet? In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 54–61, New York, NY, USA, 2001. ACM Press.

[JC97] J. Janssen and H. Corporaal. Making graphs reducible with controlled node splitting. *ACM Trans. Program. Lang. Syst.*, 19(6):1031–1052, 1997.

[LH88] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 24–31, New York, NY, USA, 1988. ACM Press.

[LJWF04] David Lacey, Neil D. Jones, Eric Van Wyk, and Carl Christian Frederiksen. Compiler optimization correctness by temporal logic. *Higher Order Symbol. Comput.*, 17(3):173–206, 2004.

[Muc97] S.S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann, San Francisco, 1997.

[Nec00] G. Necula. Translation validation of an optimizing compiler. In *Proceedings of the ACM SIGPLAN Conference on Principles of Programming Languages Design and Implementation (PLDI) 2000*, pages 83–95, 2000.

[PSS98a] A. Pnueli, M. Siegel, and O. Shtrichman. The code validation tool (CVT) - automatic verification of a compilation process. *Software Tools for Technology Transfer*, 2, 1998.

[PSS98b] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Proc.* 4^{th} *Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98),* pages 151–166, 1998.

[RM00] M. Rinard and D. Marinov. Credible compilation with pointers. In *Proceedings of the Run-Time Result Verification Workshop*, Trento, July 2000.

[SS98] D. Schmidt and B. Steffen. Program analysis as model checking of abstract interpretations. In G. Levi, editor, Proc. 5^{th} Intl. Static Analysis Symposium (SAS'98), *volume 1503 of* Lect. Notes in Comp. Sci., pages 351–380. Springer-Verlag, 1998.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, New York, NY, USA, 1996. ACM Press.

[Tip95] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[WZ91] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, 1991.

[ZPFG03] L. Zuck, A. Pnueli, Y. Fang, and B. Goldberg. Voc: A methodology for the translation validation of optimizing compilers. *Jounal of Universal Computer Science*, 9, 2003.

[ZPG$^+$02] L. Zuck, A. Pnueli, B. Goldberg, C. Barrett, Y. Fang, and Y. Hu. Translation and run-time validation of loop transformations. In Proceedings of the Run-Time Result Verification Workshop, *Electronic Notes in Theoretical Computer Science (ENTCS),* volume 70, 2002.