Mobius

Information Society Technologies

Project N°: **FP6-015905**

Project Acronym: **MOBIUS**

Project Title: **Mobility, Ubiquity and Security**

Instrument: **Integrated Project**

Priority 2: **Information Society Technologies**

Future and Emerging Technologies

Deliverable D4.5

Report on proof transformation for optimizing compilers

Due date of deliverable: 2008-08-31 (T0+36)

Actual submission date: 2008-10-15

Start date of the project: **1 September 2005** Duration: **48 months**

Organisation name of lead contractor for this deliverable: **IoC**

Revision 8093M

Project co-funded by the European Commission in the Sixth Framework Programme (2002-2006)		
Dissemination level		
PU	Public	✓
PP	Restricted to other programme participants (including Commission Services)	
RE	Restricted to a group specified by the consortium (including Commission Services)	
CO	Confidential, only for members of the consortium (including Commission Services)	

Contributions

Site	Contributed to Chapter
IoC	1, 2, 3, 4, 7
INRIA	5, 6

This document was written by Gilles Barthe (INRIA), Benjamin Grégoire (INRIA), César Kunz (INRIA), Tamara Rezk (INRIA), Ando Saabas (IoC), Tarmo Uustalu (IoC).

Executive Summary:
Report on proof transformation for optimizing compilers

This document summarises Deliverable 4.5 of project FP6-015905 (MOBIUS), co-funded by the European Commission within the Sixth Framework Programme. Full information on this project, including the contents of this deliverable, is available online at `http://mobius.inria.fr/`.

The deliverable reports progress on Task 4.4 on proof-transforming compilation. A proof-transforming compiler is a tool for the code producer that not only compiles a given source program, but also transforms a given proof of it (obtained, e.g., by interactive theorem proving) automatically into a proof of the compiled form. This is a lifting of the idea of a certified compiler, which automatically produces certificate of a program's conformance to some fixed safety policy, to situations where we are also interested in functional correctness or in general conformance to freely chosen policies.

Proof transformation for a non-optimizing compiler is a relatively straightforward matter, at least in theory: it is basically identity, modulo a suitable mapping between the control and memory models of the source and the target languages. (We have discussed the technical details for Java to JVML compilation in Deliverable 4.3, which was the first deliverable from Task 4.4.)

Optimizations, however, change the picture drastically. An optimization takes a program to one whose semantics is generally only similar to the given one, so the proof must also change. Moreover, an optimization may change a program's structure, in which case also the structure of the proof (not just the assertions and entailments between them) will change.

The purpose of this deliverable is to study exactly the situation of program optimizations, covering both optimizations of high-level programs and optimizations of low-level code. We look at ways to document analyses and optimizations and to organize proof transformation guided by the same analysis information that decides the outcome of the program optimization. We describe one technology for transformation of Hoare logic proofs and another for transformation of weakest precondition calculus proofs (i.e., of proof obligations from a verification condition generator (VCGen) and their proofs). Our conclusion is that automatic proof transformation for program optimizations is perfectly viable, but setting it up requires care. Essentially, proof transformation amounts to exploiting the general soundness of the optimization, instantiated potentially to any program of interest. So the issue at stake is the same as with optimization soundness—how to organize its proof in the best (most transparent and least error-prone) possible way.

The emphasis of the deliverable is on syntactic frameworks for describing analyses and optimizations that can support automatic proof transformation in a systematic way. Since this is exploratory work at an initial stage, it is intentional that we do not consider full Java/JVML. Neither do we concentrate on any specific fixed set of optimizations for Java to JVML compilation, but take the liberty to look at a range of optimizations, including some that are quite challenging.

Contents

1 Introduction: Proof transformation for program optimizations **5**

2 Program optimization and proof transformation of Hoare logic proofs with type systems **8**

3 Stack usage optimizations **11**

4 Partial redundancy elimination: a case study on optimizations modifying control flow **14**

5 Proof transformation for wp proofs **16**

6 An abstract interpretation model of proof transforming compilers **19**

7 Conclusions **22**

Chapter 1

Introduction: Proof transformation for program optimizations

Proof-carrying code (PCC) is based on the idea that in a security-critical code transmission setting, the code producer should provide some evidence that the program she distributes is safe and/or functionally correct. The code consumer would thus receive the program together with a certificate (proof) that attests that the program has the desired properties. For simple properties, automated generation of certificates can be used to prove the program correct. However, for more complex policies, the code producer would require some (interactive) verification environment to prove her source program. The question is how to communicate the verification result to the code consumer who will not have access to the source code of the program. It is clear that there should be a mechanism to allow compilation of the program proof together with the program.

It has been shown [4, 11] that proof compilation (automatic transformation of a program's proof alongside compiling the program) is simple in the case of a non-optimizing compiler. However the same does not hold, if optimizations take place—a valid proof of a program may not be valid for the optimized version of the same program.

A simple example illustrating this issue is given in Figure 1.1. In the left column, we have a program which, given suitable values for i, s, p and n, computes $c * n$ and c^n and saves the the results to s and p, respectively. The pre- and postcondition for the program are $\{s = 0 \wedge p = 1 \wedge i = 0 \wedge n \geq 0\}$ and $\{s = c * n \wedge p = c^n\}$ and a suitable loop invariant is $\{s = c * i \wedge p = c^i \wedge i \leq n\}$. If we now perform dead code elimination on the program assuming p to be dead at the end of the program (one could also call it program slicing), we would obtain the new program given in Figure 1.1 b. It is obvious that we can not state anything about p in the postcondition anymore. Consequently, the precondition and the loop invariant should also be relaxed.

It might seem that the reason why the assertions for the program (and therefor also the proof) need to be transformed in the given example is that the transformation did not preserve the semantics of the program (while the vast majority of standard program transformations are semantics preserving). But in fact the same issues apply for semantics preserving optimizations. This can be shown on the example of common sub-expression elimination, given in Figure 1.2. The original program is shown in column a). The pre- and postcondition for the program are $\{i = 0 \wedge x = 0 \wedge n \geq 0\}$ and $\{x = n * (a + b)\}$. A suitable loop invariant for proving the program correct is $x = i * (a + b) \wedge i \leq n$. Column b) shows the program after common subexpression elimination has been applied. The given loop invariant is not strong enough for the transformed program any more, since there is no reference to $(a + b)$ in the optimized loop. The transformation of the invariant into $x = i * t' \wedge i \leq n$ in accordance with the optimization is not sufficient either, since it would be too weak to imply the postcondition. Instead, the invariant needs to be strengthened to $\{x = i * (a + b) \wedge i \leq n \wedge t' = a + b\}$, recording the knowledge that t' is equal to $a + b$ in the loop. What can be observed on this example is that while the *whole program* optimization is indeed semantics preserving, the same can not be said about its *sub-statements*: for example $x := x + (a + b)$ is replaced by $x := x + t'$,

$a)$

$\{s = 0 \wedge p = 1 \wedge i = 0 \wedge n \geq 0\}$
while $i < n$ do
$\{s = c * i \wedge p = c^i \wedge i \leq n\}$
$s := s + c;$
$p := p * c;$
$i := i + 1;$
$\{s = c * n \wedge p = c^n\}$

$b)$

$\{s = 0 \wedge i = 0 \wedge n \geq 0\}$
while $i < n$ do
$\{s = c * i \wedge i \leq n\}$
$s := s + c;$
skip;
$i := i + 1;$
$\{s = c * n\}$

Figure 1.1: Example of annotation transformations required for dead code elimination

$a)$

$\{i = 0 \wedge x = 0 \wedge n \geq 0\}$

$t = a + b;$
while $i < n$ do
$\{x = i * (a + b) \wedge i \leq n\}$
$x := x + (a + b);$
$i := i + 1;$
$\{x = n * (a + b)\}$

$b)$

$\{i = 0 \wedge x = 0 \wedge n \geq 0\}$
$t' = a + b;$
$t = t';$
while $i < n$ do
$\{x = i * (a + b) \wedge i \leq n \wedge t' = a + b\}$
$x := x + t';$
$i := i + 1;$
$\{x = n * (a + b)\}$

Figure 1.2: Annotation transformations required for common subexpression elimination

which are obviously not semantically equivalent in general, but only when some additional assumptions can be made about the state. This is exactly the reason why the proof needs to be modified: the extra assumptions that are being made need to be recorded in the assertions.

These examples also make it clear that in general assertion transformation is not just strengthening nor weakening, but the assertions for the original and optimized program can be incomparable. In the dead code elimination example, *less* can be stated about the final result of the program after the optimization, so the corresponding assertion needs to be weakened. Same holds for the the loop invariant. Since less has to be assumed from the pre-state for the invariant to hold, the precondition can also be weakened. In the common subexpression elimination example, we saw that assertions needed to be strengthened. If we combined these two optimizations, the new assertions would be incomparable to the original ones.

Note also that the problems described here apply both to Hoare logic-based approach to program proofs and to wp-calculus proofs. In the following chapters, we look at both of these approaches and describe how proof transformation can be performed.

This deliverable is based on five published papers [13, 12, 14, 2, 3], produced by the partners that have been contributing to Task 4.4.

The report is organized as follows. In Chapter 2, summarizing [13], we report on a general method of describing program analyses and optimizations as type systems with a program transformation component and a method for obtaining mechanical transformation of Hoare logic proofs based thereupon. We explain the method on two simple but instructive examples for a high-level language, viz., dead assignment elimination and common subexpression elimination. The salient features of the type-systematic method are simple (metatheoretic) soundness proofs of optimizations (proof transformation is in fact a consequence of this) and also of their improvement properties.

In Chapter 3, summarizing [12], we show that the method scales to unstructured low-level languages and

consider the particular case of a JVML-like stack-based language. We demonstrate proof transformation for a number of stack-usage optimizations (like dead store elimination, load-pop elimination), some of which rely on bidirectional analyses, which is also of independent interest.

In Chapter 4, reporting on [14], we show that the approach scales up to complex and subtle optimizations that also change a program's control flow by edge-splitting. We consider the classical example of partial redundancy elimination, this time again for a high-level language.

Chapter 5 summarizes [2]. Here we study transformation of wp-calculus proofs. The language considered is a low-level language with expressions; the optimizations considered are constant propagation and dead assignment elimination. In this approach, ghost variables are needed to deal with dead assignment elimination, as only analyses that strengthen the proof annotations of the given program can be handled cleanly.

Chapter 6, describing [3], elaborates on the previous chapter and generalizes it by explaining the underlying ideas in terms of abstract interpretation. The resulting method can handle a number of analyses and optimizations and via a modification also the case where the analysis information is non-strengthening.

Chapter 2

Program optimization and proof transformation of Hoare logic proofs with type systems

In this paper [13], we develop a systematic approach for transforming Hoare program proofs alongside programs, using type systems as the means for describing dataflow-analysis and optimizations that stem from them. Using type systems offers a number of benefits compared to the standard, algorithmic descriptions. Type systems can explain analyses and optimizations well in a declarative fashion, separating the issues of determining what counts as a valid analysis or optimization result and how to find one. This makes soundness simple to prove by structural induction on type derivations. Automatic transformation of program proofs can then be seen as a formal version of the constructive content of these semantic arguments.

The paper demonstrates the type-systematic approach on two standard optimizations. We explain our general approach to translating dataflow analyses and optimizations to type systems on the example of dead code elimination, which is a simple optimizations requiring only one analysis to be performed. Still, it is interesting in being one of the few common optimizations which require the weakening of assertions in program proofs. The other example we look at is common subexpression elimination. This is an interesting optimization since it requires two analyses (with the second analysis relying on the results of the first) for linking of expression evaluation points to value reuse points and coordinated modifications of the program near both ends of such links, which seems to go against compositionality. We show how this can easily be overcome by a combined type system that reflects the combination of the two analyses. For an overview of the standard analyses underlying dead code and common subexpression elimination, the reader may turn to [5].

In our approach, types and subtyping relation correspond to the elements and ordering relation of the poset underlying the analysis. For example in the case of dead code elimination, a type is a set of variables (i.e. variables which are live) and subtyping is the reversed inclusion relation. The general form of a typing judgement corresponding to a dataflow analysis is $s : d \to d'$, stating that if a property d holds before running a program, then the property d' holds after running the program. Depending on the direction of the analysis, it can be stated contrapositively: if a property d' does not hold after the program run, then the property d does not hold before the program run. This is how the typing judgement for liveness analysis (the analysis used for dead code elimination) is interpreted: $s : \mathit{live} \to \mathit{live}'$ says that if the variables in live' are live in the end of the program, then variables in live must have been live in the beginning of the program. As an example, the typing rule for assignment in case of liveness analysis (the analysis underlying dead code elimination) is

$$\frac{}{x := a : \mathit{live} \setminus \{x\} \cup FV(a) \to \mathit{live}}$$

where $FV(a)$ is the set of free variables in a. The rule corresponds exactly to the intuition behind liveness analysis: using a variable in an expression makes it live, and assigning to variable kills it. This is

specified in the pretype, since the analysis is backward. For non-primitive statements, the typing rules are straightforward, for example the rule for an if-statement becomes

$$\frac{s_t : live \rightarrow live' \quad s_f : live \rightarrow live'}{\textsf{if } b \textsf{ then } s_t \textsf{ else } s_f : live \cup FV(b) \rightarrow live'}$$

A big difference of the type system from the analysis algorithm is that while the algorithm computes the weakest preproperty for a given postproperty, the type system approves any valid pretype-posttype pair. Stronger pretypes are easy to get from the weakest one with the subsumption rule. The analysis algorithm can in fact be seen as an algorithm for principal type inference: given a statement s and a posttype $live'$, one attempts to construct a type derivation. Constructing the tightest one takes calculation of greatest fixpoints with respect to $\leq$ to obtain the invariant-types of the loops and as a result one learns the weakest pretype *live*. This type declares only these variables to be possibly live initially that really have some chance of being live. In type systems jargon, it makes sense to call *live* the principal type of s with respect to $live'$.

The typing judgement for optimization takes the form $s : d \rightarrow d' \hookrightarrow s'$ and states that program s, having the pre- and posttype d and d' can be rewritten to s'. For example in the case of dead code elimination, assignment has two rules, for the case where it can be removed (replaced with a skip) and when it remains unchanged:

$$\frac{x \in live}{x := a : live \setminus \{x\} \cup FV(a) \rightarrow live \hookrightarrow x := a} \qquad \frac{x \notin live}{x := a : live \rightarrow live \hookrightarrow \textsf{skip}}$$

The second rule is where the optimization actually happens: if we know x to be certainly dead in the posttype, we can remove the assignment to it.

Using the type-system approach, it is easy to formulate and prove soundness of both the analysis, and the optimizations with respect to the natural semantics "relationally". For liveness analysis, let $\sigma \sim_{live} \sigma_*$ denote that two states σ and σ_* agree on all variables in a set *live*, i.e., $\bigwedge_{x \in live} \sigma(x) = \sigma_*(x)$. Soundness states that any program is simulated by itself with respect to $\sim$. For the optimization, the formulation is analogous, only we state that the original program s and its optimized version s' simulate each other. Proving soundness is now simple, using structural induction on the structure of the type derivation.

Soundness of the analysis and optimization with respect to the natural semantics has a formal counterpart in terms of derivations in the Hoare logic (which is a formal description of the semantics). Let $P|_{live}$ abbreviate the formula $\exists[v(x)|x \notin live](P[v(x)/x|x \notin live])$, where v is some assignment of unique logic variable names to program variables (so that, informally, $P|_{live}$ is obtained from P by quantifying out all program variables not in *live*). For example for assertion $P =_{\mathrm{df}} x = 2 \wedge y = 7$ and type $live = \{x\}$, $P|_{live}$ is $\exists y' \, (x = 2 \wedge y' = 7)$. Essentially, this mean that we do not assume anything from, or guarantee anything about dead variables in the optimized program. We then obtain the following theorem: *If* $s : live \rightarrow live' \hookrightarrow s'$*, then 1)* $\{P\}\, s\, \{Q\}$ *implies* $\{P|_{live}\}\, s'\, \{Q|_{live'}\}$.

The constructive content of the first half of this theorem gives us "proof optimization". Given a Hoare triple derivation for an original program, we get a modified Hoare triple and its derivation for its optimized form.

The other example we look at is common subexpression elimination, an optimization to avoid re-evaluation of non-trivial expressions by saving the evaluation result in a temporary variable. This is technically interesting since the standard presentation requires linking of expression evaluation points to potential value reuse points and coordinated modifications of the program near both ends of such links, which seems to go against compositionality. We solve the problem with a combined type system for a combination of two analyses, with the second analysis relying on the results of the first. The first analysis is available expressions analysis, a forward analysis which computes which expressions have already been precomputed, and later not modified at a program point. The second analysis, conditional partial anticipability (*cpant*), uses this information: as the name of the analysis implies, an expression is considered partially anticipable only if it is also available. The typing judgement thus has the form $s : av, cpant \rightarrow av' cpant'$. Like with dead code elimination, soundness of the optimization can easily be stated "relationally": if at corresponding program points the states of the two programs agree on the normal program variables and,

moreover, if some expression is in the *cpant* type, then its value in the state of the original program and the value of the corresponding auxiliary variable in the state of the optimized program coincide, then the two programs simulate each other. Again, a similar statement can be made about Hoare logic. We define $P|_{cpant}$ to abbreviate $P \wedge \bigwedge_{a \in cpant} a = nv(a)$, where $nv(a)$ is the temporary variable introduced to hold the value of the expression a. The theorem then states that if $s : av, cpant \to av' cpant' \hookrightarrow s'$, then $\{P\}\, s\, \{Q\}$ implies $\{P|_{cpant}\}\, s'\, \{Q|_{cpant'}\}$. The constructive content of this theorem gives us the algorithm for automatic transformation of Hoare proofs for common subexpression elimination.

Chapter 3

Stack usage optimizations

In the previous chapter we looked at transformations that operate on high-level structured code and corresponding Hoare proof transformations. Also, we looked only at standard unidirectional analyses i.e. analyses which propagate information only forward or backward in the control flow graph (CFG). While the majority of analyses are of this form, there are many useful bidirectional ones. One place where bidirectional analysis appear naturally are optimization analyses for bytecode. We look at them in [12], where we address a simple bytecode language. The unstructured nature of it facilitates the use of CFG based description of the analyses (and thus the proof transformation) and the presence of stacks requires many non-trivial analyses to be bidirectional.

In the paper, we give uniform formal declarative descriptions, soundness proofs and (in the full version) algorithms for a number of optimizations and their underlying analyses. Like in [13], the tool we use for this purpose are type systems with a transformation component.

Optimizing bytecode offers challenges not present in high or intermediate-level program optimizations. A few reasons can be outlined. Firstly, expressions and statements are not explicit. In a naive approach, a reconstruction of expression trees from instructions would be required for many optimizations. Secondly, related instructions are not necessarily next to each other. A value could be put on the stack, a number of other instructions executed and only then the value used and popped. This means that related instructions, for example those that put a value on a stack, and those that consume it, can be arbitrarily far apart. Links between them need to be found during the analysis. Thirdly, a single expression can span several different basic blocks. The Java Virtual Machine specification does not require zero stack depth at control flow junctions, so an expression used in a basic block can be partially computed in other basic blocks.

Most work done in the area of intraprocedural optimizations takes the approach of only optimizing code inside basic blocks [6, 8]. One of the reasons is that analyzing bytecode across basic block boundaries is significantly more subtle than analyzing only code inside basic blocks. Secondly, in compiled code, expressions that span basic blocks are rare (although they do arise, e.g., from Java's ?-expressions). Some prominent bytecode optimizers, such as Soot [16], use an approach where class files are first converted to three-address intermediate representation, optimized using standard techniques and converted back to bytecode.

The analyses and optimizations in the paper address dead stores, load-pop pairs, duplicating loads and store-load pairs, which are typical optimization situations in stack-based code. They are designed to work on general code, i.e., they do not make any assumptions about its form. The code need not be the compiled version of a high-level program. Also, the analyses and optimizations are not in any way "intra-basic block", but work across basic block boundaries. We show that optimizations modifying pairs of instructions across basic block boundaries require bidirectional analyses, as information must be propagated both forward and backward during an analysis.

A typical optimization requiring bidirectional analysis is load-pop pair optimization, an optimization used in conjunction with dead store elimination to perform dead code elimination on bytecode. This analysis tries to find pop instructions with corresponding load/push instructions and eliminate them. The

optimization introduces a subtlety that is present in all bytecode transformations which remove pairs of stack height changing instructions across basic block boundaries.

This is illustrated in Figure 3.1 (where the ls nodes denote level sequences of instructions[1]). Looking at this example, it might seem that the load x instruction can be eliminated together with pop. Closer examination reveals that this is not the case: since load y is used by store z, the pop instruction cannot be removed, because then, after taking branch 2, the stack would not be balanced. This in turn means that load x cannot be removed. As can be seen from this example, a unidirectional analysis is not enough to come to such conclusion: information that a stack position is definitely needed flows backward from store z to load y along branch 3, but then the same information flows forward along path 2, and again backward along path 1.

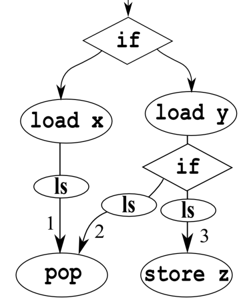

Figure 3.1: Example program

The type system we introduce for describing the optimization uses a global context of label types in the style of Stata and Abadi [15]. For example the typing rules for load are

$$\frac{\text{mnd} :: \Sigma_\ell = \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{load}\ x) \hookrightarrow (\ell, \mathsf{load}\ x)} \qquad \frac{\text{opt} :: \Sigma_\ell = \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{load}\ x) \hookrightarrow (\ell, \mathsf{nop})}$$

Here, Σ is a global context of label types, i.e. a mapping from labels to types (so Σ_ℓ is a label type). The typing judgement $\Sigma \vdash (\ell, instr) \hookrightarrow (\ell', instr')$ signifies that Σ is a valid global type context for the labelled instruction $(\ell, instr)$ (in this case, load x), and the instruction can be rewritten to $(\ell', instr')$ in this context. A label type for this particular analysis is a list of elements opt and mnd, the first denoting an element that is not needed on the stack (optional) and the second those that are (mandatory). For example the second typing rule for load states that the instruction can be replaced with a nop, if the value put on stack by load is opt (does not influence the semantics). Note that the typing rules do not have explicit subsumption rule. Normal subsumption rule is not applicable for bidirectional type systems, since changing the type in one program point potentially affects the type of all other program points.

Just like for the high-level language we showed in the previous chapter, optimization is easily stated to be sound relationally, by using a label-type indexed similarity relation on states. The similarity relation expresses that two states are related in a type, if they agree everywhere except for the optional stack positions in the first state, which must be omitted in the second. Then soundness states that running the original code and its optimized form from a related pair of prestates takes them to a related pair of poststates (including equi-termination).

Again the soundness of the analysis has a formal counterpart that can be expressed in terms of a programming logic. This has an application in proof transformation, where a proof can be transformed alongside a program, guided by the same typing information. Assume we have a program logic in the style of Bannwart and Müller [1] for reasoning about bytecode programs, with entailments $P \vdash (\ell, instr)$ for instructions and $P \vdash c$ for programs. Here, P is a map from labels to assertions, where assertions can contain the extralogical constant stk to refer to the current state of the stack and stk_n referring to the nth position down the stack. The interpretation of an entailment $P \vdash c$ is standard, i.e. if c starts in a state satisfying P and finishes, the final state satisfies P.

It is then easy to show that if the original program c is optimized to c_* according to the type Σ ($\Sigma \vdash c \hookrightarrow c_*$) then any proof for $P \vdash c$ can be transformed into a corresponding proof for $P|_\Sigma \vdash c_*$, where $(P|_\Sigma)_\ell =_{\text{df}} \exists w.\ w \sim_{\Sigma_\ell} stk \wedge P_\ell[w/stk]$ and $\sim_{\Sigma_\ell}$ is the similarity relation. (Note that $P|_\Sigma$ is the map from labels to modified assertions, so $(P|_\Sigma)_\ell$ is the modified assertion at label ℓ.) Informally, each $P|_{\Sigma_\ell}$ is obtained from P_ℓ by quantifying out stack positions which are opt, i.e., stack values which are absent in the optimized program. Of course this changes the height of the stack, so any stack position below the

[1] A sequence of instructions is a *level sequence*, if the net change of the stack height by these instructions is 0 and the instructions do not consume any values that were already present in the stack before executing these instructions.

removed one is shifted up. For example, if we have an assertion $P|_\ell =_{\mathrm{df}} 2 * stk_0 = x \wedge stk_1 = 6$ and type $\Sigma_\ell = \mathrm{opt} :: \mathrm{mnd} :: []$, the assertion $(P|_\Sigma)_\ell$ becomes $\exists v. 2 * v = x \wedge stk_0 = 6$. Again, the constructive content of the proof preservation argument is an algorithm for transforming the low level proofs.

Although we describe the type systems on a simple bytecode language, it is easy to extend them to cover the whole JVML. We have implemented the four analyses presented in the paper for full JVML (except for Jsr/Ret, which have been deprecated in Java's SDK since version 6).

Chapter 4

Partial redundancy elimination: a case study on optimizations modifying control flow

In [14] we investigate the scalability of our type-systematic approach for proof transformations. As the case study, we use partial redundancy elimination (PRE). It is a widely used compiler optimization that eliminates computations that are redundant on some but not necessarily all computation paths of a program. As a consequence, it performs both common subexpression elimination and expression motion. This optimization is notoriously tricky and has been extensively studied since it was invented by Morel and Renvoise [9]. There is no single canonical definition of the optimization. Instead, there is a plethora of subtly different ones and they do not all achieve exactly the same. The most modern and clear versions by Paleri et al. [10] and Xue and Knoop [17] are based on four unidirectional dataflow analyses.

As a case study, the optimization is interesting in several aspects. As already mentioned it is a highly nontrivial optimization. It is also interesting in the sense that it modifies program structure by inserting new nodes into the edges of the control flow graph. This makes automatic proof transformation potentially more difficult. Due to its complexity, it is not at all obvious that the transformation is always optimizing, thus it makes sense to prove that the algorithm is optimal in the non-strict sense, i.e. it does not make the program runtime worse.

An example application of partial redundancy elimination is shown in Figure 4.1 where the graph in Figure 4.1(a) is an original program and the graph in Figure 4.1(b) is the program after partial redundancy elimination optimization. Computations of $y + z$ in nodes 2 and 3 are partially redundant in the original program and can be eliminated. The result of computation of $y+z$ at node 5 can be saved into an auxiliary variable t (thus a new node is added in front of node 5). Additional computations of $y+z$ are inserted into the edge leading from node 1 to node 2 and the edge entering node 3. This is the optimal arrangement of computations, since there is one less evaluation of $y+z$ in the loop and on the path leading from node 3 to 2. Furthermore, the number of evaluations of the expression has not increased on any path through the program.

We introduce two optimization algorithms and corresponding type systems: simple PRE and full PRE. Simple PRE is a simplified version of PRE that is more conservative than full PRE. It does not eliminate all partial redundancies, but is more easily presentable, relying on two dataflow analyses. For full PRE, we look at the formulation by Paleri et al. [10], which consists of four interdependable dataflow analyses. Both of the versions perform edge splitting to place moved expression evaluations. In the type system this corresponds to new assignments appearing at subsumption inferences.

In the paper, we also show that it is possible to prove more than just correctness of the optimization using the relational method. One can also show that the optimization is actually an improvement in the sense that the number of evaluations of an expression on any given program path cannot increase. This means that no new computations can be introduced which are not used later on in the program. This is not

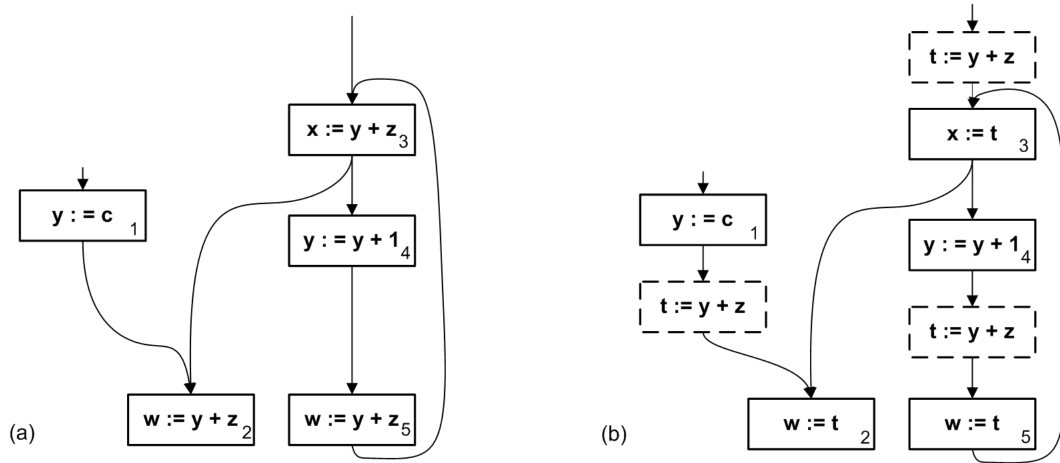

Figure 4.1: Example application of PRE

obvious, since code motion might introduce unneeded evaluations.

To show this property, there must be a way to count the expression uses. This is done via a simple instrumented semantics, which counts the number of evaluations of every expression. In the instrumented semantics a state is a pair (σ, r) of a standard state (an assignment of integer values to variables) and a "resource" state, associating to every nontrivial arithmetic expression a natural number for the number of times it has been evaluated. The rules of the semantics are as those for the standard semantics, except that for assignments of nontrivial expressions, an evaluation count is recorded in the resource state r.

The similarity relation then states that the count of evaluations for the optimized program can not be larger than the count in the original program, *except* when it is explicitly allowed in the type. The interesting point here is that types serve us as an "amortization" mechanism. The intuitive meaning of an expression being in the type of a program point is that there will be a use of this expression somewhere in the future, where this expression will be replaced with a variable already holding its value. Thus it is possible that a computation path of an optimized program has one more evaluation of the expression before this point than the corresponding computation path of the original program due to an introduced pre-computation. This does not break the improvement argument, since this increase contains a promise that this evaluation will be taken advantage of ("amortized") in the future.

We can also achieve an automatic proof transformation corresponding to the improvement property. This allows us to invoke a performance bound of a given program to obtain one for its optimized version.

Similarly to semantic improvement, where we needed an instrumented semantics, we need an instrumented Hoare logic. We extend the signature of the standard Hoare logic with an extralogical constant $count(a)$ for any nontrivial expression a. The inference rules of the instrumented Hoare logic are analogous to those for the standard Hoare logic except that the axiom for nontrivial assignment also reflects the evaluation count increase. The instrumented Hoare logic is sound and relatively complete wrt. the instrumented semantics.

In our semantic improvement statement we claimed that the optimized program performs not worse than possibly by one extra evaluation for precomputed expressions than the original one. The corresponding automatic proof transformation is only as "smart", and can not claim anything stronger (like stating that the optimized program will have a stricter bound on resource consumption than the original one). But this was not the goal, rather we are content with the observation that constructive and structured semantic arguments can be given formal counterparts in the form of automatic proof transformations.

Chapter 5

Proof transformation for wp proofs

In [2] we explore certificate translation for optimizations applied in an intermediate three-address code representation. Typically, as stated in compiler textbooks, compilation proceeds gradually by successive transformations: the original imperative program is gradually translated into a series of lower level representations, bringing the program closer to the actual machine code representation. For each step, we build an appropriate certificate translator. Since it is in these intermediate representations in which compiler optimizations are performed, we choose an RTL language to be the target of program optimizations. Although it is a simple low level language, it showed to be enough to represent common compiler textbook optimizations.

With respect to the verification environment, we adopt a weakest precondition based VCgen, which is part of standard PCC infrastructures. The lack of structure in low level languages has naturally made popular the use of weakest precondition or strongest postcondition methods. Program verification consists then in extracting a set of verification conditions, expressed as first order logic formulae, from a partially annotated program, and discharging this set of proof obligations. Therefore, in addition to a verification environment, we need to describe a certificate infrastructure to represent and check the proofs of the verification conditions computed by the VCgen, and which are the subject to transformations along program optimizations. Figure 5.1 shows the main components of a typical verification setting.

data/.style=rectangle,rounded corners,draw=black!80,top color=black!20,bottom color=black!60, very thick, inner sep=1pt,minimum size=8mm op/.style=rectangle,draw=black!20, color=black!80,very thick, inner sep=1pt,minimum size=8mm arrow2/.style=¿=stealth',thick

Instead of committing to a particular representation of certificates to study the existence of certificate translators, we propose an abstract notion of proof algebra. Then, the existence of certificate translators for common program optimizations is shown under the assumption that the algebra is closed under a set of operations that represent the application of logic rules, such as introduction and elimination for the $\wedge$ and $\vee$ connectives and for the $\forall$ quantifier, as well as substitutions of equals for equals.

In this setting, certificate translation consists in finding a relation between the proof obligations extracted from the original and optimized programs, and in the basis of this analysis a suitable transformation of the original certificate is proposed. Given a compiler represented by the function $\llbracket \cdot \rrbracket$, a function $\llbracket \cdot \rrbracket_{\text{spec}}$ to transform specifications, and certificate checkers (expressed as a ternary relation "c is a certificate that P adheres to ϕ" and written $c : P \models \phi$), a certificate translator is a function $\llbracket \cdot \rrbracket_{\text{cert}}$ such that for all programs p, policies ϕ, and certificates c,

$$c : p \models \phi \quad \Longrightarrow \llbracket c \rrbracket_{\text{cert}} : \llbracket p \rrbracket \models \llbracket \phi \rrbracket_{\text{spec}}$$

We provide a classification of program optimizations in terms of the transformations that must be made to the specification and the certificates. For instance, constant propagation and common subexpression elimination do not preserve proof obligations, and may render the program unprovable unless intermediate invariants are transformed as well.

Other optimizations eliminate instructions without computational role, e.g. dead variable elimination, and thus may eliminate information needed to prove intermediate assertions.

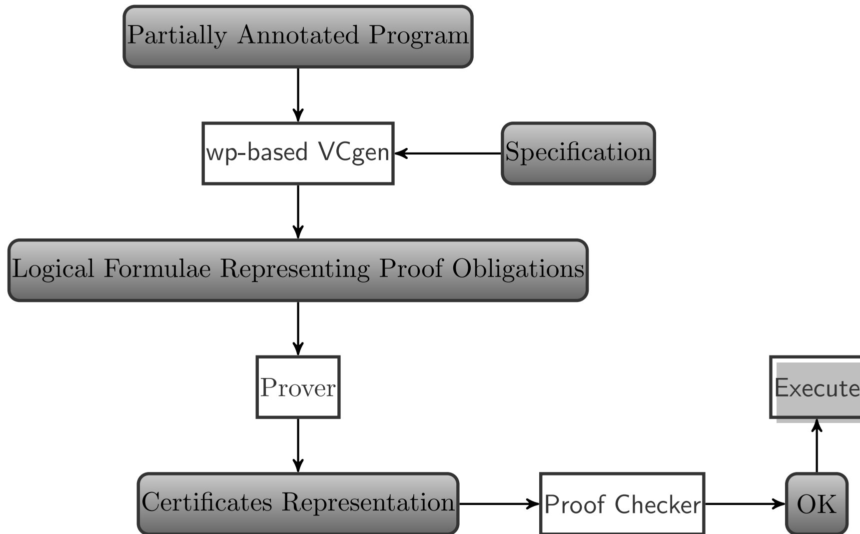

Figure 5.1: Overall description of the verification infrastructure.

Proof transformation is studied for three common optimizations: constant propagation, loop induction variable strength reduction and dead register elimination.

Constant propagation is a simple case of optimizations that perform arithmetic simplifications. In this case, the conditions that justify an optimization opportunity must be propagated through every intermediate assertion. Therefore, a certifying analyzer must be defined in order to certify the analysis result upon which the optimizations rely. Consider the following piece of code to which constant propagation is applied:

$r_1 := 1$	$r_1 := 1$
$\{\mathsf{true}\}$	$\{\mathsf{true}\}$
$r_2 := r_1$	$r_2 := 1$
$\{r_1 = r_2\}$	$\{r_1 = r_2\}$

Proof obligations for the assignment instruction to r_2 are $\mathsf{true} \Rightarrow r_1 = r_1$ and $\mathsf{true} \Rightarrow r_1 = 1$ for the original and optimized version respectively. The second proof obligation is unprovable, since this proof obligation is unrelated to the sequence of code containing the assignment $r_1 := 1$. To solve this difficulty, the intermediate invariant true must be strengthened with the condition $r_1 = 1$. Incorporating the information computed by the analysis to the original certificate requires a certificate validating the representation of this information in the verification logic. A certificate translation procedure takes as argument, in addition of the original certificate, the certificate produced by the certifying analyzer. Then, an inductively defined gluing function merges the the original certificate with the certificate produced by the certifying analyzer to generate a certificate for the optimized program. The inductive definition of the gluing function boils down to verifying that transformed instructions are point-to-point equivalent to the original ones, provided the assertions automatically computed by analysis are valid.

In a second instance, we describe a certificate translation procedure for loop induction variable strength reduction. The optimization consists in reducing the strength of operations that update an induction variable along the execution of the loop body. Translating the certificate is a similar process to constant propagation in the sense that it requires invariant strengthening by means of a certifying analyzer. However, this transformation shows a novelty due to the presence of loops. Indeed, the certifying analyzer must rely on the fact that the loop detection analysis is correct, since it requires that the loop can only be entered through its header.

As shown in the article, dead register elimination constitutes a special case for certificate transformation since the presence of variables in intermediate annotations introduces a difficulty when considering the removal of assignments to dead variables. Considering the following example, after performing constant

propagation

$$
\begin{array}{c}
x := n; \\
\vdots \\
\{x = n\} \\
y := x; \\
\vdots \\
\{x = y\}
\end{array}
\quad \longrightarrow \quad
\begin{array}{c}
x := n; \\
\vdots \\
\{x = n\} \\
y := n; \\
\vdots \\
\{x = y\}
\end{array}
$$

the variable x becomes dead. However, one cannot simply remove the first assignment since proof obligations referring to dead registers ($x = n$ in this case) cannot be proved because all hypotheses about these registers would be lost. In order to define a certificate translator to perform dead variable elimination, we propose to use an ad-hoc technique that performs simultaneously dead variable elimination in instructions and assertions.

We extend the definition of liveness to consider a register as live in assertion when it occurs in a reachable assertion. An ad-hoc technique is then proposed to solve this difficulty. The solution is based on the idea of ghost variables, commonly used to assist specification and verification and on the introduction of ghost assignments, i.e. instructions with no computational role, but that affect the computation of the verification conditions.

Chapter 6

An abstract interpretation model of proof transforming compilers

In [3] we study certificate translation in an abstract interpretation framework. We provide sufficient conditions for transforming a certificate of a program G into a certificate of a program G', as long as G' is derived from G by a semantically justified program transformation.

The results presented in the previous chapters postulate the existence of proof transformation procedures in particular programming and verification settings. In this chapter, we follow a more foundational approach, with the use of an abstract and unifying framework, to assess the scalability and generality of certificate translation. In addition to formulate a general analysis and verification frameworks, we consider an abstract and, hence, extensive representation of programs and a wider set of program transformations. Previous results can then be recovered by instantiating and discharging the proof obligations stated in our abstract setting.

First, we use the setting of abstract interpretation as a common model for the two main components of a certificate translation infrastructure: the verification environment in which the original certificate is produced and the static analysis that justifies a program optimization. The advantage of modeling the analysis and the verification infrastructure as abstract interpretations under domains of different level of abstraction allows one to study more precisely their interaction in certificate translation.

A common means to verify program properties is to consider (pre or post) fixpoints of the transfer functions of an abstract interpretation. On the other hand, to provide evidence to a code consumer that a program follows a specified policy, the code producer binds the program with a (partial) precomputed solution. The code client just needs to check that a labeling satisfies a set of inequalities to gain confidence about the program correctness. Abstraction Carrying Code is and instance of PCC where programs come equipped with a solution in an abstract interpretation domain that can be used to specify the consumer policy. However, in some abstract domains, checking the constraints required for the solution of the analysis is too costly or even undecidable. That is the case, for instance, with the domain of polyhedra or the domain of logical formulae in which our PCC is based. To model this scenario, we provide a mild extension of the abstract interpretation model to incorporate a notion of certificates. We do not commit to a particular representation of certificates; instead, we define an abstract notion of proof algebra, that includes a set of functions, closed in the domain of certificates, that are used to define a certificate translator for each program transformation.

One can see that the VCgen framework defined in previous chapter is a particular instance of this extended abstract interpretation framework. Indeed, checking that a total annotation (commonly computed from a partial annotation) is a fixpoint entails discharging a set of constraints, verification conditions, represented as logical implications in the lattice of logical formulae.

In this setting, we consider the analysis that justifies a program transformation as a lower level of abstraction than the domain of the verification environment. Therefore, we assume the existence of a monotone concretization function that maps elements from the domain of the analysis to its representation

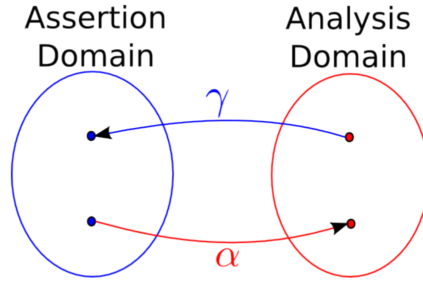

Figure 6.1: Logical interpretation of analysis results through a concretization function γ

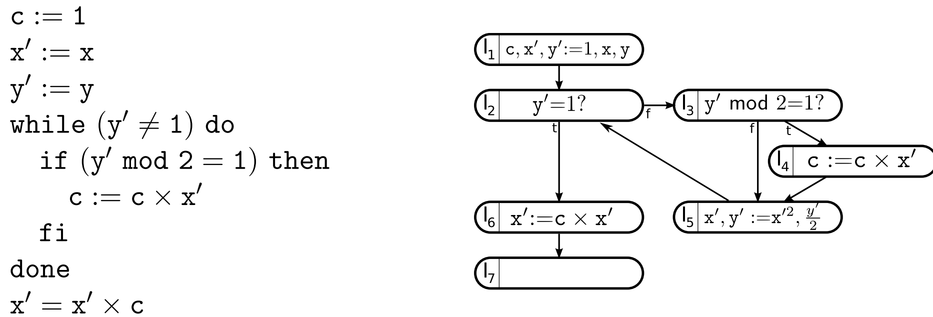

Figure 6.2: Program representation as graphs

as a logical assertion. An overall picture of the relation between the domain of logical assertions and the domain of the analysis is shown in Figure 6.1.

As explained in previous chapter, a certifying analyzer (i.e. an extension of a standard analysis procedure that outputs in addition to the analysis result a proof of its validity in the underlying certificate infrastructure) is a main component in the definition of a certificate translator. A well known concept relating two abstract interpretations, is that of the consistency of their corresponding transfer functions. Given a certificate infrastructure associated to an abstract interpretation, we found convenient to certify the consistency between two abstract interpretations. Indeed, proving the existence (and defining) a certifying analyzer boils down to discharging the verification conditions required to prove consistency of the analysis with respect to the verification environment.

To cover a wide range of programming languages and paradigms, we adopt an abstract representation of programs, including labeled structured languages, sequences of low level bytecode and the parallel composition of sequential components. A program representation consists of a flow graph, i.e. of a set of nodes, each one representing a distinguished execution state, and a set of directed edges representing the states to which execution may flow. Figure 6.2 illustrates the graph representation of the control flow of a fast exponentiation algorithm.

Typically nodes are associated to one (or a sequence of) syntactic program points, as it is needed in typical program verification environments to associate invariants to intermediate program points. Program semantics are abstractly described for illustrative purposes, since if has no role in certificate translation other than stating the presumed soundness of the verification setting.

These requirements over the transformations are expressed as proof obligations on the abstract certificate infrastructure. This set of requirements can then be opportunely instantiated to a particular analysis and verification domain (and a particular transformation) to yield the proof obligations to be discharged in order to complete the definition of the certificate translator.

Instead of considering specific program optimizations, we study certificate translation in the presence of basic transformations that, combined, can represent a wide range of standard program optimizations. This approach is relevant not only because it enable us to analyze arbitrary program optimizations extending, thus, the range of certificate translation, but because it decomposes certificate translation in less complex

tasks, and hence simpler proof obligations.

In previous work, in the presence of program optimizations, we have often followed the approach of strengthening annotations with the result of the analysis in order to translate original certificates. This is the case clearly for many common program optimizations, that rely on a safety analysis to justify semantic preserving local transformations. In the abstract lattice domain, strengthening is represented via the corresponding *meet* operator. This technique cannot be always applied, for instance with optimizations that rely on a liveness analysis to justify a semantic preserving transformation, as is the case for dead variable elimination. In this situation, an ad-hoc solution was proposed in previous chapter to define a certificate translator for dead variable elimination. The approach consists on the introduction of ghost variables and ghost assignments, a facility without computational role that assists specification and verification. We also provided a mild generalization of the technique based on invariant strengthening. It consists on merging the result of the analysis with the original specification by means of an abstract composition operator. The approach is general enough to model certificate translation for dead variable elimination, which entails (existentially) quantifying out dead variables from the intermediate assertions.

Chapter 7

Conclusions

In this deliverable, we have demonstrated that proof transformation for program optimizations is viable. It can be accomplished both for structured high-level languages and unstructured low-level languages and the central issue is how to record and exploit information from the underlying analyses in the best way. In particular, it is possible to handle optimizations based on bidirectional analyses and optimizations modifying program structure.

Proof transformation is possible both for general Hoare logic proofs and for wp proofs. In the latter case, care needs to be taken to propagate information from optimization points to assertion-annotated control-flow junctions, unless one is willing to accept insertion of extra assertions at these points.

Further work concerns the following issues:

- further work on frameworks of describing analyses and proving their soundness and improvement properties (especially combining our ideas with those from the Rhodium project [7])
- further work on the theory of bidirectional analyses (esp with analyses with non-trivial edge flows)
- further exploration of the duality between optimization soundness and proof transformation, applications of its consequences,
- analyses and optimizations specific to object-orientation, possibly multi-threading, their soundness and proof transformation,
- implementation of proof transformation for selected optimizations for Java to JVML compilation for integration of the work described here into the main line of Task 4.4 (non-optimizing Java to JVML compilation).

The results of this deliverable will serve as an input for the final deliverable of Task 4.4, Deliverable 4.6.

Bibliography

[1] F. Y. Bannwart and P. Müller. A program logic for bytecode. *Electronic Notes in Theoretical Computer Science*, 141:255–273, 2005.

[2] G. Barthe, B. Grégoire, C. Kunz, and T. Rezk. Certificate translation for optimizing compilers. In K. Yi, editor, *Static Analysis Symposium*, number 4134 in Lecture Notes in Computer Science, pages 301–317. Springer-Verlag, 2006.

[3] G. Barthe and C. Kunz. Certificate translation in abstract interpretation. In S. Drossopoulou, editor, *European Symposium on Programming*, volume 4960 of *Lecture Notes in Computer Science*, pages 368–382. Springer-Verlag, 2008.

[4] G. Barthe, T. Rezk, and A. Saabas. Proof obligations preserving compilation. In T. Dimitrakos, F. Martinelli, P. Ryan, and S. Schneider, editors, *Workshop on Formal Aspects in Security and Trust*, number 3866 in Lecture Notes in Computer Science, pages 112–126. Springer-Verlag, 2005.

[5] C. Hankin, F. Nielson, and H. R. Nielson. *Principles of Program Analysis*. Springer-Verlag, 2005. Second Ed.

[6] P. J. Koopman. A preliminary exploration of optimized stack code generation. *Forth Journal*, 6(3):241–251, 1994.

[7] S. Lerner, T. Millstein, E. Rice, and C. Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *Principles of Programming Languages*, pages 364–377, New York, NY, USA, 2005. ACM.

[8] M. Maierhofer and M. A. Ertl. Local stack allocation. In *International Conference on Compiler Construction*, number 1383 in Lecture Notes in Computer Science, pages 189–203. Springer-Verlag, 1998.

[9] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, 1979.

[10] V. K. Paleri, Y. N. Srikant, and P. Shankar. Partial redundancy elimination: a simple, pragmatic, and provably correct algorithm. *Science of Computer Programming*, 48(1):1–20, 2003.

[11] A. Saabas and T. Uustalu. A compositional natural semantics and Hoare logic for low-level languages. *Theoretical Computer Science*, 373(3):273–302, 2007.

[12] A. Saabas and T. Uustalu. Type systems for optimizing stack-based code. In M. Huisman and F. Spoto, editors, *Bytecode Semantics, Verification, Analysis and Transformation*, volume 190(1) of *Electronic Notes in Theoretical Computer Science*, pages 103–119. Elsevier, 2007.

[13] A. Saabas and T. Uustalu. Program and proof optimizations with type systems. *Journal of Logic and Algebraic Programming*, 77(1–2):131–154, 2008.

[14] A. Saabas and T. Uustalu. Proof optimization for partial redundancy elimination. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, pages 91–101. ACM Press, 2008.

[15] R. Stata and M. Abadi. A type system for Java bytecode subroutines. *ACM Transactions on Programming Languages and Systems*, 21(1):90–137, January 1999.

[16] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot – a java bytecode optimization framework. In *Proc. of Conf. on the Centre of Advanced Studies for Collaborative Research*, pages 125–135, 1999.

[17] J. Xue and J. Knoop. A fresh look at pre as a maximum flow problem. In A. Mycroft and A. Zeller, editors, *International Conference on Compiler Construction*, number 3923 in Lecture Notes in Computer Science, pages 139–154. Springer-Verlag, 2006.