Type Systems for Optimizing Stack-based Code

Ando Saabas and Tarmo Uustalu[1]

Institute of Cybernetics, Tallinn University of Technology,
Akadeemia tee 21, EE-12618 Tallinn, Estonia

Abstract

We give a uniform type-systematic account of a number of optimizations and the underlying analyses for a bytecode-like stack-based low-level language, including analysis soundness proofs and strongest analysis (principal type inference) algorithms. Specifically, we treat dead store instructions, load-pop pairs, duplicating load instructions, store-load pairs. The load-pop pairs and store-load pairs elimination optimizations are built on top of bidirectional analyses, facilitating correct elimination of instruction pairs spanning across basic block boundaries. As a result, no assumptions are needed about input code (it need not be the compiled form of a high-level source program, the stack need not be empty at basic block boundaries and not even need it be checked for safety before the analysis). The soundness proofs and strongest analysis algorithms are simple and uniform.

Keywords: stack-based low-level languages, data-flow analyses, optimizations, type systems, certification, bidirectional analyses, optimization soundness

1 Introduction

Popular Java compilers such as Sun's `javac` or Eclipse's Java Compiler are very conservative with optimizations, since most optimizations are presumed to be performed in the virtual machine by the just-in-time compiler. On the other hand, ahead-of-time optimizations are still important, especially in the context of mobile devices, where just-in-time compilers are not as powerful as on desktops or servers [4] and the size of the distributed binaries should be as small as possible.

Optimizing bytecode directly offers challenges not present in high or intermediate-level program optimizations. The following reasons can be outlined:

- Expressions and statements are not explicit. In a naive approach, a reconstruction of expression trees from instructions would be required for many optimizations.
- 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

[1] Emails: {ando|tarmo}@cs.ioc.ee

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

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.

- 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 [7,9,19,13]. There are probably two main reasons for this. First, analyzing bytecode across basic block boundaries is significantly more subtle than analyzing only code inside basic blocks. Second, 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 [17], use an approach where class files are first converted to three-address intermediate representation, optimized using standard techniques and converted back to bytecode.

In this paper, we give uniform formal declarative descriptions, soundness proofs and (in the full version) algorithms for a number of optimizations and their underlying analyses for a simple, bytecode-like stack-based low-level (unstructured) language. The tool we use for this purpose are type systems with a transformation component. This combines several lines of work, reviewed in the related work section below.

The analyses and optimizations in this 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". On the contrary, they work across basic block boundaries and do not require that the stack is empty at these. 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. Pleasantly, it turns out that this level of generality does not bring about any significant overhead for code of specifically well-behaved forms or compiled code.

The type-systematic approach has several benefits. A prominent feature of type-systematic descriptions of analyses and optimizations is that they provide a separation between a declarative description of what qualifies as a valid analysis (the rules of the system) and how the strongest analysis can be computed (principal type inference). They also lend themselves well for stating and showing soundness of optimizations, based on a relational method. Moreover, analysis type derivations (type annotations) can be used as optimization certificates in a proof-carrying code [12] like scenario, which is an advantage of a formal approach over an informal mathematical one.

The paper is organized as follows. In the remainder of this section, we fix the object language of this paper. Section 2 is devoted to dead code elimination, organized in two stages, dead stores elimination followed by load-pop pairs elimination.

$$\frac{(\ell, \mathsf{store}\ x) \in c \quad n \in \mathbb{Z}}{c \vdash (\ell, n :: zs, \sigma) \twoheadrightarrow (\ell + 1, zs, \sigma[x \mapsto n])}\ \text{store} \qquad \frac{(\ell, \mathsf{load}\ x) \in c}{c \vdash (\ell, zs, \sigma) \twoheadrightarrow (\ell + 1, \sigma(x) :: zs, \sigma)}\ \text{load}$$

$$\frac{(\ell, \mathsf{push}\ \ n) \in c}{c \vdash (\ell, zs, \sigma) \twoheadrightarrow (\ell + 1, n :: zs, \sigma)}\ \text{push} \qquad \frac{(\ell, \mathsf{add}) \in c \quad n_0, n_1 \in \mathbb{Z}}{c \vdash (\ell, n_0 :: n_1 :: zs, \sigma) \twoheadrightarrow (\ell + 1, n_0 + n_1 :: zs, \sigma)}\ \text{add}$$

$$\frac{(\ell, \mathsf{pop}) \in c}{c \vdash (\ell, z :: zs, \sigma) \twoheadrightarrow (\ell + 1, zs, \sigma)}\ \text{pop} \qquad \frac{(\ell, \mathsf{dup}) \in c}{c \vdash (\ell, z :: zs, \sigma) \twoheadrightarrow (\ell + 1, z :: z :: zs, \sigma)}\ \text{dup}$$

$$\frac{(\ell, \mathsf{goto}\ m) \in c}{c \vdash (\ell, zs, \sigma) \twoheadrightarrow (m, zs, \sigma)}\ \text{goto}$$

$$\frac{(\ell, \mathsf{gotoF}\ m) \in c}{c \vdash (\ell, \mathrm{tt} :: zs, \sigma) \twoheadrightarrow (\ell + 1, zs, \sigma)}\ \text{gotoF}^{\text{tt}} \qquad \frac{(\ell, \mathsf{gotoF}\ m) \in c}{c \vdash (\ell, \mathrm{ff} :: zs, \sigma) \twoheadrightarrow (m, zs, \sigma)}\ \text{gotoF}^{\text{ff}}$$

Fig. 1. Small-step semantics rules

In this section we also introduce the general type-systematic approach to data-flow analyses and optimizations, specialized to unstructured low-level languages, as well as how soundness is stated and proved in this approach. We also demonstrate that load-pop pairs elimination on general code must be bidirectional and show that such analyses are in fact a fairly gentle generalization of the more customary unidirectional analyses. In Section 3 we replay the same project for store-load combinations. Section 4 comments on the related work while Section 5 summarizes our conclusions. For space reasons, we only give the strongest analyses algorithms in the full version of the paper.

1.1 The object language

The object of our study is a simple operand-stack based low-level language.

The building blocks of the syntax of the language are labels $\ell \in \mathbf{Label} =_{\mathrm{df}} \mathbb{N}$ (natural numbers) and instructions $instr \in \mathbf{Instr}$. We assume having a countable set of program variables (registers) $x \in \mathbf{Var}$. The instructions of the language are defined by the grammar

$$instr ::= \mathsf{load}\ x \mid \mathsf{store}\ x \mid \mathsf{push}\ n \mid \mathsf{add} \mid \ldots \mid \mathsf{pop} \mid \mathsf{dup} \mid \mathsf{goto}\ \ell \mid \mathsf{gotoF}\ \ell$$

A piece of code $c \in \mathbf{Code}$ is a finite set of labelled instructions, i.e., a set of pairs of a label and an instruction where no label may label two different instructions: $\mathbf{Code} =_{\mathrm{df}} \{c \in \mathcal{P}_{\mathrm{fin}}(\mathbf{Label} \times \mathbf{Instr}) \mid \forall \ell, instr, instr'.(\ell, instr) \in c \wedge (\ell, instr') \in c \supset instr = instr'\}$. In other words, a piece of code is a partial function from labels to instructions.

The *domain* of a piece of code is the support of the partial function: $\mathrm{dom}(c) =_{\mathrm{df}} \{\ell \in \mathbf{Label} \mid \exists instr.(\ell, instr) \in c\}$. If $\ell \in \mathrm{dom}(c)$, we write c_ℓ for the unique instruction that ℓ labels in c.

The small-step (reduction) semantics of the language is given in terms of states, which are triples of a pc value (a label), a stack state and a store: $\mathbf{State} =_{\mathrm{df}} \mathbf{Label} \times \mathbf{Stack} \times \mathbf{Store}$. A stack state is a list of integers and booleans: $zs \in \mathbf{Stack} =_{\mathrm{df}} (\mathbb{Z} + \mathbb{B})^*$. A store is a mapping of variables to integers: $\sigma \in \mathbf{Store} =_{\mathrm{df}} \mathbf{Var} \to \mathbb{Z}$. The standard small-step operational semantics of pieces of code is given via an indexed single-step reduction relation $\twoheadrightarrow \in \mathbf{Code} \to \mathcal{P}(\mathbf{State} \times \mathbf{State})$ defined by the rules in Figure 1. The associated multi-step reduction relation $\twoheadrightarrow^*$ is defined as its reflexive-transitive closure.

2 Dead code elimination

Standard dead code elimination optimization removes from a program statements that do not affect the values of variables that are live at the end of the program. On high-level programs or intermediate (expressions-based low-level) code, the optimization is trivial to perform after live variables analysis and typically involves removing assignments to variables which are known to be dead immediately after the assignment. In stack-based code, where expressions are not explicit (nor are related instructions necessarily next to each other), removing dead code is not so straightforward. For example the program $x := z + y$ could be compiled into

```
0, load z
1, load y
2, add
3, store x
4,
```

If the analysis shows that x is dead, then in the intermediate code, the assignment to x can be deleted. In the stack-based code however, not only the store instruction on line 3, but also lines 0-2 should be deleted.

Another issue which sets stack-based code optimizations apart from optimizations in the intermediate language, is that statements and expression can span several basic blocks (or, put in another way, basic blocks are not necessarily entered into or exited from with an empty stack). A simple example of such code is the following stack-based low-level equivalent of if b then $x := z$ else $y := z$ where z is loaded only once, and in both branches, only the store instruction is applied:

```
0, load z
1, load b
2, gotoF 5
3, store x
4, goto 6
5, store y
6,
```

If live variable analysis reveals that the variable x is dead, the store instruction at line 3 cannot simply be removed, since if the true branch were taken, the unassigned value of the variable x would be left on the stack after exiting the branch. Also the load instruction on line 0 cannot be deleted, because, while it is used by a dead store, it is also used by a live store in the false branch. In such cases, without moving instructions, the best thing to do is replacing instruction 3 with a pop.

We approach dead code elimination in two stages. In the first stage (we call it dead stores elimination), all dead store, add and conditional jump instructions are replaced with pop instructions (so that the optimization does not affect the stack height at any label) based on an analogue of the standard live variables analysis. In the second stage, pop instructions with corresponding preceding load/push instruction(s) are eliminated, if possible, and care is taken that stack heights remain consistent after this transformation.

Both of these analyses are completely general, do not make any assumptions on the form of bytecode, and work across basic block boundaries.

2.1 Dead stores elimination

The live variables analysis of our stack-based language is similar to live variables analysis for languages with expressions, except that the stack is also accounted for.

This means that in addition to variables being possibly live or certainly dead, stack positions can also be either possibly live or certainly dead. For example, if we know that immediately after an execution of a $\mathsf{store}\ x$ instruction, the variable x is dead, then we also know that before the execution the top of the stack is dead, meaning that whatever value the stack top holds, it does not affect the value of a live variable.

We describe both the analysis and the optimization in terms of a type system. The type language and subtyping formalize the underlying poset of the analysis whereas typing defines valid analyses for a given piece of code (independently of fixing a specific algorithm for finding the strongest analysis satisfying some given condition).

For live variable analysis, a code type $\Sigma \in \mathbf{CodeType}$ is an assignment of a label type to every label: $\mathbf{CodeType} =_{\mathrm{df}} (\mathbf{Label} \to \mathbf{LabelType})$. A label type $\tau \in \mathbf{LabelType}$ is either a pair of a stack and store type or a special type $*$ for "any state": $\mathbf{LabelType} =_{\mathrm{df}} (\mathbf{StackType} \times \mathbf{StoreType}) + \{*\}$. Stack types $es \in \mathbf{StackType}$ and store types $d \in \mathbf{StoreType}$ are lists resp. assignments to variables of location types "possibly live" and "certainly dead": $\mathbf{StackType} =_{\mathrm{df}} \mathbf{LocType}^*$, $\mathbf{StoreType} =_{\mathrm{df}} \mathbf{Var} \to \mathbf{LocType}$, $\mathbf{LocType} =_{\mathrm{df}} \{L, D\}$. We will use the shorthand Σ_ℓ for $\Sigma(\ell)$.

The subtyping and typing rules are given in Figure 2. In all type systems of this paper, the subtyping judgement $\Sigma_\ell \leq \Sigma'_\ell$ denotes that the first label type is a subtype of the second (i.e., stronger). Similarly, the subtyping judgement $\Sigma \leq \Sigma'$ denotes that the first code type is a subtype of the other. The typing judgement $\Sigma \vdash (\ell, \mathit{instr})$ signifies that the labelled instruction (ℓ, instr) admits type Σ, i.e., that Σ is a valid analysis of this particular labelled instruction (independent of any possible code context in which it may occur). The typing judgement $\Sigma \vdash c$ means that the code c types with Σ, i.e., that Σ is a valid analysis of c as a whole. (Ignore the bits $\hookrightarrow (\ell, \mathit{instr}')$ and $\hookrightarrow c'$ for a moment; they pertain to the optimization justified by a conducted analysis.)

The typing rules of our particular analysis only allow a variable or stack position to be marked "dead" at a label ℓ in a valid code type, if there cannot be a path from ℓ to a label ℓ' such that the instruction at ℓ' contains a useful use of that position and the variable or stack position is not redefined on the path. Otherwise it must be marked "live". A typical useful use of the stack top is storing it in a variable marked "live" at the successor label. The stack top and next-to-top positions are usefully used by an addition, provided the stack top is marked "live" at the successor label. The stack top used by a pop is certainly dead, since its value is lost and does not affect the values of any location. For $\mathsf{load}\ x$, the type of x depends on its type and the type of the stack top at the successor label: if x was live at the successor label already, it stays live, otherwise if the top of the stack was live (thus needed), x also becomes live.

The analysis (type inference) algorithm finds the weakest valid code type that is stronger than a given one, using the given one as the initial value of an iterated backward propagation (this corresponds to weakest pretype calculation for a high-level language). Typically, the given code type sets the stack type to be empty and the store to be "all live" at the exit labels (successor labels outside the domain). Elsewhere, the label type has the default value "any".

$$\frac{}{L \le D} \qquad \frac{}{e \le e} \qquad \frac{}{[\,] \le [\,]} \qquad \frac{e \le e' \quad es \le es'}{e :: es \le e' :: es'} \qquad \frac{\forall x. d(x) \le d'(x)}{d \le d'} \qquad \frac{es \le es' \quad d \le d'}{es, d \le es', d'} \qquad \frac{}{\tau \le \star}$$

$$\frac{\forall \ell \in \mathbf{Label}. \Sigma_\ell \le \Sigma'_\ell}{\Sigma \le \Sigma'}$$

$$\frac{\Sigma_\ell \le (L : es, d[x \mapsto D]) \quad (es, d) = \Sigma_{\ell+1} \quad d(x) = L}{\Sigma \vdash (\ell, \mathsf{store}\ x) \hookrightarrow (\ell, \mathsf{store}\ x)}\ \mathrm{store}_1 \qquad \frac{\Sigma_\ell \le (D : es, d) \quad (es, d) = \Sigma_{\ell+1} \quad d(x) = D}{\Sigma \vdash (\ell, \mathsf{store}\ x) \hookrightarrow (\ell, \mathsf{pop})}\ \mathrm{store}_2$$

$$\frac{\Sigma_\ell \le (es, d[x \mapsto d(x) \wedge e]) \quad (e : es, d) = \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{load}\ x) \hookrightarrow (\ell, \mathsf{load}\ x)}\ \mathrm{load} \qquad \frac{\Sigma_\ell \le (es, d) \quad (e : es, d) = \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{push}\ n) \hookrightarrow (\ell, \mathsf{push}\ n)}\ \mathrm{push}$$

$$\frac{\Sigma_\ell \le (L :: L :: es, d) \quad (L :: es, d) = \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{add}) \hookrightarrow (\ell, \mathsf{add})}\ \mathrm{add}_1 \qquad \frac{\Sigma_\ell \le (D :: D :: es, d) \quad (D :: es, d) = \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{add}) \hookrightarrow (\ell, \mathsf{pop})}\ \mathrm{add}_2$$

$$\frac{\Sigma_\ell \le (D : es, d) \quad (es, d) = \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{pop}) \hookrightarrow (\ell, \mathsf{pop})}\ \mathrm{pop} \qquad \frac{\Sigma_\ell \le ((e_0 \wedge e_1) :: es, d) \quad (e_0 :: e_1 :: es, d) = \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{dup}) \hookrightarrow (\ell, \mathsf{dup})}\ \mathrm{dup}$$

$$\frac{\Sigma_\ell \le \Sigma_m}{\Sigma \vdash (\ell, \mathsf{goto}\ m) \hookrightarrow (\ell, \mathsf{goto}\ m)}\ \mathrm{goto}$$

$$\frac{m \ne \ell + 1 \quad \Sigma_\ell \le (L :: es, d) \quad (es, d) = \Sigma_{\ell+1} \wedge \Sigma_m}{\Sigma \vdash (\ell, \mathsf{gotoF}\ m) \hookrightarrow (\ell, \mathsf{gotoF}\ m)}\ \mathrm{gotoF}_1 \qquad \frac{\Sigma_\ell \le (D :: es, d) \quad (es, d) = \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{gotoF}\ \ell + 1) \hookrightarrow (\ell, \mathsf{pop})}\ \mathrm{gotoF}_2$$

$$\frac{* = \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathit{instr}) \hookrightarrow (\ell, \mathit{instr})}\ \mathrm{nonjump} \qquad \frac{* = \Sigma_{\ell+1} \wedge \Sigma_m}{\Sigma \vdash (\ell, \mathsf{gotoF}\ m) \hookrightarrow (\ell, \mathsf{gotoF}\ m)}\ \mathrm{gotoF}$$

$$\frac{\forall \ell \in \mathrm{dom}(c).\, \Sigma \vdash (\ell, c_\ell) \hookrightarrow (\ell, c'_\ell)}{\Sigma \vdash c \hookrightarrow c'}\ \mathrm{code}$$

Fig. 2. Type system for live variables analysis and dead stores elimination

The type system also has a transformation component for transforming a given piece into an optimized variant, guided by a valid code type. The judgement $\Sigma \vdash c \hookrightarrow c'$ denotes that, based on a valid analysis Σ, c can be optimized to c'.

The instructions that can be optimized (those that decrease the stack height by 1) have two rules. A $\mathsf{store}\ x$ instruction can be optimized, if x is marked "dead" in the posttype (i.e., at the successor label of the instruction). An add instruction can be optimized, if the top of the stack is "dead" in the posttype. A gotoF instruction can be optimized, if the jump target is the next label (as then there is no real need to use the boolean condition on the top of the stack).

Note that while we have not spelled them out, optimizations for load and push instructions could also be added to the type system. Namely, if in the posttype of a load instruction the top of the stack is "dead", it is obvious that the value is not used on any forward path, thus the concrete value put on the stack does not matter (only the correct stack height does). Thus, the instruction could be replaced with the cheapest instruction that puts some value of the correct type on the stack, e.g., $\mathsf{push}\ 0$.

To illustrate that the analysis does not need related instructions to be next to each other, we present the program $y := x; x := w + 1$ as a piece of code where the assignment $y := x$ has been moved to the middle (instructions 3-4). For this example, we assume the variable y to be live and variable x to be dead at the end of the program. The analysis gives the following result.

Σ_ℓ		ℓ, c_ℓ	ℓ, c'_ℓ
$[]$	$[y \mapsto D, x \mapsto L]$	0, push 1	0, push 1
$[D]$	$[y \mapsto D, x \mapsto L]$	1, load w	1, load w
$[D, D]$	$[y \mapsto D, x \mapsto L]$	2, add	2, pop
$[D]$	$[y \mapsto D, x \mapsto L]$	3, load x	3, load x
$[L, D]$	$[y \mapsto D, x \mapsto D]$	4, store y	4, store y
$[D]$	$[y \mapsto L, x \mapsto D]$	5, store x	5, pop
$[]$	$[y \mapsto L, x \mapsto D]$	6,	6,

The optimization replaces the store x and add instructions with pop instructions (since in both cases variable x resp. the stack top is dead in the type of the successor). This leaves the stack balanced. Our next analysis (Subsection 2.2) will show that the pop instructions on lines 2 and 5 can be removed together with the instructions on lines 0 and 1, since stack usage will remain consistent after those transformations too.

The optimization is easily stated to be sound using a label-type indexed similarity relation on states, defined as follows:

$$\frac{z \in \mathbb{Z} \quad z_* \in \mathbb{Z}}{z \approx z_*} \qquad \frac{z \in \mathbb{B} \quad z_* \in \mathbb{B}}{z \approx z_*} \qquad \frac{}{[] \approx []} \qquad \frac{z \approx z_* \quad zs \approx zs_*}{z :: zs \approx z_* :: zs_*}$$

$$\frac{}{[] \sim_{[]} []} \qquad \frac{zs \sim_{es} zs_*}{z :: zs \sim_{L::es} z :: zs_*} \qquad \frac{z \approx z_* \quad zs \sim_{es} zs_*}{z :: zs \sim_{D::es} z_* :: zs_*} \qquad \frac{\forall x \in \mathbf{Var}. d(x) = L \supset \sigma(x) = \sigma_*(x)}{\sigma \sim_d \sigma_*}$$

$$\frac{zs \sim_{es} zs_* \quad \sigma \sim_d \sigma_*}{(\ell, zs, \sigma) \sim_{(es,d)} (\ell, zs_*, \sigma_*)} \qquad \frac{zs \approx zs_*}{(\ell, zs, \sigma) \sim_* (\ell, zs_*, \sigma_*)}$$

We can see that two states are related by a proper label type (a stack and store type), if they agree up to locations marked "live". They are related by $*$, if the stack heights and the value types of the stack elements agree. Reducing an original piece of code and its optimized form from a related pair of states must maintain this relation. We obtain the following soundness theorem.

Theorem 2.1 (Soundness of dead stores elimination) *If $\Sigma \vdash c \hookrightarrow c'$, then:*

(i) *If $(\ell, zs, \sigma) \sim_{\Sigma_\ell} (\ell_*, zs_*, \sigma_*)$, then*

— *for any (ℓ', zs', σ') such that $c \vdash (\ell, zs, \sigma) \twoheadrightarrow (\ell', zs', \sigma')$ there exist $(\ell'_*, zs'_*, \sigma'_*)$ such that $(\ell', zs', \sigma') \sim_{\Sigma_{\ell'}} (\ell'_*, zs'_*, \sigma'_*)$ and $c' \vdash (\ell_*, zs_*, \sigma_*) \twoheadrightarrow (\ell'_*, zs'_*, \sigma'_*)$,*

— *for any $(\ell'_*, zs'_*, \sigma'_*)$ such that $c' \vdash (\ell_*, zs_*, \sigma_*) \twoheadrightarrow (\ell'_*, zs'_*, \sigma'_*)$ there exist (ℓ', zs', σ') such that $(\ell', zs', \sigma') \sim_{\Sigma_{\ell'}} (\ell'_*, zs'_*, \sigma'_*)$ and $c \vdash (\ell, zs, \sigma) \twoheadrightarrow (\ell', zs', \sigma')$.*

(ii) *If $(\ell, zs, \sigma) \sim_{\Sigma_\ell} (\ell_*, zs_*, \sigma_*)$, then*

— *for any (ℓ', zs', σ') such that $c \vdash (\ell, zs, \sigma) \twoheadrightarrow^* (\ell', zs', \sigma') \not\twoheadrightarrow$ there exist $(\ell'_*, zs'_*, \sigma'_*)$ such that $(\ell', zs', \sigma') \sim_{\Sigma_{\ell'}} (\ell'_*, zs'_*, \sigma'_*)$ and $c' \vdash (\ell_*, zs_*, \sigma_*) \twoheadrightarrow^* (\ell'_*, zs'_*, \sigma'_*) \not\twoheadrightarrow$.*

— *for any $(\ell'_*, zs'_*, \sigma'_*)$ such that $c' \vdash (\ell_*, zs_*, \sigma_*) \twoheadrightarrow^* (\ell'_*, zs'_*, \sigma'_*) \not\twoheadrightarrow$ there exist (ℓ', zs', σ') such that $(\ell', zs', \sigma') \sim_{\Sigma_{\ell'}} (\ell'_*, zs'_*, \sigma'_*)$ and $c \vdash (\ell, zs, \sigma) \twoheadrightarrow^* (\ell', zs', \sigma') \not\twoheadrightarrow$.*

Proof. Part (i) is easily checked by inspecting the typing/transformation rules for labelled instructions. Part (ii) follows from part (i) by induction on the length of the reduction sequence. □

Note that part (i) states "preservation" and we have not stated "progress". The appropriate progress property is not guaranteed for code typed in just the type system of Figure 2 alone, as typability does not ensure that stacks of fixed height

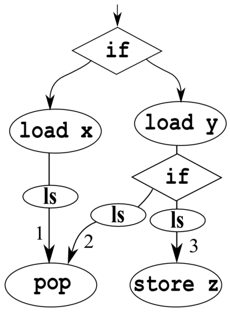

Fig. 3. Example program

contain operands of right types and that stacks in "any state" do not underflow. Progress is only provable for code which is safe. To keep the type system simple, we chose to left out the safety component, but it can be integrated. Alternatively one can always check the safety of a piece of code before type inference in our system.

2.2 Load-pop pairs elimination

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 (where the ls nodes denote level sequences of instructions[2]). 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. Thus a bidirectional analysis is needed, which at each node propagates information both forward and backward. We also see that we are not really dealing with pairs, but webs of instructions in general.

In the appropriate type system, a code type $\Sigma \in \mathbf{CodeType}$ is again an assignment of a label type to every label: $\mathbf{CodeType} =_{\mathrm{df}} \mathbf{Label} \to \mathbf{LabelType}$. Here, label types $\tau \in \mathbf{LabelType}$ are stack types or the special type $*$ for "any state". Stack types $es \in \mathbf{StackType}$ are lists of location types "mandatory" and "optional": $\mathbf{StackType} =_{\mathrm{df}} \mathbf{LocType}^*$ and $\mathbf{LocType} =_{\mathrm{df}} \{\mathrm{mnd}, \mathrm{opt}\}$.

The typing and subtyping rules are given in Figure 4. The typing rules state that, if at some label a stack element is marked "mandatory", then at all other labels of its lifetime, this particular element is also considered "mandatory". Thus the typing rules explain which optimizations are acceptable. The rule for store instructions states that the instruction always requires a "mandatory" element on

[2] 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.

$$\frac{}{\mathrm{mnd} \leq \mathrm{opt}} \qquad \frac{}{e \leq e} \qquad \frac{}{[\,] \leq [\,]} \qquad \frac{e \leq e' \quad es \leq es'}{e :: es \leq e' :: es'} \qquad \frac{}{\tau \leq *} \qquad \frac{\forall \ell \in \mathbf{Label}. \Sigma_\ell \leq \Sigma'_\ell}{\Sigma \leq \Sigma'}$$

$$\frac{\Sigma_\ell = \mathrm{mnd} :: \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{store}\ x) \hookrightarrow (\ell, \mathsf{store}\ x)}\ \mathrm{store}$$

$$\frac{\mathrm{mnd} :: \Sigma_\ell = \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{load}\ x) \hookrightarrow (\ell, \mathsf{load}\ x)}\ \mathrm{load}_1 \qquad \frac{\mathrm{opt} :: \Sigma_\ell = \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{load}\ x) \hookrightarrow (\ell, \mathsf{nop})}\ \mathrm{load}_2$$

$$\frac{\mathrm{mnd} :: \Sigma_\ell = \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{push}\ n) \hookrightarrow (\ell, \mathsf{push}\ (n)}\ \mathrm{push}_1 \qquad \frac{\mathrm{opt} :: \Sigma_\ell = \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{push}\ n) \hookrightarrow (\ell, \mathsf{nop})}\ \mathrm{push}_2$$

$$\frac{\Sigma_l = \mathrm{mnd} :: \mathrm{mnd} :: es \quad \mathrm{mnd} :: es = \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{add}) \hookrightarrow (\ell, \mathsf{add})}\ \mathrm{add}_1 \qquad \frac{\Sigma_l = \mathrm{opt} :: \mathrm{opt} :: es \quad \mathrm{opt} :: es = \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{add}) \hookrightarrow (\ell, \mathsf{nop})}\ \mathrm{add}_2$$

$$\frac{\Sigma_\ell = \mathrm{mnd} :: \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{pop}) \hookrightarrow (\ell, \mathsf{pop})}\ \mathrm{pop}_1 \qquad \frac{\Sigma_\ell = \mathrm{opt} :: \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{pop}) \hookrightarrow (\ell, \mathsf{nop})}\ \mathrm{pop}_2$$

$$\frac{\Sigma_\ell = \mathrm{mnd} :: es \quad \mathrm{mnd} :: \mathrm{mnd} :: es = \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{dup}) \hookrightarrow (\ell, \mathsf{dup})}\ \mathrm{dup}_1 \qquad \frac{\Sigma_\ell = e_1 :: es \quad \mathrm{opt} :: e_1 :: es = \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{dup}) \hookrightarrow (\ell, \mathsf{nop})}\ \mathrm{dup}_2$$

$$\frac{\Sigma_\ell = \Sigma_m}{\Sigma \vdash (\ell, \mathsf{goto}\ m) \hookrightarrow (\ell, \mathsf{goto}\ m)}\ \mathrm{goto}$$

$$\frac{\Sigma_\ell = \mathrm{mnd} :: es \quad es = \Sigma_m \quad es = \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{gotoF}\ m) \hookrightarrow (\ell, \mathsf{gotoF}\ m)}\ \mathrm{gotoF}$$

$$\frac{\Sigma_\ell = * \quad * = \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{load}\ x) \hookrightarrow (\ell, \mathsf{nop})}\ \mathrm{load}_2 \qquad \frac{\Sigma_\ell = * \quad * = \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{push}\ n) \hookrightarrow (\ell, \mathsf{nop})}\ \mathrm{push}_2$$

$$\frac{\forall \ell \in \mathrm{dom}(c).\, \Sigma \vdash (\ell, c_\ell) \hookrightarrow (\ell, c'_\ell)}{\Sigma \vdash c \hookrightarrow c'}\ \mathrm{code}$$

Fig. 4. Type system for load-pop pairs elimination

the stack, thus its predecessors must definitely leave a value on top of the stack. Instructions that put elements on the stack “do not care”: if an element is required, they can push a value (a mnd element on the stack in the posttype), otherwise the instruction could be omitted (an opt element on the stack in the posttype). The same holds for pop: if an element is definitely left on the stack, a pop instruction is not removed, otherwise it can be removed.

The analysis (type derivation) algorithm, as mentioned above, is bidirectional. While bidirectional analyses may seem much more involved that unidirectional analyses, it has been shown that bidirectional problems are inherently no more complex than unidirectional ones [6].

The intuition behind the algorithm is the following. The types definitely required at some labels should be given (typically the types at the exit labels of the code are set to be the empty stack, possibly also the type at the entry label). All other types are initialized to the default type “any state”. The algorithm then computes the weakest valid type of the code that is stronger than the given type. At each label, information is gathered from all its successors and predecessors. The constraints initiate from the given types and the store and conditional jump instructions, which require that a value is present on the stack for them (i.e. a mnd element has to be on top of the stack type). Other instructions that produce or consume a value from the stack can initially be assumed to produce or consume “useless” values (denoted by opt in the type system). Type information arriving from different directions to a program point can be intersected according to subtyping relations given in Figure 4. This guarantees that, if an instruction definitely needs a value on the stack, this information is propagated to all of its predecessors. Similarly, if an instruction definitely must produce a value on the stack (since some subsequent instruction may need it), this information is propagated to its successor.

Looking at the example in Figure 3, where we initialize the posttype of the control-flow graph to be the empty stack, the pretype of store z requires a mnd element on the top of the stack. This means that the posttype of load y has to have mnd on the top of the stack. This information propagates to the pop instruction, and from there to load x instruction. Thus the analysis shows that no instruction can be deleted. If, on the other hand, store z were not present, the postlabels of the two load instructions and prelabel of the pop instruction would keep their initial opt stack top types, and the three instructions could be deleted.

A piece of code corresponding to Figure 3 is given in the following example in the left column (where the level sequences of instructions have been omitted to simplify presentation). It gets a type showing that no optimization is possible. In the right column, we consider a minimally different piece of code where store z instruction has been replaced with pop. Here the analysis shows that both the pop and corresponding load instructions can be removed.

Σ_ℓ	ℓ, c_ℓ	ℓ, c'_ℓ	Σ_ℓ	ℓ, c_ℓ	ℓ, c'_ℓ
$[]$	$0,$ load b	$0,$ load b	$[]$	$0,$ load b	$0,$ load b
$[\mathrm{mnd}]$	$1,$ gotoF 9	$1,$ gotoF 9	$[\mathrm{mnd}]$	$1,$ gotoF 9	$1,$ gotoF 9
$[]$	$2,$ load y	$2,$ load y	$[]$	$2,$ load y	$2,$ nop
$[\mathrm{mnd}]$	$3,$ load b'	$3,$ load b'	$[\mathrm{opt}]$	$3,$ load b'	$3,$ load b'
$[\mathrm{mnd}, \mathrm{mnd}]$	$4,$ gotoF 7	$4,$ gotoF 7	$[\mathrm{mnd}, \mathrm{opt}]$	$4,$ gotoF 7	$4,$ gotoF 7
$[\mathrm{mnd}]$	$5, \underline{\mathsf{store}\ z}$	$5,$ store z	$[\mathrm{opt}]$	$5, \underline{\mathsf{pop}}$	$5,$ nop
$[]$	$6,$ goto 11	$6,$ goto 11	$[]$	$6,$ goto 11	$6,$ goto 11
$[\mathrm{mnd}]$	$7,$ pop	$7,$ pop	$[\mathrm{opt}]$	$7,$ pop	$7,$ nop
$[]$	$8,$ goto 11	$8,$ goto 11	$[]$	$8,$ goto 11	$8,$ goto 11
$[]$	$9,$ load x	$9,$ load x	$[]$	$9,$ load x	$9,$ nop
$[\mathrm{mnd}]$	$10,$ goto 7	$10,$ goto 7	$[\mathrm{opt}]$	$10,$ goto 7	$10,$ goto 7
$[]$	$11,$	$11,$	$[]$	$11,$	$11,$

The type-indexed similarity relation on states for establishing soundness of the optimization is defined as follows:

$$\frac{}{[] \sim_{[]} []} \qquad \frac{zs \sim_{es} zs_*}{z :: zs \sim_{\mathrm{mnd}::es} z :: zs_*} \qquad \frac{zs \sim_{es} zs_*}{z :: zs \sim_{\mathrm{opt}::es} zs_*} \qquad \frac{zs \sim_{es} zs_*}{(\ell, zs, \sigma) \sim_{es} (\ell, zs_*, \sigma)} \qquad \frac{}{(\ell, zs, \sigma) \sim_* (\ell, [], \sigma)}$$

The rules state that two states are related, if they agree everywhere except for the optional stack positions in the first state, which must be omitted in the second.

The soundness statement is the same as in Theorem 2.1 and the proof is analogous. The same will hold for the following two optimizations in the next section.

3 Store/load+ elimination

In this section, we deal with one of the more widely used bytecode optimizations—redundant store/load computations. This optimization is based on the observation that, if a store is followed by a reload of the same variable to the same stack position (and the variable is not redefined in the meantime), then, provided there are no future uses of that variable, both the store and the load instruction can be eliminated. Similarly, if there is a store followed by n loads, then the store and loads can be replaced by $n - 1$ dup instructions. Note that for these optimizations, the store and the loads do not necessarily have to be next to each other, there can

$$\frac{}{x \leq \mathrm{nac}} \qquad \frac{}{e \leq e} \qquad \frac{}{[\,] \leq [\,]} \qquad \frac{e \leq e' \quad es \leq es'}{e :: es \leq e' :: es'} \qquad \frac{}{\emptyset \leq \tau} \qquad \frac{\forall \ell \in \mathbf{Label}. \Sigma_\ell \leq \Sigma'_\ell}{\Sigma \leq \Sigma'}$$

$$\frac{\Sigma_\ell = e :: es \quad \mathrm{replace}(x, \mathrm{nac}, es) \leq \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{store}\ x) \hookrightarrow (\ell, \mathsf{store}\ x)}\ \mathrm{store}$$

$$\frac{x :: \Sigma_\ell \leq \Sigma_{\ell+1} \quad \forall es. \Sigma_l \neq x :: es}{\Sigma \vdash (\ell, \mathsf{load}\ x) \hookrightarrow (\ell, \mathsf{load}\ x)}\ \mathrm{load}_1 \qquad \frac{x :: \Sigma_\ell \leq \Sigma_{\ell+1} \quad \Sigma_l = x :: es}{\Sigma \vdash (\ell, \mathsf{load}\ x) \hookrightarrow (\ell, \mathsf{dup})}\ \mathrm{load}_2$$

$$\frac{\mathrm{nac} :: \Sigma_\ell \leq \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{push}\ n) \hookrightarrow (\ell, \mathsf{push}\ n)}\ \mathrm{push} \qquad \frac{\Sigma_\ell = e_0 :: e_1 :: es \quad \mathrm{nac} :: es \leq \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{add}) \hookrightarrow (\ell, \mathsf{add})}\ \mathrm{add}$$

$$\frac{\Sigma_\ell = e :: es \quad es \leq \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{pop}) \hookrightarrow (\ell, \mathsf{pop})}\ \mathrm{pop} \qquad \frac{\Sigma_\ell = e :: es \quad e :: e :: es \leq \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{dup}) \hookrightarrow (\ell, \mathsf{dup})}\ \mathrm{dup}$$

$$\frac{\Sigma_\ell \leq \Sigma_m}{\Sigma \vdash (\ell, \mathsf{goto}\ m) \hookrightarrow (\ell, \mathsf{goto}\ n)}\ \mathrm{goto} \qquad \frac{\Sigma_\ell = e :: es \quad es \leq \Sigma_{\ell+1} \quad es \leq \Sigma_m}{\Sigma \vdash (\ell, \mathsf{gotoF}\ m) \hookrightarrow (\ell, \mathsf{gotoF}\ n)}\ \mathrm{gotoF}$$

$$\frac{\Sigma_\ell = \emptyset}{\Sigma \vdash (\ell, \mathit{instr}) \hookrightarrow (\ell, \mathit{instr})}\ \mathrm{instr}$$

$$\frac{\forall \ell \in \mathrm{dom}(c).\, \Sigma \vdash (\ell, c_\ell) \hookrightarrow (\ell, c'_\ell)}{\Sigma \vdash c \hookrightarrow c'}\ \mathrm{code}$$

Fig. 5. Type system for duplicating loads elimination

be intervening instructions, as long as the stack height remains the same after the instructions and the values below the top are not consumed by them.

We approach this optimization in two stages. First, a simple forward copy propagation analysis determines whether some load instructions can be replaced with dup instructions. In the second stage, store/load pairs are detected and transformed.

3.1 Duplicating loads elimination

This analysis is a simple copy propagation analysis, which tries to determine if before a $\mathsf{load}\ x$ instruction, the value of x is already on top of the stack. If this is the case, the $\mathsf{load}\ x$ instruction can be replaced with a dup instruction. It is a unidirectional, forward analysis.

In the type system, label types are stack types or a special type "no state": $\mathbf{LabelType} =_{\mathrm{df}} \mathbf{StackType} + \{\emptyset\}$. Stack types $es \in \mathbf{StackType}$ are again lists of location types, which this time are elements of $x \in \mathbf{Var}$ (signifying that the location is certainly a copy of the variable x) and a special type "not a copy" (the location is possibly not a copy of any variable): $\mathbf{StackType} =_{\mathrm{df}} \mathbf{LocType}^*$, $\mathbf{LabelType} =_{\mathrm{df}} \mathbf{StackType} + \{\emptyset\}$ and $\mathbf{LocType} =_{\mathrm{df}} \mathbf{Var} + \{\mathrm{nac}\}$.

The subtyping and typing rules are given in Figure 5. The typing rules state that a stack position can be marked a variable at label ℓ, if on all paths to this label, this variable is put on the stack in this position, and later not modified. In other words, at label ℓ, the value in the corresponding position in the stack is necessarily equal to the value of the variable. If a stack type holds a nac element in some position it means that this position may not be a copy (e.g., since on some path to ℓ, a numeral is pushed to that position).

Thus the typing rule for load reflects that, after the instruction, the value on top of the stack and the value of the corresponding variable are necessarily equal. A $\mathsf{store}\ x$ explicitly kills all variables x in the stack, since the values in the stack and the new value of the variable cannot be guaranteed to be consistent anymore.

An optimization can be made, if a variable x is on top of the stack before a $\mathsf{load}\ x$ instruction. In this case, the load can be replaced with a dup, as in the following example:

Σ_ℓ	(ℓ, c_ℓ)	(ℓ, c'_ℓ)
$[]$	$0, \mathsf{load}\ x$	$0, \mathsf{load}\ x$
$[x]$	$1, \mathsf{push}\ 1$	$1, \mathsf{push}\ 1$
$[\mathrm{nac}, x]$	$2, \mathsf{store}\ y$	$2, \mathsf{store}\ y$
$[x]$	$3, \mathsf{load}\ x$	$3, \mathsf{dup}$
$[]$	$4,$	$4,$

This optimization could be improved by not only keeping track of copies of variables in the stack, but also in the variables. A location type would then be a set of variables (those variables of which the given location is certainly a copy; the empty set will signify that the location is possibly not a copy of anything). Then, even if there were consecutive loads from different variables, a dup could be introduced, provided that the two variables were actually copies of each other.

The label-type indexed similarity relation on states to establish soundness of the optimization is defined as follows:

$$\frac{}{[] \sim^\sigma_{[]} []} \qquad \frac{zs \sim^\sigma_{es} zs_*}{z :: zs \sim^\sigma_{\mathrm{nac}::es} z :: zs_*} \qquad \frac{zs \sim^\sigma_{es} zs_*}{\sigma(x) :: zs \sim^\sigma_{x::es} \sigma(x) :: zs_*} \qquad \frac{zs \sim^\sigma_{es} zs_*}{(\ell, zs, \sigma) \sim_{es} (\ell, zs_*, \sigma)}$$

(Note that no states are in the relation $\sim_\emptyset$.)

3.2 Store-load pairs elimination

The store-load pairs analysis tries to find store instructions followed by a load instruction to the same stack position and referring to the same variable (before any new store to the same variable takes place). Provided that this variable is not used later on, both instructions could be eliminated. Since the possible future use of a variable requires a live variable analysis, we take the approach to only remove the load instruction, but keep the store instruction and precede it with a dup, as in the following example:

$0, \mathsf{store}\ x$	$0, \mathsf{dup}; \mathsf{store}\ x$
$1, \mathsf{load}\ x$	$1, \mathsf{nop}$
$2, \ldots$	$2, \ldots$

The benefit of this approach is that a check for future uses of x can be omitted. If it turns out that the variable is not needed, a dead code elimination optimization would remove the dup and store instructions later on.

Since this optimization manipulates pairs of instructions, a bidirectional analysis is needed, as was the case with load-pop pairs.

Similarly to the previous analysis, label types are again stack types or the special type $\emptyset$ for “no state”: **LabelType** $=_{\mathrm{df}}$ **StackType** $+ \{\emptyset\}$. Stack types $es \in$ **StackType** are lists of location types, which are elements x of **Var** (a position to be inserted in the optimized code to keep a copy of variable x in the stack) and “mandatory” (original positions): **StackType** $=_{\mathrm{df}}$ **LocType***, **LocType** $=_{\mathrm{df}}$ **Var** $+ \{\mathrm{mnd}\}$.

The subtyping and typing rules are given in Figure 6. The typing rules say that, if a label has some stack type, then every time this label is reached in the execution of the code the size of the stack will be the number of mnd elements in the stack. In

$$\frac{}{[\,] \leq [\,]} \qquad \frac{es \leq es'}{e :: es \leq e :: es'} \qquad \frac{es \leq es'}{x :: es \leq es'} \qquad \frac{}{\emptyset \leq \tau} \qquad \frac{\forall \ell \in \mathbf{Label}. \Sigma_\ell \leq \Sigma'_\ell}{\Sigma \leq \Sigma'}$$

$$\frac{\Sigma_\ell = \mathrm{mnd} :: es \quad es = \Sigma_{\ell+1} \quad \neg\mathrm{member}(x, es)}{\Sigma \vdash (\ell, \mathsf{store}\ x) \hookrightarrow (\ell, \mathsf{store}\ x)}\ \mathrm{store}_1 \qquad \frac{\Sigma_\ell = \mathrm{mnd} :: es \quad x :: es = \Sigma_{\ell+1} \quad \neg\mathrm{member}(x, es)}{\Sigma \vdash (\ell, \mathsf{store}\ x) \hookrightarrow (\ell, \mathsf{dup}; \mathsf{store}\ x)}\ \mathrm{store}_2$$

$$\frac{\Sigma_\ell = es \quad \mathrm{mnd} :: es = \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{load}\ x) \hookrightarrow (\ell, \mathsf{load}\ x)}\ \mathrm{load}_1 \qquad \frac{\Sigma_\ell = x : es \quad \mathrm{mnd} :: es = \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{load}\ x) \hookrightarrow (\ell, \mathsf{nop})}\ \mathrm{load}_2$$

$$\frac{\mathrm{mnd} :: \Sigma_\ell = \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{push}\ n) \hookrightarrow (\ell, \mathsf{push}\ n)}\ \mathrm{push} \qquad \frac{\Sigma_\ell = \mathrm{mnd} :: \mathrm{mnd} :: es \quad \Sigma_{\ell+1} = \mathrm{mnd} :: es}{\Sigma \vdash (\ell, \mathsf{add}) \hookrightarrow (\ell, \mathsf{add})}\ \mathrm{add}$$

$$\frac{\Sigma_\ell = \mathrm{mnd} :: \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{pop}) \hookrightarrow (\ell, \mathsf{pop})}\ \mathrm{pop} \qquad \frac{\Sigma_\ell = \mathrm{mnd} :: es \quad \mathrm{mnd} :: \mathrm{mnd} :: es = \Sigma_{\ell+1}}{\Sigma \vdash (\ell, \mathsf{dup}) \hookrightarrow (\ell, \mathsf{dup})}\ \mathrm{dup}$$

$$\frac{\Sigma_\ell = \Sigma_m}{\Sigma \vdash (\ell, \mathsf{goto}\ m) \hookrightarrow (\ell, \mathsf{goto}\ m)}\ \mathrm{goto} \qquad \frac{\Sigma_\ell = \mathrm{mnd} :: es \quad es = \Sigma_{\ell+1} \quad es = \Sigma_m}{\Sigma \vdash (\ell, \mathsf{gotoF}\ m) \hookrightarrow (\ell, \mathsf{gotoF}\ m)}\ \mathrm{gotoF}$$

$$\frac{\Sigma_\ell = \emptyset \quad \emptyset = \Sigma_{\ell+1}}{\Sigma \vdash (\ell, instr) \hookrightarrow (\ell, instr)}\ \mathrm{nonjump} \qquad \frac{\Sigma_\ell = \emptyset \quad \emptyset = \Sigma_{\ell+1} \quad \emptyset = \Sigma_m}{\Sigma \vdash (\ell, \mathsf{gotoF}\ m) \hookrightarrow (\ell, \mathsf{gotoF}\ m)}\ \mathrm{gotoF}$$

$$\frac{\forall \ell \in \mathrm{dom}(c).\, \Sigma \vdash (\ell, c_\ell) \hookrightarrow (\ell, c'_\ell)}{\Sigma \vdash c \hookrightarrow c'}\ \mathrm{code}$$

Fig. 6. Type system for store-load pairs elimination

addition, if at some label the stack type contains an element of **Var**, it means that, if the code was optimized according to the typing rules, then at that label in the optimized code, the stack would hold an additional copy of that variable between the positions corresponding to the positions of the original code.

As an example, a piece of code could be typed in the following way:

Σ_ℓ	ℓ, c_ℓ	ℓ, c'_ℓ
$[\,]$	$0, \mathsf{push}\ 1$	$0, \mathsf{push}\ 1$
$[\mathrm{mnd}]$	$1, \mathsf{store}\ x$	$1, \mathsf{dup}; \mathsf{store}\ x$
$[x]$	$2, \mathsf{push}\ 1$	$2, \mathsf{push}\ 1$
$[\mathrm{mnd}, x]$	$3, \mathsf{store}\ y$	$3, \mathsf{store}\ y$
$[x]$	$4, \mathsf{load}\ x$	$4, \mathsf{nop}$
$[\mathrm{mnd}]$	$5, \mathsf{store}\ z$	$5, \mathsf{store}\ z$
$[z]$	$6,$	$6,$

(Note that this is the principal type that the analysis algorithm would derive when the type of label 6 is initialized to $\emptyset$. It is also acceptable to type label 6 with $[\,]$. That is also what the algorithm would deliver, if the type of label 6 were initialized to $[\,]$.)

The analysis algorithm works as follows. The lowest acceptable types of some labels should be given (typically the type of the entry point of the program, i.e., label 0, is set to $[\,]$ and optionally also the types of the exit labels). The rest of the labels are initialized to be of the default type $\emptyset$. The algorithm then tries to compute the actual and potential stack heights for each label. As mentioned before, the mnd elements in the type mean the actual elements; the variables in the type mean potential elements present after optimization. For store and load, "speculative" types can initially be given, since a store could potentially be replaced with a dup and a store (so that after the transformation an extra value would be on the stack) and load with nop (since the extra value already present removes the need of reloading it). The types of the successors and predecessors of a label are used to compute the local label type. The types arriving from different directions are combined by a non-deterministic union operation determined by the subtyping relation. (Both $[x]$ and $[y]$ are greater than $[x, y]$ and $[y, x]$, so there is no unique least upper bound for them, only minimal upper bounds.) As a result, any provisional additional position is dropped from the stack, if it turns out that on some incoming or outgoing path the potential added stack height is not really viable.

To simplify presentation, the transformation does not carry out any relabelling of instructions. Instead, two instructions (the dup and the store) share a label. In fact, appropriate relabelling would be unproblematic: any instruction label and jump target should be increased by the number of variable names in its type. To keep the rules simpler, we have refrained from doing this.

The label-type indexed similarity relation to establish soundness of the optimization is defined as follows:

$$\frac{}{[\,] \sim^{\sigma}_{[\,]} [\,]} \qquad \frac{zs \sim^{\sigma}_{es} zs_*}{z :: zs \sim^{\sigma}_{\mathrm{mnd} :: es} z :: zs_*} \qquad \frac{zs \sim^{\sigma}_{es} zs_*}{zs \sim^{\sigma}_{x :: es} \sigma(x) :: zs_*} \qquad \frac{zs \sim^{\sigma}_{es} zs_*}{(\ell, zs, \sigma) \sim_{es} (\ell, zs_*, \sigma)}$$

(Again no two states are in the relation $\sim_{\emptyset}$.) Now two states are related, if the second one has appropriate additional positions in its stack component, agreeing suitably with the store component (additional positions copy variables).

4 Related work

The related work falls into two categories: work on optimizations for stack-based low-level languages and work on type-systematic data-flow analysis and optimization, including the necessary ingredients from data-flow analysis theory.

Optimization of stack-based low-level code

As mentioned in the introduction, one of the more well-known bytecode transformation tools is Soot [17]. Soot's approach to bytecode optimization is to transform bytecode into 3-address intermediate code, use standard techniques to optimize the intermediate code, and then translate the code back into bytecode. The back and forth translation can introduce several inefficiencies into bytecode, such as redundant store/load computations. This is tackled by either transforming the intermediate code into an aggregated form, using some peephole optimization techniques and converting it to bytecode using standard tree traversal techniques, or by translating the intermediate code into a streamlined form of bytecode, and performing store-load optimizations on its basic blocks [18]. The benefit of Soot's approach is of course that optimizations on the intermediate language are routine to perform. The drawback is that multiple transformations between different representations make the optimizations performed non-transparent and the code can lose some properties that were present before. This can become an issue, when preservation of properties beyond the standard semantics (e.g., code size) is desired.

The Java bytecode analyzer Julia [14] provides a framework for implementing different static analyses on Java bytecode. Analyses implemented so far include escape analysis, rapid type analysis, information-flow analysis, static initialisation analysis and several others. To our knowledge the analyses outlined here have not been implemented in Julia.

The jDFA framework performs basic analysis of liveness and constant propagation on bytecode [10]. Liveness information is only computed for local registers (not for the stack). More complicated analyses, which would allow removal of dead code or constant folding, are not implemented. Further implementation of this framework seems to have stopped.

VanDrunen et al. [19] give a formalization and soundness proofs for specific pattern based eliminations of store-load pairs in basic blocks. Their work is partly motivated by that of Shpeisman and Tikir [13], who list specific instances for code replacement in Java bytecode, considering variations of dup instruction available there, but do not formalize these transformations.

Type-systematic data-flow analysis and data-flow analysis theory

Our analyses and optimizations are presented in the form of type systems, following the approach of Stata and Abadi [16], who described the Java bytecode verifier as a type system. A similar design has been used in program logics for stack-based languages and a subset of Java bytecode [1,2]. Bidirectional analyses for high-level languages have been described in great detail by Khedker and Dhamdhere [6]. Such analyses have been used in high-level optimizations such as the Morel-Renvoise algorithm for partial redundancy elimination [11].

The method of defining data-flow analyses for imperative high-level languages with type systems appears in Laud et al. [8]. The extension of this paradigm to optimizations, again for high-level programs, has been studied by Saabas and Uustalu [15], including optimization of programs together with their functional correctness proofs, based on type derivations. Benton's [3] work on relational soundness proofs for analyses and optimizations of high-level programs promotes similar ideas.

5 Conclusions

We have described four different data-flow analyses for optimizing stack-based code. Our main goal was to give general and formal descriptions for both the analyses and optimizations based on them. We have outlined the difficulties associated to bytecode optimizations that modify pairs of stack height changing instructions across basic block boundaries. The analyses have been presented in the form of type systems, the benefit being that a type derivation (type annotation) can serve as a certificate in the context of proof carrying code. We have also hinted at the algorithms for computing strongest analyses, which we could not include for space reasons.

We have implemented the analyses and optimizations presented here for Java bytecode (except for jsr/ret instructions).

As follow-up work, we plan to look at analyses for aligning store-load and load-load pairs via movement of instructions, to make the optimizations of Section 3 applicable to a wider variety of bytecode. We will also investigate optimizations which include movement of blocks of bytecode, such as loop-invariant code-motion.

Acknowledgement

We acknowledge the constructive remarks of our referees. This work was supported by the Estonian Science Foundation grants No. 5567 and 6940 and by the EU FP6 IST integrated project No. 15905 MOBIUS.

References

[1] Bannwart, F. and P. Müller, *A program logic for bytecode*, in "Proc. of 1st Wksh. on Bytecode Semantics, Verification, Analysis and Transformation, Bytecode 2005," Vol. 141(1) of Electr. Notes in Theor. Comput. Sci., pp. 255–273, Elsevier, 2005

[2] Benton, N., *A typed logic for stacks and jumps*, draft, 2004

[3] Benton, N., *Simple relational correctness proofs for static analyses and program transformations*, in "Proc. of 31st ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL 2004," pp. 14–25, ACM Press, 2004

[4] Debbabi, M., A. Mourad, C. Talhi and H. Yahyaoui, *Accelerating embedded Java for mobile devices*, IEEE Commun. **43**(9), pp. 80–85, 2005

[5] Khedker, U. P. and D. M. Dhamdhere, *Bidirectional data flow analysis: myths and reality*, ACM SIGPLAN Notices **34**(6), pp. 47–57, 1999

[6] Khedker, U. P. and D. M. Dhamdhere, *A generalized theory of bit vector data flow analysis*, ACM Trans. on Program. Lang. and Syst. **16**(5), pp. 1472–1511, 1994

[7] Koopman, P. J., *A preliminary exploration of optimized stack code generation*, J. of Forth Applications and Research **6**(3), pp. 241–251, 1994

[8] Laud, P., T. Uustalu and V. Vene, *Type systems equivalent to data-flow analyses for imperative languages*, Theor. Comput. Sci. **364**(3), pp. 292–310, 2006

[9] Maierhofer, M. and M. A. Ertl, *Local stack allocation*, in "Proc. of 7th Int. Conf. on Compiler Construction, CC '98," Vol. 1383 of Lect. Notes in Comput. Sci., pp. 189–203, Springer, 1998

[10] Mohnen, M., *An open framework for data-flow analysis in Java*, in "Proc. of 2nd Wksh. on Intermediate Representation Engineering for Virtual Machines, IRE 2002," 2002

[11] Morel, E. and C. Renvoise, *Global optimization by suppression of partial redundancies*, Commun. of ACM **22**(2), pp. 96–103, 1979

[12] Necula, G. C., *Proof-carrying code*, in "Proc. of 24th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL 1997," pp. 106–119, ACM Press, 1997

[13] Shpeisman, T. and M. Tikir, *Generating efficient stack code for Java.* Tech. Rep. CS-TR-4069, University of Maryland, 1999

[14] Spoto, F., *Julia: A generic static analyser for the Java bytecode*, in "Proc. of 7th Wksh. on Formal Techniques for Java-like Programs, FTfJP 2005," 2005

[15] Saabas, A. and T. Uustalu, *Program and proof optimizations with type systems*, manuscript, 2006

[16] Stata, R. and M. Abadi, *A type system for Java bytecode subroutines*, ACM Trans. on Program. Lang. and Syst. **21**(1), pp. 90–137, 1999

[17] Vallée-Rai, R., P. Co, E. Gagnon, L. Hendren, P. Lam and V. Sundaresan, *Soot - a Java bytecode optimization framework*, in "Proc. of 1999 Conf. on the Centre of Advanced Studies for Collaborative Research, CASCON 1999", pp. 125–135, 1999

[18] Vallée-Rai, R., E. Gagnon, L. Hendren, P. Lam, P. Pominville and V. Sundaresan, *Optimizing Java bytecode using the Soot framework: is it feasible?*, in "Proc. of 9th Int. Conf. on Compiler Construction, CC 2000", Vol. 1781 of Lect. Notes in Comput. Sci., pp. 18–34, Springer, 2000

[19] VanDrunen, T., A. L. Hosking and J. Palsberg, *Reducing loads and stores in stack architectures*, manuscript, 2000.