

# Completeness of a bytecode verifier and a certifying Java-to-JVM compiler

Robert F. Stärk ([staerk@inf.ethz.ch](mailto:staerk@inf.ethz.ch))  
*ETH Zürich*

Joachim Schmid ([joachim.schmid@tydo.de](mailto:joachim.schmid@tydo.de))  
*Siemens AG, Munich*

**Abstract.** During an attempt to prove that the Java-to-JVM compiler generates code that is accepted by the Bytecode Verifier we found examples of legal Java programs that are rejected by the verifier. We propose therefore to restrict the rules of definite assignment for the try-finally statement as well as for the labeled statement so that the example programs are no longer allowed. Then we can prove, using the framework of Abstract State Machines, that each program from the slightly restricted Java language is accepted by the Bytecode Verifier. In the proof we use a new notion of bytecode type assignment without subroutine call stacks.

**Keywords:** Java, bytecode verification, certifying compilation

## 1. Introduction

We do not see bytecode verification as an isolated problem. We see bytecode verification as a link in a chain that starts at the source code level. Source code programs are statically analyzed by the compiler to ensure that local variables are definitely assigned when they are accessed. After compilation, the bytecode is executed on a virtual machine (VM). Since the VM cannot distinguish between bytecode generated by a correct compiler and bytecode produced by a different source to exploit the VM, bytecode has to pass verification before it is executed on the VM. Moreover, the verifier annotates the bytecode with type information that is later used by the just-in-time compiler (JIT) and the garbage collector. From the security point of view, it is important that bytecode which is accepted by the verifier is type-safe at run-time and does not corrupt the memory. Therefore, one has to prove the following property.

PROPERTY 1 (Soundness of the verifier).

*Bytecode which is accepted by the verifier is type-safe at run-time.*

The property is usually shown for a *defensive* VM. A defensive VM is an interpreter of bytecode that performs additional checks prior to the execution of an instruction. For example, it checks that there are

enough operands of the right types on the operand stack, or that the registers are initialized when they are accessed. The defensive VM does not work with raw bytes but with typed values and can distinguish between values representing integers and values representing pointers. The term ‘type-safe at run-time’ in property 1 is then read as ‘does not violate the checks of the defensive VM’. As a consequence of property 1, the time consuming checks of the defensive VM can be omitted for verifiable bytecode, since they are always true.

From the practical point of view, property 1 is not enough. Property 1 is, for example, satisfied for the trivial (and useless) verifier that rejects every bytecode program. Hence, in addition to property 1 one has to prove that the class of bytecode programs which are accepted by the verifier is large enough. It should at least contain all bytecode programs that are generated by a correct compiler from valid source code programs. Hence, one has to prove the following property which is not a property of the compiler alone, but a property of the source code language (rules of definite assignment), the compiler *and* the bytecode verifier.

**PROPERTY 2.** *The Java-to-JVM compiler generates verifiable bytecode.*

Several proofs have been published for property 1 for different bytecode verifiers and different fragments of the VM. Property 2 has been neglected, maybe since everybody thought that it is true for Sun’s Java-to-JVM compiler. A surprising result of [16], however, is that property 2 is not true for a compiler that follows the guidelines of the Java Virtual Machine specification [11].

When the following legal Java programs are compiled using a standard compiler like Sun’s JDK 1.3 or 1.4 compiler, the generated bytecode is rejected by most bytecode verifiers including JDK 1.3, JDK 1.4, Netscape 4.73–4.76, Microsoft VM for Java 5.0 and 5.5 and the Kimera Verifier [15].

```

class Test1 {
    void test(boolean b) {
        int i;
        try {i = 1;}
        finally {if (b) i = 2;}
        int j = i;
    }
}

class Test2 {
    void test(boolean b) {
        int i;
        L: { try {i = 1; if (b) break L;}
            finally {if (b) i = 2;}
            i = 3;
        }
        int j = i;
    }
}

```

What is the problem?—In both examples the compiler infers on the source code level that the local variable *i* has a value when it is assigned

to  $j$  at the end of the body of the method. On the bytecode level, however, the verifier believes that there is a path to the corresponding `Load( $i$ )` instruction on which no value is stored into  $i$ . Hence, the verifier rejects the bytecode, because  $i$  could be uninitialized when it is used at the end of the method. Let us look in more detail at the bytecode for the first example `Test1`:

		$b$	$i$	$e$	$r$
$A$ : <code>Const(1)</code>	<code>[]</code>	<code>int</code>	<code>Undef</code>	<code>Undef</code>	<code>Undef</code>
<code>Store(<math>i</math>)</code>	<code>[int]</code>	<code>int</code>	<code>Undef</code>	<code>Undef</code>	<code>Undef</code>
<code>Jsr(<math>S</math>)</code>	<code>[]</code>	<code>int</code>	<code>int</code>	<code>Undef</code>	<code>Undef</code>
<code>Goto(<math>C</math>)</code>	<code>[]</code>	<code>int</code>	<code>Undef</code>	<code>Undef</code>	<code>Undef</code>
$H$ : <code>Store(<math>e</math>)</code>	<code>[Throwable]</code>	<code>int</code>	<code>Undef</code>	<code>Undef</code>	<code>Undef</code>
<code>Jsr(<math>S</math>)</code>	<code>[]</code>	<code>int</code>	<code>Undef</code>	<code>Throwable</code>	<code>Undef</code>
<code>Load(<math>e</math>)</code>	<code>[]</code>	<code>int</code>	<code>Undef</code>	<code>Throwable</code>	<code>Undef</code>
<code>Throw</code>	<code>[Throwable]</code>	<code>int</code>	<code>Undef</code>	<code>Throwable</code>	<code>Undef</code>
$S$ : <code>Store(<math>r</math>)</code>	<code>[retAddr(<math>S</math>)]</code>	<code>int</code>	<code>Undef</code>	<code>Undef</code>	<code>Undef</code>
<code>Load(<math>b</math>)</code>	<code>[]</code>	<code>int</code>	<code>Undef</code>	<code>Undef</code>	<code>retAddr(<math>S</math>)</code>
<code>Ifne(<math>B</math>)</code>	<code>[int]</code>	<code>int</code>	<code>Undef</code>	<code>Undef</code>	<code>retAddr(<math>S</math>)</code>
<code>Goto(<math>R</math>)</code>	<code>[]</code>	<code>int</code>	<code>Undef</code>	<code>Undef</code>	<code>retAddr(<math>S</math>)</code>
$B$ : <code>Const(2)</code>	<code>[]</code>	<code>int</code>	<code>Undef</code>	<code>Undef</code>	<code>retAddr(<math>S</math>)</code>
<code>Store(<math>i</math>)</code>	<code>[int]</code>	<code>int</code>	<code>Undef</code>	<code>Undef</code>	<code>retAddr(<math>S</math>)</code>
$R$ : <code>Ret(<math>r</math>)</code>	<code>[]</code>	<code>int</code>	<code>Undef</code>	<code>Undef</code>	<code>retAddr(<math>S</math>)</code>
$C$ : <code>Load(<math>i</math>)</code>	<code>[]</code>	<code>int</code>	<code>Undef</code>	<code>Undef</code>	<code>Undef</code>
<code>Store(<math>j</math>)</code>	<code>[]</code>				

The subroutine  $S$  that corresponds to the finally block is called from two different places, once at the end of the code for the try block, where the variable  $i$  has type `int`, and once in the default handler, where  $i$  has type `Undef`. Therefore, at the beginning of the subroutine  $S$ , the verifier has to assume that  $i$  is of type `Undef`. Since there is a path from  $S$  to  $R$  on which there is no store instruction for  $i$ , the type of  $i$  is still `Undef` at  $R$ .

Since there is a store instruction for  $i$  between  $S$  and  $R$ , the variable  $i$  is considered to be modified by the subroutine  $S$ . Hence, the type `int` of  $i$  before the first `Jsr( $S$ )` instruction cannot be propagated to the `Goto( $C$ )` instruction and the verifier propagates the type `Undef` from  $R$  instead. Finally, at the `Load( $i$ )` instruction at  $C$ , the verifier finds that  $i$  is of type `Undef` and rejects the program.

Hence the problem is that the *rules for definite assignment* by which the compiler determines when a variable may be treated as initialized in the Java source code program do not exactly correspond to the static analysis of the bytecode verifier.

An obvious solution of the problem is to restrict the rules of definite assignment for the try-finally statement and for the labeled statement so that the problematic programs are no longer legal and therefore already rejected by the compiler. In case of the try-finally statement the

change is that, if a variable  $x$  is definitely assigned after the try block but not after the finally block and if there is an assignment to  $x$  in the finally block, then the variable  $x$  is no longer considered to be definitely assigned after the try-finally statement. Notice that an assignment to  $x$  in the finally block makes  $x$  a modified variable of the corresponding subroutine and that is what we have to avoid in the case where  $x$  is not definitely assigned after the finally block.

Since changing the rules of definite assignment could create situations which are not very intuitive for programmers, another solution is to compile the problematic programs in a different way. In example `Test1`, where the variable `i` is definitely assigned after the try block but not after the finally block and where there is an assignment to `i` in the finally block, the compiler could insert additional code that initializes `i` with the default value zero. The resulting bytecode program would then be accepted by the verifier.

More generally, for every variable that is definitely assigned after a try-finally statement or after a labeled statement according to the official Java Language Specification in [5, §16] but not according to the more restricting rules of this article, the compiler could insert such initialization code. This alternative compilation strategy corresponds to a source code transformation that maps the problematic programs into the slightly restricted Java language for which we can prove that the compiler will generate verifiable bytecode.

Another solution of the problem would be to change the bytecode verifier in a fundamental way so that programs like `Test1` and `Test2` can be verified when they are compiled in the standard way. The change would be to verify subroutines not only once but several times for each call of the subroutine. Equivalently the verifier could assign more than one stack map (type frame) to instructions in subroutines as it has been proposed by Coglio [2], Haase [7], Henrio and Serpette [8]. Both approaches, however, lead to an exponential behavior in the worst case (for nested subroutines).

An even more progressive solution is proposed by Leroy in [10]. He forbids that in a method body the same local variable can be used with different types for different purposes. Moreover, he requires that the virtual machine has to initialize every local variable with a default value (which is currently not required for reasons of efficiency). Then the problems described in this paper disappear and bytecode verification becomes much simpler and is even feasible on smart cards.

The bytecode for the two example programs is also rejected by most verifiers that have been proposed in the literature. For example, Stata and Abadi [17] treat subroutines in a simplistic way. They do not consider the possibility to jump out of a subroutine to an enclosing

subroutine via a `break` or `throw`. The examples still cannot be typed in the extended and refined system of Freund and Mitchell [4]. They can also not be typed in the system of O’Callahan [12] which is based on ideas of type systems for continuations and polymorphic recursion. It seems that any verifier that checks each subroutine only once will reject legal Java programs like the two above. This includes also various systems by Qian [13].

Some of the cited articles use subroutine call stacks either directly as part of the type system or as a means to show the correctness of the type system. We do not follow this approach. There are several reasons for that. First of all, the same subroutine may be called from several different subroutines, for example, when the try block of a try-finally statement contains again try-finally statements. Hence it is not sufficient to assign a single subroutine call stack to each instruction, but in general one needs a set of possible subroutine call stacks. Moreover, it is possible to leave a finally block via an exception or a break statement. This means that in the bytecode a subroutine can be left by a simple `Goto` instruction or a jump via the exception table to an enclosing subroutine. In this case several subroutines have to be popped from the subroutine call stack at once. Finally, it is not clear to us how to compare subroutine call stacks and how to define a notion of most specific subroutine call stacks as we do it for bytecode type assignments below in Def. 16.

In this article we prove property 1 and 2 for the fragment of Java defined in Table I. The fragment contains the problematic example programs. We introduce a notion of bytecode type assignments and show the following:

1. Bytecode which can be typed does not violate the checks of the defensive VM at run-time (Theorem 15).
2. If bytecode is accepted by the verifier, it can be typed (Theorem 17).
3. If bytecode can be typed, then it is accepted by the verifier and the verifier computes a principal type assignment (Theorem 18).
4. The Java-to-JVM compiler generates bytecode that can be typed (Theorem 40).

Similar theorems are proved in [16] for the full Java language. In this article we focus on the essential problems which are already contained in the fragment in Table I.

For the specification of the defensive virtual machine and the bytecode verification algorithm we use the framework of Abstract State Machines [1, 6]. The specifications are executable in the AsmGofer system [14]. For the reader’s convenience we summarize here the most important concepts and notations that are used in the ASMs in this

Table I. Syntax of a fragment of Java.

$Exp$	$::=$	$Literal \mid Variable \mid (Exp \ Op \ Exp)$
$Stm$	$::=$	$Variable = Exp; \mid Block \mid \mathbf{while} (Exp) Stm$ $\mid Label: Stm \mid \mathbf{break} Label; \mid \mathbf{throw} Exp;$ $\mid \mathbf{if} (Exp) Stm \ \mathbf{else} \ Stm$ $\mid \mathbf{try} Block \ \mathbf{catch} (Type \ Variable) Block$ $\mid \mathbf{try} Block \ \mathbf{finally} Block$
$Block$	$::=$	$\{Bstm \ \dots \ Bstm\}$
$Bstm$	$::=$	$Type \ Variable; \mid Stm$

article. An abstract state of an ASM is given by a collection of dynamic functions. Nullary dynamic functions correspond to ordinary state variables. Formally all functions are total. They may, however, return the special element *undef* if they are not defined at an argument. In each step the machine updates in parallel some of the functions at certain arguments. The updates are programmed using transition rules  $P$ ,  $Q$  with the following meaning:

$f(s) := t$	Update $f$ at the argument $s$ to $t$ .
$\mathbf{if} \ \varphi \ \mathbf{then} \ P \ \mathbf{else} \ Q$	If $\varphi$ is true, then execute $P$ , else $Q$ .
$P \ Q$	Execute the updates of $P$ and $Q$ in parallel.
$\mathbf{do-seq} \ P \ Q$	Execute first $P$ , then $Q$ .
$\mathbf{choose} \ x \ \mathbf{with} \ \varphi \ \mathbf{do} \ P$	Choose an $x$ satisfying $\varphi$ and execute $P$ .
$\mathbf{forall} \ x \ \mathbf{with} \ \varphi \ \mathbf{do-seq} \ P$	Execute in an arbitrary order $P$ sequentially for each $x$ satisfying $\varphi$

The **do-seq** is not a standard ASM construct. We use it in the verification algorithm in Fig. 3 to avoid possible conflicts in the updates.

*Notations.* We use the following notations for lists. A list  $l$  with elements  $x_0, \dots, x_{n-1}$  is written as  $[x_0, \dots, x_{n-1}]$ . The  $i$ th element  $x_i$  of  $l$  can be accessed by  $l(i)$ ;  $first(l)$  is the first element  $x_0$  of  $l$ ;  $top(l)$  is the last element  $x_{n-1}$ ;  $drop(l, m)$  is the list of the first  $n - m$  elements of  $l$ ;  $l_1 \cdot l_2$  is the result of the concatenation of  $l_1$  and  $l_2$ . For functions  $f$  and  $g$  we denote by  $f \oplus g$  the combination of the two functions, where  $g$  overrides values which are defined in  $f$ . Hence, if  $f$  and  $g$  are represented as sets of pairs, then  $f \oplus g = \{(x, y) \in f \mid x \notin dom(g)\} \cup g$ .

## 2. The defensive Virtual Machine

We use a VM with an instruction set large enough to compile the fragment of Java described in Table I. The VM has the following instructions:

<code>Load(Register)</code>	<code>Store(Register)</code>	<code>Jsr(Index)</code>
<code>Const(Literal)</code>	<code>Goto(Index)</code>	<code>Ret(Register)</code>
<code>Prim(Op)</code>	<code>Ifne(Index)</code>	<code>Throw</code>

The effect of the instructions on the dynamic state of the VM is described in Fig. 1 using an Abstract State Machine. The dynamic state of the VM is given by the program counter  $pc$ , the operand stack  $opd$  and the registers  $reg$ . The  $pc$  contains the index of the instruction that has to be executed;  $opd$  is a list of values that grows from left to right;  $reg$  is a function that assigns values to register numbers (it assigns *undef* to un-initialized registers). Values are pairs consisting of a type and a bitstring. If  $v$  is a value, then  $type(v)$  is the type of the value  $v$ . The types of the VM are either primitive types (`int`, `float`, ...) or reference types (class types, interface types). In the initial state,  $pc$  is 0,  $opd$  is empty, and  $reg(x)$  is *undef* for each register  $x$ .

The instructions of the bytecode program are stored in an array with the name *code*. Hence,  $code(i)$  is the  $i$ th instruction. Information about handling exceptions is contained in the exception table *excs*, a list of entries (*from*, *up*, *type*, *handler*). An upper bound for the operand stack (computed by the compiler) is given by the constant *maxOpd*. In summary, the dynamic part of the VM is given by  $pc$ ,  $opd$ ,  $reg$  and the static part by *code*, *excs*, *maxOpd*.

The `Load(x)` instruction pushes the value of register  $x$  on top of the operand stack and increments  $pc$  by 1. The `Const(c)` instruction pushes the constant  $c$  on top of the operand stack and increments  $pc$  by 1. The `Prim(op)` instruction replaces the topmost two values of the operand stack by the result of applying the operand  $op$  to them and increments  $pc$  by 1. The result is *undef*, if one of the two values is *undef*. [Note: We do not model exceptions that arise, for example, in a division by zero.] The `Store(x)` instruction pops the topmost value from the operand stack, stores it into register  $x$  and increments  $pc$  by 1. The `Goto(i)` instruction sets  $pc$  to  $i$ . The `Ifne(i)` instruction pops the topmost value from the operand stack. If the value is not equal to zero it sets  $pc$  to  $i$ , otherwise it increments  $pc$  by 1. The `Jsr(i)` instruction pushes the return address  $pc + 1$  on top of the operand stack and sets  $pc$  to  $i$ . We assume that the type of the return address is `retAddr(i)` indicating that it is a return address for the subroutine  $i$ . The `Ret(x)` instruction sets  $pc$  to the value stored in register  $x$ . The

```

DEFENSIVEVM =
  if VALIDCODEINDEX(pc) and CHECK(pc, type(reg), type(opd))
  then EXECUTE(code(pc))
  else halt := "Check failed"

VALIDCODEINDEX(i) : $\iff$  0  $\leq$  i  $\wedge$  i < length(code)
type(reg) = {(x, type(v)) | (x, v)  $\in$  reg}
type(opd) = [type(v) | v  $\in$  opd]

EXECUTE(instr) = case instr of
  Load(x)  $\rightarrow$ 
    opd := opd · [reg(x)]
    pc := pc + 1
  Const(c)  $\rightarrow$ 
    opd := opd · [c]
    pc := pc + 1
  Prim(op)  $\rightarrow$ 
    let rest · [v1, v2] = opd in
    opd := rest · [v1 op v2]
    pc := pc + 1
  Store(x)  $\rightarrow$ 
    let rest · [v] = opd in
    opd := rest
    reg(x) := v
    pc := pc + 1
  Goto(i)  $\rightarrow$  pc := i
  Ifne(i)  $\rightarrow$ 
    let rest · [v] = opd in
    opd := rest
    if v  $\neq$  0 then pc := i
    else pc := pc + 1
  Jsrr(i)  $\rightarrow$ 
    opd := opd · [pc + 1]
    pc := i
  Ret(x)  $\rightarrow$ 
    pc := reg(x)
  Throw  $\rightarrow$ 
    let rest · [v] = opd in
    opd := [v]
    pc := handler(e)
    where
      e = first([e  $\in$  excs | MATCH(pc, v, e)])

MATCH(i, v, (f, u, t, -)) : $\iff$  f  $\leq$  i  $\wedge$  i < u  $\wedge$  type(v)  $\preceq$  t

CHECK(i, regT, opdT) : $\iff$ 
  case code(i) of
    Load(x)  $\rightarrow$ 
      regT(x)  $\neq$  Undef  $\wedge$ 
       $\neg$ OVERFLOW(opdT, 1)
    Const(c)  $\rightarrow$ 
       $\neg$ OVERFLOW(opdT, 1)
    Prim(op)  $\rightarrow$ 
      opdT  $\sqsubseteq_{\text{suf}}$  argTypes(op)
    Store(x)  $\rightarrow$ 
      length(opdT) > 0
    Goto(i)  $\rightarrow$  true
    Ifne(i)  $\rightarrow$ 
      opdT  $\sqsubseteq_{\text{suf}}$  [int]
    Jsrr(i)  $\rightarrow$ 
       $\neg$ OVERFLOW(opdT, 1)
    Ret(x)  $\rightarrow$ 
       $\exists$ s (regT(x) = retAddr(s))
    Throw  $\rightarrow$ 
      opdT  $\sqsubseteq_{\text{suf}}$  [Throwable]

opdT  $\sqsubseteq_{\text{suf}}$  ts : $\iff$  let m = length(ts), n = length(opdT) in
  m  $\leq$  n  $\wedge$   $\forall$ i < m (opdT(n - m + i)  $\preceq$  ts(i))
OVERFLOW(opdT, n) : $\iff$  length(opdT) + n > maxOpd

```

Figure 1. The defensive Virtual Machine.

**Throw** instruction takes the topmost value  $v$  of the operand stack. It clears the operand stack and pushes  $v$  on the empty stack. It then scans the exception table  $excS$  from left to right until it finds the first entry  $(f, u, t, h)$  so that  $pc$  is in the interval protected by the entry, i.e.  $f \leq pc < u$ , and the type of the exception  $v$  is compatible with  $t$ . Finally, it sets  $pc$  to  $h$ . [Note: We do not model the case where  $v$  is a null pointer.]

Before executing an instruction the defensive VM checks whether the  $pc$  is still valid and points into the code array and, depending on the instruction, whether certain conditions are satisfied. The checks for the single instructions operate on the list of types of the values on the operand stack ( $opdT$ ) and the types of the values stored in the registers ( $regT$ ) that are extracted using the function  $type$ . For example, the check for the `Prim(op)` instruction tests, whether there are at least two operands on the operand stack and that they are compatible with the argument types  $argTypes(op)$  of the operation. Since the checks use only the type information, they can later be re-used in Def. 11 (bytecode type assignment) and in the verifier (Sect. 4).

We assume that  $\preceq$  is a partial order (subtype relation) on the types  $A, B, C$  of the defensive VM. There is a topmost type `Undef` representing undefined (or unusable) values so that  $A \preceq \text{Undef}$  for each type  $A$ . Moreover,  $A \sqcup B$  is the least upper bound of two types  $A$  and  $B$ :

- $A \preceq A \sqcup B$  and  $B \preceq A \sqcup B$ .
- If  $A \preceq C$  and  $B \preceq C$ , then  $A \sqcup B \preceq C$ .

A primitive type (like `int` or `float`) has the following properties:

- If  $A$  is primitive and  $B \preceq A$ , then  $B = A$ .
- If  $A$  is primitive and  $A \preceq B$ , then  $B = A$  or  $B = \text{Undef}$ .

Return address types `retAddr(i)` are treated as primitive types.

A pair  $(regT, opdT)$  consisting of types for the registers and a list of types of an operand stack is called a *type frame*. Two type frames  $\sigma = (regS, opdS)$  and  $\tau = (regT, opdT)$  are compared as follows:

$$\begin{aligned} \sigma \sqsubseteq \tau & \iff regS \sqsubseteq regT \wedge opdS \sqsubseteq opdT \\ regS \sqsubseteq regT & \iff \forall x (regS(x) \preceq regT(x)) \\ opdS \sqsubseteq opdT & \iff length(opdS) = length(opdT) \wedge \\ & \quad \forall i < length(opdS) (opdS(i) \preceq opdT(i)) \end{aligned}$$

Hence a type frame  $\sigma$  is more specific than a type frame  $\tau$ , if the registers of  $\sigma$  have more specific types than the registers of  $\tau$ , the operand stacks of  $\sigma$  and  $\tau$  have the same length, and the type of

an operand in  $\sigma$  is more specific than the type of the corresponding operand in  $\tau$ . In particular, if a register or operand of  $\tau$  is defined (has a type different from `Undef`), then it is defined in  $\sigma$ , too. [Note: In [16],  $regT$  is a partial function. Here, it is a total function that assigns to every register a type.]

The checks of the defensive VM in Fig. 1 are monotonic. If a check is true for a code index in a type frame, then it is also true for the same code index in a more specific type frame.

**LEMMA 3** (Monotonicity of checks of the defensive VM).  
*If  $\sigma \sqsubseteq \tau$  and  $CHECK(i, \tau)$  is true, then  $CHECK(i, \sigma)$  is true.*

*Proof.* By a case distinction on the instruction  $code(i)$ .  $\square$

### 3. Bytecode type assignments

The verifier executes the bytecode instructions symbolically. It takes an instruction at a given code index together with a type frame and computes the type frames for all possible successor code indices as defined in the function `SUCC` in Fig. 2. The function `SUCC` is used by the bytecode verifier in the next section as well as in the definition of bytecode type assignment below.

The set of possible successor frames for the instruction in Fig. 2 includes also successors via the exception table. For each instruction, except for `Jsr` and `Goto` (which are used for the compilation of the break statement), all handlers which protect the code index are included into the set of possible successors by the function `ALLHANDLERS`. This conservative assumption about possible exceptions or errors of instructions is not really necessary for our simple VM which does not throw exceptions except for the `Throw` instruction. Nevertheless we make the assumption in order to get a robust verifier that can easily be extended to the real JVM.

The successors of the `Jsr` and `Ret` instructions are treated in a special way (in Def. 11) below, since they do not have the following monotonicity property.

**LEMMA 4** (Monotonicity of successor type frames).  
*If  $\sigma \sqsubseteq \tau$  and  $CHECK(pc, \tau)$  is true, then for each  $(i, \sigma') \in \text{SUCC}(pc, \sigma)$  there exists  $(i, \tau') \in \text{SUCC}(pc, \tau)$  so that  $\sigma' \sqsubseteq \tau'$ .*

We divide the instructions of the VM into normal instructions and control transfer instructions. The control transfer instructions are `Goto`( $i$ ),

```

SUCC( $pc, regT, opdT$ ) =
  let  $instr = code(pc)$  in
    ALLHANDLERS( $instr, pc, regT$ )  $\cup$ 
    case  $instr$  of
      Load( $x$ )  $\rightarrow$   $\{(pc + 1, regT, opdT \cdot [regT(x)])\}$ 
      Const( $c$ )  $\rightarrow$   $\{(pc + 1, regT, opdT \cdot [type(c)])\}$ 
      Prim( $op$ )  $\rightarrow$   $\{(pc + 1, regT, drop(opdT, 2) \cdot retType(op))\}$ 
      Store( $x$ )  $\rightarrow$   $\{(pc + 1, regT \oplus \{(x, top(opdT))\}, drop(opdT, 1))\}$ 
      Goto( $i$ )  $\rightarrow$   $\{(i, regT, opdT)\}$ 
      Ifne( $i$ )  $\rightarrow$   $\{(pc + 1, regT, drop(opdT, 1)), (i, regT, drop(opdT, 1))\}$ 
      Jsr( $i$ )  $\rightarrow$   $\{(i, regT, opdT \cdot [retAddr(i)])\}$ 
      Ret( $x$ )  $\rightarrow$   $\emptyset$ 
      Throw  $\rightarrow$   $\emptyset$ 

ALLHANDLERS(Jsr(-),  $pc, regT$ ) =  $\emptyset$ 
ALLHANDLERS(Goto(-),  $pc, regT$ ) =  $\emptyset$ 
ALLHANDLERS(-,  $pc, regT$ ) =  $\{(h, regT, [t]) \mid (f, u, t, h) \in excs, f \leq pc < u\}$ 

```

Figure 2. Computing successor type frames.

Ifne( $i$ ), Jsr( $i$ ), Ret( $x$ ), Throw. Each code index has a (possibly empty) set of successor indices which can be reached in the next step of the execution of the VM.

DEFINITION 5 (Successor index). A code index  $j$  is called a successor index of  $i$ , if one of the following conditions is true:

1.  $code(i)$  is not a control transfer instruction and  $j = i + 1$ , or
2.  $code(i) = \text{Goto}(j)$ , or
3.  $code(i) = \text{Ifne}(k)$  or  $code(i) = \text{Jsr}(k)$  and  $j \in \{i + 1, k\}$ , or
4. there exists a handler  $(f, u, -, j)$  in the exception table such that  $f \leq i < u$  and  $code(i)$  is neither Jsr nor Goto.

Note that the index of a Jsr instruction has two possible successor indices. The type frame associated to a successor instruction of Jsr in Fig. 2 has only one element (except for type frames of possible exception handlers). This is because in the intended “pairing” of Jsr and Ret instructions, the subroutine called by Jsr returns to the instruction immediately following Jsr. In the following definition of reachability it is important that both successor indices of Jsr are included.

DEFINITION 6 (Reachable). A code index  $j$  is called reachable from  $i$  iff there exists a finite (possibly empty) sequence of successor steps from  $i$  to  $j$  according to Def. 5.

Since the treatment of subroutines is rather complicated we have to define precisely what we mean by a subroutine.

DEFINITION 7 (Subroutine). If  $i$  is reachable from 0 and  $code(i)$  is  $\text{Jsr}(s)$ , then the code index  $s$  is called a subroutine.

We make the following assumptions about subroutines. We assume that the first instruction of a subroutine is always a **Store** instruction that stores the return address in a register. This assumption is reasonable, since if a subroutine ever returns, it has to take its return address from a register and not from the operand stack in the **Ret** instruction. Hence, the return address has to be stored in a register somewhere in the subroutine. We assume that this is done at the beginning. Our second assumption is that the code generated by a Java compiler for a finally block is connected. The code may contain several **Ret** instructions, but the code indices for the block must form an interval. Under these assumptions, the possible returns from a subroutine are easy to identify.

DEFINITION 8 (Return from subroutine). A code index  $r$  is a return from subroutine  $s$ , if  $code(s) = \text{Store}(x)$ ,  $code(r) = \text{Ret}(x)$  and  $r$  is reachable from  $s + 1$  on a path that does not use any  $\text{Store}(x)$  instruction.

The instructions which belong to a subroutine are simply those which are in the interval between the first instruction of the subroutine and any possible return from the subroutine (there could be several or none).

DEFINITION 9 (Belongs to a subroutine). A code index  $i$  belongs to subroutine  $s$ , if there exists a return  $r$  from  $s$  so that  $s \leq i \leq r$ . The set of all indices which belong to  $s$  is denoted by  $\text{BELONGS TO}(s)$ .

For the treatment of polymorphic subroutines the set of registers that are modified by a subroutine includes also the registers that are used in other subroutines called by this subroutine as well as the registers that are used in implicitly called exception handlers, as long as they belong to the subroutine.

DEFINITION 10 (Modified registers). A register  $x$  is modified by the subroutine  $s$ , if there exists a code index  $i$  which belongs to  $s$  so that  $code(i) = \text{Store}(x)$ .

The set of registers that are modified by a subroutine is used to restrict the type assignment to registers when a subroutine returns

with a  $\text{Ret}(x)$  instruction. At this time, the register  $x$  must be of type  $\text{retAddr}(s)$  for some subroutine  $s$ . The types of the registers modified by  $s$  have to be returned to the callers of  $s$ . Since  $\text{Ret}$  instructions have no successor type frames, they have to be treated in a special way in the following definition.

DEFINITION 11 (Bytecode type assignment). A bytecode type assignment with domain  $\mathcal{D}$  is a family  $(\text{reg}T_i, \text{opd}T_i)_{i \in \mathcal{D}}$  of type frames that satisfies the following conditions:

- T1.  $\mathcal{D}$  is a set of valid code indices.
- T2. Code index 0 belongs to  $\mathcal{D}$ .
- T3.  $\text{reg}T_0(x) = \text{Undef}$  for each register  $x$ .
- T4. The list  $\text{opd}T_0$  is empty.
- T5. If  $i \in \mathcal{D}$ , then  $\text{CHECK}(i, \text{reg}T_i, \text{opd}T_i)$  is true.
- T6. If  $i \in \mathcal{D}$  and  $(j, \text{reg}S, \text{opd}S) \in \text{SUCC}(i, \text{reg}T_i, \text{opd}T_i)$ , then  $j \in \mathcal{D}$ ,  $\text{reg}S \sqsubseteq \text{reg}T_j$  and  $\text{opd}S \sqsubseteq \text{opd}T_j$ .
- T7. If  $i \in \mathcal{D}$ ,  $\text{code}(i) = \text{Ret}(x)$  and  $\text{reg}T_i(x) = \text{retAddr}(s)$ , then for every  $j \in \mathcal{D}$  with  $\text{code}(j) = \text{Jsr}(s)$  and every register  $y$ :
  - a)  $j + 1 \in \mathcal{D}$
  - b) if  $y$  is modified by  $s$ , then  $\text{reg}T_i(y) \preceq \text{reg}T_{j+1}(y)$
  - c) if  $y$  is not modified by  $s$ , then  $\text{reg}T_j(y) \preceq \text{reg}T_{j+1}(y)$
  - d)  $\text{opd}T_i \sqsubseteq \text{opd}T_{j+1}$
  - e) if  $y$  is not modified by  $s$  and  $\text{reg}T_{j+1}(y) = \text{retAddr}(\ell)$ , then each code index which belongs to  $s$  belongs to  $\ell$
- T8. If  $i \in \mathcal{D}$  and  $\text{retAddr}(s)$  occurs in  $\text{reg}T_i$ , then  $i$  belongs to  $s$ .  
If  $i \in \mathcal{D}$  and  $\text{retAddr}(s)$  occurs in  $\text{opd}T_i$ , then  $i = s$ .

If the *code* array has a type assignment with T1–T8, then it does not violate the checks when it runs on the defensive VM. One can show that at run-time the program counter is always a valid code index. The values stored in the registers and the values of the operands have at run-time the types indicated by the type assignment. Moreover, the operand stack has at run-time exactly the same length as the list of types which are assigned to the operand stack.

The crucial point is how to define what it means that a return address is of type  $\text{retAddr}(s)$ . The idea is that  $j + 1$  belongs to the type  $\text{retAddr}(s)$ , if  $\text{code}(j)$  is the instruction  $\text{Jsr}(s)$  and, for all registers  $x$  which are not modified by the subroutine  $s$ , the value of  $x$  is of the type assigned to  $x$  at index  $j + 1$ . Therefore a return address has to be typed with respect to the registers.

DEFINITION 12 (Typing rules). Let  $(regT_i, opdT_i)_{i \in \mathcal{D}}$  be a bytecode type assignment. The typing judgment  $reg \vdash v: A$  is inductively defined by the following rules:

$$\frac{type(v) \preceq A, A \neq \mathbf{retAddr}(-)}{reg \vdash v: A}$$

$$\frac{j \in \mathcal{D}, code(j) = \mathbf{JsR}(s), s \text{ does not return}}{reg \vdash j + 1: \mathbf{retAddr}(s)}$$

$$\frac{j \in \mathcal{D}, code(j) = \mathbf{JsR}(s), s \text{ returns}}{reg \vdash reg(x): regT_{j+1}(x) \text{ for each } x \text{ not modified by } s} \\ reg \vdash j + 1: \mathbf{retAddr}(s)$$

We say that a subroutine  $s$  returns, if there exists an  $i \in \mathcal{D}$  so that  $code(i) = \mathbf{Ret}(x)$  and  $regT_i(x) = \mathbf{retAddr}(s)$ .

The typing rules for return addresses depend on the registers and the type assignment. Since a **Store** instruction changes the contents of registers, a return address could possibly loose its type. The following coincidence lemma therefore states a convenient condition under which a return address keeps its type. It says that the typing of a return address from a subroutine depends only on the registers which are not modified by the subroutine.

LEMMA 13 (Coincidence). *If  $reg \vdash j + 1: \mathbf{retAddr}(s)$  and, for each  $x$  not modified by  $s$ ,  $reg(x) = reg'(x)$ , then  $reg' \vdash j + 1: \mathbf{retAddr}(s)$ .*

*Proof.* By induction on the length of a derivation of  $reg \vdash v: A$ .

If  $s$  does not return, we obtain  $reg' \vdash j + 1: \mathbf{retAddr}(s)$  by the second typing rule. Otherwise, by the third typing rule we have

1.  $j \in \mathcal{D}$ ,  $code(j) = \mathbf{JsR}(s)$ ,  $s$  returns,
2.  $reg \vdash reg(x): regT_{j+1}(x)$  for each  $x$  not modified by  $s$ .

Let  $x$  be a register which is not modified by  $s$ . Since  $reg(x) = reg'(x)$ , it follows by (2) that  $reg \vdash reg'(x): regT_{j+1}(x)$ . If  $regT_{j+1}(x)$  is not a return address type, then, by the first typing rule, we immediately obtain  $reg' \vdash reg'(x): regT_{j+1}(x)$ . Otherwise, there is an  $\ell$  so that  $regT_{j+1}(x) = \mathbf{retAddr}(\ell)$ . By T7 (e) and Def. 10, it follows that each register modified by  $s$  is also modified by  $\ell$ . Hence,  $reg(x) = reg'(x)$  for each  $x$  not modified by  $\ell$ . By the induction hypothesis, we obtain that  $reg' \vdash reg'(x): \mathbf{retAddr}(\ell)$ .

Thus, the three premises of the third typing rule are satisfied and we can conclude that  $reg' \vdash j + 1: \mathbf{retAddr}(s)$ .  $\square$

LEMMA 14.

1. If  $reg \vdash v: A$ , then  $type(v) \preceq A$ .
2. If  $reg \vdash v: A$  and  $A \preceq B$ , then  $reg \vdash v: B$ .

*Proof.* By induction on the derivation of  $reg \vdash v: A$ .  $\square$

That bytecode type assignments are sound can now be proved by a simple induction on the run of the defensive VM.

THEOREM 15 (Soundness of bytecode type assignments).

Let  $(regT_i, opdT_i)_{i \in \mathcal{D}}$  be a bytecode type assignment. Then the following invariants are satisfied at run-time on the defensive VM:

- (pc)**  $pc \in \mathcal{D}$  and  $pc$  is a valid code index.
- (reg)**  $reg \vdash reg(x): regT_{pc}(x)$  for each register  $x$ .
- (length)**  $length(opd) = length(opdT_{pc}) \leq maxOpd$ .
- (opd)**  $reg \vdash opd(i): opdT_{pc}(i)$  for each  $i < length(opd)$ .
- (check)**  $CHECK(pc, type(reg), type(opd))$  is true.

*Proof.* We show first that the invariant (check) follows from the other invariants:

Since  $pc \in \mathcal{D}$ , by T5 it follows that  $CHECK(pc, regT_{pc}, opdT_{pc})$  is true. Let  $regS = types(reg)$  and  $opdS = types(opd)$  be the types associated to the registers  $reg$  and the operand stack  $opd$ . From (reg), (length) and (opd) we can deduce using Lemma 14 that  $regS \sqsubseteq regT_{pc}$  and  $opdS \sqsubseteq opdT_{pc}$ . Since the checks of the defensive VM are monotonic (Lemma 3), it follows that  $CHECK(pc, regS, opdS)$  is true as well.

The remaining invariants are proved by an induction on the run of the defensive VM. In the initial state, when  $pc$  is 0, the invariants are satisfied due to T2–T4. In the induction step we have to consider different cases depending on the instruction  $code(pc)$ :

**Case 1.**  $code(pc) = \mathbf{JsR}(s)$ : The new  $pc$  is  $s$  and the new operand stack is  $opd \cdot [pc + 1]$ . The type assignment condition T6 ensures that  $s \in \mathcal{D}$  and

- $regT_{pc} \sqsubseteq regT_s$
- $opdT_{pc} \cdot [\mathbf{retAddr}(s)] \sqsubseteq opdT_s$

This implies by the transitivity of  $\preceq$  and the induction hypothesis the invariants (pc), (reg), (length) and (opd) at  $s$ , except for the return address on top of the operand stack for which one has to show that  $reg \vdash pc + 1: \mathbf{retAddr}(s)$ . The induction hypothesis (reg) yields that  $reg \vdash reg(x): regT_{pc}(x)$  for every register  $x$ . If  $s$  does not return, then by the second typing rule of Def. 12,  $reg \vdash pc + 1: \mathbf{retAddr}(s)$ . Otherwise, if  $s$  returns, then by T7 (c),  $regT_{pc}(x) \preceq regT_{pc+1}(x)$  for each  $x$  not modified by  $s$ , and hence  $reg \vdash reg(x): regT_{pc+1}(x)$ . By the third typing rule of Def. 12, it follows that  $reg \vdash pc + 1: \mathbf{retAddr}(s)$ .

**Case 2.**  $code(pc) = \mathbf{Ret}(x)$ : Condition T5 and the check for the **Ret** instruction in Fig. 1 ensure that there exists a subroutine  $s$  so that  $regT_{pc}(x) = \mathbf{retAddr}(s)$  and therefore  $s$  returns. The induction hypothesis for  $x$  implies that  $reg \vdash reg(x): \mathbf{retAddr}(s)$ . Let  $reg(x) = j + 1$ . The judgement  $reg \vdash reg(x): \mathbf{retAddr}(s)$  cannot be inferred using the first or second typing rule in Def. 12. It can only be inferred using the third typing rule, hence

$$reg \vdash reg(y): regT_{j+1}(y) \text{ for each } y \text{ not modified by } s.$$

The new  $pc$  is  $j + 1$ . The typing rules in Def. 12 also yield that  $j \in \mathcal{D}$  and  $code(j) = \mathbf{JsR}(s)$ . The type assignment conditions T7 (a), (b), (d) ensure that

- $j + 1 \in \mathcal{D}$  (hence  $j + 1$  is a valid code index by T1)
- if  $y$  is modified by  $s$ , then  $regT_{pc}(y) \preceq regT_{j+1}(y)$
- $opdT_{pc} \sqsubseteq opdT_{j+1}$

Thus, the invariant (reg) follows also for registers  $y$  which are modified by  $s$ . The invariants (length) and (opd) follow immediately from the induction hypothesis, since the operand stack is propagated to  $j + 1$  without modifications.

**Case 3.**  $code(pc) = \mathbf{Store}(x)$ : The type assignment condition T6 ensures that there exist  $opdS$  and  $\tau$  so that

- $opdT_{pc} = opdS \cdot [\tau]$
- $opdS \sqsubseteq opdT_{pc+1}$
- $regT_{pc} \oplus \{(x, \tau)\} \sqsubseteq regT_{pc+1}$

The new  $pc$  is  $pc + 1$ , the new operand stack is  $opd'$  and the new registers are  $reg'$ , where  $opd = opd' \cdot [v]$  and  $reg' = reg \oplus \{(x, v)\}$ . Invariants (pc), (reg), (length) and (opd) for  $pc + 1$  follow from the induction hypothesis using part 1 of Def. 12, except for operands and registers

of return address type. Assume that  $\mathbf{retAddr}(\ell)$  occurs in  $regT_{pc+1}$  or  $opdT_{pc+1}$ . Then  $\mathbf{retAddr}(\ell)$  already occurs in  $regT_{pc}$  or  $opdT_{pc}$  and, by T8,  $pc$  belongs to subroutine  $\ell$ . Therefore  $x$  is modified by  $\ell$  (Def. 10) and  $reg(y) = reg'(y)$  for each  $y$  not modified by  $\ell$ . We can apply the Coincidence Lemma 13 and see that, if  $reg \vdash j + 1 : \mathbf{retAddr}(\ell)$ , then  $reg' \vdash j + 1 : \mathbf{retAddr}(\ell)$ . Hence, the invariants are also satisfied for return addresses at code index  $pc + 1$ .

**Case 4.**  $code(pc) = \mathbf{Throw}$ : Condition T5 and the check for the **Throw** instruction in Fig. 1 ensure that the operand stack is not empty and  $type(v) \preceq \mathbf{Throwable}$  for the topmost element  $v$  of  $opd$ . Let  $(f, u, t, h)$  be an entry in the exception table  $exc$ s such that  $f \leq pc < u$  and  $type(v)$ . (That the VM selects the first such entry is not important here.) Then the new  $pc$  is  $h$  and the new operand stack is  $[v]$ . By assumption each target of a handler is a valid code index and hence the invariant (pc) is still satisfied. The function **ALLHANDLERS** in Fig. 2 includes  $(h, regT_{pc}, [t])$  into the set of successor type frames of  $pc$ . Hence, by T6,  $regT_{pc} \sqsubseteq regT_h$  and  $[t] \sqsubseteq opdT_h$ . The invariant (reg) follows using the induction hypothesis. The invariants (opd) and (length) are true, since the new operand stack is  $[v]$  and  $type(v) \preceq t$ .

The remaining cases for **Load**( $x$ ), **Const**( $c$ ), **Prim**( $op$ ), **Goto**( $i$ ), and **Ifne**( $i$ ) shown in a similar way.  $\square$

#### 4. The bytecode verification algorithm

The bytecode verification algorithm decides whether there exists a type assignment with T1–T8 for the code array. In the positive case it computes a most specific type assignment. The relation ‘more specific’ is defined as follows:

DEFINITION 16 (More specific bytecode type assignment).

A type assignment  $(regV_i, opdV_i)_{i \in \mathcal{V}}$  is more specific than the type assignment  $(regT_i, opdT_i)_{i \in \mathcal{D}}$  iff the following conditions are satisfied:

1.  $\mathcal{V} \subseteq \mathcal{D}$ ,
2.  $regV_i \sqsubseteq regT_i$  for each  $i \in \mathcal{V}$ ,
3.  $opdV_i \sqsubseteq opdT_i$  for each  $i \in \mathcal{V}$ .

Hence a more specific type assignment assigns type frames to less code indices. It assigns more specific types to the registers and the operand stacks. A most specific type assignment to bytecode is called a *principal type assignment*.

The bytecode verification algorithm attempts to compute a bytecode type assignment  $(regV_i, opdV_i)_{i \in \mathcal{V}}$  for the *code* array. At the beginning the set  $\mathcal{V}$  consists of code index zero only. More indices are added to  $\mathcal{V}$  whenever they can be reached by the algorithm which tries to propagate the already computed type frames to the possible successor indices. It can happen that during this process an index is revisited and the type frame for the index has to be changed. In this case the index is added to a set  $\mathcal{C}$  of changed indices whereby it becomes subject to yet another verification step, namely the attempt to propagate its new type frame. The bytecode verifier proceeds until the set  $\mathcal{C}$  is empty.

The bytecode verification algorithm in Fig. 3 uses the following dynamic functions:

$$\begin{array}{ll} regV : Index \rightarrow Map(Register, Type) & visited : Index \rightarrow Bool \\ opdV : Index \rightarrow List(Type) & changed : Index \rightarrow Bool \end{array}$$

The set of visited and changed code indices are:

$$\mathcal{V} = \{i \mid visited(i) = \mathbf{true}\} \quad \mathcal{C} = \{i \mid changed(i) = \mathbf{true}\}$$

In the initial state,  $\mathcal{V} = \mathcal{C} = \{0\}$ ,  $opdV_0 = []$  and  $regV_0(x) = \mathbf{Undef}$  for each register  $x$ .

When type frames are propagated in PROPAGATESUCC, register types are merged so that  $(regS \sqcup regT)(x) = regS(x) \sqcup regT(x)$  for each register  $x$ . Operand stacks can be merged only if they have the same length. If this is the case, they are merged component-wise, too. Note, that by our assumptions on types (in Sect. 2) we have, for example, that  $\mathbf{int} \sqcup \mathbf{boolean} = \mathbf{Undef}$  and  $\mathbf{int} \sqcup \mathbf{retAddr}(i) = \mathbf{Undef}$ .

**THEOREM 17** (Soundness of the verification algorithm).

*During the bytecode verification the following invariants are satisfied:*

- I1.  $\mathcal{C} \subseteq \mathcal{V}$  and  $\mathcal{V}$  is a set of valid code indices.
- I2. Code index 0 belongs to  $\mathcal{V}$ .
- I3.  $regV_0(x) = \mathbf{Undef}$  for each register  $x$ .
- I4. The list  $opdV_0$  is empty.
- I5. If  $i \in \mathcal{V} \setminus \mathcal{C}$ , then  $\mathbf{CHECK}(i, regV_i, opdV_i)$  is true.

```

VERIFY = if  $\exists i$  changed( $i$ ) = true then
  choose  $i$  with changed( $i$ ) = true do
    if CHECK( $i$ , reg $V_i$ , opd $V_i$ ) = true then do-seq
      changed( $i$ ) := false
      PROPAGATE( $i$ )
    else halt := "Verification failed (check violated)"

PROPAGATE( $i$ ) = do-seq
  forall ( $s$ , reg $S$ , opd $S$ )  $\in$  SUCC( $i$ , reg $V_i$ , opd $V_i$ ) do-seq
    PROPAGATESUCC( $s$ , reg $S$ , opd $S$ )
  case code( $i$ ) of
    JsR( $s$ )  $\rightarrow$ 
      forall  $j$  with visited( $j$ ) = true  $\wedge$  changed( $j$ ) = false  $\wedge$ 
        code( $j$ ) = Ret( $x$ )  $\wedge$  reg $V_j$ ( $x$ ) = retAddr( $s$ )
      do-seq PROPAGATEJSR( $i$ ,  $s$ ,  $j$ )
    Ret( $x$ )  $\rightarrow$  let retAddr( $s$ ) = reg $V_i$ ( $x$ ) in
      forall  $j$  with visited( $j$ ) = true  $\wedge$  changed( $j$ ) = false  $\wedge$ 
        code( $j$ ) = JsR( $s$ )
      do-seq PROPAGATEJSR( $j$ ,  $s$ ,  $i$ )

PROPAGATESUCC( $s$ , reg $S$ , opd $S$ ) =
  if visited( $s$ ) = false then
    if VALIDCODEINDEX( $s$ ) then
      reg $V_s$  := {( $x$ , if VALIDREG( $t$ ,  $s$ ) then  $t$  else Undef) | ( $x$ ,  $t$ )  $\in$  reg $S$ }
      opd $V_s$  := [if VALIDOPD( $t$ ,  $s$ ) then  $t$  else Undef |  $t \in$  opd $S$ ]
      visited( $s$ ) := true
      changed( $s$ ) := true
    else halt := "Verification failed (invalid code index)"
  elseif reg $S \sqsubseteq$  reg $V_s \wedge$  opd $S \sqsubseteq$  opd $V_s$  then skip
  elseif length(opd $S$ ) = length(opd $V_s$ ) then
    reg $V_s$  := reg $V_s \sqcup$  reg $S$ 
    opd $V_s$  := opd $V_s \sqcup$  opd $S$ 
    changed( $s$ ) := true
  else halt := "Verification failed (propagation not possible)"

PROPAGATEJSR( $j$ ,  $s$ ,  $i$ ) =
  PROPAGATESUCC( $j + 1$ , reg $J \cup$  reg $R$ , opd $V_i$ ) where
    reg $R$  = {( $x$ ,  $t$ )  $\in$  reg $V_i$  |  $x$  modified by  $s$ }
    reg $S$  = {( $x$ ,  $t$ )  $\in$  reg $V_j$  |  $x$  not modified by  $s$ }
    reg $J$  = {( $x$ , if VALIDJUMP( $t$ ,  $s$ ) then  $t$  else Undef) | ( $x$ ,  $t$ )  $\in$  reg $S$ }

VALIDREG( $t$ ,  $s$ ) : $\iff$  ( $t =$  retAddr( $\ell$ )  $\Rightarrow s \in$  BELONGSTO( $\ell$ ))
VALIDOPD( $t$ ,  $s$ ) : $\iff$  ( $t =$  retAddr( $\ell$ )  $\Rightarrow s = \ell$ )
VALIDJUMP( $t$ ,  $s$ ) : $\iff$  ( $t =$  retAddr( $\ell$ )  $\Rightarrow$  BELONGSTO( $s$ )  $\subseteq$  BELONGSTO( $\ell$ ))

```

Figure 3. The bytecode verification algorithm.

- I6. If  $i \in \mathcal{V} \setminus \mathcal{C}$  and  $(j, \text{reg}S, \text{opd}S) \in \text{SUCC}(i, \text{reg}V_i, \text{opd}V_i)$ , then  $j \in \mathcal{V}$ ,  $\text{reg}S \sqsubseteq \text{reg}V_j$  and  $\text{opd}S \sqsubseteq \text{opd}V_j$ .
- I7. If  $i \in \mathcal{V} \setminus \mathcal{C}$ ,  $\text{code}(i) = \mathbf{Ret}(x)$  and  $\text{reg}V_i(x) = \mathbf{retAddr}(s)$ , then for every  $j \in \mathcal{V} \setminus \mathcal{C}$  with  $\text{code}(j) = \mathbf{Jsr}(s)$  and every register  $y$ :
- a)  $j + 1 \in \mathcal{V}$
  - b) if  $y$  is modified by  $s$ , then  $\text{reg}V_i(y) \preceq \text{reg}V_{j+1}(y)$
  - c) if  $y$  is not modified by  $s$ , then  $\text{reg}V_j(y) \preceq \text{reg}V_{j+1}(y)$
  - d)  $\text{opd}V_i \sqsubseteq \text{opd}V_{j+1}$
  - e) if  $y$  is not modified by  $s$  and  $\text{reg}V_{j+1}(y) = \mathbf{retAddr}(\ell)$ , then each code index which belongs to  $s$  belongs to  $\ell$
- I8. If  $i \in \mathcal{V}$  and  $\mathbf{retAddr}(s)$  occurs in  $\text{reg}V_i$ , then  $i$  belongs to  $s$ .  
If  $i \in \mathcal{V}$  and  $\mathbf{retAddr}(s)$  occurs in  $\text{opd}V_i$ , then  $i = s$ .

*Proof.* By induction on the run of the verifier in Fig. 3.

The set  $\mathcal{C}$  of changed indices is always a subset of the set  $\mathcal{V}$  of visited code indices, since a new code index  $s$  is added to  $\mathcal{C}$  only in the  $\text{PROPAGATESUCC}(s, -, -)$  rule. The set  $\mathcal{V}$  contains only valid code indices, since in  $\text{PROPAGATESUCC}(s, -, -)$  either the index  $s$  is visited for the first time and is added to  $\mathcal{V}$  because it is a valid code index, or, in case of a merge, we know that  $s$  has already been visited. Hence invariant I1 is satisfied.

The verification starts at code index 0 with empty operand stack and each register of type  $\mathbf{Undef}$ . Hence, the invariants I2–I4 are satisfied.

At the beginning of the verification process I5 holds because by initialization  $\mathcal{V} \setminus \mathcal{C}$  is empty. A code index is removed from  $\mathcal{C}$  only if its type frame satisfies the  $\mathbf{CHECK}$  predicate, so that I5 is satisfied also during the verification process.

If a code index  $i$  is removed from  $\mathcal{C}$ , then  $\text{PROPAGATE}(i)$  enforces the invariant I6 about the successor indices.

When  $\text{PROPAGATEJSR}(j, s, i)$  is called, we know that  $i$  and  $j$  are in  $\mathcal{V} \setminus \mathcal{C}$ ,  $\text{code}(j) = \mathbf{Jsr}(s)$ ,  $\text{code}(i) = \mathbf{Ret}(x)$ , and  $\text{reg}V_i(x) = \mathbf{retAddr}(s)$ . The types of registers which are propagated to  $j + 1$  are a combination of the types at  $i$  (for registers that are modified by  $s$  in I7 b) and the types at  $j$  (for registers that are not modified by  $s$  in I7 c). If the type of a register at  $j$  is  $\mathbf{retAddr}(\ell)$  and the subroutine  $s$  is not contained in  $\ell$ , then the type  $\mathbf{Undef}$  is propagated (I7 e).

Invariant I8 remains true, since in  $\text{PROPAGATESUCC}$  invalid types are replaced by  $\mathbf{Undef}$ , when a new code index is added to  $\mathcal{V}$ .  $\square$

If the set  $\mathcal{C}$  of changed code indices is empty, then the invariants I1–I8 are equivalent to the conditions T1–T8 of Def. 11 and the verifier has computed a bytecode type assignment. Conversely, if the code to be verified has a bytecode type assignment, then the bytecode type assignment computed by the verifier will be more specific. In other words, the verifier is complete. The algorithm terminates, since each time when a code index is re-visited, its type frame is merged and only finitely many merges are possible.

**THEOREM 18** (Completeness of the verification algorithm).

*If the code array has a bytecode type assignment  $(regT_i, opdT_i)_{i \in \mathcal{D}}$ , then during the verification process the type assignment  $(regV_i, opdV_i)_{i \in \mathcal{V}}$  is always more specific than  $(regT_i, opdT_i)_{i \in \mathcal{D}}$  and no **VerifyError** occurs.*

*Proof.* The initial assignment  $(regV_0, opdV_0)$  with which the verifier starts satisfies the completeness condition, namely by properties T1–T4 of bytecode type assignments.

We prove now that the condition is preserved under each verification step. Assume that **VERIFY** chooses a code index  $i \in \mathcal{C}$ . Since  $\mathcal{C}$  is a subset of  $\mathcal{V}$  and  $\mathcal{V}$  is by the induction hypothesis a subset of  $\mathcal{D}$ , we know by T5 that the function  $\text{CHECK}(i, regT_i, opdT_i)$  is true. By the induction hypothesis  $regV_i$  and  $opdV_i$  are more specific than  $regT_i$  and  $opdT_i$  so that by the monotonicity of the checks (Lemma 3), it follows that  $\text{CHECK}(i, regV_i, opdV_i)$  is true as well and the verifier proceeds and propagates the successors.

Assume that  $(s, regS, opdS)$  is one of the successors with respect to  $regV_i$  and  $opdV_i$ . By the monotonicity of successors (Lemma 4), we know that there exists also a successor  $(s, regU, opdU)$  of  $i$  with respect to the less specific type frame  $(regT_i, opdT_i)$  so that

$$regS \sqsubseteq regU \text{ and } opdS \sqsubseteq opdU.$$

By T6 in Def. 11,  $s \in \mathcal{D}$ ,  $regU \sqsubseteq regT_s$  and  $opdU \sqsubseteq opdT_s$ . By the transitivity of  $\sqsubseteq$ , it follows that

$$regS \sqsubseteq regT_s \text{ and } opdS \sqsubseteq opdT_s.$$

By T1,  $s \in \mathcal{D}$  implies that  $s$  is a valid code index.

If the index  $s$  is visited for the first time in **PROPAGATESUCC**, then  $regV_s$  is set to  $regS'$  and  $opdV_s$  to  $opdS'$ , where  $regS'$  and  $opdS'$  are obtained from  $regS$  and  $opdS$  by replacing all invalid return addresses by **Undef**. A return address  $\text{retAddr}(\ell)$  for a register is valid, if  $s$

belongs to the subroutine  $\ell$  (Def. 9). A return address  $\mathbf{retAddr}(\ell)$  is valid on the operand stack, only if  $s = \ell$ . We have to show that

$$\mathit{regS}' \sqsubseteq \mathit{regT}_s \text{ and } \mathit{opdS}' \sqsubseteq \mathit{opdT}_s.$$

Let  $x$  be a register so that  $\mathit{regS}(x)$  is  $\mathbf{retAddr}(\ell)$  and  $s$  does not belong to  $\ell$ . Since  $\mathit{regS}(x) \sqsubseteq \mathit{regT}_s(x)$ , it follows that  $\mathit{regT}_s(x)$  is either  $\mathbf{retAddr}(\ell)$  or  $\mathbf{Undef}$ . By T8, the first case is not possible and hence  $\mathit{regT}_s(x)$  is  $\mathbf{Undef}$  and  $\mathit{regS}'(x) \sqsubseteq \mathit{regT}(x)$ .

Assume that  $k$  is a location on the operand stack so that  $\mathit{opdS}(k)$  is  $\mathbf{retAddr}(\ell)$  and  $s \neq \ell$ . Since  $\mathit{opdS}(k) \sqsubseteq \mathit{opdT}_s(k)$ , it follows that  $\mathit{opdT}_s(k)$  is either  $\mathbf{retAddr}(\ell)$  or  $\mathbf{Undef}$ . By T8, the first case is not possible and hence  $\mathit{opdT}_s(k)$  is  $\mathbf{Undef}$  and  $\mathit{opdS}'(k) \sqsubseteq \mathit{opdT}_s(k)$ .

If the successor index has been seen before we know by the induction hypothesis that  $\mathit{regV}_s \sqsubseteq \mathit{regT}_s$  and  $\mathit{opdV}_s \sqsubseteq \mathit{opdT}_s$ .

If  $(\mathit{regS}, \mathit{opdS})$  is more specific than  $(\mathit{regV}_s, \mathit{opdV}_s)$ , then the verifier does not change the so far computed type assignment. Otherwise, since  $\mathit{opdS} \sqsubseteq \mathit{opdT}_s$  and  $\mathit{opdV}_s \sqsubseteq \mathit{opdT}_s$ , both stacks have the same length and can be merged. By the properties of least upper bounds, we can conclude that  $\mathit{regV}_s \sqcup \mathit{regS} \sqsubseteq \mathit{regT}_s$  and  $\mathit{opdV}_s \sqcup \mathit{opdS} \sqsubseteq \mathit{opdT}_s$ , and the new type assignment remains more specific.

When  $\mathbf{PROPAGATEJSR}(j, s, i)$  is called we know by the induction hypothesis that

$$\begin{aligned} \mathit{regV}_j &\sqsubseteq \mathit{regT}_j \text{ and } \mathit{opdV}_j \sqsubseteq \mathit{opdT}_j, \\ \mathit{regV}_i &\sqsubseteq \mathit{regT}_i \text{ and } \mathit{opdV}_i \sqsubseteq \mathit{opdT}_i. \end{aligned}$$

We have to show that in the next step of the verification, after the successors have been propagated to  $j + 1$ , the following holds:

$$\mathit{regV}_{j+1} \sqsubseteq \mathit{regT}_{j+1} \text{ and } \mathit{opdV}_{j+1} \sqsubseteq \mathit{opdT}_{j+1}$$

It is sufficient to show that

$$\mathit{regJ} \cup \mathit{regR} \sqsubseteq \mathit{regT}_{j+1} \text{ and } \mathit{opdV}_i \sqsubseteq \mathit{opdT}_{j+1}.$$

Let  $x$  be a register that is modified by  $s$ . Then  $\mathit{regR}(x) = \mathit{regV}_i(x)$ . By T7 (b),  $\mathit{regT}_i(x) \preceq \mathit{regT}_{j+1}(x)$  and hence  $\mathit{regR}(x) \preceq \mathit{regT}_{j+1}(x)$ .

Let  $x$  be a register that is not modified by  $s$ . Then, by T7 (c), we have  $\mathit{regT}_j(x) \preceq \mathit{regT}_{j+1}(x)$  and hence  $\mathit{regV}_j(x) \preceq \mathit{regT}_{j+1}(x)$ . Assume that  $\mathit{regV}_j(x) = \mathbf{retAddr}(\ell)$  and that the subroutine  $s$  is not contained in  $\ell$ . Then  $\mathit{regJ}(x) = \mathbf{Undef}$  and  $\mathit{regT}_{j+1}(x)$  is either  $\mathbf{retAddr}(\ell)$  or  $\mathbf{Undef}$ . By T7 (e), the first case is not possible. Hence,  $\mathit{regJ}(x)$  and  $\mathit{regT}_{j+1}(x)$  are both  $\mathbf{Undef}$  and  $\mathit{regJ}(x) \preceq \mathit{regT}_{j+1}(x)$ .

By T7 (d),  $\mathit{opdT}_i \sqsubseteq \mathit{opdT}_{j+1}$  and hence  $\mathit{opdV}_i \sqsubseteq \mathit{opdT}_{j+1}$ .  $\square$

## 5. Java-to-VM compilation

Table I contains the syntax of a fragment of the Java language. The fragment has been chosen so that it comprises the features of the Java programming language that are critical for bytecode verification (except for object-initialization). A Java block has to satisfy the following conditions:

- Local variables and catch parameters cannot be re-declared in the scope of a declaration of the same identifier. (The scope of a local variable declaration consists of the statements in the block to the right of the declaration.)
- The same name for a label cannot be used in the scope of the label. (The scope of  $l$  in  $l: stm$  is  $stm$ .)
- A statement `break  $l$`  must be inside a statement with label  $l$ .

For the reader's convenience we summarize the semantics of the try-finally statement. The semantics of `try  $b$  finally  $s$`  is to execute  $s$  after  $b$ , no matter how control leaves  $b$ . That is,  $s$  is guaranteed to be executed whether  $b$  completes normally or abruptly (via a `break` or an exception). If  $s$  completes normally, then the result of `try  $b$  finally  $s$`  is the result of the execution of  $b$ . If  $s$  completes abruptly, then the result of `try  $b$  finally  $s$`  is the result of the execution of  $s$  whereby a possible reason for an abrupt termination of  $b$  is discarded.

### 5.1. CODE GENERATION FOR THE FRAGMENT OF JAVA

Two functions  $\mathcal{E}$  and  $\mathcal{S}$  for the compilation of expressions and statements are defined in Fig. 4. To improve readability, we suppress the details of a consistent assignment of VM register numbers  $\bar{x}$  to (occurrences of) Java variables  $x$  and of the generation of labels. We assume that the result of the compilation is an array of instructions and view the labels as indices of the code array. The function  $\mathcal{X}$  in Fig. 5 computes the exception table.

We do not consider problems that arise in the compilation of boolean expressions. The only exception is the while statement. If the test expression in a while statement is a constant expression with value `true`, then the while statement is compiled as a direct loop with a `Goto` instruction instead of the `Ifne` in Fig. 4. Otherwise, the end of the code for the while statement would be reachable from the beginning although the while statement cannot complete normally.

*Compilation of try-catch statements:* The try block is compiled as if the `try` were not present. If no exception is thrown during the execution

$\mathcal{E}(\text{var}) =$ Load( $\overline{\text{var}}$ )	$\mathcal{S}(\text{if } (exp) \text{ stm}_1 \text{ else } \text{stm}_2) =$ $\mathcal{E}(exp)$ Ifne( <i>if</i> ) $\mathcal{S}(\text{stm}_2)$ Goto( <i>end</i> ) if: $\mathcal{S}(\text{stm}_1)$ end:
$\mathcal{E}(\text{lit}) =$ Const( <i>lit</i> )	
$\mathcal{E}(exp_1 \text{ op } exp_2) =$ $\mathcal{E}(exp_1)$ $\mathcal{E}(exp_2)$ Prim( <i>op</i> )	$\mathcal{S}(\text{try block catch } (E x) \text{ stm}) =$ beg: $\mathcal{S}(\text{block})$ Goto( <i>end</i> ) hdl: Store( $\overline{x}$ ) $\mathcal{S}(\text{stm})$ end:
$\mathcal{S}(\text{type var};) = []$	
$\mathcal{S}(\text{var} = exp;) =$ $\mathcal{E}(exp)$ Store( $\overline{\text{var}}$ )	$\mathcal{S}(\text{try block finally } \text{stm}) =$ beg: $\mathcal{S}(\text{block})$ Jsr( <i>fin</i> ) Goto( <i>end</i> ) dft: Store( $\overline{exc}$ ) Jsr( <i>fin</i> ) Load( $\overline{exc}$ ) Throw fin: Store( $\overline{ret}$ ) $\mathcal{S}(\text{stm})$ Ret( $\overline{ret}$ ) end:
$\mathcal{S}(\{stm_1 \dots stm_n\}) =$ $\mathcal{S}(\text{stm}_1)$ : $\mathcal{S}(\text{stm}_n)$	
$\mathcal{S}(\text{while } (exp) \text{ stm}) =$ Goto( <i>tst</i> ) whl: $\mathcal{S}(\text{stm})$ tst: $\mathcal{E}(exp)$ Ifne( <i>whl</i> )	$\mathcal{S}(\text{break } lab;) =$ Jsr( <i>fin</i> <sub>1</sub> ) : Jsr( <i>fin</i> <sub><i>k</i></sub> ) Goto( <i>lab</i> ) <b>where</b> [ <i>fin</i> <sub>1</sub> , ..., <i>fin</i> <sub><i>k</i></sub> ] = <i>finallyUntil</i> ( <i>lab</i> )
$\mathcal{S}(lab: \text{stm}) =$ $\mathcal{S}(\text{stm})$ lab:	
$\mathcal{S}(\text{throw } exp;) =$ $\mathcal{E}(exp)$ Throw	

Figure 4. Compilation of the fragment of Java of Table I.

of the try block, then control jumps to the end of the try-catch statement. If an exception is thrown in the try block and it is compatible with the type of the parameter of the catch clause, then control is transferred to the handler for the exception. There, the exception is stored in the parameter of the catch clause and the code for the catch statement is executed. An exception table entry is created that protects the code interval of the try block with the code of the catch clause.

$\mathcal{X}(\text{try block catch } (E\ x)\ stm)$	$= \mathcal{X}(\text{block}) \cdot$ $(beg, hdl, E, hdl) \cdot \mathcal{X}(stm)$
$\mathcal{X}(\text{try block finally } stm)$	$= \mathcal{X}(\text{block}) \cdot$ $(beg, dfl, \text{Throwable}, dfl) \cdot \mathcal{X}(stm)$
$\mathcal{X}(\{stm_1 \dots stm_n\})$	$= \mathcal{X}(stm_1) \cdots \mathcal{X}(stm_n)$
$\mathcal{X}(\text{while } (exp)\ stm)$	$= \mathcal{X}(stm)$
$\mathcal{X}(\text{lab: } stm)$	$= \mathcal{X}(stm)$
$\mathcal{X}(\text{if } (exp)\ stm_1\ \text{else } stm_2)$	$= \mathcal{X}(stm_1) \cdot \mathcal{X}(stm_2)$
$\mathcal{X}(-)$	$= []$

Figure 5. Generation of exception tables.

*Compilation of try-finally statements:* After the code for the try block a call to the subroutine that implements the finally block is inserted. On the return from the subroutine control jumps to the end of the try-finally statement. The subroutine call works as follows: The `Jsr` instruction pushes the return address onto the operand stack before jumping to the subroutine. There, the return address is stored in a temporary register and the code of the finally block is executed. At the end of the finally block, the `Ret` instruction loads the return address from the temporary register and jumps back. A special default handler is inserted between the code for the try block and the code of the finally block. The default handler catches any exception that is thrown in the try block, stores it in a special register, jumps to the subroutine, and after return from the subroutine pushes the exception back onto the operand stack and throws it again.

*Compilation of break statements:* For the compilation of a break statement the list of finally blocks that have to be executed before control can break out is computed as follows. Starting at the break statement the computation goes upwards in the syntax tree. On each exit of a try block the corresponding finally block is appended to the list. The walk stops when the label of the break statement is reached. For each finally block in the list, a `Jsr` instruction to the index of the corresponding subroutine has to be inserted before the final `Goto` instruction that breaks out.

## 5.2. STATIC ANALYSIS OF JAVA PROGRAMS

The bytecode verifier has to ensure that the program counter is always a valid code index and that “execution never falls off the bottom of the code array” [11, §4.8.2]. This property of bytecode can only be guaranteed, if already on the Java source level it is ensured that each method is left via a return statement or an exception. Therefore, every

$\alpha$ <i>type var</i> ;	$normal(\alpha) \iff reachable(\alpha)$
$\alpha$ <i>var = exp</i> ;	$normal(\alpha) \iff reachable(\alpha)$
$\alpha \{ \beta_1 stm_1 \dots \beta_n stm_n \}$	$reachable(\beta_1) \iff reachable(\alpha)$ $reachable(\beta_{i+1}) \iff normal(\beta_i)$ $normal(\alpha) \iff normal(\beta_n)$
$\alpha$ <b>while</b> ( $\beta$ <i>exp</i> ) $\gamma$ <i>stm</i>	$reachable(\gamma) \iff reachable(\alpha)$ and $\beta$ <i>exp</i> is not a constant expression with value <b>false</b> $normal(\alpha) \iff reachable(\alpha)$ and $\beta$ <i>exp</i> is not a constant expression with value <b>true</b>
$\alpha$ <i>lab</i> : $\beta$ <i>stm</i>	$reachable(\beta) \iff reachable(\alpha)$ $normal(\alpha) \iff normal(\beta)$ or there exists a <i>reachable</i> statement <b>break lab</b> inside $\beta$ <i>stm</i> that can exit $\beta$ <i>stm</i> (Def. 19)
$\alpha$ <b>break lab</b> ;	$normal(\alpha) = \mathbf{false}$
$\alpha$ <b>throw exp</b> ;	$normal(\alpha) = \mathbf{false}$
$\alpha$ <b>if</b> ( <i>exp</i> ) $\beta$ <i>stm</i> <sub>1</sub> <b>else</b> $\gamma$ <i>stm</i> <sub>2</sub>	$reachable(\beta) \iff reachable(\alpha)$ $reachable(\gamma) \iff reachable(\alpha)$ $normal(\alpha) \iff normal(\beta) \vee normal(\gamma)$
$\alpha$ <b>try</b> $\beta$ <i>block</i> <b>catch</b> ( <i>E x</i> ) $\gamma$ <i>stm</i>	$reachable(\beta) \iff reachable(\alpha)$ $reachable(\gamma) \iff reachable(\alpha)$ and <i>block</i> can throw an exception <i>F</i> with $F \preceq E$ or $E \preceq F$ $normal(\alpha) \iff normal(\beta) \vee normal(\gamma)$
$\alpha$ ( <b>try</b> $\beta$ <i>block</i> <b>finally</b> $\gamma$ <i>stm</i> )	$reachable(\beta) \iff reachable(\alpha)$ $reachable(\gamma) \iff reachable(\alpha)$ $normal(\alpha) \iff normal(\beta) \wedge normal(\gamma)$

Figure 6. Reachability constraints for the fragment of Java of Table I.

Java compiler must carry out a conservative flow analysis to make sure that all statements are reachable and that method bodies cannot complete normally [5, §14.20]. For this purpose two predicates *reachable* and *normal* are computed at compile time. The predicates have the following intended meanings:

$reachable(\alpha) \iff$  The statement at position  $\alpha$  is reachable.

$normal(\alpha) \iff$  The statement at position  $\alpha$  can complete normally.

Initially, at the root position of a method body, the predicate *reachable* is true. For the other positions in the block the predicates *reachable* and *normal* can then be computed in a top-down manner. Instead of explaining exactly how the predicates are computed, we state the equivalences the predicates have to satisfy in Fig. 6.

The constraints for the labeled statement in Fig. 6 depend on the notion ‘an abruption can exit a statement’. We define this notion as follows<sup>1</sup> capturing the behavior of finally statements as explained above.

**DEFINITION 19** (Abruption can exit). An abruption at position  $\alpha$  can exit  ${}^\beta stm$ , if for every substatement **try**  $\gamma b$  **finally**  $\delta s$  of  ${}^\beta stm$  so that  $\alpha$  is in the try block  $\gamma b$  the predicate  $normal(\delta)$  is true for the finally statement. We say that a break statement at position  $\alpha$  can exit  $stm$ , if an abruption at position  $\alpha$  can exit  $stm$ .

The bytecode verifier checks at every **Load**( $x$ ) instruction whether the type of the register  $x$  is different from **Undef**. If this is the case, it can conclude that on each path leading to the **Load**( $x$ ) instruction there is at least one **Store**( $x$ ) instruction and thus  $x$  has a value which can be loaded onto the operand stack.

In order that the code generated by the Java-to-JVM compiler is accepted by the verifier, the compiler has to make a similar analysis on the source code level. It has to ensure that a local variable is definitely assigned when it is used in an expression. The static analysis is specified by *rules of definite assignment* [5, §16]. Since local variables are not initialized with default values, a Java program that does not obey the rules of definite assignment would not be type safe. For example, a local pointer variable which is not initialized could point to an undefined location on the heap.

In order to precisely specify all the cases of definite assignment, functions *before*, *after* and *vars* are computed at compile time. These functions assign sets of variables (identifiers) to each position in the body of a method. The functions have the following intended meanings:

$x \in before(\alpha) \iff$  The variable  $x$  is definitely assigned before the execution of the statement at position  $\alpha$ .

$x \in after(\alpha) \iff$  The variable  $x$  is definitely assigned after the statement at position  $\alpha$  when this statement completes normally.

$x \in vars(\alpha) \iff$  The position  $\alpha$  is in the scope of the local variable, formal parameter or catch parameter  $x$ .

At the root position  $\alpha$  of the body of a method  $m(t_1 x_1, \dots, t_n x_n)$  the initial condition is  $before(\alpha) = \{x_1, \dots, x_n\}$ , because when the body is invoked there are always values assigned to the formal parameters  $x_1, \dots, x_n$ . For the other statement positions in the body, the functions *before* and *after* can then be computed in a top-down manner. Instead

---

<sup>1</sup> The notion ‘can exit’ is not defined in the official JLS and compilers interpret it differently.

${}^\alpha \text{type } \text{var};$	$\text{after}(\alpha) = \text{before}(\alpha)$
${}^\alpha \text{var} = \text{exp};$	$\text{after}(\alpha) = \text{before}(\alpha) \cup \{\text{var}\}$
${}^\alpha \{\beta_1 \text{stm}_1 \dots \beta_n \text{stm}_n\}$	$\text{before}(\beta_1) = \text{before}(\alpha), \text{before}(\beta_{i+1}) = \text{after}(\beta_i)$ $\text{after}(\alpha) = \text{after}(\beta_n) \cap \text{vars}(\alpha)$
${}^\alpha \text{while } ({}^\beta \text{exp}) \ \gamma \text{stm}$	$\text{before}(\gamma) = \text{before}(\alpha), \text{after}(\alpha) = \text{before}(\alpha)$
${}^\alpha \text{lab}: {}^\beta \text{stm}$	$\text{before}(\beta) = \text{before}(\alpha)$ $\text{after}(\alpha) = \text{after}(\beta) \cap \text{break}(\beta, \text{lab})$
${}^\alpha \text{break } \text{lab};$	$\text{after}(\alpha) = \text{vars}(\alpha)$
${}^\alpha \text{throw } \text{exp};$	$\text{after}(\alpha) = \text{vars}(\alpha)$
${}^\alpha \text{if } (\text{exp}) \ \beta \text{stm}_1 \ \text{else } \ \gamma \text{stm}_2$	$\text{before}(\beta) = \text{before}(\alpha), \text{before}(\gamma) = \text{before}(\alpha)$ $\text{after}(\alpha) = \text{after}(\beta) \cap \text{after}(\gamma)$
${}^\alpha \text{try } \ \beta \text{block}$ $\quad \text{catch } (E \ x) \ \gamma \text{stm}$	$\text{before}(\beta) = \text{before}(\alpha), \text{before}(\gamma) = \text{before}(\alpha) \cup \{x\}$ $\text{after}(\alpha) = \text{after}(\beta) \cap \text{after}(\gamma)$
${}^\alpha (\text{try } \ \beta \text{block } \text{finally } \ \gamma \text{stm})$	$\text{before}(\beta) = \text{before}(\alpha), \text{before}(\gamma) = \text{before}(\alpha)$ $\text{after}(\alpha) = \text{after}(\gamma) \cup$ $\quad \{x \in \text{after}(\beta) \mid \text{there is no } x = \text{exp} \text{ in } \ \gamma \text{stm}\}$

Figure 7. The rules of definite assignment.

of explaining exactly how the functions are computed, we just state the equations the functions have to satisfy in Fig. 7. For a position  $\alpha$  in an expression we define  $\text{before}(\alpha) = \text{after}(\alpha) = \text{before}(\xi)$ , where  $\xi$  is the position of the nearest enclosing statement.

For a statement with label  $l$  at position  $\alpha$  the set  $\text{break}(\alpha, l)$  is defined as follows:

DEFINITION 20. A variable  $x$  belongs to  $\text{break}(\alpha, l)$ , if  $x \in \text{vars}(\alpha)$  and the following two conditions are true:

1.  $x$  is in  $\text{before}(\beta)$  for each statement  ${}^\beta \text{break } l$  inside the statement at position  $\alpha$  that can exit  $\alpha$  (Def. 19) and
2.  $x$  is in  $\text{after}(\beta)$  for each statement  ${}^\beta (\text{try } b \text{ finally } s)$  inside  $\alpha$  so that the try block  $b$  contains a  $\text{break } l$  that can exit  $\alpha$  and there is an assignment  $x = \text{exp}$  in the finally statement  $s$ .

The constraints in Fig. 7 differ from the rules of definite assignment in [5, §16] on two points. First, the official JLS defines for a try-finally statement:

$$\text{after}(\alpha) = \text{after}(\beta) \cup \text{after}(\gamma).$$

Our definition in Fig. 7 is more restrictive. A variable  $x$  in  $\text{after}(\beta)$  which is not in  $\text{after}(\gamma)$  belongs to  $\text{after}(\alpha)$  only if there is no assignment  $x = \text{exp}$  in  $\gamma \text{stm}$ .

Why do we restrict the set in this way?—We want to ensure that, if a variable is definitely assigned after a try-finally statement, then the verifier is able to infer that the variable has a type different from `Undef` at the end of the bytecode instructions for the statement. Suppose that the variable  $x$  is definitely assigned after the try block but not after the finally statement and that there exists an assignment  $x = exp$  in the finally statement. Then there will be a `Store(x)` instruction in the corresponding subroutine and  $x$  is considered to be modified by the subroutine. Since  $x$  is not definitely assigned after the try statement,  $x$  cannot be definitely assigned before the try-finally statement. Hence  $x$  will have the type `Undef` at the beginning of the subroutine (due to the default handler) and also at the end of the subroutine. Since  $x$  is modified by the subroutine, the type `Undef` will be propagated to the end of the code of the try-finally statement. [Note: This happens exactly in the program `Test1` of Sect. 1.]

The effect of our restriction of the rules of definite assignment for the try-finally statement is that a variable which is definitely assigned after the try-finally statement but not after the finally block is—in the eyes of the bytecode verifier—not modified by the subroutine that implements the finally block.

The second difference between Fig. 7 and the rules of definite assignment in the official JLS affects the labeled statement. The second clause of Def. 20 is not contained in the official JLS. Hence, according to the JLS a variable is definitely assigned after a labeled statement, if it is definitely assigned after the statement and before every `break` that can exit the labeled statement. Now suppose that a labeled statement contains a try-finally statement and inside the try-block there is a `break` statement. Suppose that there is a variable  $x$  that is definitely assigned before the `break` statement and also at the end of the labeled statement but not after the try-finally statement and that there is an assignment  $x = exp$  in the finally block. This means that  $x$  will have the type `Undef` at the end of the subroutine and is modified by the subroutine. Therefore, at the `Jsr` instruction before the `Goto` instruction that corresponds to the `break` statement, the type `Undef` is propagated to the `Goto` and further to the end of the labeled statement. Hence  $x$  will have the type `Undef` at the end of the code of the labeled statement although it is definitely assigned according to the official JLS. [Note: This happens exactly in the program `Test2` of Sect. 1.]

Both of our restrictions of the rules of definite assignments are chosen so that we can prove that the Java-to-JVM compiler generates verifiable code (Theorem 40).

## 5.3. CERTIFYING COMPILATION

We want to show that the code generated by the compiler from a method body that satisfied the reachability constraints in Fig. 6 and the rules of definite assignment in Fig. 7 is accepted by the verifier. For that purpose we extend the compiler so that it generates also type frames for the instructions. Hence, the result of the extended compilation is not only an array of bytecode instructions but an array of triples  $(instr, regT, opdT)$ , where  $(regT, opdT)$  is a type frame for the instruction. We then prove that the so generated type frames satisfy the conditions T1–T8 of Def. 11 for bytecode type assignments.

We call the extended compiler a *certifying* compiler, because the type frames it generates can be understood as a certificate that the generated bytecode behaves in a well-defined manner on the virtual machine (cf. [3]). What the certifying compiler does is similar to the *off-device pre-verification* in [19]. Our extended compiler, however, does not inline subroutines as it is done in CLDC (*Connected, Limited Device Configuration*).

The certifying compiler is defined in Fig. 8. The compilation function  $\mathcal{E}(\alpha exp, opdT)$  compiles the expression at position  $\alpha$  under the assumption that the operand stack contains values of type  $opdT$  and that  $\mathcal{T}(\alpha)$  is the compile time type of the expression. The compilation functions  $\mathcal{E}$  and  $\mathcal{S}$  use auxiliary functions  $\mathcal{A}$  and  $\mathcal{B}$  for the construction of the register types (the second components of the triples). These functions will be explained in detail below. In the compilation of a break statement the pseudo instruction **Break**( $i$ ) is used as an alternative name for **Goto**( $i$ ). The **Break**( $i$ ) simplifies the notations in the proofs below.

We assume that the generated code array for the given method body is the list of instructions

$$[code(0), \dots, code(n - 1)]$$

and the corresponding type frames generated by the extended compiler are

$$(regT_i, opdT_i)_{0 \leq i < n}.$$

We associate to each position  $\alpha$  in the body of the method a code interval

$$[code(i) \mid beg_\alpha \leq i < end_\alpha]$$

where the index  $beg_\alpha$  is the index of the first instruction which belongs to the compiled code for the phrase (expression or statement) at position  $\alpha$  and the index  $end_\alpha$  is the index of the first instruction immediately following the code for the phrase at position  $\alpha$ . If the

$\mathcal{E}(\alpha \text{ var}, \text{opdT}) =$ Load( $\overline{\text{var}}$ ), $\mathcal{A}(\alpha)$ , $\text{opdT}$	$S(\alpha \text{ if } (\text{exp})^\beta \text{ stm}_1 \text{ else } \gamma \text{ stm}_2) =$ $\mathcal{E}(\text{exp}, [])$ Ifne( $\text{if}$ ), $\mathcal{B}(\beta)$ , [] $\mathcal{S}(\text{stm}_2)$ Goto( $\text{end}$ ), $\mathcal{A}(\gamma)$ , [] if: $\mathcal{S}(\text{stm}_1)$ end:
$\mathcal{E}(\alpha \text{ lit}, \text{opdT}) =$ Const( $\text{lit}$ ), $\mathcal{A}(\alpha)$ , $\text{opdT}$	$S(\alpha (\text{try }^\beta \text{ block catch } (E x)^\gamma \text{ stm})) =$ beg: $\mathcal{S}(\text{block})$ Goto( $\text{end}$ ), $\mathcal{A}(\beta)$ , [] hdl: Store( $\overline{x}$ ), $\mathcal{B}(\beta)$ , [ $E$ ] $\mathcal{S}(\text{stm})$ end:
$\mathcal{E}(\alpha^\beta \text{ exp}_1 \text{ op } \gamma \text{ exp}_2), \text{opdT}) =$ $\mathcal{E}(\text{exp}_1, \text{opdT})$ $\mathcal{E}(\text{exp}_2, \text{opdT} \cdot [\mathcal{T}(\beta)])$ Prim( $\text{op}$ ), $\mathcal{A}(\alpha)$ , $\text{opdT} \cdot [\mathcal{T}(\beta), \mathcal{T}(\gamma)]$	$S(\alpha \text{ try }^\beta \text{ block finally } \gamma \text{ stm}) =$ beg $_\alpha$ : $\mathcal{S}(\text{block})$ end $_\beta$ : JsR( $\text{fin}_\alpha$ ), $\mathcal{A}(\beta)$ , [] Goto( $\text{end}_\alpha$ ), $\mathcal{A}(\alpha)$ , [] dfl $_\alpha$ : Store( $\overline{e}$ ), $\mathcal{B}(\beta)$ , [Throw] JsR( $\text{fin}_\alpha$ ), $\mathcal{B}(\beta) \oplus \{(\overline{e}, \text{Throw})\}$ , [] Load( $\overline{e}$ ), $\mathcal{B}(\beta) \oplus \{(\overline{e}, \text{Throw})\}$ , [] Throw, $\mathcal{B}(\beta)$ , [Throw] fin $_\alpha$ : Store( $\overline{\text{ret}}$ ), $\mathcal{B}(\alpha)$ , [retAddr( $\text{fin}_\alpha$ )] $\mathcal{S}(\text{stm})$ end $_\gamma$ : Ret( $\overline{\text{ret}}$ ), $\mathcal{A}(\gamma)$ , [] end $_\alpha$ :
$\mathcal{S}(\text{type var};) = []$	$S(\alpha \text{ break lab};) =$ JsR( $\text{fin}_{\beta_1}$ ), $\text{regB} \oplus \mathcal{B}(\beta_1)$ , [] : JsR( $\text{fin}_{\beta_k}$ ), $\text{regB} \oplus \mathcal{B}(\beta_k)$ , [] br: Break( $\text{lab}$ ), $\text{regB}$ , [] where $^\gamma \text{ lab: stm}$ encloses $\alpha$ and $[\beta_1, \dots, \beta_k] = \text{finallyCode}(\alpha, \text{lab})$ and $\text{regB} = \text{if } br \in \mathcal{D} \text{ then } \mathcal{A}(\gamma) \text{ else } \emptyset$
$\mathcal{S}(\alpha \text{ var} = \text{exp};) =$ $\mathcal{E}(\text{exp}, [])$ Store( $\overline{\text{var}}$ ), $\mathcal{A}(\alpha)$ , [ $\mathcal{T}(\alpha)$ ]	
$\mathcal{S}(\{\text{stm}_1 \dots \text{stm}_n\}) =$ $\mathcal{S}(\text{stm}_1)$ : $\mathcal{S}(\text{stm}_n)$	
$\mathcal{S}(\alpha \text{ while } (\text{exp}) \text{ stm}) =$ Goto( $\text{tst}$ ), $\mathcal{B}(\alpha)$ , [] whl: $\mathcal{S}(\text{stm})$ tst: $\mathcal{E}(\text{exp}, [])$ Ifne( $\text{whl}$ ), $\mathcal{B}(\alpha)$ , [int]	
$\mathcal{S}(\text{lab: stm}) =$ $\mathcal{S}(\text{stm})$ lab:	
$\mathcal{S}(\alpha \text{ throw exp};) =$ $\mathcal{E}(\text{exp}, [])$ Throw, $\mathcal{A}(\alpha)$ , [Throwable]	

Figure 8. Certifying compilation.

statement at position  $\alpha$  is not empty, then  $\text{beg}_\alpha$  is less than  $\text{end}_\alpha$ , otherwise it is equal to  $\text{end}_\alpha$ .

What is the domain  $\mathcal{D}$  of the bytecode type assignment generated by the extended compiler?—It cannot be the set of all code indices, as the following example shows.

EXAMPLE 21. In the following code, the index of the `Goto(B)` instruction cannot belong to the domain of the bytecode type assignment, because otherwise the variable `i` would have the type `Undef` at label `B`.

void m(E x,boolean b) {	Load(b)	<i>fin</i> : Store( <i>r</i> )
int i;	Ifne( <i>if</i> )	Load( <i>x</i> )
if (b) { i = 2; }	A: Jsr( <i>fin</i> )	Throw
else {	Goto(B)	Ret( <i>r</i> )
try { }	<i>dfl</i> : Store( <i>e</i> )	<i>if</i> : Const(2)
finally { throw x; }	Jsr( <i>fin</i> )	Store( <i>i</i> )
}	Load( <i>e</i> )	B: Load( <i>i</i> )
int j = i;	Throw	Store( <i>j</i> )
}		

Therefore, we define  $\mathcal{D}$  as follows:

DEFINITION 22 (Domain of type assignment). The domain  $\mathcal{D}$  for the type assignment generated by the certifying compiler is the least (w.r.t. set inclusion) set of natural numbers with the following properties:

1.  $0 \in \mathcal{D}$ .
2. If  $i \in \mathcal{D}$  and  $code(i)$  is not a control transfer instruction, then  $i + 1 \in \mathcal{D}$ .
3. If  $i \in \mathcal{D}$  and  $code(i) = \text{Goto}(j)$ , then  $j \in \mathcal{D}$ .
4. If  $i \in \mathcal{D}$  and  $code(i) = \text{Ifne}(j)$ , then  $i + 1 \in \mathcal{D}$  and  $j \in \mathcal{D}$ .
5. If  $i \in \mathcal{D}$  and  $code(i) = \text{Jsr}(j)$ , then  $j \in \mathcal{D}$ .
6. If  $i \in \mathcal{D}$ ,  $(f, u, -, h) \in excs$ ,  $f \leq i < u$  and  $code(i)$  is neither `Jsr` nor `Goto`, then  $h \in \mathcal{D}$ .
7. If  $i \in \mathcal{D}$ ,  $code(i) = \text{Jsr}(fin_\alpha)$  and  ${}^\alpha(\text{try } {}^\beta b \text{ finally } {}^\gamma s)$  is a statement with  $end_\gamma \in \mathcal{D}$ , then  $i + 1 \in \mathcal{D}$ .

Condition 7 says that the code index immediately following a `Jsr` instruction belongs to  $\mathcal{D}$ , if the index of the `Ret` instruction at the end of the code for the corresponding finally block belongs to  $\mathcal{D}$ .

The certifying compiler computes the types for the local registers based on the rules of definite assignment using the functions  $\mathcal{A}$  and  $\mathcal{B}$  which are defined as follows:

DEFINITION 23. For a subset  $S$  of  $vars(\alpha)$  we denote by  $\mathcal{R}(\alpha, S)$  the function that assigns the declared types to the (register numbers of) variables in the set  $S$ , the return address types to registers that contain return addresses and the type `Undef` to other registers.

1. If  $x$  is in  $S$  of declared type  $t$ , then  $(\bar{x}, t)$  belongs to  $\mathcal{R}(\alpha, S)$ .
2. If  $\alpha$  is a position in the finally block of  ${}^\beta(\text{try } \gamma b \text{ finally } {}^\delta s)$  with  $\text{end}_\delta \in \mathcal{D}$ , then  $(\overline{\text{ret}}_\beta, \text{retAddr}(\text{fin}_\beta))$  belongs to  $\mathcal{R}(\alpha, S)$ . [Note: We assume that  $\overline{\text{ret}}_\beta$  is different from  $\bar{x}$  for all  $x \in \text{vars}(\beta)$  and different from  $\overline{\text{ret}}_\xi$ , if the position  $\beta$  is in a finally block of a try-finally statement at position  $\xi$ .]
3. For every other register  $i$ ,  $(i, \text{Undef})$  belongs to  $\mathcal{R}(\alpha, S)$ .

The functions  $\mathcal{A}$  and  $\mathcal{B}$  are defined as follows:

$$\mathcal{A}(\alpha) = \mathcal{R}(\alpha, \text{after}(\alpha)) \quad \mathcal{B}(\alpha) = \mathcal{R}(\alpha, \text{before}(\alpha))$$

In the rest of this section we prove a sequence of technical lemmas about the code generated by the compiler in Fig. 4 so that at the end, in Theorem 40, we can show that the type frames  $(\text{reg}T_i, \text{opdT}_i)_{i \in \mathcal{D}}$  generated by the extended compiler in Fig. 8 satisfy the conditions T1–T8.

We start with some simple properties of the generated code. The first observation is that the nesting of expressions and statements is preserved by the compiler. If a phrase is a subphrase of another phrase, then the code interval of the phrase is contained in the code interval of the enclosing phrase.

LEMMA 24. *If  $\beta$  is a position inside  ${}^\alpha$  phrase, then  $\text{beg}_\alpha \leq \text{beg}_\beta \leq \text{end}_\beta \leq \text{end}_\alpha$ .*

*Proof.* By induction on the size of  ${}^\alpha$  phrase.  $\square$

Since condition 7 in Def. 22 is more restrictive than the corresponding definition for the successor indices of a `Jsr` instruction in Def. 5, it is obvious that  $\mathcal{D}$  is a subset of the reachable code indices of the method.

LEMMA 25. *If  $i \in \mathcal{D}$ , then  $i$  is reachable from code index 0 via a path with elements from  $\mathcal{D}$ .*

*Proof.* By induction on the generation of the set  $\mathcal{D}$ .  $\square$

The code generated by the compiler has the property that the  $j + 1$  successor of a `Jsr` instruction at index  $j$  cannot be reached on a path avoiding index  $j$ , for example via a `Goto(j + 1)`. Therefore, if  $j + 1$  belongs to  $\mathcal{D}$ , then the `Ret` instruction of the corresponding subroutine belongs to  $\mathcal{D}$ , too. To prove this fact one needs the following lemma.

LEMMA 26. *If  $code(j) = \mathbf{Jsr}(-)$  and  $j + 1$  is a successor index of  $i$ , then  $i = j$ .*

*Proof.* The following facts are used:

1. The code for expressions is free of  $\mathbf{Jsr}$ .
2. The code for statements does not end in a  $\mathbf{Jsr}$  instruction.
3. The code for statements does not start after a  $\mathbf{Jsr}$  instruction.
4. Targets  $i$  of  $\mathbf{Goto}(i)$  and  $\mathbf{Ifne}(i)$  are not after a  $\mathbf{Jsr}$ .
5. No exception handler starts after a  $\mathbf{Jsr}$  instruction.
6. No subroutine starts after a  $\mathbf{Jsr}$  instruction.

Therefore, if  $code(j) = \mathbf{Jsr}(-)$ , then the only possible way to reach  $j + 1$  is via  $j$ .  $\square$

In the compiled code of an expression, the end of the code is reachable from the beginning of the code.

LEMMA 27. *In  $\mathcal{E}(\alpha exp)$ , the code index  $end_\alpha$  is reachable from  $beg_\alpha$  in the code interval for  $\alpha$ .*

*Proof.* By induction on the size of  $\alpha exp$ .  $\square$

Every successor of an instruction in the bytecode for a phrase is either an instruction in the code for the phrase, or the next instruction immediately following the code for the phrase, or the target of an exception handler which protects the instruction, or a jump to a subroutine of an enclosing try-finally statement, or a jump to the end of an enclosing labeled statement.

LEMMA 28. *If  $beg_\alpha \leq i < end_\alpha$  in  $\mathcal{E}(\alpha exp)$  and  $j$  is a successor index of  $i$ , then one of the following is true:*

1.  $beg_\alpha \leq j \leq end_\alpha$ , or
2. there exists a handler  $(f, u, -, j) \in excs$  with  $f \leq i < u$ .

*Proof.* By induction on the size of  $\alpha exp$ .  $\square$

LEMMA 29. *If  $beg_\alpha \leq i < end_\alpha$  in  $\mathcal{S}(\alpha stm)$  and  $j$  is a successor index of  $i$ , then one of the following is true:*

1.  $beg_\alpha \leq j \leq end_\alpha$ , or
2. there exists a handler  $(f, u, -, j) \in excs$  with  $f \leq i < u$ , or
3.  $code(i) = \mathbf{Jsr}(j)$  and there exists a statement  ${}^\beta(\mathbf{try} \ \gamma s \ \mathbf{finally} \ \delta b)$  so that  $\alpha$  is a position in  $\gamma s$  and  $j = fn_\beta$ , or
4.  $code(i) = \mathbf{Break}(j)$  and there exists a statement  $l: {}^\beta stm$  so that  $\alpha$  is a position in  ${}^\beta stm$  and  $j = end_\beta$ .

*Proof.* By induction on the size of  ${}^\alpha stm$ .  $\square$

The code for a phrase has exactly one entry point, the first instruction of the code for the phrase. Each path entering the code must go through this entry point. This fact is a consequence of the following lemma.

LEMMA 30. *If  $j$  is a successor index of  $i$  with  $beg_\alpha < j < end_\alpha$ , then  $beg_\alpha \leq i < end_\alpha$ .*

*Proof.* Top down, starting at the root position of the body of the method.  $\square$

Lemma 25 can be strengthened. If a code index in the code interval of a phrase belongs to  $\mathcal{D}$ , then the code index is reachable from the beginning of the interval via a path in  $\mathcal{D}$ . It follows that the first instruction of the code for the phrase belongs to  $\mathcal{D}$ , too.

LEMMA 31. *If  $i \in \mathcal{D}$  and  $beg_\alpha \leq i < end_\alpha$ , then  $i$  is reachable from  $beg_\alpha$  via a path in  $\{j \mid beg_\alpha \leq j < end_\alpha\} \cap \mathcal{D}$ .*

*Proof.* By Lemma 25, there exists a path from code index 0 to  $i$  with elements from  $\mathcal{D}$ . If  $beg_\alpha = i$ , we are done. Otherwise, consider the last element on the path which is not in  $\{j \mid beg_\alpha < j < end_\alpha\} \cap \mathcal{D}$ . By Lemma 30, it follows that this last element must be  $beg_\alpha$ .  $\square$

If the **Ret** instruction at the end of the code of a finally block belongs to  $\mathcal{D}$ , then the instruction is reachable from the beginning of the block. Hence, the **Ret**( $\bar{r}$ ) instruction is a possible return from the subroutine in the sense of Def. 8, since in the code of the finally block no **Store**( $\bar{r}$ ) is used.

LEMMA 32. *If  $end_\gamma \in \mathcal{D}$  in  ${}^\alpha(\mathbf{try} \ \beta b \ \mathbf{finally} \ \gamma s)$ , then  $end_\gamma$  is reachable from  $beg_\gamma$  via a path in  $\{i \mid beg_\gamma \leq i \leq end_\gamma\} \cap \mathcal{D}$ .*

*Proof.* If  $end_\gamma$  is in  $\mathcal{D}$ , then by Lemma 31 there exists a path from  $beg_\alpha$  to  $end_\gamma$  in  $\{j \mid beg_\alpha \leq j < end_\alpha\} \cap \mathcal{D}$ . Since the path has to leave the code of  ${}^\beta block$ , it must reach  $fin_\alpha$  via  $Jsr(fin_\alpha)$  and hence reaches also  $beg_\gamma$ . The end piece of the path is in  $\{j \mid beg_\gamma \leq j \leq end_\gamma\} \cap \mathcal{D}$  and connects  $beg_\gamma$  with  $end_\gamma$ .  $\square$

**COROLLARY 33.** *If  $end_\gamma \in \mathcal{D}$  in  ${}^\alpha(\text{try } {}^\beta b \text{ finally } \gamma s)$ , then  $end_\gamma$  is a possible return from subroutine  $fin_\alpha$ .*

**EXAMPLE 34.** In the following code,  $end_\alpha$  is reachable from  $beg_\alpha$ , but the predicate  $normal(\alpha)$  is false according to Fig. 6.

```

L:  ${}^\alpha$ try {                 $beg_\alpha$  : Jsr( $fin$ )           $fin$ : Store( $r$ )
      break L;              Goto( $end_\alpha$ )          Load( $b$ )
    } finally {             $dfl$ : Store( $e$ )          Ifne( $end_\alpha$ )
      if (b) break L;      Jsr( $fin$ )              Ret( $r$ )
    }                       Load( $e$ )                 $end_\alpha$ :
  }                          Throw

```

Hence, in Lemma 36 below, we need a stronger assumption than simple reachability. We assume that  $end_\alpha$  is reachable from  $beg_\alpha$  via a path which is entirely in the set  $\mathcal{D}$  and which is *not critical* for  $\alpha$ .

**DEFINITION 35 (Critical path).** A path from  $beg_\alpha$  to  $end_\alpha$  is a critical path for  ${}^\alpha stm$ , if the last step in the path is a **Break**( $l$ ), the position  $\alpha$  is in the scope of the label  $l$  and  $l = end_\alpha$ .

Note, that the path in Lemma 32 which connects the beginning of a subroutine with the end is not critical for the finally block.

**LEMMA 36.** *Let  ${}^\alpha stm$  be a statement.*

1. *If code index  $end_\alpha$  is reachable from code index  $beg_\alpha$  via a path in the set  $\{j \mid beg_\alpha \leq j \leq end_\alpha\} \cap \mathcal{D}$  which is not critical for  ${}^\alpha stm$ , then  $normal(\alpha)$  is true.*
2. *If  $beg_\alpha \leq i < end_\alpha$ ,  $i \in \mathcal{D}$ ,  $code(i) = \text{Break}(l)$  and  $end_\alpha \leq l$ , then the corresponding **break**  $l$  can exit  $\alpha$ .*

*Proof.* By induction on the size of  ${}^\alpha stm$ . We consider the case of a statement  ${}^\alpha(\text{try } {}^\beta block \text{ finally } \gamma stm)$ .

Consider a path from  $beg_\alpha$  to  $end_\alpha$  in  $\{j \mid beg_\alpha \leq j \leq end_\alpha\} \cap \mathcal{D}$  which is not critical for  $\alpha$ . We can assume that the path does not loop back to  $beg_\alpha$ . Otherwise, we take an end piece of the path.

By Lemma 29, the different possibilities for the path to leave the code of  ${}^\beta block$  are via  $end_\beta$ ,  $dfl_\alpha$ ,  $Jsr(fin_\alpha)$ , or  $Break(end_\alpha)$ .

If the path leaves  ${}^\beta block$  via  $Break(end_\alpha)$ , then it would be critical for  $\alpha$ .

Suppose that the path reaches  $fin_\alpha$ . From  $fin_\alpha$  the path proceeds to  $beg_\gamma$ . Then, again by Lemma 29, it can leave the code of  ${}^\gamma block$  via  $end_\gamma$  or  $Break(end_\alpha)$ . If the path reaches  $end_\gamma$ , then it gets stuck at the following **Ret** instruction contradicting the assumption that it reaches  $end_\alpha$ . If the path leaves the code of  ${}^\gamma block$  via  $Break(end_\alpha)$ , then the path would be critical for  $\alpha$ . Hence, the path cannot reach  $fin_\alpha$ .

Suppose that the path reaches  $dfl_\alpha$ . Then the path must reach  $fin_\alpha$  because otherwise it gets stuck at the **Throw** instruction.

The only possibility that remains for leaving the code of  ${}^\beta block$  is via  $end_\beta$ . The part of the path from  $beg_\beta$  to  $end_\beta$  is not critical for  $\beta$ , since a  $Break(end_\delta)$  at the end of a critical path would jump below  $end_\alpha$ . Hence we can apply the induction hypothesis 1 to  $\beta$  and obtain that  $normal(\beta)$  is true.

At the instruction  $Jsr(fin_\alpha)$  at  $end_\beta$  the path proceeds to  $Goto(end_\alpha)$ . Otherwise, it would go through  $fin_\alpha$ . Since the path is in  $\mathcal{D}$ , the instruction  $Goto(end_\alpha)$  belongs to  $\mathcal{D}$ . By Def. 22 and Lemma 26 it follows that  $end_\gamma$  is in  $\mathcal{D}$ .

Since  $end_\gamma$  is in  $\mathcal{D}$ , by Lemma 32 there exists a path from  $beg_\gamma$  to  $end_\gamma$  in  $\{j \mid beg_\gamma \leq j \leq end_\gamma\} \cap \mathcal{D}$  which is not critical for  $\gamma$ . Hence we can apply the induction hypothesis 1 to  ${}^\gamma block$  and obtain that  $normal(\gamma)$  is true.

From Fig. 6 it follows that  $normal(\alpha)$  is true.

For the second part of the lemma, the difficult case is a  $Break(l)$  instruction in the code of  ${}^\beta block$ . By the induction hypothesis 2 for  $\beta$ , it follows that the corresponding **break**  $l$  statement can exit  $\beta$ . In order to conclude that it can also exit  $\alpha$ , we have to show that  $normal(\gamma)$  is true. According to Fig. 8 the  $Break(l)$  is preceded by a cascade of **Jsr** instructions. One of them is  $Jsr(fin_\alpha)$ . Since the  $Break(end_\delta)$  is in  $\mathcal{D}$ , by Def. 22 and Lemma 26 it follows that  $end_\gamma$  is in  $\mathcal{D}$ . As in the proof of the first part of the lemma it follows that  $normal(\gamma)$  is true. Hence, the **break**  $l$  statement can exit  $\alpha$ .  $\square$

In Def. 10, the set of register numbers modified by a subroutine is defined. We extend the notion and define the set of variables modified by an arbitrary subphrase of the body of the method.

**DEFINITION 37.** *A variable  $x$  belongs to  $mod(\alpha)$ , if there exists an  $i$  with  $beg_\alpha \leq i < end_\alpha$  and  $code(i) = Store(\bar{x})$ .*

For subroutines the two definitions are related in the following sense.

LEMMA 38. *Let  ${}^\alpha(\text{try } {}^\beta \text{finally } {}^\gamma s)$  be a statement with  $end_\gamma \in \mathcal{D}$ . If  $x \in \text{mod}(\gamma)$ , then  $\bar{x}$  is modified by  $fin_\alpha$ .*

*Proof.* Let  $x \in \text{mod}(\gamma)$ . This means that there exists a code index  $i$  with  $beg_\gamma \leq i < end_\gamma$  and  $code(i) = \text{Store}(\bar{x})$ . By Corollary 33, the code index  $end_\gamma$  is a possible return from subroutine  $fin_\alpha$ . Hence  $\bar{x}$  is modified by the subroutine  $fin_\alpha$  according to Def. 10.  $\square$

An obvious conjecture is that variables which are definitely assigned after a statement but not before the statement occur in **Store** instructions in the code interval for the statement. In general, however, the conjecture is false. For example, in the code for a throw statement there are no **Store** instructions, although every variable is considered to be definitely assigned after the throw statement (Fig. 7). Even if  $end_\alpha$  is reachable from  $beg_\alpha$ , not every variable which is definitely assigned after the statement but not before the statement is used in a **Store** instruction, since a **Break** may jump to  $end_\alpha$  from almost any position in the statement.

LEMMA 39. *Let  ${}^\alpha stm$  be a statement.*

1. *If code index  $end_\alpha$  is reachable from code index  $beg_\alpha$  via a path in the set  $\{j \mid beg_\alpha \leq j \leq end_\alpha\} \cap \mathcal{D}$  which is not critical for  ${}^\alpha stm$ , then  $after(\alpha) \setminus before(\alpha) \subseteq \text{mod}(\alpha)$ .*
2. *If  $beg_\alpha \leq i < end_\alpha$ ,  $i \in \mathcal{D}$ ,  $code(i) = \text{Break}(l)$  and  $\alpha$  is in the scope of the label  $l$ , then  $break(\alpha, l) \setminus before(\alpha) \subseteq \text{mod}(\alpha)$ .*

*Proof.* By induction on the size of  ${}^\alpha stm$ . We consider the case of a finally statement  ${}^\alpha(\text{try } {}^\beta \text{block finally } {}^\gamma stm)$ .

According to Fig. 7,  $before(\alpha) = before(\beta) = before(\gamma)$  and

$$after(\alpha) \setminus before(\alpha) \subseteq (after(\beta) \setminus before(\beta)) \cup (after(\gamma) \setminus before(\gamma)).$$

Consider a path from  $beg_\alpha$  to  $end_\alpha$  in  $\{j \mid beg_\alpha \leq j \leq end_\alpha\} \cap \mathcal{D}$  which is not critical for  $\alpha$ . As in the proof of Lemma 36 it follows that the first part of the path goes from  $beg_\beta$  to  $end_\beta$  and is not critical for  $\beta$ . Hence we can apply the induction hypothesis 1 to  $\beta$  and obtain that

$$after(\beta) \setminus before(\beta) \subseteq \text{mod}(\beta) \subseteq \text{mod}(\alpha).$$

Following further the proof of Lemma 36 we see that  $end_\gamma$  is in  $\mathcal{D}$  and that there exists a path from code index  $beg_\gamma$  to code index  $end_\gamma$  in

$\{j \mid \text{beg}_\gamma \leq j \leq \text{end}_\gamma\} \cap \mathcal{D}$  which is not critical for  $\gamma$ . Hence we can apply the induction hypothesis 1 to  ${}^\gamma \text{stm}$  and obtain that

$$\text{after}(\gamma) \setminus \text{before}(\gamma) \subseteq \text{mod}(\gamma) \subseteq \text{mod}(\alpha).$$

Hence, statement 1 of the lemma is shown for  $\alpha$ .

Statement 2 is proved in a similar way.  $\square$

In the main theorem we prove that the type frames computed by the extended compiler restricted to the domain  $\mathcal{D}$  of Def. 22 is a bytecode type assignment in the sense of Def. 11.

**THEOREM 40** (Completeness of the compiler).

*The family  $(\text{reg}T_i, \text{opdT}_i)_{i \in \mathcal{D}}$  of type frames generated by the certifying compiler for the body of the method is a bytecode type assignment.*

*Proof.* We have to show that conditions T1–T8 of Def. 11 are satisfied.

T1 says that  $\mathcal{D}$  must be a set of valid code indices. So we have to show that, if  $i \in \mathcal{D}$ , then  $0 \leq i < \text{length}(\text{code})$ . Obviously, each index  $i \in \mathcal{D}$  is less than or equal to  $\text{length}(\text{code})$ . Suppose that  $\text{length}(\text{code})$  belongs to  $\mathcal{D}$ . By Lemma 25, it follows that  $\text{length}(\text{code})$  is reachable from code index 0 via a path in  $\mathcal{D}$ . Since the body of the method is not contained in an enclosing labeled statement, the path is not critical and Lemma 36 yields that the body of the method is *normal*. This is a contradiction, since the body of a method is not allowed to be *normal*. Hence  $\text{length}(\text{code}) \notin \mathcal{D}$  and T1 is satisfied.

T2 is satisfied, since 0 belongs to  $\mathcal{D}$  according to Def. 22.

Next we prove that the triples generated by the certifying compiler for expressions and statements have the following properties.

If  $\mathcal{E}(\alpha \text{ exp}, \text{opdT}) = [\text{code}(i) \mid m \leq i < n]$ , then the following is true:

E1.  $\mathcal{B}(\alpha) \sqsubseteq \text{reg}T_m$  and  $\text{opdT} = \text{opdT}_m$ .

E2. If  $m \leq i < n$  and  $i \in \mathcal{D}$ , then  $\text{CHECK}(i, \text{reg}T_i, \text{opdT}_i)$  is true.

E3. If  $m \leq i < n$ ,  $i \in \mathcal{D}$  and  $(j, \text{reg}S, \text{opdS}) \in \text{SUCC}(i, \text{reg}T_i, \text{opdT}_i)$ , then  $j \in \mathcal{D}$  and one of the following is true:

- a)  $m \leq j < n$ ,  $\text{reg}S \sqsubseteq \text{reg}T_j$  and  $\text{opdS} \sqsubseteq \text{opdT}_j$ , or
- b)  $j = n$ ,  $\text{reg}S \sqsubseteq \mathcal{A}(\alpha)$  and  $\text{opdS} \sqsubseteq \text{opdT} \cdot \mathcal{T}(\alpha)$ , or
- c) there exists an entry  $(f, u, t, j)$  in *excs* such that  $f \leq i < u$ ,  $\text{reg}S \sqsubseteq \mathcal{B}(\alpha)$  and  $\text{opdS} = [t]$ .

E4. If  $m \leq i < n$ ,  $i \in \mathcal{D}$  and  $regT_i(x) = \mathbf{retAddr}(s)$ , then  $i$  belongs to the subroutine  $s$ .

For  $\mathcal{S}(\alpha stm) = [code(i) \mid m \leq i < n]$  the following is true:

- S1. If  $m \leq n$ , then  $\mathcal{B}(\alpha) \sqsubseteq regT_m$  and  $opdT_m = []$ .
- S2. If  $m \leq i < n$  and  $i \in \mathcal{D}$ , then  $\mathbf{CHECK}(i, regT_i, opdT_i)$  is true.
- S3. If  $m \leq i < n$ ,  $i \in \mathcal{D}$  and  $(j, regS, opdS) \in \mathbf{SUCC}(i, regT_i, opdT_i)$ , then  $j \in \mathcal{D}$  and one of the following is true:
  - a)  $m \leq j < n$ ,  $regS \sqsubseteq regT_j$  and  $opdS \sqsubseteq opdT_j$ , or
  - b)  $j = n$ ,  $regS \sqsubseteq \mathcal{A}(\alpha)$  and  $opdS = []$ , or
  - c) there exists an entry  $(f, u, t, j)$  in  $excS \setminus \mathcal{X}(\alpha)$  so that  $f \leq i < u$ ,  $regS \sqsubseteq \mathcal{B}(\alpha)$  and  $opdS = [t]$ , or
  - d) there exists a statement  ${}^\beta(lab: s)$  such that  $\alpha$  is in  $s$ ,  $j = lab$ ,  $regS \sqsubseteq \mathcal{A}(\beta)$  and  $opdS = []$ , or
  - e) there exists a statement  ${}^\beta(\mathbf{try } b \mathbf{ finally } s)$  so that  $\alpha$  is in  $b$ ,  $j = \mathit{fin}_\beta$ ,  $regS \sqsubseteq \mathcal{B}(\beta)$  and  $opdS = [\mathbf{retAddr}(\mathit{fin}_\beta)]$ .
- S4. If  $m \leq i < n$ ,  $i \in \mathcal{D}$  and  $regT_i(x) = \mathbf{retAddr}(s)$ , then  $i$  belongs to the subroutine  $s$ .

Properties E1–E4 and S1–S4 are proved by induction on the size of  $\alpha exp$  and  $\alpha stm$  using the extended compiler in Fig. 8. [Note: S3 (c) would not be true for break statements if `Jsr` and `Goto` were included in `ALLHANDLERS` in Fig. 2.]

Consider now the body of the method with

$$\mathcal{S}(body) = [code(i) \mid 0 \leq i < \mathit{length}(code)].$$

The properties S1–S3 are true for the body.

From S1 we obtain for the root position  $\alpha$  that  $\mathcal{B}(\alpha) \sqsubseteq regT_0$  and  $opdT_0 = []$ , hence T3 and T4 are satisfied.

From S2 it follows that the checks are true for every code index  $i \in \mathcal{D}$ , hence T5 is satisfied.

Condition T6 follows from S3, since the cases (b)–(e) are not possible for a successor  $j$  of an index  $i \in \mathcal{D}$ . Case (b) is not possible, since  $\mathit{length}(code) \notin \mathcal{D}$  as we have seen above. Case (c) is not possible, since the exception table  $excS$  is defined to be  $\mathcal{X}(body)$ . Cases (d) and (e) are not possible, since  $body$  cannot be contained in another statement in the body of the method.

The first part of condition T8 follows from S4. The second part of condition T8 follows directly from Fig. 8.

It remains to show that (a)–(e) of T7 are satisfied.

Assume that  $i \in \mathcal{D}$ ,  $code(i) = \mathbf{Ret}(x)$ ,  $regT_i(x) = \mathbf{retAddr}(s)$ ,  $j$  is in  $\mathcal{D}$  and  $code(j) = \mathbf{Jsr}(s)$ . In Fig. 8 we see that  $\mathbf{Ret}$  instructions are used only for try-finally statements, hence there exists a position  $\alpha$  so that  $x = \overline{ret}_\alpha$  and  $s = fin_\alpha$ . Since  $i$  and  $j$  are in  $\mathcal{D}$ , by Def. 22 (7), it follows that  $j + 1$  is in  $\mathcal{D}$ , hence T7 (a) is shown. For T7 (b)–(e) we have to consider the  $\mathbf{Jsr}(s)$  instructions which occur in Fig. 8.

Consider the first  $\mathbf{Jsr}(fin_\alpha)$  instruction in Fig. 8 in the code of the statement  ${}^\alpha(\mathbf{try} \ {}^\beta block \ \mathbf{finally} \ {}^\gamma stm)$ . Then  $i = end_\gamma$ ,  $j = end_\beta$ ,  $regT_i = \mathcal{A}(\gamma)$ ,  $regT_j = \mathcal{A}(\beta)$  and  $regT_{j+1} = \mathcal{A}(\alpha)$ .

For T7 (b) we have to show that  $\mathcal{A}(\gamma)(x) \preceq \mathcal{A}(\alpha)(x)$  for each  $x$  modified by  $fin_\alpha$ .

Let  $x$  be a variable in  $after(\alpha)$  so that  $\bar{x}$  is modified by  $fin_\alpha$ . If  $x$  is in  $after(\gamma)$ , then  $\mathcal{A}(\gamma)(x) = \mathcal{A}(\alpha)(x)$  and we are done. Otherwise,  $x$  is in  $after(\beta)$  and, by definition of  $after(\alpha)$ , there is no subexpression  $x = exp$  in  ${}^\gamma stm$ . Hence, there is no  $\mathbf{Store}(\bar{x})$  instruction between  $beg_\gamma$  and  $end_\gamma$  and, since  $\bar{x}$  is different from  $\overline{ret}_\alpha$ ,  $\bar{x}$  is not modified by  $fin_\alpha$ . Therefore  $x$  cannot be in  $after(\beta)$ . [Note: This proof step is not valid for the example **Test1** from Sect. 1. In the first proof attempt, we constructed at exactly this point in the proof the counter example **Test1**.]

For T7 (c) we have to show that  $\mathcal{A}(\beta)(x) \preceq \mathcal{A}(\alpha)(x)$  for each  $x$  not modified by  $fin_\alpha$ .

Let  $x$  be a variable in  $after(\alpha)$  so that  $\bar{x}$  is not modified by  $fin_\alpha$ . If  $x$  is in  $after(\beta)$ , then  $\mathcal{A}(\beta)(x) \preceq \mathcal{A}(\alpha)(x)$  and we are done. Otherwise,  $x$  is in  $after(\gamma)$  and not in  $before(\gamma)$ . By Lemma 32 (which is applicable because  $end_\gamma = i \in \mathcal{D}$ ), Lemma 39 and Lemma 38, it follows that  $\bar{x}$  is modified by  $fin_\alpha$ . By the assumption,  $x$  is not modified by  $fin_\alpha$ , and therefore the second case is not possible.

T7 (d) is satisfied, since  $opdT_i$  as well as  $opdT_{j+1}$  are empty.

For T7 (e) assume that  $\mathcal{A}(\alpha)(x) = \mathbf{retAddr}(l)$ , where  $x$  is not modified by  $fin_\alpha$ . By item 2 of Def. 23, it follows that  $\alpha$  is a position in a finally block with subroutine  $l$ . Hence, using Corollary 33 to guarantee the existence of a possible return from subroutine  $l$ , the code for subroutine  $fin_\alpha$  belongs to  $l$  in the sense of Def. 9.

Let us now consider the  $\mathbf{Jsr}$  instruction in the default handler. For T7 (b) we have to show that

$$\mathcal{A}(\gamma)(x) \preceq (\mathcal{B}(\beta) \oplus \{(\bar{e}, \mathbf{Throwable})\})(x)$$

```

 $\gamma$ lab: {
   $\beta_2$ try {
     $\varepsilon$ try { ... }
    finally {
       $\beta_1$ try {  $\alpha$ break lab; }
      finally  $\delta_1$ { ... }
    }
  } finally  $\delta_2$ { ... }
}

```

Figure 9. The general situation for break statements.

for each  $x$  not modified by  $fin_\alpha$ .

This is true, since  $before(\beta) = before(\gamma) \subseteq after(\gamma)$ , by Fig. 7, and the variable  $\bar{e}$  has been chosen by the compiler to be sufficiently fresh so that there is no  $\mathbf{Store}(\bar{e})$  instruction between  $beg_\gamma$  and  $end_\gamma$ . T7 (c)–(e) follow as above.

Finally we move to the  $\mathbf{Jsr}$  instructions in  $\mathcal{S}(\alpha\mathbf{break\ }lab;)$ . The difficult case is when the instruction  $\mathbf{Break}(lab)$  is in  $\mathcal{D}$  and  $regB = \mathcal{A}(\gamma)$ . If  $\mathbf{Break}(lab)$  is in  $\mathcal{D}$ , then by Lemma 36, it follows that  $\alpha\mathbf{break\ }lab$  can exit (Def. 19) the labeled statement  $\gamma(lab : stm)$  enclosing  $\alpha$ .

The general situation is pictured in Fig. 9.

Consider the instruction  $\mathbf{Jsr}(fin_{\beta_1})$ . For T7 (b) we have to show that

$$\mathcal{A}(\delta_1)(x) \preceq (\mathcal{A}(\gamma) \oplus \mathcal{B}(\beta_2))(x)$$

for every  $x$  modified by  $fin_{\beta_1}$ . By the nesting of the try-finally statements, we have

$$before(\beta_2) \subseteq before(\beta_1) = before(\delta_1) \subseteq after(\delta_1).$$

Let  $x$  be a variable in  $after(\gamma)$  which is modified by  $fin_{\beta_1}$ . Since  $x$  is modified by  $fin_{\beta_1}$ , there is a  $\mathbf{Store}(x)$  instruction in the code for the block at position  $\delta_1$ . This can only be, if there is an assignment  $x = exp$  in the block at  $\delta_1$ . By the rules of definite assignment in Fig. 7 and item 2 of Def. 20,  $x$  is in  $after(\beta_1)$  because the  $\alpha\mathbf{break\ }lab$  can exit  $\gamma$ . By the rules of definite assignment for try-finally, it follows that  $x \in after(\delta_1)$ . [Note: This proof step is not valid for the example **Test2** from Sect. 1. In the first proof attempt, we constructed at exactly this point in the proof the counter example **Test2**.]

For T7 (c) we have to show that

$$(\mathcal{A}(\gamma) \oplus \mathcal{B}(\beta_1))(x) \preceq (\mathcal{A}(\gamma) \oplus \mathcal{B}(\beta_2))(x)$$

for each  $x$  not modified by  $fin_{\beta_1}$ . This is true because  $before(\beta_2)$  is contained in  $before(\beta_1)$ . T7 (d) and (e) follow as above.  $\square$

## 6. Conclusion

The proof that the Java-to-VM compiler generates verifiable code is longer and more complicated than the other proofs in this article. It requires several lemmas about properties of the generated code. For example, the only entry point into the code interval of a statement is the first instruction of the interval. Another lemma says that under certain assumptions there exists a path from the beginning of a subroutine to the end of the subroutine. Some of these lemmas cannot simply be proved by structural induction. For example, in Lemma 30, we show that every predecessor of an instruction which is not the first instruction in the code interval of a statement belongs also to the same code interval. This property is certainly true for the given method body. Moreover, it can be shown that if it is true for a statement, then it is also true for the immediate substatements. Hence the proof of Lemma 30 proceeds top down starting at the root position of the method body down to the atomic statements.

As far as we know, the only publication that also considers the problem of compiling Java into verifiable code is [9, 18], where the Isabelle proof assistant is used to prove that, for a fragment of Java, the compiler generates verifiable code. Unfortunately the main problems that we have faced in this article, namely subroutines, break statements, exceptions, and the rules of definite assignment, are not yet treated in the formalization in Isabelle.

Finally, the problem of generating verifiable code seems to play a more important role in Microsoft's .NET environment than in the Java world. Microsoft's .NET is a multi-language environment and several compilers already exist for different programming languages. Most compiler writers want to generate verifiable IL (intermediate language). We plan to use the methods developed in this paper to show that their compilers actually generate verifiable code.

## References

1. Börger, E.: 2002, 'The Origins and the Development of the ASM Method for High Level System Design and Analysis'. *J. of Universal Computer Science* **8**(1), 2–74.

2. Coglio, A.: 2002, 'Simple Verification Technique for Complex Java Bytecode Subroutines'. In: *Proc. 4th ECOOP Workshop on Formal Techniques for Java-like Programs*.
3. Colby, C., P. Lee, G. C. Necula, F. Blau, M. Plesko, and K. Cline: 2000, 'A certifying compiler for Java'. In: *SIGPLAN Conference on Programming Language Design and Implementation*. pp. 95–107.
4. Freund, S. N. and J. C. Mitchell: 1999, 'The type system for object initialization in the Java bytecode language'. *ACM Transactions on Programming Languages and Systems* **21**(6), 1196–1250.
5. Gosling, J., B. Joy, G. Steele, and G. Bracha: 2000, *The Java(tm) Language Specification*. Addison Wesley, second edition.
6. Gurevich, Y.: 1993, 'Evolving Algebras 1993: Lipari Guide'. In: E. Börger (ed.): *Specification and Validation Methods*. Oxford University Press, pp. 9–36.
7. Haase, E.: 2001, 'JustIce: An Implementation of a Free Class File Verifier for Java'. Technical report, Institut für Informatik, Freie Universität Berlin. <http://bcel.sourceforge.net/justice/>.
8. Henrio, L. and B. Serpette: 2001, 'A Framework for Bytecode Verifiers: Application to Intra-Procedural Continuations'. Technical report, Inria Sophia-Antipolis.
9. Klein, G. and M. Strecker: 2002, 'Verified bytecode verification and type-certifying compilation'. Technical report, Technical University Munich.
10. Leroy, X.: 2001, 'On-card bytecode verification for Java Card'. In: I. Attali and T. Jensen (eds.): *Smart Card Programming and Security (E-smart 2001)*. pp. 150–164.
11. Lindholm, T. and F. Yellin: 1999, *The Java(tm) Virtual Machine Specification*. Addison Wesley, second edition.
12. O'Callahan, R.: 1998, 'A simple, comprehensive type system for Java bytecode subroutines'. In: *Proc. 26th ACM Symposium on Principles of Programming Languages*. pp. 70–78.
13. Qian, Z.: 2000, 'Standard Fixpoint Iteration for Java Bytecode Verification'. *ACM Transactions on Programming Languages and Systems* **22**(4), 638–672.
14. Schmid, J.: 1999, 'Executing ASM specifications with AsmGofer'. Web pages at <http://www.tydo.de/AsmGofer>.
15. Sirer, E., S. McDirmid, and B. Bershad: 1997, 'Kimera: A Java system security architecture'. <http://kimera.cs.washington.edu/>.
16. Stärk, R. F., J. Schmid, and E. Börger: 2001, *Java and the Java Virtual Machine—Definition, Verification, Validation*. Springer-Verlag.
17. Stata, R. and M. Abadi: 1999, 'A Type System for Java Bytecode Subroutines'. *ACM Transactions on Programming Languages and Systems* **21**(1), 90–137.
18. Strecker, M.: 2002, 'Investigating type-certifying compilation with Isabelle'. In: *Proc. Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*.
19. Sun Microsystems: 2000, 'Connected, Limited Device Configuration, Specification 1.0, Java 2 Platform Micro Edition'.