Java Definite Assignment in Isabelle/HOL*

Norbert Schirmer

Technische Universität München
Department of Informatics
80290 Munich, Germany
Email: schirmer@in.tum.de

Abstract. In Java the compiler guarantees that each local variable is initialised when we attempt to access it at runtime. This prohibits access to uninitialised memory during execution and is a key ingredient for type safety. We have formalised the definite assignment analysis of the Java compiler in the theorem prover Isabelle/HOL and proved it correct.

1 Motivation

One potential violation of type safety during runtime is caused by uninitialised variables. An uninitialised piece of memory may contain an arbitrary sequence of bits. If we regard them as proper values and read them, we can easily produce some unpredictable behaviour. Think of an uninitialised object variable. The bit sequence is interpreted as a reference to an object in main memory and since the variable is not initialised we may read or write to an arbitrary memory location. In Java special care is taken that access to uninitialised memory does not occur. If an object is created, the fields of the object are initialised with default values. For example the default value for an object reference is `null`. For local variables the compiler and bytecode-verifier have to ensure that all variables have a *definitely assigned* value when any access to its value occurs. The terminology "Definite Assignment" stems from the Java Language Specification (JLS)[1]. The definite assignment analysis of the compiler is basically a data-flow analysis, which ensures that there is an assignment to a local variable on every possible execution path before a read of the variable content. The analysis is similar to a live variable analysis or "Definition-Use" chains [3].

The work presented in this paper is part of a comprehensive research effort aiming at formalising and verifying key aspects of the Java programming language. In particular we have a type system and an operational semantics (with a proof of type soundness) and an axiomatic semantics (with a proof of its equivalence to the operational semantics) for a large subset of Java [5,6]. All these formalisations and proofs have been carried out in the theorem prover Isabelle/HOL [4]. The definite assignment analysis was not yet part of our type system. As far as we know, the only formal model of Java source language which also treats definite assignment is the work of Stärk et al. [7]. In that work the analysis is

* This research is funded by the EU project *VerifiCard*

described with data flow equations and the proofs are carried out by pencil and paper. We present the analysis as type systems. The underlying data flow analysis does not require a fixed point iteration. Therefore the presentation as type systems appears to be easier to understand as the data flow equations. Moreover our model directly implies an implementation of the analysis and provides the first machine checked formalisation. The complete sources and documentation are available online [9].

2 The definite assignment analysis

In this section we introduce a formal model of the definite assignment analysis. Due to space limitations we cannot explain all details. We only focus on the analysis of a few Java language constructs. Our aim is to illustrate how we analyse the `while` loop with all its complications, like constant conditions and `break`s.

We define a relation written as B »t» A where B is the set of initialised (or definitely assigned) local variable *b*efore evaluation of the Java term t and A is the set of definite assigned variables *a*fter its evaluation. For example for an assignment `y = x` the relation, {`x`} »`y = x`» {`x`,`y`} is a valid analysis result. If we start in a state where the variable `x` was already assigned a value, then the assignment to `y` will be legal and we will end up in a state where both `x` and `y` are assigned. On the one hand the analysis will ensure that the variables we read from (like `x`) have an assigned value and on the other hand the analysis will calculate the set of variables that will be assigned after execution of the term. Let us look at a first example, taken from the JLS:

```
int k;
if (v > 0 && (k = System.in.read()) >= 0)
        System.out.println(k);
```

In this example the compiler recognises that `k` has been definitely assigned a value before it is printed out. Since the print method is only executed if the condition has evaluated to *True*, both sides of the conjunction `&&` must have been evaluated. Therefore the assignment to variable `k` was evaluated. Note that the operator `&&` only evaluates the right expression if the left one evaluated to *True*. The definite assignment analysis propagates knowledge about the outcome of a condition while analysing a branching statement. We introduce an auxiliary function *assigns-if* that calculates the set of assigned variables of a condition depending on the current branch. The semantical intuition of the term *assigns-if* b e is: "What are the assigned variables if the expression e evaluates to the boolean value b". In the example the function will yield the following results for the condition evaluating to *True* or *False*:

assigns-if True `(v > 0 && (k = System.in.read()) >= 0)` = {`k`}
assigns-if False `(v > 0 && (k = System.in.read()) >= 0)` = {}

The next example considers a `while` loop:

```
int k;
while (true) {
    k = n;
    if (k >= 5) break;
}
System.out.println(k);
```

The definite assignment rules will allow to print out `k` in this example, too. Since the condition of the `while` statement is constantly `true` the only way out of the loop is through the `break` in the body of the loop. For the analysis this means, that we have to take constant boolean conditions into account and take care of normal and of abrupt completion due to `break`. The constant value propagation in boolean conditions, to statically decide which path of evaluation will be chosen, is very basic. Only boolean literals are propagated over boolean connectives.

To properly analyse the `while` loop we need to calculate the sets of definitely assigned variables in the loop body, for both normal and `break` completion. Since `while` and `break` may also carry labels to indicate which `while` a nested `break` will complete, we have to analyse the definitely assigned variables for each of those labels. In our generalised model every `break` carries a label. In an analysis $B \gg t \gg A$ the output A is not just a set of assigned variables, but a pair of analysis results for normal and `break` completion: starting in a state where the local variables in set B are definitely assigned, if evaluation of term t completes normally, then all variables in set $nrm\ A$ are definitely assigned; if evaluation of term t completes abruptly because of a `break` with label l, then all variables in set $(brk\ A)\ l$ are definitely assigned. The analysis is designed to yield a safe approximation of the assigned variables at runtime. So whenever the analysis regards a variable as initialised, the variable will actually be initialised at the corresponding point of execution in any program run. The rules that set up the relation are defined inductively, for most terms of the language there is exactly one rule.

First we look at sequential composition written as $c_1;;\ c_2$ in our setting.

$$\frac{\begin{array}{c} B \gg\langle c_1\rangle\gg C_1 \qquad nrm\ C_1 \gg\langle c_2\rangle\gg C_2 \\ nrm\ A = nrm\ C_2 \qquad brk\ A = (brk\ C_1) \Rightarrow\!\cap\ (brk\ C_2)\end{array}}{B \gg\langle c_1;;\ c_2\rangle\gg A} \quad (\textsc{Comp})$$

The sequential composition of two statements c_1 and c_2 completes normally if both statements complete normally and completes abruptly if either of the statements completes abruptly. The second statement c_2 is only executed if the first one has completed normally. Therefore we feed the result $nrm\ C_1$ of the first part of the analysis into the second part. The result for normal completion of the second statement $nrm\ C_2$ is also the result for normal completion of the combined statement, since we know that both statements must have completed normally. For abrupt completion either the first or the second statement completed abruptly. For each label we intersect the results of both paths. That is what the infix function $\Rightarrow\cap$ is for:

def $A \Rightarrow\cap\ \ B \equiv \lambda\ l.\ A\ l \cap B\ l$

$$\frac{\begin{array}{c} B \gg\langle e\rangle\gg E \\ (B \cup \textit{assigns-if True } e) \gg\langle c_1\rangle\gg C_1 \\ (B \cup \textit{assigns-if False } e) \gg\langle c_2\rangle\gg C_2 \\ \textit{nrm } A = \textit{nrm } C_1 \cap \textit{nrm } C_2 \qquad \textit{brk } A = \textit{brk } C_1 \Rightarrow\cap \textit{ brk } C_2 \end{array}}{B \gg\langle \textit{If}(e)\ c_1\ \textit{Else}\ c_2\rangle\gg A} \quad (\text{IF})$$

To analyse the `if` statement, we first ensure that the condition e passes the definite assignment analysis. By that we ensure that the variable accesses in e are valid, although we do not further use the output E of this analysis. We can gain a more precise approximation as input for the two conditional branches. If evaluation continues with the first branch c_1 we know that the condition e must have evaluated to *True*. In the other case, when evaluation continues with c_2, we know that the condition e must have been *False*. This extra knowledge about the value of the condition is exploited by the function *assigns-if*. The results C_1 and C_2 of the analysis of the two possible execution paths are intersected to gain the overall result of the `if` statement. Since special care of constant values is taken by *assigns-if*, this intersection also works fine for constant branching conditions as the following example will illustrate:

```
int k;
if (true) {k = 5;} else {...}
System.out.println(k);
```

For this code fragment we get *assigns-if True* `true` $= \{\}$ and *assigns-if False* `true` $=$ *UNIV*, where *UNIV* is the universal set in Isabelle. We can safely regard all variables to have an assigned value in case the condition will evaluate to `false` since this will never happen. The `else` block is unreachable. This leads to $\textit{nrm } C_1 = \{k\}$ and $\textit{nrm } C_2 = \textit{UNIV}$. Therefore, intersecting these two sets will leave us with $\textit{nrm } A = \{k\}$. The same idea carries over to the `break` map as well. If we are in a path of execution that will never be reached, the set of assigned variables will be *UNIV*. When we then encounter a `break` with label l this set will be inserted in the `break` map at position l (see rule JMP). The lifted intersection $\Rightarrow\cap$ will work for the `break` map in the same manner as the ordinary intersection $\cap$ for the normal sets.

$$\frac{\begin{array}{c} B \gg\langle c\rangle\gg C \\ \textit{nrm } A = \textit{nrm } C \cap (\textit{brk } C)\ l \qquad \textit{brk } A = \textit{rmlab } l\ (\textit{brk } C) \end{array}}{B \gg\langle l\bullet\ c\rangle\gg A} \quad (\text{LAB})$$

If a `break` to a label l occurs inside of a labelled statement $l\bullet\ c$, which carries the same label, the `break` will be absorbed and the labelled statement will complete normally. The set of variables that are definitely assigned for normal completion then is given by the intersection of the variables for both possible paths out of c: for normal completion and for abrupt completion because of a `break` with label l. Since the labelled statement $l\bullet\ c$ absorbs a `break` to l it will itself never complete with such a `break`. Therefore we reset the entry for the label l in $brk\ C$ with the function $rmlab$:

$$\textbf{def}\ rmlab\ l\ A \equiv \lambda\ k.\ if\ k{=}l\ then\ UNIV\ else\ A\ k$$

$$\frac{\begin{array}{c} nrm\ A = UNIV \\ brk\ A = (case\ jump\ of \\ \quad Break\ l \Rightarrow \lambda\ k.\ if\ k{=}l\ then\ B\ else\ UNIV \\ \quad \mid Cont\ l \Rightarrow \lambda\ k.\ UNIV \\ \quad \mid Ret \Rightarrow \lambda\ k.\ UNIV) \end{array}}{B \gg\langle Jmp\ jump\rangle\gg A} \quad (\textsc{Jmp})$$

The jump statements in our model combines all non exceptional abrupt completions: `break`, `continue` and `return`. Execution of a jump will never complete normally. Therefore $nrm\ A = UNIV$. Note that this implies that the definite assignment analysis will never complain about any code in sequence after the jump since this code is unreachable. The `break` map is more interesting. Since definite assignment only cares about `break`s we can take the trivial map $\lambda\ k.\ UNIV$ for `continue` and `return`. In case of a `break` with label l we take the currently assigned variables B as entry for label l. For other labels we again use the trivial map.

$$\frac{\begin{array}{c} B \gg\langle e\rangle\gg E \\ (B \cup assigns\text{-}if\ True\ e) \gg\langle c\rangle\gg C \\ nrm\ A = nrm\ C \cap (B \cup assigns\text{-}if\ False\ e) \qquad brk\ A = brk\ C \end{array}}{B \gg\langle l\bullet\ While(e)\ c\rangle\gg A} \quad (\textsc{Loop})$$

First of all the loop in our model of Java only handles continue jumps itself. A `break` inside the loop is handled by an enclosing label statement. A labelled loop in Java like `l: while (...) ...` is modelled by $l\bullet\ (l\bullet\ While(\ldots)\ \ldots)$, where the inner label is directly part of the while (to handle `continue l`) and the outer one is an additional label statement (to handle `break l`). Splitting up these two concepts unclutters the definite assignment analysis for the loop statement and makes the basic ideas clearer. They are the same as for the `if` statement. The loop body is analysed with the extra knowledge that the condition must have evaluated to *True*. The assigned variables for normal completion are given by the intersection of the two possible execution paths, one if we enter the body at least once and the second one if we do not enter the body at all (when the branching condition e evaluates to *False*). The analysis for constant conditions like `while (true)` works in the same fashion as explained for the `if` statement. Now let us look at the `break` map. Since evaluation of the branching condition will never end up in a `break` we can directly take the `break` map from the analysis of the loop body. Let me illustrate how the rules LAB, JMP and LOOP work together to `break` out of a labelled `while`:

```
int j;
l: { while (true) {        nrm  | brk
         j = 5;            {j}  | λ k. UNIV
         break l;          UNIV | λ k. if k=l then {j} else UNIV
     }                     UNIV | λ k. if k=l then {j} else UNIV
}                          {j}  | λ k. UNIV
System.out.println(j);
```

After the assignment `j = 5` the set of definitely assigned variables that will be the input for the analysis of `break l` will be $\{j\}$. So applying the JMP rule will yield UNIV for normal completion and ($\lambda\ k.\ \mathit{if}\ k{=}l\ \mathit{then}\ \{j\}\ \mathit{else}\ \mathit{UNIV}$) for `break` completion. We will refer to this intermediate result as C. Since the condition of the `while` statement is trivially `true` this will also be the result of the whole loop (since *assigns-if False* `true` $= \mathit{UNIV}$). Then the LAB rule has to be considered. For normal completion it will calculate $\mathit{nrm}\ C \cap (\mathit{brk}\ C)\ l = \mathit{UNIV} \cap \{j\} = \{j\}$. For completion because of a `break` it will yield $\mathit{rmlab}\ l\ (\mathit{brk}\ C) = \lambda\ k.\ \mathit{UNIV}$ which simply expresses that no `break` will actually leave the label statement.

Since the `break` map is important for the analysis of the `while` statement we want to take a look at how the `break` map is calculated for the `finally` statement c_1 *Finally* c_2. Even if a `break` occurs in c_1 the block c_2 will be executed and can assign some variables. We take this into account in the analysis.

$$\frac{\begin{array}{c} B \gg\!\langle c_1\rangle\!\gg C_1 \qquad B \gg\!\langle c_2\rangle\!\gg C_2 \\ \mathit{nrm}\ A = \mathit{nrm}\ C_1 \cup \mathit{nrm}\ C_2 \\ \mathit{brk}\ A = ((\mathit{brk}\ C_1) \Rightarrow\cup_\forall (\mathit{nrm}\ C_2)) \Rightarrow\cap (\mathit{brk}\ C_2) \end{array}}{B \gg\!\langle c_1\ \mathit{Finally}\ c_2\rangle\!\gg A} \qquad (\textsc{Fin})$$

Regardless of how the first statement c_1 will complete the `finally` block c_2 will be executed. If an abruption occurs in either statement it is (re-)raised after the completion of `finally`. If both statements terminate abruptly the second one takes precedence.

If the whole statement completes normally we can conclude that both c_1 and c_2 were executed and completed normally. Therefore we take the union of both analysis results for normal completion. In case the whole statement completes abruptly because of a `break` we regard two possibilities: Does c_2 complete normally or abruptly with a `break`. First if c_2 completes normally then the `break` has already occurred in the first statement c_1. The assigned variables for normal completion of c_2 can be added to the `break` map of C_1. We augment every set in the `break` map of C_1 with the set for normal completion of C_2. This is expressed in $((\mathit{brk}\ C_1) \Rightarrow\cup_\forall (\mathit{nrm}\ C_2))$ with the auxiliary operator $\Rightarrow\cup_\forall$:

def $A \Rightarrow\cup_\forall B \equiv \lambda\ k.\ A\ k \cup B$

Secondly if c_2 completes abruptly with a `break` this will show up in the overall result, because the abnormality of the second statement takes precedence. Statically we do not know whether the first statement c_1 has completed normally or abruptly. That is why we start the analysis of c_2 with the set B assigned before the whole `finally` statement and only regard $(brk\ C_2)$ to be definitely assigned if c_2 completes with a `break`. The overall result of the `break` analysis for the `finally` statement is given by the lifted intersection of these two main paths.

3 Safe approximation

The definite assignment analysis has to be a safe approximation. This means that if the analysis will infer a variable as definitely assigned at a certain program point, then this variable will actually be assigned at that point during every execution of the program. This property is a key ingredient for type safety. Only if we know for sure that a local variable has an assigned value we can safely read this value and trust the type of the value. We have adapted the type safety proof of [5] which did not yet take definite assignment into account. To ensure type safety there, all local variables were initialised by the semantics itself. To reestablish type safety after removing this default initialisation, the key property about definite assignment we need is: the approximation of the assigned variables that the analysis yields is a subset of the variables that will actually be assigned in a program run:

theorem *da-safe-approx*:
assumes *eval*: $prg\ Env \vdash s_0\ -t\succ\rightarrow\ (v, s_1)$
assumes *wt*: $Env \vdash t{::}T$
assumes *da*: $dom\ (locals\ (store\ s_0))\ »t»\ A$
assumes *wf*: $wf\text{-}prog\ (prg\ Env)$
shows $(normal\ s_1 \longrightarrow nrm\ A \subseteq\ dom\ (locals\ (store\ s_1))) \wedge$
$\quad(\forall l.\ abrupt\ s_1 = Some\ (Jump\ (Break\ l)) \wedge normal\ s_0$
$\quad\quad\longrightarrow brk\ A\ l \subseteq dom\ (locals\ (store\ s_1)))$

Let me first explain the assumptions of the theorem. With *eval* we look at evaluation of a Java term t. We start in the initial state s_0 and evaluate the term t to the result value v and the final state s_1. The semantics is an operational big step semantics. Java statements and expressions are generalised to terms in this semantics. Statements evaluate to a dummy result. With *wt* we assume that the term t is welltyped in the typing environment *Env*. The program component of the typing environment *prg Env* describes the class and interface hierarchy and is also a parameter of the evaluation relation. The term must have passed the definite assignment analysis: $dom\ (locals\ (store\ s_0))\ »t»\ A$. The state is decomposed into an *abrupt*- and a *store*-component. The *abrupt*-component signals all kinds of possible reasons for an abrupt completion (exceptions, `break`, `continue` and `return`). The *store*-component contains the contents of the heap and the local variables. The already assigned variables in the current state s_0 are the input variables for the definite assignment analysis. Formally we can get hold of the already assigned variables by taking the domain (*dom*) of the local variable map *locals* of the *store* in state s_0. Finally, assumption *wf* constrains the programs under consideration. We only look at wellformed programs. The wellformedness predicate encapsulates a lot of context conditions that we usually have in mind for a Java program. Particularly interesting for the definite assignment analysis, wellformedness ensures that all methods and static initialisers have been analysed and are welltyped. The conclusion of the theorem summarises the different aspects of definite assignment analysis that we had in mind during the design of the analysis. The main structure is a conjunction of the two constituents for normal and `break` completion:

– If evaluation completes normally (*normal* s_1) then the analysis result for normal completion *nrm A* will be a subset of the actually assigned local variables *dom* (*locals* (*store* s_1)).
– If evaluation starts in a normal state (*normal* s_0) and completes abruptly because of a `break` to label l (*abrupt* s_1 = *Some* (*Jump* (*Break* l))), the analysis result *brk A* l for the corresponding `break` set has to be a subset of the actually assigned local variables *dom* (*locals* (*store* s_1)).

The `break` part of the conclusion carries the precondition that the evaluation started in a normal state. This has to be motivated by our evaluation model used to define the operational semantics and how abrupt completion is handled. If the initial state is not normal, then abrupt completion is already signalled in the initial state and therefore the evaluation will just be skipped. The theorem is proved by induction on the evaluation relation.

4 Conclusion

In this paper we have formalised the definite assignment analysis of Java in the theorem prover Isabelle/HOL and proved that the analysis yields a safe approximation of the assigned variables at runtime. This is a key property to ensure type safe execution of a Java program. Our Java formalisation [5,6] was sufficiently mature to let us add and analyse the new concept. This again shows, that it is feasible to investigate aspects of a realistic programming language completely formally in a theorem prover. For future research it would be interesting to investigate how the source language model for definite assignment fits together with the analysis of the bytecode-verifier. The link between the source and the bytecode typesystem has already been studied [7,8]. Stärk et al. [7] have reported on some discrepancy of the definite assignment analysis of the JLS and their model of bytecode-verification. However, the bytecode-verifier presented by Klein [2] should be able to deal with it.

Acknowledgements I am grateful to Gerwin Klein, Martin Strecker and the anonymous referees for comments on this paper.

References

1. James Gosling, Bill Joy, Guy L. Steele Jr., and Gilad Bracha. *The Java Language Specification, Second Edition*. Addison Wesley, 2000.
2. Gerwin Klein. *Verified Java Bytecode Verification*. PhD thesis, Institut für Informatik, Technische Universität München, 2003.
3. Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.
4. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer, 2002. LNCS 2283.
5. David von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001.
6. Norbert Schirmer. Analysing the Java Package/Access Concepts in Isabelle/HOL. Technical Report NIII-R0204, Computing Science Department, University of Nijmegen, 2002.
7. Robert Stärk, Joachim Schmid, and Egon Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer, 2001.
8. Martin Strecker. Investigating type-certifying compilation with Isabelle. In *Proc. Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 2514 of *Lecture Notes in Computer Science*. Springer Verlag, 2002.
9. Verificard at Munich. Available from http://isabelle.in.tum.de/verificard.