Precondition Generation for a Java Subset

Nicole Rauch[1]

rauch@informatik.uni-kl.de
Universität Kaiserslautern, Germany

1 Introduction

In order to achieve a better software quality, it is an interesting aspect to verify the correctness of a program at the source code level with respect to a given specification. We are currently developing an interactive verification system called Jive [7] which operates on a Java subset using a Hoare-style programming logic. It is very tedious to verify each and every line of the code to be examined in a separate step, even for those code parts that are straightforward to handle. Thus it is desirable that larger pieces of code can be verified automatically. This can be achieved by using a predicate transformer. It can handle a code sequence by generating a precondition for each element of the sequence, starting with a given postcondition. If it can be shown that a given precondition implies the one generated for the whole sequence, the code indeed satisfies its specification.

This paper presents a predicate transformer called "practical weak precondition transformation" (*pwp*) which operates on the statements of a sequential Java subset called Java-KE. This subset covers object-oriented features like dynamic method binding and exceptions. References, recursion and iteration are supported as well. Details of the embedding of the *pwp* predicate transformer into our verification tool Jive are given in [12].

Predicate transformers have been examined in detail in the literature. Many other papers deal with aspects like modelling the object store or handling exceptions. This paper puts these pieces together and presents a predicate transformer that adapts parts of these different approaches to our Java subset and integrates them into the formal setting used in Jive.

Related Work Dijkstra was one of the first to present a weakest precondition calculus [4]. His work was revised and extended by Gries [5]. These calculi are based on a sequential language. More recent publications include Cavalcanti and Naumann's refinement calculus for an object-oriented language [3]. This calculus supports classes, visibility, dynamic binding, and recursion. Leino presents in [6] a weakest precondition calculus for an object-oriented language with exceptions. He identifies a part of the object store with each pointer type. David von Oheimb embeds in [9] a large part of Java Card, a sequential Java sublanguage, into the theorem prover Isabelle/HOL, including exceptions and an object store model, and performs meta-theory on the language.

Overview Section 2 provides an overview of the Java subset we consider in this paper and explains our model of the object store and exceptions. Section 3 presents the practical weak precondition generation mechanism *pwp*. It first introduces the language constructs for which a weakest precondition has been found, subsequently followed by those language constructs that can not be treated as easily. For the latter constructs, *pwp* does not yield weakest preconditions, but it provides "practical weak" preconditions, i.e. preconditions that are weak enough to be useful in practice.

2 Verifying Java

2.1 The Language Java-KE

The predicate transformer *pwp* operates on a sequential Java subset called JAVA-KE. This language encompasses basic data types, interface and class types, subtyping and inheritance, a simple exception mechanism, dynamic binding, most statement kinds in a simplified form, and expressions without side-effects.

2.2 Modelling the Object Store and Exceptions

The object store (or heap) is considered a global program variable denoted by \$. The variable \$ can be used in pre- and postconditions and holds values of type *Store*. Type *Store* is an abstract data type with the following operations:

$$
\begin{array}{lll}
\_\langle\_ := \_\rangle & : \mathit{Store} \times \mathit{InstVar} \times \mathit{Value} & \rightarrow \mathit{Store} \\
\_\langle\_\rangle & : \mathit{Store} \times \mathit{TypeId} & \rightarrow \mathit{Store} \\
\_(\_) & : \mathit{Store} \times \mathit{InstVar} & \rightarrow \mathit{Value} \\
\mathit{alive} & : \mathit{Value} \times \mathit{Store} & \rightarrow \mathit{Bool} \\
\mathit{new} & : \mathit{Store} \times \mathit{TypeId} & \rightarrow \mathit{Value}
\end{array}
$$

InstVar denotes the set of instance variables. $OS\langle IV := V\rangle$ yields the object store that is obtained from OS by updating instance variable IV with value V. $OS\langle T\rangle$ yields the object store that is obtained from OS by allocating a new object of type T. $OS(IV)$ yields the value of instance variable IV in store OS. If V is an object reference, $\mathit{alive}(V, OS)$ tests whether the referenced object is alive in OS. $\mathit{new}(OS, \mathit{TID})$ yields a reference to an object of type *TID* that is not alive in OS. (An axiomatization of these operations is presented in [11].) In addition, it is assumed that there exists an initial value for each type. This value is provided by $\mathit{init}(T)$. T may be an interface type or basic type; $\mathit{init}(T)$ may yield `null`.

To handle exceptions, our proof tool uses a special program variable `exc` that records whether an exception has occurred and if so, of what kind. If a statement terminated normally, `exc` equals `null` in the postconditional state. Otherwise `exc` contains a value of an exceptional type. Currently, the exceptional types `NullPExc` for null pointer exceptions and `CastExc` for class cast exceptions are supported. In preconditional states, the semantics guarantees `exc` = `null`.

3 The Predicate Transformer *pwp*

The predicate transformer *pwp* presented here treats all JAVA-KE statements. We show for each statement how a given postcondition is transformed into a weak precondition. The generated precondition is weakest for all statements except the `while`-loop, method invocation and method call (see below).

3.1 Weakest Preconditions

The precondition generation rules presented in this subsection yield weakest preconditions wrt. a given statement `stmt` and postcondition **P**. This means that any execution of `stmt` starting in any state S that satisfies the generated precondition, denoted by $\mathit{pwp}(\text{“}\texttt{stmt}\text{”}, \mathbf{P})$, leads to a state S' that satisfies the given postcondition **P**.

The first rule, *pwp-field-read*, operates on a given postcondition **P** and the *field-read* statement. This statement operates on the object referenced by y. It can lead to two results: If y

points to some object in the object store, the value of the field a of the object referenced by y is assigned to the local variable x, and the statement terminates normally. If, on the other hand, the reference y equals `null`, the statement terminates exceptionally. A new null pointer exception object of type `NullPExc` is created in the current object store, and a reference to this object is stored in `exc`. The notation $\mathbf{R}[a/b]$ represents the result of substituting a for the free occurrences of b in $\mathbf{R}$. The current object store is denoted by \$.

pwp-field-read: $pwp(\text{“}x = y.a;\text{”}, \mathbf{P}) =_{def} (y \neq \texttt{null} \ \wedge \ \mathbf{P}[\$(y.a)/x])$
$\vee \ (y = \texttt{null} \ \wedge \ \mathbf{P}[\$\langle\texttt{NullPExc}\rangle/\$, new(\$, \texttt{NullPExc})/\texttt{exc}])$

In the *field-write* statement, the value of the expression e is assigned to the field a of the object referenced by x. Again, the statement can terminate normally, which leads to an update of the object x in the current object store, or exceptionally, which causes a null pointer exception object to be generated as described above.

pwp-field-write: $pwp(\text{“}x.a = e;\text{”}, \mathbf{P}) =_{def} (x \neq \texttt{null} \wedge \mathbf{P}[\$\langle x.a := e\rangle/\$])$
$\vee (x = \texttt{null} \wedge \mathbf{P}[\$\langle\texttt{NullPExc}\rangle/\$, new(\$, \texttt{NullPExc})/\texttt{exc}])$

The *cast* statement casts the value of the expression e to type T. This is only successful if the type of this value, denoted by *typeof(e)*, is indeed a subtype of T (the subtype relation is depicted by $\preceq$). If this is the case, the result of the cast operation is assigned to the local variable x. Otherwise, a cast exception object of type `CastExc` is created and assigned to the variable `exc`.

pwp-cast: $pwp(\text{“}x = (\mathrm{T})e;\text{”}, \mathbf{P}) =_{def} (typeof(e) \preceq \mathrm{T} \ \wedge \ \mathbf{P}[e/x]) \vee$
$(typeof(e) \not\preceq \mathrm{T} \ \wedge \ \mathbf{P}[\$\langle\texttt{CastExc}\rangle/\$, new(\$, \texttt{CastExc})/\texttt{exc}])$

The *new* statement generates a new object of type T on the object store and assigns a reference to it to the local variable x. We do not consider out of memory errors as they are not related to the correctness of the code. Thus, the *new* statement cannot raise an exception.

pwp-new: $pwp(\text{“}x = \texttt{new}\ \mathrm{T}();\text{”}, \mathbf{P}) =_{def} \mathbf{P}[new(\$, \mathrm{T})/x, \$\langle\mathrm{T}\rangle/\$]$

The *seq* statement represents the sequential composition of two statements. If an exception occurs during the execution of the first statement s_1, the execution of the second statement s_2 is skipped, otherwise s_2 is executed after s_1.

pwp-seq: $pwp(\text{“}s_1\ s_2\text{”}, \mathbf{P}) =_{def} pwp(\text{“}s_1\text{”}, (\texttt{exc} \neq \texttt{null} \ \wedge \ \mathbf{P}) \ \vee \ (\texttt{exc} = \texttt{null} \ \wedge \ pwp(\text{“}s_2\text{”}, \mathbf{P})))$

The *if* statement executes the statement s_1 if the expression e is evaluated to `true`. Otherwise, it executes s_2.

pwp-if: $pwp(\text{“}\texttt{if}(e)\{s_1\}\texttt{else}\{s_2\}\text{”}, \mathbf{P}) =_{def} (pwp(\text{“}s_1\text{”}, \mathbf{P}) \wedge e) \vee (pwp(\text{“}s_2\text{”}, \mathbf{P}) \wedge \neg e)$

The *block* statement declares a local variable v of type T. Then, it executes the statement s. $init(\mathrm{T})$ yields the initial value for variables of type T. This value is initially stored in v.

pwp-block: $pwp(\text{“}\mathrm{T}\ v; s\text{”}, \mathbf{P}) =_{def} pwp(\text{“}s\text{”}, \mathbf{P})[init(\mathrm{T})/v]$

The *try-catch* statement executes the statement s_0. If the execution terminates normally, the rest of the *try-catch* statement is ignored. If an exception occurs in s_0, then the type of the exception can either be a subtype of T or not. In the first case, the reference to the exception object, which is stored in `exc`, is assigned to the local variable e, then `exc` is set to `null`, and the statement s_2 is executed. In the second case, the rest of the *try-catch* statement is ignored again.

pwp-catch: $pwp(\text{“}\texttt{try}\{s_0\}\texttt{catch}(\mathrm{T}\ e)\{s_1\}\text{”}, \mathbf{Q}) =_{def}$
$pwp(\text{“}s_0\text{”}, ((\texttt{exc} = \texttt{null} \vee \mathrm{typeof}(\texttt{exc}) \not\preceq \mathrm{T}) \wedge \mathbf{Q}) \vee$
$(\texttt{exc} \neq \texttt{null} \ \wedge \ \mathrm{typeof}(\texttt{exc}) \preceq \mathrm{T} \ \wedge \ pwp(\text{“}s_1\text{”}, \mathbf{Q})[\texttt{null}/\texttt{exc}, \texttt{exc}/e]))$

3.2 Practical Weak Preconditions

The following *pwp*-rules are not sharp, which means that there exist weaker preconditions that lead to the same postcondition. But this might only be of theoretical value if the sharp rule is not practically manageable. A more detailed discussion follows below.

pwp-while: The `while`-loop causes the usual problems of finding the loop invariant. We assume here that the user has annotated each while loop with the according loop invariant before the application of *pwp*. In our proof system JIVE we let the user determine the loop invariant interactively. We are planning to investigate invariant generation as presented in [13] in the near future.

The *while* statement repeats the execution of statement s as long as the expression e yields true (which is checked before any execution of s) and no exception occurs in s.

$$\begin{aligned} \textit{pwp-while:}\ \ \textit{pwp}(\text{“}\texttt{while}(e)\{s\}\text{”}, \mathbf{Q}) =_{def} \mathbf{I} \ \wedge\ & (\forall \boldsymbol{v}, \$: \mathbf{I} \ \wedge\ e \Rightarrow \textit{pwp}(\text{“}s\text{”}, \mathbf{I})) \\ & \wedge\ (\forall \boldsymbol{v}, \$: \mathbf{I} \ \wedge\ (\texttt{exc} \neq \texttt{null} \ \vee\ \neg e) \Rightarrow \mathbf{Q}) \end{aligned}$$

Let $\boldsymbol{v}$ be the vector that contains the values of all variables that occur in $\mathbf{I}$, e, $\textit{pwp}(\text{“}s\text{”}, \mathbf{I})$ or $\mathbf{Q}$. Then the above quantification ranges over all possible values of variables and all possible object stores, denoted by the quantification over the current object store \$.

Dijkstra gives in [4] a recursive version of the weakest precondition of loops which relies on the fact that the given postcondition already is an invariant of the loop. The result depends on the number of iterations of the loop. Berghammer presents in [1] a more general variant of this rule which requires the calculation of a countably infinite set of formulae that are derived from the postcondition.

pwp-invocation: The *method invocation* statement invokes a method m on the object referenced by y, passing the expression e as argument. If y does not equal `null`, the method m is evaluated and the return value is stored in x. Otherwise, a null pointer exception is created.

In order for this *pwp*-rule to work, we assume that each method of the given program is annotated with suitable specifications. The *pwp*-rule basically uses the precondition $\mathbf{P}$ of the specification of the invoked method m, with some slight modifications, as precondition of the method invocation site. To be able to do this, it has to guarantee that the postcondition $\mathbf{R}$ of the method specification implies the postcondition at the invocation site. This implication must hold for a suitable set of results and object stores, namely for those results that can be produced by m if invoked in the preconditional state, and for those object stores that can be achieved from the preconditional object store by means of invoking m. It is safe to assume the implication for all results and object stores, but this precondition can usually be weakened by constraining their choice. Details of such constraints are given below. If, however, the method is invoked on an object that equals `null`, the resulting precondition can directly be derived from the postcondition, without regarding any method specifications.

$$\begin{aligned} \textit{pwp-invocation:}\ \ \textit{pwp}(\text{“}x = y.m(e);\text{”}, \mathbf{Q}) =_{def} & (y \neq \texttt{null} \ \wedge\ \mathbf{P}[y/\texttt{this}, e/\texttt{par}] \wedge \\ & (\forall E, H : \rho(y, e, \$, H, E) \ \wedge\ \mathbf{R}[E/\texttt{res}, H/\$] \Rightarrow \mathbf{Q}[E/x, H/\$])) \\ & \vee\ (y = \texttt{null} \ \wedge\ \mathbf{Q}[\$\langle\texttt{NullPExc}\rangle/\$, \textit{new}(\$, \texttt{NullPExc})/\texttt{exc}]) \end{aligned}$$

H and E must be fresh variables that do not appear in $\mathbf{Q}$ nor $\mathbf{R}$. The term $\rho(y, e, \$, H, E)$ represents some general constraints on the objects y on which the method can be invoked, on the arguments e, on the current object store \$, on the possible postconditional object stores H, and on the possible results E. A good ρ makes the precondition weaker and the rule sharper. A simple solution is to take ρ to be true everywhere. In this case, no constraints are regarded, which strengthens the precondition. This does not result in a sharp rule.

Which constraints are reasonable to be imposed on ρ to restrict the range of the quantification and to weaken the precondition? We start in the preconditional state with a given object store \$. The modifications that m can perform on \$ are quite limited. Only object stores that result from imposing those modifications on \$ can possibly play the role of the object store of the poststate. This especially includes that all objects that are alive in the preconditional object store \$ have still to be alive in the postconditional object store because objects cannot be destroyed in JAVA-KE. Additionally, we can restrict the possible result values E. First of all, E must be alive, which means that it must have been created in the program flow leading to the state regarded here. E must also represent an object or value which is a legal return value of m: it must be of the declared return type, and it must only contain information that m has access to, i.e. that can be reached via the implicit `this` reference which points to the object referenced by y or via a reference that is possibly passed to m as argument, denoted by e in the method invocation[1]. This leads to the next constraint on the object store, namely that m can only modify those objects it can reach. In the method invocation "$y.m(e)$", m can only reach objects that are reachable via the references y and possibly e. For further examples of constraints as well as for a definition of the *reach* property see [10].

A more general version of this discussion can be applied to the *pwp-while* rule as well.

Sharp rules for method invocation have already been examined in the literature. Bijlsma [2] and Naumann [8] both present a sharp rule, but neither of them models the global object store. Thus, it is not as difficult as in our scenario to get a sharp rule because the external constraints listed above can be disregarded. Still, it might be interesting to integrate their work into our rule to make it sharper than it is, which we are currently investigating.

pwp-call: This rule is used for method invocations that are statically bound. In JAVA-KE, only method invocations of the kind `super.m()` fulfill this requirement as the keyword `private` is not part of the language.

The line of argumentation equals the one used for *pwp-invocation*. The only difference is that here the method is not invoked on a given object y but on the implicit parameter (`this`). Therefore, we need not regard the case that y equals `null` because the JAVA-KE semantics always guarantees `this` $\neq$ `null`, which simplifies the generated precondition.

pwp-call: $pwp(\text{“}x = \texttt{super.m(}e\texttt{);”}, \mathbf{P}) =_{def} \mathbf{P}[e/\texttt{par}] \wedge$
$(\forall E, H : \rho(e, \$, H, E) \wedge \mathbf{R}[E/\texttt{res}, H/\$] \Rightarrow \mathbf{Q}[E/x, H/\$])$

4 Conclusions

In this paper we presented a predicate transformer that operates on a sequential Java subset. It yields practical weak preconditions for all statements of the language. These preconditions are weakest for all statements except for the `while` loop and method invocation. The preconditions generated for the latter statements are weak enough to be practically useful.

References

1. Rudolf Berghammer. Soundness of a purely syntactical formalization of weakest preconditions. In Dieter Spreen, editor, *Electronic Notes in Theoretical Computer Science*, volume 35. Elsevier Science Publishers, 2000.
2. A. Bijlsma, P. A. Matthews, and J. G. Wiltink. A sharp proof rule for procedures in wp semantics. *Acta Informatica*, 26:409–419, 1989.

[1] Of course e need not be a reference; it can be any expression.

3. Ana Cavalcanti and David A. Naumann. A weakest precondition semantics for refinement of object-oriented programs. *IEEE Transactions on Software Engineering*, 26(8):713–728, 2000.
4. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
5. David Gries. *The Science of Programming*. Springer-Verlag, New York, 1981.
6. K. Rustan M. Leino. Toward reliable modular programs. Technical Report TR95-03, California Institute of Technology, Pasadena, California, 5, 1995.
7. J. Meyer and A. Poetzsch-Heffter. An architecture for interactive program provers. In S. Graf and M. Schwartzbach, editors, *TACAS00, Tools and Algorithms for the Construction and Analysis of Systems*, volume 276 of *Lecture Notes in Computer Science*, pages 63–77, 2000.
8. David A. Naumann. Calculating sharp adaptation rules. *Information Processing Letters*, 77(2-4):201–208, 2001.
9. David von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001. `http://www4.in.tum.de/~oheimb/diss/`.
10. A. Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitation thesis, Technical University of Munich, January 1997.
11. A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In D. Swierstra, editor, *ESOP '99*, volume 1576 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.
12. N. Rauch and A. Poetzsch-Heffter. Predicate transformation as a proof strategy. In *Proc. 4th ECOOP Workshop: Formal Techniques for Java-like Programs*, 2002. To appear.
13. A. Tiwari, H. Rueß, H. Saïdi, and N. Shankar. A technique for invariant generation. In T. Margaria and W. Yi, editors, *TACAS01, Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 113–127, 2001.