Dealing with Numeric Fields in Termination Analysis of Java-like Languages *

Elvira Albert[1], Puri Arenas[1], Samir Genaim[2], and Germán Puebla[2]

[1] DSIC, Complutense University of Madrid (UCM), Spain
[2] CLIP, Technical University of Madrid (UPM), Spain

Abstract. Termination analysis tools strive to find proofs of termination for as wide a class of (terminating) programs as possible. Though several tools exist which are able to prove termination of non-trivial programs, when one tries to apply them to realistic programs, there are still a number of open problems. In the case of Java-like languages, one of such problems is to find a practical solution to prove termination when the termination behaviour of loops is affected by *numeric fields*. We have performed statistics on the Java libraries to see how often this happens in practice and we found that in 12.95% of cases, the number of iterations of loops (and therefore termination) explicitly depends on values stored in fields and, in the vast majority of cases, such fields are numeric. Inspired by the examples found in the libraries, this paper identifies a series of difficulties that need to be solved in order to deal with numeric fields in termination and propose some ideas towards a lightweight analysis which is able to prove termination of sequential Java-like programs in the presence of numeric fields.

1 Termination Analysis and Numeric Fields

Termination analysis tools strive to find proofs of termination for as wide a class of (terminating) programs as possible. Termination analysis is about the study of *loops*, which are the program constructs which may introduce non-termination. Loops may correspond to iterative constructs or to recursion. The boolean conditions which determine whether the loop should be executed again or not are called *guards*. Automated techniques for proving termination are typically based on analyses which track *size* information, such as the value of numeric data or array indexes, or the size of data structures. In particular, analysis should keep track of how the (size of the) data involved in loop guards changes when the loop goes through its iterations. This information is used for specifying a *ranking function* for the loop [14], which is a function which strictly decreases on a

* This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* project, by the Spanish Ministry of Education (MEC) under the TIN-2005-09207 *MERIT* project, and the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project. S. Genaim was supported by a *Juan de la Cierva* Fellowship awarded by MEC.

well-founded domain at each iteration of the loop, thus guaranteeing that the loop will be executed a finite number of times.

In the last two decades, a variety of sophisticated termination analysis tools have been developed. Several analyses and tools exist, primarily for less-widely used programming languages, including term rewrite systems [8], and logic and functional languages [11, 6, 10]. Termination-proving techniques are also emerging in the imperative paradigm [5, 7, 8], even for dealing with large industrial code [7].

Termination analysis of realistic object-oriented programming languages faces new difficulties due to the existence of advanced features such as exceptions, virtual method invocation, references, heap-allocated data-structures, objects, fields. Focusing on Java, termination analyzers for Java bytecode programs [1] and for Java source [9] are being developed which are able to accurately handle a good number of the features mentioned above. However, interesting open problems still remain. In particular, it is well known that the *heap* poses important difficulties to static analysis. Some reasons for this are that the heap is a global data structure whose contents are not accessed using named variables, but rather using (possibly chained) references. Therefore, the same location in the heap may be modified using different aliased references and, furthermore, references may be reassigned several times, and thus they may point to different locations during execution. When loop guards involve information stored in the heap, such as object *fields*, tracking size information becomes rather complex and accurate aliasing information is required in order to track all possible updates of the corresponding fields (see e.g. [12]).

A partial solution to this problem is already solved by the *path-length* domain [9] which allows proving termination of loops which traverse acyclic heap-allocated data structures (i.e., linked lists, trees, etc.). Path-length is an abstract domain which, for reference values, provides a safe approximation of the length of the longest reference chain reachable from it. Unfortunately, though the path-length domain is a useful abstraction for fields which contain references, it does not capture any information about fields which contain numbers. In this work we look into the Sun implementation of the Java libraries for J2SE 1.4.2 in order to estimate how often loop termination depends on *numeric* values stored in fields and to try to come up with sufficient conditions for termination which are able to cover a large fraction of those loops whose termination is not provable using current techniques, such as those in [9, 1].

2 Motivating Examples from the Java Libraries

Since termination is an undecidable problem, all techniques for proving termination provide sufficient (but not necessary) conditions for termination. Therefore, for any termination proving technique it is possible to find terminating programs where the given technique fails to prove termination. Thus, usually the practicality of termination analyses is measured by applying the analyses to a representative set of real programs. In this work, the design of the analysis is

driven by common programming patterns for loops that we have found in the Java libraries. By looking at Sun's implementation of the J2SE (version 1.4.2_13) libraries, which contain 71432 methods, we have found 7886 loops (for, while, and do) from which 1021 (12.95%) explicitly involve fields in their guards. By inspecting these 1021 loops, we have observed, among others, the following three kinds of common patterns in the Java libraries.

Pattern #1: Loops in this category use numeric fields as bounds for loop counters and, moreover, the value of those fields is not updated within the loop. This is demonstrated in the following loop of the method public void or(BitSet set) of library java.util.BitSet, where unitsInUse is a field of type int:

```
for(; i<set.unitsInUse; i++) bits[i]=set.bits[i];
```

Pattern #2: Loops in this category are similar to those in the previous category. The difference is that, rather than corresponding to the value of a numeric field, the bound of the loop counter corresponds to the length of an array which is stored in a field. In this case, even if the elements of the array may be updated within the loop, if the field itself does not, the length of the array remains constant. This is demonstrated in the following example, corresponding to method public void fixupVariables(java.util.Vector vars, int globalsSize) of library org. apache.xpath.functions.FunctionMultiArgs where m_args is a field of type Expression[ ]:

```
for(int i=0; i<m_args.length; i++) m_args[i].fixupVariables(vars,globalsSize);
```

Pattern #3: Loops in this category use numeric fields as loop counters, which means that the field value is updated within the loop, but none of the references in the *path* to the field (in this example, the chain just consists of the reference this) are re-assigned within the loop, i.e., all updates correspond to the same object on the heap. This is demonstrated in the following loop of the method public synchronized void setLength(int newLength) in the library java.lang.StringBuffer, in which count is a field of type int:

```
for(; count<newLength; count++) value[count] = '\0';
```

In this paper we concentrate on proving termination of loops that fall in the above categories by providing (uniform) conditions under which proving termination of such loops becomes possible. The Java libraries include also other patterns such as loops that: (1) increase/decrease an integer variable until it reaches a given upper/lower bound; (2) traverse a non-cyclical data structure or an array; (3) look for an element in an input stream, which is common in classes that manipulate structured text such as parsing XML documents; and (4) look for a non-null element in a given array in a circular way, which is very common in the multi-threading classes. The first two patterns are the major part of the loops, and they are already handled in [1]. The other patterns are planned for future research and are not addressed in this paper.

3 Dealing with Fields in Termination

In a Java-like language, objects are stored in the *heap* and they are accessed by means of references (or pointers). References can take the value null or *point* to an object in the heap. Given a reference l which points to an object o, $l.f$ denotes the value of the field f in the object o. We say that a syntactic construction of the form $l.f$ is a *field access*. Each field f has a unique signature, which consists of the class where it is declared, its type, and its name.

Objects are global in that they survive the execution of methods. Typically, when a method starts execution, a large number of objects may exist in the heap. One approach to analyzing programs with objects is to compute an abstraction of the heap (see [13]) which approximates the execution context of each method. This usually requires computing abstractions of all possible objects in the program, which might turn out to be too expensive in practice if one wants to deal with real programs. However, in most cases, only a small fraction of such objects affects the execution of the method. We seek for a more lightweight approach which tries to approximate the contents of only a subset of the objects in the heap. The approach must remain correct by making safe assumptions about the objects (and fields) whose contents are not taken into consideration.

Another disadvantage of computing an abstraction of the heap, in addition to its computational complexity, is that we end up obtaining termination information which is *context-dependent*. Though context dependent analysis is in principle more precise, the results obtained are not extrapolable to other execution contexts. In particular, in the case of libraries, ideally we would like to prove termination in a *context-independent* way, i.e., regardless of what the contents of the heap are when the method is executed.

We now introduce the concept of *local* field access. In particular, we are interested in finding field accesses which are local to a *loop*. Though termination analysis in our context aims at proving termination of methods, in the rest of the paper we will concentrate on *loops* since they are the main subject of termination analysis.

Definition 1 (local field access). *We say that a field access $l.r_1.\ldots.r_n.f$, where f is a numeric field, is* local *to a loop L if*

(i) No prefix of $l.r_1.\ldots.r_n$ changes its value within L, i.e., they remain constant.
(ii) If the value of $l.r_1.\ldots.r_n.f$ changes within L, then all write accesses have to be done explicitly through the field access $l.r_1.\ldots.r_n.f$.

Condition (i) guarantees that all occurrences of the field access within the loop refer to the same memory location in the heap. Note that the prefixes of $l.r_1.\ldots.r_n$, i.e., l, $l.r_1$, $l.r_1.r_2$, ... are references which altogether form a chain to an object where the numeric field f is stored. Condition (ii) guarantees that all write accesses to the field can be syntactically identified. Note that this condition can be violated due to aliasing, since we can have different field access which update the same memory location.

Given a loop L, we denote by *g-fields(L)* the set of field accesses $l.r_1 \ldots r_n.f$, where f is a numeric field, which explicitly appear inside the guard of L. For instance, for the three loops in Section 2, the sets *g-fields(L)* are, respectively, {this.unitsInUse}, {m_args.length} and {this.count}. These three fields are locally accessed within their corresponding loops. The practical implication is: if we ensure that a field in *g-fields*(L) is local, then we are able to treat this field in the same way as if it were a local variable, as regards the analysis of L. Essentially, given a loop L, the analysis proceeds as follows:

1. Compute the set *g-fields*(L).
2. Compute the set *l-g-fields*(L), which is the subset of *g-fields*(L) which contains the field accesses whose locality condition has been proved.
3. Analyze the termination of L by considering those field accesses in *l-g-fields*(L) as if they were local variables.

The method is applied locally to all nested loops in L. Note that the termination of a method is ensured if all loops *involved* in its body are terminating. By involved we mean not only those loops occurring explicitly in the body but also those coming from possible calls to some other methods.

3.1 Syntactic Inference of the Locality Condition on Field Accesses

The above approach is practical only if we provide effective mechanisms to prove the locality condition on field accesses. In this section, we consider only loops that do not contain method invocations. Later, in Section 4, we take method invocations into account. Now, we present sufficient *syntactic* conditions for ensuring that a field access is local. The following conditions ensure that a numeric field access $l.r_1.\ldots.r_n.f$ is local to a loop L:

1. The reference variable l remains constant in L. This can be ensured by checking that there is no assignment to l within L.
2. All reference fields $l.r_1, \ldots, l.r_1 \ldots r_n$ are constant in L. This can be ensured by checking that there is no assignment within L to a field with the same signature as any of r_i.
3. All assignments to a field with the same signature as f in L are done through the field access $l.r_1.\ldots.r_n.f$.

Let us briefly explain each of the above conditions. Conditions 1 and 2 ensure point (i) of Definition 1. The reason why we separate it into two conditions is due to the way in which it is syntactically checked in each case. For the reference variable l, we check that there is no assignment to it. These conditions guarantee that we do not incorrectly consider a loop of the form *while (l.size < 10) {l.size++; l=new C(); }* as terminating. Note that this loop is not guaranteed to terminate since l potentially changes the location of *size* and hence its value.

Condition 2 guarantees that we do not change any of the intermediary reference fields $l.r_1, \ldots, l.r_1 \ldots r_n$. Note that if we modify a reference field $l.r_1 \ldots r_i$ then

we fail to ensure constancy of the local field access. For instance, we would fail to prove termination of this loop *while (l.r_1.size < 10)* {*l.r_1.size++; l'.r_1=z;* }. This is a safe assumption, as without knowledge about the aliasing of l and l', we might be changing the reference to *size*.

Condition 3 is a sufficient condition to ensure that the field is not updated due to possible aliasing with another object (point (*ii*) in Definition 1). This condition is not satisfied in a loop of the form *while (l.size < 10)* {*l.size++; l'.size--;* } and therefore we do not prove termination for it. This is reasonable, as l and l' might be aliased during the execution.

Example 1. Reconsider the third loop in Section 2. For clarity, we replace the access to the field count to explicitly include the this path variable:

```
for(; this.count<newLength; this.count++) value[this.count] = '\0';
```

We can prove that this.count is local to the loop by checking the syntactic conditions stated above: the reference this does not change; and all updates to this.count are done through the field access this.count. The key point is that, since this.count is local, we can safely treat it as local variable. Consequently, existing termination analysers [3] are able to infer that this.count is increasing at each iteration. Besides, as newLength remains constant in the loop, the analyzer finds out that newLength-this.count is a decreasing well-founded measure and thus termination is guaranteed. □

4 Termination with (Virtual) Method Invocations

In this section, we address the more challenging problem of proving the termination of loops which contain method invocations. As notation, we denote by $M(L)$ the set of methods transitively invoked within the scope of a loop L. We now study what are the conditions that the methods in $M(L)$ must satisfy in order to preserve the locality condition on *g-fields*(L).

Consider a method m invoked within L, we distinguish three possible scenarios. In the first two ones, the implementation of m is available at analysis time and thus we can apply the techniques to detect local field accesses to the code in m. As our method is purely syntactic, in order to check the conditions on m, first we must do a *renaming* between the variables in the call and the formal parameters in m, as parameter passing does. Note that, when a method m is invoked from a reference l, the *this* reference in m is renamed to l in order to check the conditions. In the first scenario, method m does not modify the value of the (numeric) field, whereas in the second one it does. In the third one, the implementation of m either it is not available (i.e., it is an abstract or native method) or it has been redefined by means of subclassing. We aim at proving *modular* termination of the loop by making assumptions on m. We study these scenarios in more detail below.

Scenario 1. Consider method test_1 at the top of the right-hand column in Fig. 1. Due to dynamic dispatching, the execution of $\mathsf{a.m}_1()$ can correspond to method

```
class A {
  int f,g;
  int m1(){return 1;}
};
abstract class B extends A {
  int m1(){return 2;}
  void m2(){
    f = f + 1;
  }
  abstract void m3();
};
class C extends B {
  void m3() { g=g-1; }
};
```

```
void test1(A a,int k) {
  while (a.f < k) a.f = a.f + a.m1();
}
void test2(B b,int k) {
  while (b.f < k) b.m2();
}
void test3(B a,int k) {
  while (a.f < k){
     a.m3();
     a.f = a.f + a.m1();}
}
```

Fig. 1. Termination with fields and method invocations

m_1 in class A or to method m_1 in class B. Since, in both cases, the reference variable a remains constant and the field a.f is not updated within either implementation of m_1, we can guarantee that the field access a.f is local to (the loop in) test_1. Proving termination now is straightforward since both implementations of m_1 return a positive number.

Scenario 2. Now, we consider the case that, even if the field access is local to the loop, the field is updated during the execution of the invoked method. This happens, for example, in method test_2 where the call $\mathsf{b.m}_2\mathsf{()}$ increments the value of b.f. Indeed, method m_2 is responsible for the termination of test_2. In this case, we need to track the variations in the field b.f in an inter-procedural manner. One way to do it is by *inlining* the invoked method. However, this cannot always be done, as it is problematic for recursive methods. Another approach is to transform the methods in such a way that they carry as additional parameters the fields that must be tracked. When we have virtual invocations and several instances of the same method can be executed at runtime, we need to do such transformation to all the possible instances. Doing it at the level of Java would require a more sophisticated transformation, since parameters are passed by value. It could, however, be easily integrated in a termination analyzer like [1], as it works on an intermediate representation with permits multiple output parameters. We plan to develop this part in an extended version of this work.

Scenario 3. If the code of a method m in $M(L)$ is not available or the implementation of the method has been redefined, unfortunately we can say very little about the termination of L. For instance, if m is an abstract method, it is customary that the user defines a new class which implements m and it is always possible that it modifies the fields which affect the termination of the loop. Also, the new implementation might introduce callbacks which endanger termination. Clearly, one possibility is, once the implementation is available, to re-analyze

the loop with the new method. More interestingly, we can try to prove *modular* termination of the loop by assuming that (1) the method terminates, (2) it does not update any field access in *g-fields* and (3) it does not have callbacks. Once the new implementation is available, we actually have to ensure that the method m does not introduce a termination problem in L by checking the first two syntactic conditions in Sect. 3.1 as well as proving termination of m by applying our method to m again. For instance, consider method test_3, which is similar to test_1, but where a call to the (abstract) method m_3 has been added in the body of the loop. Assume that the class C is not available, then we make the assumption that m_3 is terminating and does not update a.f. Under these assumptions, we can prove modular termination of the loop. Consider now that the user defines class C at the bottom. Trivially, this method terminates and besides we can ensure that a.f is never updated from it. Note that, if the update inside m_3 was on f instead of on g, we would fail to ensure that that m_3 does not interfere with the guard. Indeed, the loop does not terminate in this case.

4.1 Method Invocations in the Java Libraries

It is common to find loops for scenarios 1 and 3 in the Java libraries. For instance, the loop of *Pattern* #2 of Section 2 is an example of scenario 3. The method fixupVariables invoked by m_args[i] is an abstract method of the library org.apache.xpath.Expression. The code is not available, thus we can only aim at proving termination modularly. We first make the assumption that fixupVariables will not introduce a termination problem in the loop. Under this assumption, we can prove termination of the loop. Note that, for actual implementation of fixupVariables, we will have to check that the local access condition holds and that it terminates.

We found many loops for scenario 1. For instance, the following loop appears in method public int indexOf(Object elem) of the library java.util.ArrayList:

```
for (int i = 0;i<size;i++)
    if (elem.equals(elementData[i])) return i;
```

where size is a field of type int. Its termination depends on the termination of the calls to elem.equals(elementData[i]), where elem and elementData[i] are objects of class java.lang.Object. The implementation of equals is available and contains as unique instruction return (this==obj), which ensures the local field access of size. Thus the loop is definitely terminating. It is rare in the libraries to find loops for scenario 2, indeed we have not found any. Though we believe it is necessary to provide solutions for them in order to handle the termination of user-defined programs which rely on the libraries and define methods which actually update the fields.

It is important to note that the solution we have proposed for this scenario is valid as long as the implementation of the missing methods does not use static fields. The reason for this is that static fields can be, similarly to global variables, used in the code without being passed as arguments to the method. Therefore, the set of classes reachable from a method signature, as obtained by

the procedure above, is not guaranteed to be a safe approximation of the actual classes reached by execution in the presence of static fields.

5 Perspectives for Future Work

The state of the practice in termination analysis is moving beyond less-widely used programming languages to realistic *object-oriented* languages. This paper draws attention to some difficulties that need to be solved if object *fields* are to be supported by termination analyzers. In particular, tracking size information becomes rather complex, and accurate *aliasing* information is required in order to track all possible updates of the corresponding fields. Motivated by examples found in the Java libraries, we have proposed some ideas towards dealing with numeric fields in a practical manner. The perspectives on the application of our technique include to infer *termination annotations* for as many methods in the Java libraries as possible. Applying termination tools on realistic programs which use libraries is a challenging problem, as there are many dependencies between the library classes and, in our experience, even small applications require analyzing a high number of library methods. By using precomputed annotations, the analyzer can safely assume the termination of those annotated methods in the Java libraries (and those that they depend upon).[3]

Although our ideas have not been experimentally evaluated yet, we believe that most of the patterns found in the libraries match those presented in Sec. 2. We, nevertheless, plan to improve the accuracy of the analysis in order to cover a broader range of patterns. For instance, as a starting point, we have proposed to check the local field access condition on those fields which appear explicitly in the guards, denoted *g-fields*. There are, of course, other possibilities and enhancements:

- Ideally, we should try to prove the locality condition not only on *g-fields*, but also on those fields which may interact with *g-fields*. For instance, in a loop of the form *while (l.size < 10)* {*l.size+= l'.size;* }, unless we track some information about $l'.size$ (in this case, its sign would suffice), we will fail to prove termination. Unfortunately, it is not always trivial to determine the minimal set of fields which may interact with *g-fields*. In particular, a simple syntactic inspection is not enough.
- To simplify the above point, another idea would be to try and prove the locality condition on *all* fields which appear inside the scope of the loop. This approach would be in general more accurate (e.g., would solve the above problem) but more expensive. Importantly, even if not all fields are local to the loop, the termination analysis proceeds (step 3 in Sect. 3). As long as the non-local field accesses do not affect the termination behaviour, the analysis can still succeed to prove termination.

[3] Note that precomputed assertions are valid as long as the user does not redefine methods which have been used (and analysed) to infer the assertions.

– Another interesting refinement is to consider not only the fields which appear explicit in the guards but also those which are accessed through *getter* methods like *while (l.getSize() < 10)* {...}. For this, we should go through the code of the methods invoked in the loop guards and identify those fields. A simple solution to this problem is inlining the method. Afterwards, the same basic techniques explained in the paper could be applied.

It can be seen that in some cases there is an accuracy vs efficiency tradeoff and also that, what it is optimal for one example might not be good for others. We need to perform experimental evaluation to assess the different options.

From an implementation perspective, we plan to enhance the COSTA system [3] with the ideas presented in this paper. COSTA is a cost and termination analyzer which works directly on the bytecode (and has no knowledge about the source Java). The termination module is based on the techniques proposed in [1] and the cost module on the method described in [2]. To carry out the implementation, the first issue is to incorporate the syntactic conditions to prove whether fields are accessed locally. Condition 3 can be easily checked on the bytecode by seeing that there is no `putfield` to the corresponding field signatures. Checking that the object does not change (conditions 1 and 2) requires to track dependencies between stack variables and local variables. This happens because, in the bytecode, the access to a field is done by first pushing the variable (on which the condition is to be checked) to the stack and then the field is accessed from the stack variable. This check can be done syntactically in most cases due to the elimination of stack variables [4]. Once the syntactic conditions are checked, we will implement the extensions to treat fields as local variables during analysis. This is straightforward to do in COSTA, as the tool converts the bytecode into a rule-based representation where the local variables (and the stack positions) appear as arguments of these rules. We can just add the required fields as additional arguments to them. Size analysis will directly treat them as it does with local variables in order to infer how they increase/decrease over the program.

References

1. E. Albert, P. Arenas, M. Codish, S. Genaim, G. Puebla, and D. Zanardini. Termination Analysis of Java Bytecode. In *FMOODS*, LNCS. Springer-Verlag, 2008.
2. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of java bytecode. In *ESOP'07*, LNCS, 2007.
3. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. COSTA: A Cost and Termination Analyzer for Java Bytecode. In *Proc. of BYTECODE Workshop*, ENTCS. Elsevier, 2008. To appear.
4. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Removing Useless Variables in Cost Analysis of Java Bytecode. In *Proc. SAC*. ACM Press, 2008.
5. A. Bradley, Z. Manna, and H. Sipma. Termination of polynomial programs. In *VMCAI*, 2005.
6. M. Codish and C. Taboch. A semantic basis for the termination analysis of logic programs. *J. Log. Program.*, 41(1):103–123, 1999.

7. B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI*, 2006.
8. J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic Termination Proofs in the Dependency Pair Framework. In *IJCAR*, 2006.
9. P. Hill, E. Payet, and F. Spoto. Path-length analysis of object-oriented programs. In *Proc. EAAI*. Elsevier, 2006.
10. C. Lee, N. Jones, and A. Ben-Amram. The size-change principle for program termination. In *Proc. POPL*. ACM, 2001.
11. N. Lindenstrauss and Y. Sagiv. Automatic termination analysis of logic programs. In *ICLP*, 1997.
12. C. Marché and C. Paulin-Mohring. Reasoning about java programs with aliasing and frame conditions. In J. Hurd and T. F. Melham, editors, *TPHOLs*, volume 3603 of *Lecture Notes in Computer Science*, pages 179–194. Springer, 2005.
13. A. Miné. Field-sensitive value analysis of embedded c programs with union types and pointer arithmetics. In M. J. Irwin and K. D. Bosschere, editors, *LCTES*, pages 54–63. ACM, 2006.
14. A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI*, 2004.