

# NUMERIC FIELDS IN TERMINATION ANALYSIS OF JAVA-LIKE LANGUAGES

**Elvira Albert**<sup>(1)</sup>, Puri Arenas<sup>(1)</sup>, Samir Genaim<sup>(2)</sup>,  
Germán Puebla<sup>(2)</sup>

(1) DSIC, COMPLUTENSE UNIVERSITY OF MADRID

(2) CLIP, TECHNICAL UNIVERSITY OF MADRID

**10th Workshop on Formal Techniques  
for Java-like Programs FTfJP 2008**

8 July, 2008

# The Problem

- **Termination analysis**: find termination proofs for as wide a class of (terminating) programs as possible.
- **How?**: study of **loops** which are the program constructs which may introduce non-termination.

# The Problem

- **Termination analysis**: find termination proofs for as wide a class of (terminating) programs as possible.
  - **How?**: study of **loops** which are the program constructs which may introduce non-termination.
- ⇒ Track *size* information:

```
while (l!=null) l=l.next;
```

how the (size of the) data involved in loop guards changes when the loop goes through its iterations

# The Problem

- **Termination analysis**: find termination proofs for as wide a class of (terminating) programs as possible.
- **How?**: study of **loops** which are the program constructs which may introduce non-termination.

⇒ Track *size* information:

```
while (l!=null) l=l.next;
```

how the (size of the) data involved in loop guards changes when the loop goes through its iterations

⇒ Specify **ranking function**: function which strictly decreases at each iteration of the loop

```
size(l) is a ranking function
```

# The Problem

- **Termination analysis**: find termination proofs for as wide a class of (terminating) programs as possible.
- **How?**: study of **loops** which are the program constructs which may introduce non-termination.

⇒ Track *size* information:

```
while (l!=null) l=l.next;
```

how the (size of the) data involved in loop guards changes when the loop goes through its iterations

⇒ Specify **ranking function**: function which strictly decreases at each iteration of the loop

`size(l)` is a ranking function

- **The problem**: termination behavior affected by *numeric fields*.  
`while (i < n) { i++; o.m(); }`     `n-i` is a ranking function

# The Problem

- **Termination analysis**: find termination proofs for as wide a class of (terminating) programs as possible.
- **How?**: study of **loops** which are the program constructs which may introduce non-termination.

⇒ Track **size** information:

```
while (l!=null) l=l.next;
```

how the (size of the) data involved in loop guards changes when the loop goes through its iterations

⇒ Specify **ranking function**: function which strictly decreases at each iteration of the loop

`size(l)` is a ranking function

- **The problem**: termination behavior affected by *numeric fields*.

`while (i < n) { i++; o.m(); }`      `n-i` is a ranking function

`while (i < f.n) { i++; o.m(); }`      `f.n-i` ranking function?

# Termination Analysis and Numeric Fields

- Difficulties in termination analysis of OO languages: exceptions, virtual invocation, references, heap-allocated data-structures, objects, fields.

- Difficulties in termination analysis of OO languages: exceptions, virtual invocation, references, **heap-allocated data-structures**, objects, fields.
- Why the heap poses problems to static analysis?
  - **global** data structure accessed using (chained) references,
  - same location modified using different **aliased** references
  - references may point to different locations during execution

# Termination Analysis and Numeric Fields

- Difficulties in termination analysis of OO languages: exceptions, virtual invocation, references, **heap-allocated data-structures**, objects, fields.
- Why the heap poses problems to static analysis?
  - **global** data structure accessed using (chained) references,
  - same location modified using different **aliased** references
  - references may point to different locations during execution
- What is the consequence?
  - track how (size of) data involved in loop guards changes
  - when guards involve information stored in the heap **tracking size** information rather complex,
  - **aliasing** information required to track updates of fields

- Termination analyzers handle a good number of the features:
  - **Costa**: (cost and) termination analyzer for Java bytecode
  - **Julia**: incorporates the **path-length** domain for reference fields

- Termination analyzers handle a good number of the features:
  - **Costa**: (cost and) termination analyzer for Java bytecode
  - **Julia**: incorporates the **path-length** domain for reference fields
- **Path-length**:
  - Prove termination of loops which traverse **acyclic** heap-allocated data structures (i.e., linked lists, trees, etc.).
  - Abstract domain for reference values
  - Does not capture any information about numeric fields

- Termination analyzers handle a good number of the features:
  - **Costa**: (cost and) termination analyzer for Java bytecode
  - **Julia**: incorporates the **path-length** domain for reference fields
- **Path-length**:
  - Prove termination of loops which traverse **acyclic** heap-allocated data structures (i.e., linked lists, trees, etc.).
  - Abstract domain for reference values
  - Does not capture any information about numeric fields
- **Our Goal**:
  - estimate how often loop termination depends on *numeric* fields in the Sun implementation of the Java libraries for **J2SE 1.4.2**
  - propose **sufficient conditions** for termination to cover a large fraction of those loops not provable using current techniques

# Motivating Examples from the Java Libraries

- Techniques for proving termination provide **sufficient** (but not necessary) conditions for termination
- Practicality of termination analyses is measured by applying the analyses to a **representative set** of real programs

# Motivating Examples from the Java Libraries

- Techniques for proving termination provide **sufficient** (but not necessary) conditions for termination
- Practicality of termination analyses is measured by applying the analyses to a **representative set** of real programs
- **Our design** of the analysis 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 three kinds of **common patterns** in the Java libraries.

- **Pattern #1:** numeric fields as bounds for loop counters  
for(; i<set.unitsInUse; i++) bits[i]=set.bits[i];  
library java.util.BitSet, where unitsInUse is an int field

# Patterns in Libraries

- **Pattern #1:** numeric fields as bounds for loop counters  
for(; i<set.unitsInUse; i++) bits[i]=set.bits[i];  
library java.util.BitSet, where unitsInUse is an int field
- **Pattern #2:** the bound of the loop counter corresponds to the length of an array which is stored in a field  
for(int i=0; i<m\_args.length; i++)  
    m\_args[i].fixupVariables(vars,globalsSize);  
library org.apache.xpath.functions.FunctionMultiArgs  
where m\_args is a field of type Expression[ ]

# Patterns in Libraries

- **Pattern #1:** numeric fields as bounds for loop counters  
`for(; i<set.unitsInUse; i++) bits[i]=set.bits[i];`  
library `java.util.BitSet`, where `unitsInUse` is an `int` field
- **Pattern #2:** the bound of the loop counter corresponds to the length of an array which is stored in a field  
`for(int i=0; i<m_args.length; i++)`  
`m_args[i].fixupVariables(vars,globalsSize);`  
library `org.apache.xpath.functions.FunctionMultiArgs`  
where `m_args` is a field of type `Expression[ ]`
- **Pattern #3:** numeric fields as loop counters: the field value is updated but none of the references in the *path* are re-assigned  
`for(; count<newLength; count++) value[count] = '\0';`  
library `java.lang.StringBuffer`, with `count` an `int` field

- **The Context-Dependent Approach:**
  - computes abstractions of all possible objects in the program
  - too expensive in practice to deal with real programs
  - obtains *context-dependent* termination information
  - more precise but the results not extrapolable to other contexts

- **The Context-Dependent Approach:**
  - computes abstractions of all possible objects in the program
  - too expensive in practice to deal with real programs
  - obtains *context-dependent* termination information
  - more precise but the results not extrapolable to other contexts

**Libraries:** ideally we would like to prove termination *context-independent*, i.e., regardless of what the contents of the heap are when the method is executed

# Context-Dependent vs Context-Independent Analysis

- **The Context-Dependent Approach:**

- computes abstractions of all possible objects in the program
- too expensive in practice to deal with real programs
- obtains *context-dependent* termination information
- more precise but the results not extrapolable to other contexts

**Libraries:** ideally we would like to prove termination *context-independent*, i.e., regardless of what the contents of the heap are when the method is executed

- **The Context-Independent Approach:**

- only a small fraction of objects usually affects the execution
- a lightweight approach to approximate the contents of only such subset of objects in the heap
- correct by making safe assumptions about the objects (and fields) whose contents are not taken into consideration

- **Goal:** finding field accesses which are local to a *loop*  $L$ .
- A field access  $l.r_1 \dots r_n.f$  is *local* to a loop if

- **Goal:** finding field accesses which are local to a *loop*  $L$ .
- A field access  $l.r_1 \dots r_n.f$  is *local* to a loop if
  - (i) No prefix of  $l.r_1 \dots r_n$  changes its value within  $L$

- **Goal:** finding field accesses which are local to a *loop*  $L$ .
- A field access  $l.r_1\dots r_n.f$  is *local* to a loop if
  - (i) No prefix of  $l.r_1\dots r_n$  changes its value within  $L$
  - (ii) If the value of  $l.r_1\dots r_n.f$  changes within  $L$ , then all **write accesses** are explicitly through  $l.r_1\dots r_n.f$ .
- (i) guarantees that all occurrences of the field access refer to the same memory location in the heap
- (ii) guarantees that all write accesses to the field can be syntactically identified

- **Goal:** finding field accesses which are local to a *loop*  $L$ .
  - A field access  $l.r_1\dots r_n.f$  is *local* to a loop if
    - (i) No prefix of  $l.r_1\dots r_n$  changes its value within  $L$
    - (ii) If the value of  $l.r_1\dots r_n.f$  changes within  $L$ , then all **write accesses** are explicitly through  $l.r_1\dots r_n.f$ .
  - (i) guarantees that all occurrences of the field access refer to the same memory location in the heap
  - (ii) guarantees that all write accesses to the field can be syntactically identified
- Practical implication**: if we ensure that a field is **local**, then we can treat it as if it was a **local variable**.

# The Role of the Locality Condition in Termination

- Given a loop  $L$ , we denote by  $g\text{-fields}(L)$  the set of (numeric) field accesses  $l.r_1 \dots r_n.f$  which appear in the guard of  $L$ .

# The Role of the Locality Condition in Termination

- Given a loop  $L$ , we denote by  $g\text{-fields}(L)$  the set of (numeric) field accesses  $l.r_1 \dots r_n.f$  which appear in the guard of  $L$ .
- The termination analysis is as follows:
  - ① Compute the set  $g\text{-fields}(L)$ .
  - ② Compute the set  $l\text{-}g\text{-fields}(L)$ : the subset of  $g\text{-fields}(L)$  whose locality condition has been proved.
  - ③ Analyze the termination of  $L$  by considering those field accesses in  $l\text{-}g\text{-fields}(L)$  as if they were local variables.

# The Role of the Locality Condition in Termination

- Given a loop  $L$ , we denote by  $g\text{-fields}(L)$  the set of (numeric) field accesses  $l.r_1 \dots r_n.f$  which appear in the guard of  $L$ .
- The termination analysis is as follows:
  - 1 Compute the set  $g\text{-fields}(L)$ .
  - 2 Compute the set  $l\text{-g-fields}(L)$ : the subset of  $g\text{-fields}(L)$  whose locality condition has been proved.
  - 3 Analyze the termination of  $L$  by considering those field accesses in  $l\text{-g-fields}(L)$  as if they were local variables.
- The method is applied locally to all nested loops in  $L$ .
- Termination ensured if all loops *involved* are terminating.
- *Involved* means not only those loops explicit in the body but also those coming from possible method calls.

# Syntactic Inference of the Locality Condition

- Approach practical only if we provide effective mechanisms to prove the locality condition on field accesses
- Sufficient **syntactic conditions** to ensure that a numeric field access  $l.r_1.\dots.r_n.f$  is local to  $L$ :

# Syntactic Inference of the Locality Condition

- Approach practical only if we provide effective mechanisms to prove the locality condition on field accesses
- Sufficient **syntactic conditions** to ensure that a numeric field access  $l.r_1\dots r_n.f$  is local to  $L$ :
  - ① The reference variable  $l$  remains constant in  $L$   
⇒ check there is no assignment to  $l$  within  $L$ .
  - ② All reference fields  $l.r_1, \dots, l.r_1\dots r_n$  are constant in  $L$   
⇒ check there is no assignment to a field with signature  $r_i$
  - ③ All assignments to a field with the same signature as  $f$  in  $L$  are done through the field access  $l.r_1\dots r_n.f$

# Syntactic Inference of the Locality Condition

- Approach practical only if we provide effective mechanisms to prove the locality condition on field accesses
- Sufficient **syntactic conditions** to ensure that a numeric field access  $l.r_1\dots r_n.f$  is local to  $L$ :
  - ① The reference variable  $l$  remains constant in  $L$   
⇒ check there is no assignment to  $l$  within  $L$ .
  - ② All reference fields  $l.r_1, \dots, l.r_1\dots r_n$  are constant in  $L$   
⇒ check there is no assignment to a field with signature  $r_i$
  - ③ All assignments to a field with the same signature as  $f$  in  $L$  are done through the field access  $l.r_1\dots r_n.f$
- Condition 1 guarantees that we do not consider this loop terminating: `while (l.size < 10) {l.size++; l=new C(); }`

# Syntactic Inference of the Locality Condition

- Approach practical only if we provide effective mechanisms to prove the locality condition on field accesses
- Sufficient **syntactic conditions** to ensure that a numeric field access  $l.r_1 \dots r_n.f$  is local to  $L$ :
  - ① The reference variable  $l$  remains constant in  $L$   
⇒ check there is no assignment to  $l$  within  $L$ .
  - ② All reference fields  $l.r_1, \dots, l.r_1 \dots r_n$  are constant in  $L$   
⇒ check there is no assignment to a field with signature  $r_i$
  - ③ All assignments to a field with the same signature as  $f$  in  $L$  are done through the field access  $l.r_1 \dots r_n.f$
- Condition 1 guarantees that we do not consider this loop terminating:  $\text{while } (l.size < 10) \{ l.size++; l = \text{new } C(); \}$
- Condition 2 guarantees that we do not consider this loop terminating:  $\text{while } (l.r_1.size < 10) \{ l.r_1.size++; l'.r_1 = z; \}$

# Syntactic Inference of the Locality Condition

- Approach practical only if we provide effective mechanisms to prove the locality condition on field accesses
- Sufficient **syntactic conditions** to ensure that a numeric field access  $l.r_1 \dots r_n.f$  is local to  $L$ :
  - ① The reference variable  $l$  remains constant in  $L$   
⇒ check there is no assignment to  $l$  within  $L$ .
  - ② All reference fields  $l.r_1, \dots, l.r_1 \dots r_n$  are constant in  $L$   
⇒ check there is no assignment to a field with signature  $r_i$
  - ③ All assignments to a field with the same signature as  $f$  in  $L$  are done through the field access  $l.r_1 \dots r_n.f$
- Condition 1 guarantees that we do not consider this loop terminating:  $\text{while } (l.size < 10) \{ l.size++; l = \text{new } C(); \}$
- Condition 2 guarantees that we do not consider this loop terminating:  $\text{while } (l.r_1.size < 10) \{ l.r_1.size++; l'.r_1 = z; \}$
- Condition 3 is not satisfied in a loop of the form  $\text{while } (l.size < 10) \{ l.size++; l'.size--; \}$

# An Example

- Consider the previous loop:

```
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:

# An Example

- Consider the previous loop:  
    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;
  - all updates to `this.count` are done through `this.count`

# An Example

- Consider the previous loop:  
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;
  - all updates to `this.count` are done through `this.count`
- The key point is that we can safely treat `this.count` as a local variable and hence:
  - Existing termination analyzers are able to infer that `this.count` is increasing and `newLength` constant at each iteration
  - Thus,  $\boxed{\text{newLength} - \text{this.count}}$  is a decreasing well-founded measure and thus termination is guaranteed.

# Termination with (Virtual) Method Invocations

- Conditions that methods in a loop,  $M(L)$ , must satisfy in order to preserve the locality condition on  $g\text{-fields}(L)$
- We distinguish three possible **scenarios**:

# Termination with (Virtual) Method Invocations

- Conditions that methods in a loop,  $M(L)$ , must satisfy in order to preserve the locality condition on  $g\text{-fields}(L)$
- We distinguish three possible **scenarios**:
  - ① The implementation of  $m$  is **available** at analysis time and  $m$  does **not modify** the value of the (numeric) field
  - ② The implementation of  $m$  is **available** at analysis time and  $m$  **modifies** the value of the (numeric) field
  - ③ The implementation of  $m$  either it is **not available** or it has been redefined by means of subclassing.

# Scenario 1

- Consider the following classes which define a method  $m_1$ :

```
class A {  
    int  $m_1$ () {return 1;}  
    ... };
```

<pre>class B extends A {     int <math>m_1</math>() {return 2;}     ... };</pre>
--

# Scenario 1

- Consider the following classes which define a method  $m_1$ :

```
class A {  
    int  $m_1()$ {return 1;}  
    ... };  
|  
class B extends A {  
    int  $m_1()$ {return 2;}  
    ... };
```

- We want to prove termination of the following method:

```
void  $test_1(A a, int k)$  {  
    while ( $a.f < k$ )  $a.f = a.f + a.m_1();$ }
```

# Scenario 1

- Consider the following classes which define a method  $m_1$ :

```
class A {  
    int  $m_1()$ {return 1;}  
    ... };  
|  
class B extends A {  
    int  $m_1()$ {return 2;}  
    ... };
```

- We want to prove termination of the following method:  

```
void  $test_1(A a, int k)$  {  
    while ( $a.f < k$ )  $a.f = a.f + a.m_1();$ }
```
- 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  $test_1$
- Proving termination now is straightforward

- Consider the following implementation of method  $m_2$   
class B extends A {  
    void  $m_2()$ {  
         $f = f + 1$ ;  
    }

## Scenario 2

- Consider the following implementation of method  $m_2$   
class B extends A {  
    void  $m_2()$ {  
         $f = f + 1$ ;  
    }  
}
- $m_2$  is responsible for the termination of  $test_2$ :  
void  $test_2(B\ b, int\ k)$  {  
    while ( $b.f < k$ )  $b.m_2();$ }

## Scenario 2

- Consider the following implementation of method  $m_2$   
class B extends A {  
    void  $m_2()$ {  
         $f = f + 1$ ;  
    }  
}
- $m_2$  is responsible for the termination of  $test_2$ :  
    void  $test_2(B\ b, int\ k)$  {  
        while ( $b.f < k$ )  $b.m_2()$ ;}  
    }
- Track variations in  $b.f$  in an **inter-procedural** manner.
- **Inlining** cannot always be done (problematic for recursion).

## Scenario 2

- Consider the following implementation of method  $m_2$   
class B extends A {  
    void  $m_2()$ {  
         $f = f + 1$ ;  
    }  
}
- $m_2$  is responsible for the termination of  $test_2$ :  
    void  $test_2(B\ b, int\ k)$  {  
        while ( $b.f < k$ )  $b.m_2();$ }
- Track variations in  $b.f$  in an **inter-procedural** manner.
- **Inlining** cannot always be done (problematic for recursion).
- Transform the methods to carry as **additional parameters** the fields that must be tracked.
  - At the level of Java requires a sophisticated transformation, since parameters are passed by value.
  - We can have intermediate representations (bytecode) with permit multiple output parameters.

## Scenario 3

- If the code of a method  $m$  is **not available** or the implementation of the method has been **redefined**:

# Scenario 3

- If the code of a method  $m$  is **not available** or the implementation of the method has been **redefined**:
  - e.g., if  $m$  is **abstract**, the user will usually implement  $m$  which might modify the fields that affect termination
  - also, the new implementation might introduce **callbacks** which endanger termination

## Scenario 3

- If the code of a method  $m$  is **not available** or the implementation of the method has been **redefined**:
  - e.g., if  $m$  is **abstract**, the user will usually implement  $m$  which might modify the fields that affect termination
  - also, the new implementation might introduce **callbacks** which endanger termination
- One possibility is, once the implementation is available, to **re-analyze** the loop with the new method

# Scenario 3

- If the code of a method  $m$  is **not available** or the implementation of the method has been **redefined**:
  - e.g., if  $m$  is **abstract**, the user will usually implement  $m$  which might modify the fields that affect termination
  - also, the new implementation might introduce **callbacks** which endanger termination
- 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:
  - 1 the method terminates,
  - 2 it does not update any field access in *g-fields*,
  - 3 it does not have callbacks.

# Scenario 3

- If the code of a method  $m$  is **not available** or the implementation of the method has been **redefined**:
  - e.g., if  $m$  is **abstract**, the user will usually implement  $m$  which might modify the fields that affect termination
  - also, the new implementation might introduce **callbacks** which endanger termination
- 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:
  - 1 the method terminates,
  - 2 it does not update any field access in *g-fields*,
  - 3 it does not have callbacks.
- Once the new implementation is available, we check the first two syntactic conditions on  $m$
- Then, we apply our method to  $m$  to prove that it does not introduce a termination problem

# Method Invocations in the Java Libraries

- It is common to find loops for scenario 3 in the Java libraries  
for(int i=0; i<m\_args.length; i++)  
    m\_args[i].fixupVariables(vars,globalsSize);

# Method Invocations in the Java Libraries

- It is common to find loops for scenario 3 in the Java libraries

```
for(int i=0; i<m_args.length; i++)
```

```
    m_args[i].fixupVariables(vars,globalsSize);
```

the method `fixupVariables` is an abstract method.

- Make assumption that `fixupVariables` does not introduce a termination problem and prove termination
- For actual implementation of `fixupVariables`, we have to check that the local access condition holds and that it terminates.

# Method Invocations in the Java Libraries

- It is common to find loops for scenario 3 in the Java libraries  
for(int i=0; i<m\_args.length; i++)  
    m\_args[i].fixupVariables(vars,globalsSize);  
the method `fixupVariables` is an abstract method.
  - Make assumption that `fixupVariables` does not introduce a termination problem and prove termination
  - For actual implementation of `fixupVariables`, we have to check that the local access condition holds and that it terminates.
- We found many loops for scenario 1.  
    for (int i = 0;i<size;i++)  
        if (elem.equals(elementData[i])) return i;  
in method `public int indexOf(Object elem)` of the library `java.util.ArrayList` and `size` is a field of type `int`.

# Method Invocations in the Java Libraries

- It is common to find loops for scenario 3 in the Java libraries  

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

the method `fixupVariables` is an abstract method.

  - Make assumption that `fixupVariables` does not introduce a termination problem and prove termination
  - For actual implementation of `fixupVariables`, we have to check that the local access condition holds and that it terminates.
- We found many loops for scenario 1.  

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

in method `public int indexOf(Object elem)` of the library `java.util.ArrayList` and `size` is a field of type `int`.

  - The implementation of `equals` is available and contains as unique instruction `return (this==obj)`
  - It ensures the local field access of `size` and thus the loop is definitely terminating.

# Method Invocations in the Java Libraries

- It is common to find loops for scenario 3 in the Java libraries  

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

the method `fixupVariables` is an abstract method.

  - Make assumption that `fixupVariables` does not introduce a termination problem and prove termination
  - For actual implementation of `fixupVariables`, we have to check that the local access condition holds and that it terminates.
- We found many loops for scenario 1.  

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

in method `public int indexOf(Object elem)` of the library `java.util.ArrayList` and `size` is a field of type `int`.

  - The implementation of `equals` is available and contains as unique instruction `return (this==obj)`
  - It ensures the local field access of `size` and thus the loop is definitely terminating.
- It is rare in the libraries to find loops for scenario 2

- State-of-the-practice in termination analysis moving beyond *less-widely used* programming languages to realistic *object-oriented* languages
- This work draws attention to some difficulties that need to be solved to support *fields* in termination analysis
- Motivated by examples found in the *Java libraries*, we have proposed some syntactic techniques towards dealing with numeric fields in a practical manner

# Conclusions & Perspectives

- State-of-the-practice in termination analysis moving beyond *less-widely used* programming languages to realistic *object-oriented* languages
- This work draws attention to some difficulties that need to be solved to support *fields* in termination analysis
- Motivated by examples found in the *Java libraries*, we have proposed some syntactic techniques towards dealing with numeric fields in a practical manner
- **Perspectives:** apply termination tools on realistic programs which use libraries is challenge due to many dependencies
- 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)

- **COSTA**: COST and Termination Analyzer which works directly on the bytecode (no knowledge about the Java).
  - Termination module based on techniques in **FMOODS'08**
  - Cost module based on the method in **ESOP'07**
- Enhance **COSTA** with the ideas presented here
- Tool demonstration of **COSTA** at ECOOP08  
(Wednesday at 16.15)  
(Thursday at 10.45)