

Measure extraction for termination proofs in Stainless

Rodrigo Raya

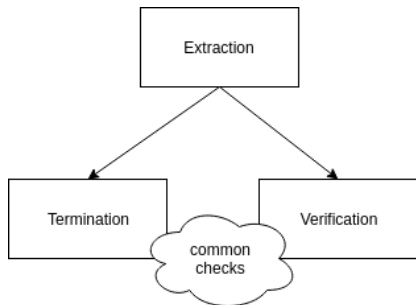
École Polytechnique Fédérale de Lausanne

September 26, 2018

- 1 Termination analysis vs measure extraction
- 2 Measure annotation
- 3 Checking measure verification conditions
- 4 While loop annotations
- 5 Final thoughts
- 6 References

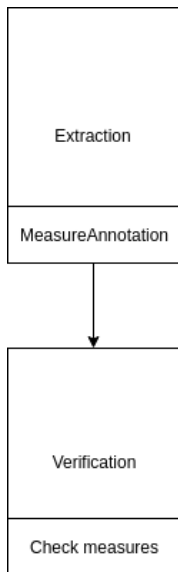
- 1 Termination analysis vs measure extraction
- 2 Measure annotation
- 3 Checking measure verification conditions
- 4 While loop annotations
- 5 Final thoughts
- 6 References

Termination analysis



- Not suitable for the new framework based on System T.
- We rely on the correctness of the analysis in the termination phase.
- There are repeated checks in both phases: termination and verification (namely positivity check for measures).

Measure extraction



- A method first proposed by Turing and later by Floyd.
- We perform the analysis as before but produce a witness called measure that certifies the termination of each function.
- The witness produced can be checked in the verification phase.
- No repeated checks/increased time for the verification process.

- Formally, a map from program state to a well-founded domain.
- To simplify, we can take the well-founded domain to be $(\mathbb{N}^I, <)$ with $<$, the lexicographic order on natural tuples.
- The program state can be modelled by a domain of values $D_1 \times \dots \times D_I$. We will write D^I .
- So a measure is a map $m : D^I \rightarrow \mathbb{N}^t$.
- Suppose that at each function call we can say that some measure decreases. Then we get a chain:

$$m_1 > m_2 > \dots m_n > \dots$$

and well-foundedness implies that chain has to be finite.

- Thus, the program terminates.

Semantics of decreases

- To check the measure decrease and positivity a 'decreases' clause is written around measures.
- For each measure we have to check that its arguments are in \mathbb{N}^t and that it decreases at each function call.
- The goal of the extraction phase is to annotate **every** function with a 'decreases' clause.

Examples: things Stainless cannot do

- Ackermann's function with reversed arguments

```
def ackermann(n: BigInt, m: BigInt): BigInt = {  
  require(m >= 0 && n >= 0)  
  if (m == 0) n + 1  
  else if (n == 0) ackermann(1,m-1)  
  else ackermann(ackermann(n - 1,m),m-1)  
} ensuring (_ >= 0)
```

- Stainless will try: decreases(n,m)
- A suitable measure: decreases(m,n)

Examples: things Stainless cannot do

- In rare examples it may not have a clue:

```
def mn(x: BigInt, y: BigInt): BigInt = {  
  require(x >= 0 && y >= 0)  
  
  if(x == 0 || y == 0){  
    BigInt(0)  
  } else {  
    if(x < y) mn(x-1,y)  
    else mn(x+1,y-1)  
  }  
}
```

- A suitable measure: $\text{decreases}(\min\{x,y\})$

Examples: things Stainless cannot do

- Most of the time, you really need to confuse the solver to avoid it proving termination:

```
def f(x: BigInt, y: BigInt): BigInt = {  
  require(x >= 0 && y >= 0)  
  
  if(x == 0 || y == 0){  
    BigInt(0)  
  } else {  
    if(ackermann(x,y) == 1) f(y,y-1)  
    else f(y,x-1)  
  }  
}
```

- A suitable measure: $\text{decreases}(\max(2 \cdot x - 1, 2 \cdot y))$

- 1 Termination analysis vs measure extraction
- 2 Measure annotation**
- 3 Checking measure verification conditions
- 4 While loop annotations
- 5 Final thoughts
- 6 References

Size-change termination principle

- Most of the framework is based on the size-change termination principle.
- This methodology builds a 'size' abstraction for the program based on the function:

$$\text{size}(x) = \begin{cases} \sum_{i=1}^N \text{size}(x_i) & \text{if } x : \text{Tuple}_N \\ 1 + \sum_{i=1}^N \text{size}(x.\text{field}_i) & \text{if } x : \text{ADT}_N \\ |x| & \text{if } x : \text{Integer} \\ |x|' & \text{if } x : \text{BitVector} \\ 0 & \text{otherwise} \end{cases}$$

- To reason about sequences of arguments in the program you need to introduce an ordering relation: $<_{\text{sum}}, <_{\text{lex}}, <_{\text{bv}}$
- Beware! $<_{\text{bv}}$ is unsound when mixing different bit-vectors types.

Bit vector ordering potential unsoundness

- Something like this leads to unsoundness (after correcting bvAbs in StructuralSize):

```
def f(l: Long, i: Int): Long = {  
  require(l >= 1L && i >= 1 && i <= 100 && l <= 100L)  
  if(i == 100) g(l - 1L,i)  
  else g(l - 1L,i + 1)  
}  
  
def g(l: Long,i: Int): Long = {  
  require(l >= 0L && i >= 2 && i <= 100 && l <= 100L)  
  if(l == 100L) f(l,i - 1)  
  else f(l + 1L, i - 1)  
}
```

- Mixing different bit-vector types is not so usual.
- Alternative orderings are weaker.

Processors and orders

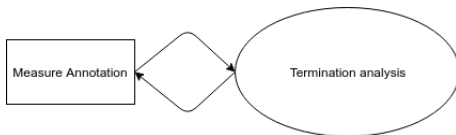
- Currently, Stainless has four termination analysis and three orderings that are combined in the following way:

Processor	Ordering
Recursion	Integer
Relation	Integer, Lexicographic, Bit-vector
Chain	Integer
Decreases	Integer
Loop	Integer

- Recursion processor looks for recursive calls using subfields.
- Relation processor looks for \leq and $<$ relations.
- Chain processor analyses chains including $>$ relations.
- Decreases processor checks the decreases annotations.
- Loop processor shows non-termination.

Reusing the termination analysis framework

- The approach is to reuse the framework for termination analysis.
- For each analysis we study what measures should be annotated.
- We store the derived measures and add them to the trees in the extraction phase.



Measure annotation

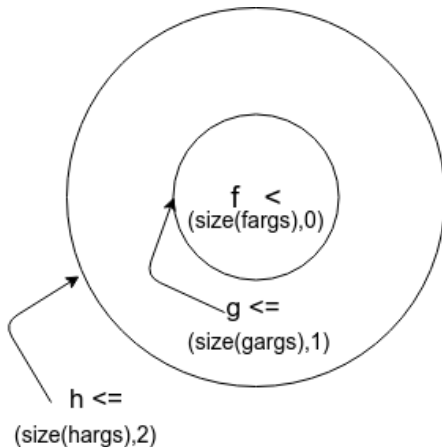
- We give each ordering a measure:

Ordering	Measure
Integer	$m(args) = \sum_{i=1}^{args.length} size(args_i)$
Lexicographic	$m(args) = (size(args_1), \dots, size(args_{args.length}))$
Bit-vector	$m(args) = \sum_{t=byte}^{long} size(\text{arguments of type } t)$

- We give each processor an algorithm to annotate measures:
 - If recursion processor finds a recursive decreasing pattern for parameter p , then we annotate $size(p)$.
 - The other three annotations need more explanation.

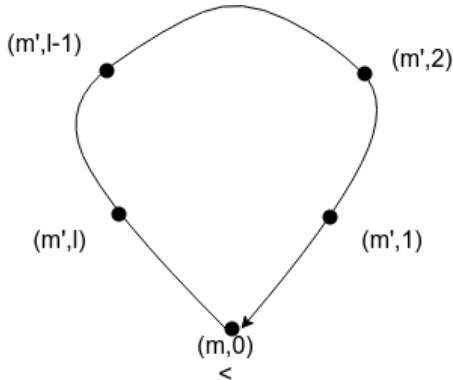
Relation processor annotation

- Reuses the fixpoint computation for finding strict decrease to annotate different measure levels.



Chain processor annotation

- Extends the global size decrease in the chain to a decrease in the whole chain.
- m' denotes the value of m after a tour around the cycle.



Chain processor annotation

- In a later stage we find problems when definitions are inlined. For instance consider the following function:

```
def simple(f: Formula) = f match{  
  case Or(lhs,rhs) => Or(simple(lhs),simple(rhs))  
  case Implies(lhs,rhs) => simple(Or(lhs,rhs))  
  case _ => f  
}
```

- Currently, the framework would infer $fullSize(formula)$ as a measure.
- The decrease is shown as the system will inline definitions to:
$$simple(Implies(lhs, rhs)) \rightarrow simple(Or(lhs, rhs)) \rightarrow simple(lhs)$$
$$simple(Implies(lhs, rhs)) \rightarrow simple(Or(lhs, rhs)) \rightarrow simple(rhs)$$
- However, the check of the measure will not pass since it will try to prove that $size(Implies(lhs, rhs)) < size(Or(lhs, rhs))$.

Chain processor annotation

- I suggest to use term rewriting to obtain the function:

```
def simple(f: Formula) = f match{  
  case Or(lhs,rhs) => Or(simple(lhs),simple(rhs))  
  case Implies(lhs,rhs) =>  
    Or(simple(lhs),simple(rhs))  
  case _ => f  
}
```

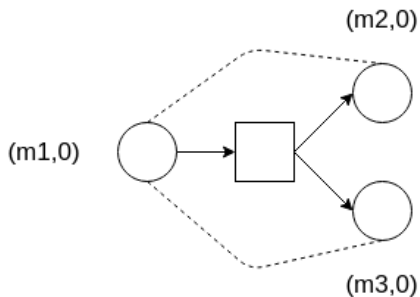
- This links with one of the recommendations given when starting the project (study the AproVe system).

Decreases processor annotation

Algorithm 1 Measure derivation in Decreases processor

```
1: for each node with measure  $m$  do
2:   Change  $m$  to  $(m, 0)$ .
3:   Find nodes without measure.
4:   for each node  $n$  without measure do
5:     Look for next nodes  $n'_i$  with measure  $m'_i$ .
6:     for each  $n'_i$  do
7:       Save measure:  $path_i \Rightarrow (m''_i, l)$  for  $n$  where:
8:        $l = length(n \rightarrow n'_i)$ 
9:        $path =$  condition for function invocation calling  $n'_i$ 
10:       $m''_i =$  value of  $m'_i$  computed from  $n$ 
11:     end for
12:   end for
13: end for
```

Decreases processor annotation



```
if(path2) (m2',1)
else if(path3) (m3',1)
....
```

- 1 Termination analysis vs measure extraction
- 2 Measure annotation
- 3 Checking measure verification conditions**
- 4 While loop annotations
- 5 Final thoughts
- 6 References

- Once all measures for the functions are annotated, we derive verification conditions of two forms:
 - Conditions to ensure positivity of the measures.
 - Conditions to ensure that measures decrease from one function to its functions calls whenever the calls transitively call the caller.
- This is done in file DefaultTactic of the verification phase.

Decreases processor annotation

stainless summary

append	meas. dec (Measure decrease for the call append(xs, l2))	valid nativez3 ./frontends/benchmarks/termination/valid/ListWithSize.scala:82:5
append	meas. pos (def append(l1: List, l2: List): List = { ...})	valid nativez3 ./frontends/benchmarks/termination/valid/ListWithSize.scala:82:5
append	postcondition	valid nativez3 ./frontends/benchmarks/termination/valid/ListWithSize.scala:82:5
append	postcondition	valid nativez3 ./frontends/benchmarks/termination/valid/ListWithSize.scala:82:5
append	match exhaustiveness	valid nativez3 ./frontends/benchmarks/termination/valid/ListWithSize.scala:82:48
appendAssoc	postcond. (ind. on xs: List / Cons)	valid nativez3 ./frontends/benchmarks/termination/valid/ListWithSize.scala:96:5
appendAssoc	postcond. (ind. on xs: List / Nil)	valid nativez3 ./frontends/benchmarks/termination/valid/ListWithSize.scala:96:5
concat	postcond. (ind. on l1: List / Cons)	valid nativez3 ./frontends/benchmarks/termination/valid/ListWithSize.scala:114:5
concat	postcond. (ind. on l1: List / Nil)	valid nativez3 ./frontends/benchmarks/termination/valid/ListWithSize.scala:114:5
concat0	meas. dec (Measure decrease for the call concat0(Nil(), ys, Cons(y, l3)))	valid nativez3 ./frontends/benchmarks/termination/valid/ListWithSize.scala:118:5
concat0	meas. dec (Measure decrease for the call concat0(xs, l2, Cons(x, l3)))	valid nativez3 ./frontends/benchmarks/termination/valid/ListWithSize.scala:118:5
concat0	meas. pos (@induct ...)	valid nativez3 ./frontends/benchmarks/termination/valid/ListWithSize.scala:118:5
concat0	postcond. (ind. on l1: List / Cons)	valid nativez3 ./frontends/benchmarks/termination/valid/ListWithSize.scala:118:5
concat0	postcond. (ind. on l1: List / Cons)	valid nativez3 ./frontends/benchmarks/termination/valid/ListWithSize.scala:118:5
concat0	postcond. (ind. on l1: List / Cons)	valid nativez3 ./frontends/benchmarks/termination/valid/ListWithSize.scala:118:5
concat0	postcond. (ind. on l1: List / Nil)	valid nativez3 ./frontends/benchmarks/termination/valid/ListWithSize.scala:118:5
concat0	postcond. (ind. on l1: List / Nil)	valid nativez3 ./frontends/benchmarks/termination/valid/ListWithSize.scala:118:5
concat0	postcond. (ind. on l1: List / Nil)	valid nativez3 ./frontends/benchmarks/termination/valid/ListWithSize.scala:118:5
concat0	match exhaustiveness	valid nativez3 ./frontends/benchmarks/termination/valid/ListWithSize.scala:118:57
concat0	match exhaustiveness	valid nativez3 ./frontends/benchmarks/termination/valid/ListWithSize.scala:119:21
content	meas. dec (Measure decrease for the call content(xs))	valid nativez3 ./frontends/benchmarks/termination/valid/ListWithSize.scala:55:5
content	meas. pos (def content(l: List): Set[Int] = { ...})	valid nativez3 ./frontends/benchmarks/termination/valid/ListWithSize.scala:55:5
content	match exhaustiveness	valid nativez3 ./frontends/benchmarks/termination/valid/ListWithSize.scala:55:39
drunk	meas. dec (Measure decrease for the call drunk(l1))	valid nativez3 ./frontends/benchmarks/termination/valid/ListWithSize.scala:64:5
drunk	meas. pos (def drunk(l: List): List = { ...})	valid nativez3 ./frontends/benchmarks/termination/valid/ListWithSize.scala:64:5
drunk	postcondition	valid nativez3 ./frontends/benchmarks/termination/valid/ListWithSize.scala:64:5
drunk	postcondition	valid nativez3 ./frontends/benchmarks/termination/valid/ListWithSize.scala:64:5
drunk	match exhaustiveness	valid nativez3 ./frontends/benchmarks/termination/valid/ListWithSize.scala:64:35

Preliminary results, problems and solutions

- Problems with caching: in current solution solver loses it cache.
- The strengthening of applications and postconditions could lead to problems if not passed to the extraction pipeline.
- Preliminary implementation allowed us to verify 71 % of the examples.

- 1 Termination analysis vs measure extraction
- 2 Measure annotation
- 3 Checking measure verification conditions
- 4 While loop annotations**
- 5 Final thoughts
- 6 References

Annotating while loops

- We added support to annotate while loops with measures.
- This amounts to a change in the ImperativeCodeElimination and the CodeExtraction.
- To take into account the while condition to prove the decrease we change the transformation into a do-while. Something like:

```
def f() = ... if(c) fwhile() else ...  
def fwhile() = b if(c) fwhile() else ...
```

- This way we can use c to prove decrease of measures.

An example from linear ranking functions context

In master, you can now prove termination of loops like this:

```
def fi(fi: BigInt, fj: BigInt, fk: BigInt): BigInt = {  
  var i = fi  
  var j = fj  
  var k = fk  
  
  while((i <= 100) && (j <= k)){  
    decreases(k-i-j+102)  
    val s = i  
    i = j  
    j = s+1  
    k = k-1  
  }  
  
  i+j+k  
}
```

- 1 Termination analysis vs measure extraction
- 2 Measure annotation
- 3 Checking measure verification conditions
- 4 While loop annotations
- 5 Final thoughts**
- 6 References

How would you annotate this example?

```
def zeros: Stream[Int] = 0 #:: zeros
```

Algebraic topology for termination proofs

⤴ I'm reading about various ways in which termination proofs of software verifiers are built: ad-hoc methods that detect recursions, term-rewriting, synthesis of lexicographic orderings...

12 From the ad-hoc methods I came with the following intuitive idea. Is it possible to formalize termination with algebraic topology? The idea would be:

⤵




- look at termination as the problem of finding loops in the code of the program.
- ★ -look these loops as a fundamental group at some origin (a point in the source code)

- test whether this fundamental group is simply connected (implying that this code is terminating)

Of course both problems are undecidable (the second one being testing the simple connectedness) but it seems to me theoretically feasible.

Has this been done before? You find it is not feasible?

- 1 Termination analysis vs measure extraction
- 2 Measure annotation
- 3 Checking measure verification conditions
- 4 While loop annotations
- 5 Final thoughts
- 6 References**

-  L. Bulwahn, A. Krauss, and T. Nipkow. “Finding lexicographic orders for termination proofs in Isabelle/HOL”. In: *Theorem Proving in Higher Order Logics* (2007), pp. 38–53.
-  B. Cook, A. Podelski, and A. Rybalchenko. “Proving program termination.”. In: *Communications of the ACM* (2011), pp. 88–98.
-  C.B. Jones. *Turing and Software Verification*. Tech. rep. Newcastle University, 2014.
-  C.S. Lee. “Ranking functions for size-change termination”. In: *ACM Transactions on Programming Languages and Systems* (2009), p. 2009.
-  N. Voirol. “Termination Analysis in a Higher-Order Functional Context”. MA thesis. EPFL: School of Computer and Communication Sciences, 2013.