Modular Termination Verification

Bart Jacobs [1] Dragan Bosnacki [2] Ruurd Kuiper [2]

[1]DistriNet, KU Leuven

[2]Eindhoven University of Technology

ECOOP 2015

Disclaimer

- This paper is **NOT about termination analysis**. No algorithms are proposed.

Disclaimer

- This paper is **NOT about termination analysis**. No algorithms are proposed.
- We propose an approach for doing pencil-and-paper proofs of termination of programs *modularly*.

Disclaimer

- This paper is **NOT about termination analysis**. No algorithms are proposed.
- We propose an approach for doing pencil-and-paper proofs of termination of programs *modularly*.
- I.e., we propose a notion of module correctness such that

Disclaimer

- This paper is **NOT about termination analysis**. No algorithms are proposed.
- We propose an approach for doing pencil-and-paper proofs of termination of programs *modularly*.
- I.e., we propose a notion of module correctness such that
  - if one succeeds in producing a paper-and-pencil proof of the correctness of each of a program's modules,

Disclaimer

- This paper is **NOT about termination analysis**. No algorithms are proposed.
- We propose an approach for doing pencil-and-paper proofs of termination of programs *modularly*.
- I.e., we propose a notion of module correctness such that
  - if one succeeds in producing a paper-and-pencil proof of the correctness of each of a program's modules,
  - then the program terminates.

Disclaimer

- This paper is **NOT about termination analysis**. No algorithms are proposed.
- We propose an approach for doing pencil-and-paper proofs of termination of programs *modularly*.
- I.e., we propose a notion of module correctness such that
  - if one succeeds in producing a paper-and-pencil proof of the correctness of each of a program's modules,
  - then the program terminates.
- Module correctness means the module satisfies its specification

Disclaimer

- This paper is **NOT about termination analysis**. No algorithms are proposed.
- We propose an approach for doing pencil-and-paper proofs of termination of programs *modularly*.
- I.e., we propose a notion of module correctness such that
  - if one succeeds in producing a paper-and-pencil proof of the correctness of each of a program's modules,
  - then the program terminates.
- Module correctness means the module satisfies its specification
  - assuming that the modules it *imports* satisfy theirs.

Disclaimer

- This paper is **NOT about termination analysis**. No algorithms are proposed.
- We propose an approach for doing pencil-and-paper proofs of termination of programs *modularly*.
- I.e., we propose a notion of module correctness such that
  - if one succeeds in producing a paper-and-pencil proof of the correctness of each of a program's modules,
  - then the program terminates.
- Module correctness means the module satisfies its specification
  - assuming that the modules it *imports* satisfy theirs.
- Main contribution:

Disclaimer

- This paper is **NOT about termination analysis**. No algorithms are proposed.
- We propose an approach for doing pencil-and-paper proofs of termination of programs *modularly*.
- I.e., we propose a notion of module correctness such that
  - if one succeeds in producing a paper-and-pencil proof of the correctness of each of a program's modules,
  - then the program terminates.
- Module correctness means the module satisfies its specification
  - assuming that the modules it *imports* satisfy theirs.
- Main contribution:
  - an approach for writing module specifications

Disclaimer

- This paper is **NOT about termination analysis**. No algorithms are proposed.
- We propose an approach for doing pencil-and-paper proofs of termination of programs *modularly*.
- I.e., we propose a notion of module correctness such that
  - if one succeeds in producing a paper-and-pencil proof of the correctness of each of a program's modules,
  - then the program terminates.
- Module correctness means the module satisfies its specification
  - assuming that the modules it *imports* satisfy theirs.
- Main contribution:
  - an approach for writing module specifications
  - that are sufficiently expressive to allow verification of client code

Disclaimer

- This paper is **NOT about termination analysis**. No algorithms are proposed.
- We propose an approach for doing pencil-and-paper proofs of termination of programs *modularly*.
- I.e., we propose a notion of module correctness such that
  - if one succeeds in producing a paper-and-pencil proof of the correctness of each of a program's modules,
  - then the program terminates.
- Module correctness means the module satisfies its specification
  - assuming that the modules it *imports* satisfy theirs.
- Main contribution:
  - an approach for writing module specifications
  - that are sufficiently expressive to allow verification of client code
  - and sufficiently abstract to allow module implementation evolution

Disclaimer

- This paper is **NOT about termination analysis**. No algorithms are proposed.
- We propose an approach for doing pencil-and-paper proofs of termination of programs *modularly*.
- I.e., we propose a notion of module correctness such that
  - if one succeeds in producing a paper-and-pencil proof of the correctness of each of a program's modules,
  - then the program terminates.
- Module correctness means the module satisfies its specification
  - assuming that the modules it *imports* satisfy theirs.
- Main contribution:
  - an approach for writing module specifications
  - that are sufficiently expressive to allow verification of client code
  - and sufficiently abstract to allow module implementation evolution
    - any modification that does not break clients should be allowed

Contents

Contents

Whole-Program Verification

$\textbf{num}\ \mathsf{sqrt}(\textbf{num}\ \mathsf{x})$
$\{\ (1 + \mathsf{x})/2\ \}$

$\textbf{num}\ \mathsf{vectorSize}(\textbf{num}\ \mathsf{x}, \textbf{num}\ \mathsf{y})$
$\{\ \mathsf{sqrt}(\mathsf{x} \cdot \mathsf{x} + \mathsf{y} \cdot \mathsf{y})\ \}$

$\textbf{void}\ \mathsf{main}()$
$\{\ \textbf{assert}\ 0 \leq \mathsf{vectorSize}(3, 4)\ \}$

$$\textbf{num}\ \mathsf{sqrt}(\textbf{num}\ \mathsf{x})$$
$$\left\{\begin{array}{l}\textbf{num}\ \mathsf{y} := (1+\mathsf{x})/2; \\ (y + x/y)/2\end{array}\right\}$$

$$\textbf{num}\ \mathsf{vectorSize}(\textbf{num}\ \mathsf{x}, \textbf{num}\ \mathsf{y})$$
$$\{\ \mathsf{sqrt}(\mathsf{x}\cdot\mathsf{x}+\mathsf{y}\cdot\mathsf{y})\ \}$$

$$\textbf{void}\ \mathsf{main}()$$
$$\{\ \textbf{assert}\ 0 \leq \mathsf{vectorSize}(3,4)\ \}$$

Modular Verification

num sqrt(**num** x)
$\{ (1 + x)/2 \}$

num vectorSize(**num** x, **num** y)
$\{ \text{sqrt}(x \cdot x + y \cdot y) \}$

void main()
{ **assert** $0 \leq \text{vectorSize}(3, 4)$ }

```
num sqrt(num x)
  req 0 ≤ x
  ens 0 ≤ result
{ (1 + x)/2 }

num vectorSize(num x, num y)
  ens 0 ≤ result
{ sqrt(x · x + y · y) }

void main()
{ assert 0 ≤ vectorSize(3, 4) }
```

Modular Verification

$\textbf{num}\ \text{sqrt}(\textbf{num}\ x)$
$\quad \textbf{req}\ 0 \leq x$
$\quad \textbf{ens}\ 0 \leq \text{result}$
? $\{\ (1 + x)/2\ \}$

$\textbf{num}\ \text{vectorSize}(\textbf{num}\ x, \textbf{num}\ y)$
$\quad \textbf{ens}\ 0 \leq \text{result}$
? $\{\ \text{sqrt}(x \cdot x + y \cdot y)\ \}$

$\textbf{void}\ \text{main}()$
? $\{\ \textbf{assert}\ 0 \leq \text{vectorSize}(3, 4)\ \}$

Modular Verification

$\textbf{num}\ \mathsf{sqrt}(\textbf{num}\ x)$
  $\textbf{req}\ 0 \leq x$
  $\textbf{ens}\ 0 \leq \mathsf{result}$
✓ $\{\ (1 + x)/2\ \}$

$\textbf{num}\ \mathsf{vectorSize}(\textbf{num}\ x, \textbf{num}\ y)$
  $\textbf{ens}\ 0 \leq \mathsf{result}$
? $\{\ \mathsf{sqrt}(x \cdot x + y \cdot y)\ \}$

$\textbf{void}\ \mathsf{main}()$
? $\{\ \textbf{assert}\ 0 \leq \mathsf{vectorSize}(3, 4)\ \}$

Modular Verification

num sqrt(**num** x)
  req $0 \leq x$
  ens $0 \leq$ result
✓ { $(1 + x)/2$ }

num vectorSize(**num** x, **num** y)
  ens $0 \leq$ result
✓ { $\mathrm{sqrt}(x \cdot x + y \cdot y)$ }

void main()
? { **assert** $0 \leq \mathrm{vectorSize}(3, 4)$ }

Modular Verification

$$\textbf{num}\ \text{sqrt}(\textbf{num}\ x)$$
$$\quad \textbf{req}\ 0 \leq x$$
$$\quad \textbf{ens}\ 0 \leq \text{result}$$
$\checkmark$ $$\{\ (1 + x)/2\ \}$$

$$\textbf{num}\ \text{vectorSize}(\textbf{num}\ x, \textbf{num}\ y)$$
$$\quad \textbf{ens}\ 0 \leq \text{result}$$
$\checkmark$ $$\{\ \text{sqrt}(x \cdot x + y \cdot y)\ \}$$

$$\textbf{void}\ \text{main}()$$
$\checkmark$ $$\{\ \textbf{assert}\ 0 \leq \text{vectorSize}(3, 4)\ \}$$

Modular Verification

```
num sqrt(num x)
    req 0 ≤ x
    ens 0 ≤ result
?   { num y := (1 + x)/2;
      (y + x/y)/2 }

    num vectorSize(num x, num y)
    ens 0 ≤ result
✓   { sqrt(x · x + y · y) }

    void main()
✓   { assert 0 ≤ vectorSize(3, 4) }
```

Modular Verification

```
num sqrt(num x)
    req 0 ≤ x
    ens 0 ≤ result
✓   { num y := (1 + x)/2;
      (y + x/y)/2 }

num vectorSize(num x, num y)
    ens 0 ≤ result
✓   { sqrt(x · x + y · y) }

void main()
✓   { assert 0 ≤ vectorSize(3, 4) }
```

Contents

Contents

Termination Verification

$\textbf{num}\ \text{sqrt}(\textbf{num}\ x)$
$\{\ (1 + x)/2\ \}$

$\textbf{num}\ \text{vectorSize}(\textbf{num}\ x, \textbf{num}\ y)$
$\{\ \text{sqrt}(x \cdot x + y \cdot y)\ \}$

$\textbf{void}\ \text{main}()$
$\{\ \textbf{assert}\ 0 \leq \text{vectorSize}(3, 4)\ \}$

```
num sqrt(num x)
   level ?
{ (1 + x)/2 }

num vectorSize(num x, num y)
   level ?
{ sqrt(x · x + y · y) }

void main()
   level ?
{ assert 0 ≤ vectorSize(3, 4) }
```

num sqrt(**num** x)
  level 0
{ $(1 + x)/2$ }

num vectorSize(**num** x, **num** y)
  level 1
{ sqrt($x \cdot x + y \cdot y$) }

void main()
  level 2
{ **assert** $0 \leq$ vectorSize(3, 4) }

num average(**num** x, **num** y)
  level 0
{ $(x + y)/2$ }

num sqrt(**num** x)
  level 0
{ average(1, x) }

num vectorSize(**num** x, **num** y)
  level 1
{ sqrt($x \cdot x + y \cdot y$) }

void main()
  level 2
{ **assert** $0 \leq$ vectorSize(3, 4) }

$\textbf{num}\ \mathsf{sqrt}(\textbf{num}\ x)$
$\{\ (1 + x)/2\ \}$

$\textbf{num}\ \mathsf{vectorSize}(\textbf{num}\ x, \textbf{num}\ y)$
$\{\ \mathsf{sqrt}(x \cdot x + y \cdot y)\ \}$

$\textbf{void}\ \mathsf{main}()$
$\{\ \textbf{assert}\ 0 \leq \mathsf{vectorSize}(3, 4)\ \}$

Classes

class Math $\left\{ \begin{array}{l} \textbf{static num } \text{sqrt}(\textbf{num } x) \\ \{\ (1 + x)/2\ \} \end{array} \right\}$

class Util $\left\{ \begin{array}{l} \textbf{static num } \text{vectorSize}(\textbf{num } x, \textbf{num } y) \\ \{\ \text{sqrt}(x \cdot x + y \cdot y)\ \} \end{array} \right\}$

class Main $\left\{ \begin{array}{l} \textbf{static void } \text{main}() \\ \{\ \textbf{assert } 0 \leq \text{vectorSize}(3, 4)\ \} \end{array} \right\}$

Import Relation

$$\textbf{class}\ \text{Math}\ \left\{\begin{array}{l}\textbf{static num}\ \text{sqrt}(\textbf{num}\ x)\\ \{\ (1+x)/2\ \}\end{array}\right\}$$

$$\textbf{class}\ \text{Util}\ \textbf{import}\ \text{Math}\ \left\{\begin{array}{l}\textbf{static num}\ \text{vectorSize}(\textbf{num}\ x, \textbf{num}\ y)\\ \{\ \text{sqrt}(x\cdot x+y\cdot y)\ \}\end{array}\right\}$$

$$\textbf{class}\ \text{Main}\ \textbf{import}\ \text{Util}\ \left\{\begin{array}{l}\textbf{static void}\ \text{main}()\\ \{\ \textbf{assert}\ 0\leq \text{vectorSize}(3,4)\ \}\\ \\ \end{array}\right\}$$

Levels: Method Names

$$\textbf{class}\ \text{Math}\ \left\{\begin{array}{l}\textbf{static num}\ \text{sqrt}(\textbf{num}\ x)\\ \quad \textbf{level}\ \text{Math.sqrt}\\ \{\ (1+x)/2\ \}\end{array}\right\}$$

$$\textbf{class}\ \text{Util}\ \textbf{import}\ \text{Math}\ \left\{\begin{array}{l}\textbf{static num}\ \text{vectorSize}(\textbf{num}\ x, \textbf{num}\ y)\\ \quad \textbf{level}\ \text{Util.vectorSize}\\ \{\ \text{sqrt}(x\cdot x + y\cdot y)\ \}\end{array}\right\}$$

$$\textbf{class}\ \text{Main}\ \textbf{import}\ \text{Util}\ \left\{\begin{array}{l}\textbf{static void}\ \text{main}()\\ \quad \textbf{level}\ \text{Main.main}\\ \{\ \textbf{assert}\ 0 \leq \text{vectorSize}(3,4)\ \}\end{array}\right\}$$

Levels: Method Names

class Math { **static num** sqrt(**num** x) **level** sqrt { $(1 + x)/2$ } }

class Util **import** Math { **static num** vectorSize(**num** x, **num** y) **level** vectorSize { sqrt($x \cdot x + y \cdot y$) } }

class Main **import** Util { **static void** main() **level** main { **assert** $0 \leq$ vectorSize(3, 4) } }

Levels: Method Names

class Math {

static num average(**num** x, **num** y)
  level average
{ $(x + y)/2$ }

static num sqrt(**num** x)
  level sqrt
{ average(1, x) }

}

class Util **import** Math {

static num vectorSize(**num** x, **num** y)
  level vectorSize
{ sqrt($x \cdot x + y \cdot y$) }

}

class Main **import** Util {

static void main()
  level main
{ **assert** $0 \leq$ vectorSize(3, 4) }

}

Specification Pattern

Specification Pattern

$$\cdots \; m(\cdots)$$
$$\textbf{level } m$$
$$\{ \; \cdots \; \}$$

Reason

Allows arbitrary upcalls.

Contents

Contents

Dynamic Binding

```
interface Function {
   num apply(num x)
}

class Util {




   static num derivative(Function f, num x)
   { f.apply(x + 1) − f.apply(x) }
}

class ZeroFunc implements Function { num apply(num x) { 0 } }

class Main imports Util, ZeroFunc {
   void main()
   { derivative(new ZeroFunc(), 0) }
}
```

Dynamic Binding

```
interface Function {
   num apply(num x)
      level ?
}
class Util {



   static num derivative(Function f, num x)
      level ?
   { f.apply(x + 1) − f.apply(x) }
}
class ZeroFunc implements Function { num apply(num x) { 0 } }
class Main imports Util, ZeroFunc {
   void main()
      level ?
   { derivative(new ZeroFunc(), 0) }
}
```

Dynamic Binding

```
interface Function {
   num apply(num x)
      level ?
}
class Util {




   static num derivative(Function f, num x)
      level ?
   { f.apply(x + 1) − f.apply(x) }
}
class ZeroFunc implements Function { num apply(num x) { 0 } }
class Main imports Util, ZeroFunc {
   void main()
      level main
   { derivative(new ZeroFunc(), 0) }
}
```

```
interface Function {
   num apply(num x)
      level classOf(this).apply
}
class Util {




   static num derivative(Function f, num x)
      level ?
   { f.apply(x + 1) − f.apply(x) }
}
class ZeroFunc implements Function { num apply(num x) { 0 } }
class Main imports Util, ZeroFunc {
   void main()
      level main
   { derivative(new ZeroFunc(), 0) }
}
```

Dynamic Binding

```
interface Function {
   num apply(num x)
      level this.apply
}
class Util {




   static num derivative(Function f, num x)
      level ?
   { f.apply(x + 1) − f.apply(x) }
}
class ZeroFunc implements Function { num apply(num x) { 0 } }
class Main imports Util, ZeroFunc {
   void main()
      level main
   { derivative(new ZeroFunc(), 0) }
}
```

```
interface Function {
  num apply(num x)
    level this.apply
}
class Util {



  static num derivative(Function f, num x)
    level derivative
  { f.apply(x + 1) − f.apply(x) }
}
class ZeroFunc implements Function { num apply(num x) { 0 } }
class Main imports Util, ZeroFunc {
  void main()
    level main
  { derivative(new ZeroFunc(), 0) }
}
```

Dynamic Binding

```
interface Function {
  num apply(num x)
    level this.apply
}
class Util {




  static num derivative(Function f, num x)
    level f.apply
  { f.apply(x + 1) − f.apply(x) }
}
class ZeroFunc implements Function { num apply(num x) { 0 } }
class Main imports Util, ZeroFunc {
  void main()
    level main
  { derivative(new ZeroFunc(), 0) }
}
```

Dynamic Binding

```
interface Function {
   num apply(num x)
      level this.apply
}
class Util {
   static num derivativeHelper(Function f, num x)
      level f.apply
   { f.apply(x + 1) − f.apply(x) }
   static num derivative(Function f, num x)
      level f.apply
   { derivativeHelper(f, x) }
}
class ZeroFunc implements Function { num apply(num x) { 0 } }
class Main imports Util, ZeroFunc {
   void main()
      level main
   { derivative(new ZeroFunc(), 0) }
}
```

Dynamic Binding: Multiset Order

```
interface Function {
  num apply(num x)
    level {[this.apply]}
}
class Util {
  static num derivativeHelper(Function f, num x)
    level {[derivativeHelper, f.apply]}
  { f.apply(x + 1) − f.apply(x) }
  static num derivative(Function f, num x)
    level {[derivative, f.apply]}
  { derivativeHelper(f, x) }
}
class ZeroFunc implements Function { num apply(num x) { 0 } }
class Main imports Util, ZeroFunc {
  void main()
    level {[main]}
  { derivative(new ZeroFunc(), 0) }
}
```

Contents

Contents

Complex Objects

```
interface Function {
  num apply(num x)

}
class Util {
  static num derivative(Function f, num x)

  { f.apply(x + 1) − f.apply(x) }
}
class ZeroFunc implements Function { num apply(num x) { 0 } }
class Plus1Func(Function f) implements Function {
  num apply(num x) { f.apply(x) + 1 }
}
class Main imports Util, ZeroFunc, Plus1Func {
  void main()

  { derivative(new Plus1Func(new ZeroFunc()), 0) }
}
```

Complex Objects

```
interface Function {
   num apply(num x)
      level {[this.apply]}
}
class Util {
   static num derivative(Function f, num x)
      level {[derivative, f.apply]}
   { f.apply(x + 1) − f.apply(x) }
}
class ZeroFunc implements Function { num apply(num x) { 0 } }
class Plus1Func(Function f) implements Function {
   num apply(num x) { f.apply(x) + 1 }
}
class Main imports Util, ZeroFunc, Plus1Func {
   void main()
      level {[main]}
   { derivative(new Plus1Func(new ZeroFunc()), 0) }
}
```

Complex Objects

```
interface Function {
   num apply(num x)
      level {[this.apply, this.f.apply]}
}
class Util {
   static num derivative(Function f, num x)
      level {[derivative, f.apply]}
   { f.apply(x + 1) − f.apply(x) }
}
class ZeroFunc implements Function { num apply(num x) { 0 } }
class Plus1Func(Function f) implements Function {
   num apply(num x) { f.apply(x) + 1 }
}
class Main imports Util, ZeroFunc, Plus1Func {
   void main()
      level {[main]}
   { derivative(new Plus1Func(new ZeroFunc()), 0) }
}
```

```
interface Function {
   num apply(num x)
      level if this instanceof Plus1Func then {[this.apply, this.f.apply]}
         else {[this.apply]}
}
class Util {
   static num derivative(Function f, num x)
      level {[derivative, f.apply]}
   { f.apply(x + 1) − f.apply(x) }
}
class ZeroFunc implements Function { num apply(num x) { 0 } }
class Plus1Func(Function f) implements Function {
   num apply(num x) { f.apply(x) + 1 }
}
class Main imports Util, ZeroFunc, Plus1Func {
   void main()
      level {[main]}
   { derivative(new Plus1Func(new ZeroFunc()), 0) }
```

```
interface Function {
  predicate valid(MethodBag depth)
  num apply(num x)
    req this.valid(d)
    level d
}
```

Complex Objects: Predicate Families, Dynamic Depths

```
interface Function {
  predicate valid(MethodBag depth)
  num apply(num x)
    req this.valid(d)
    level d
}
class ZeroFunc implements Function {
   predicate valid(MethodBag depth) =
     (depth = {[this.apply]})
  num apply(num x) { 0 }
}
```

```
interface Function {
  predicate valid(MethodBag depth)
  num apply(num x)
    req this.valid(d)
    level d
}
class ZeroFunc implements Function {
   predicate valid(MethodBag depth) =
     (depth = {[this.apply]})
  num apply(num x) { 0 }
}
class Plus1Func(Function f) implements Function {
   predicate valid(MethodBag depth) =
     (∃df. f.valid(df) ∧ depth = {[this.apply]} ⊎ df)
  num apply(num x) { f.apply(x) + 1 }
}
```

Predicate Families: Inductive Interpretation

$$\textsc{ZeroFuncValid}\quad \frac{\mathsf{classOf}(o) = \mathsf{ZeroFunc} \qquad \mathsf{depth} = \{\!\{o.\mathsf{apply}\}\!\}}{o.\mathsf{valid}(\mathsf{depth})}$$

$$\textsc{Plus1FuncValid}\quad \frac{\begin{array}{c}\mathsf{classOf}(o) = \mathsf{Plus1Func} \\ o.\mathsf{f.valid}(\mathsf{df}) \qquad \mathsf{depth} = \{\!\{o.\mathsf{apply}\}\!\} \uplus \mathsf{df}\end{array}}{o.\mathsf{valid}(\mathsf{depth})}$$

Predicate Families: Inductive Interpretation

[Parkinson and Bierman, POPL 2005]

$$\textsc{ZeroFuncValid}\ \frac{\mathsf{classOf}(o) = \mathsf{ZeroFunc} \qquad \mathsf{depth} = \{\!\{o.\mathsf{apply}\}\!\}}{o.\mathsf{valid}(\mathsf{depth})}$$

$$\textsc{Plus1FuncValid}\ \frac{\begin{array}{c}\mathsf{classOf}(o) = \mathsf{Plus1Func} \\ o.\mathsf{f}.\mathsf{valid}(\mathsf{df}) \qquad \mathsf{depth} = \{\!\{o.\mathsf{apply}\}\!\} \uplus \mathsf{df}\end{array}}{o.\mathsf{valid}(\mathsf{depth})}$$

```
interface Function {
   predicate valid(MethodBag depth)
   num apply(num x)
      req this.valid(d)
      level d
}
class Util {
   static num derivative(Function f, num x)
      req f.valid(d)
      level {[derivative]} ⊎ d
   { f.apply(x + 1) − f.apply(x) }
}
class Main imports Util, ZeroFunc, Plus1Func {
   void main()
      level {[main]}
   { derivative(new Plus1Func(new ZeroFunc()), 0) }
}
```

```
class Main imports Util, ZeroFunc, Plus1Func {
  void main()
    level {[main]}
  {
    Function f1 := new ZeroFunc();
    {f1.valid({[ZeroFunc.apply]})}
    Function f2 := new Plus1Func(f1);
    {f2.valid({[Plus1Func.apply, ZeroFunc.apply]})}
    {{[derivative, Plus1Func.apply, ZeroFunc.apply]} < {[main]}}
    derivative(f2, 0)
  }
}
```

Contents

Contents

Abstract Object Construction

```
class ZeroFunc implements Function {
   predicate valid(MethodBag depth) =
      (depth = {[this.apply]})
  num apply(num x) { 0 }
  static create()
     level {[create]}
     ens ∃d. result.valid(d) ∧ d < {[create]}
  { new ZeroFunc() }
}
```

Abstract Object Construction

```
class Plus1Func(Function f) implements Function {
   predicate valid(MethodBag depth) =
     (∃d. f.valid(d) ∧ depth = {[this.apply]} ⊎ d)
  num apply(num x) { f.apply(x) + 1 }
  static create(Function f)
    req f.valid(df)
    level {[create]} ⊎ df
    ens ∃d. result.valid(d) ∧ d < {[create]} ⊎ df
  { new Plus1Func(df) }
}
```

Abstract Object Construction

```
class Main imports Util, ZeroFunc, Plus1Func {
  void main()
    level {[main]}
  {
    Function f1 := ZeroFunc.create();
    {f1.valid(d1) ∧ d1 < {[ZeroFunc]}}
    Function f2 := Plus1Func.create(f1);
    {f2.valid(d2) ∧ d2 < {[Plus1Func, ZeroFunc]})}
    {{[derivative]} ⊎ d2 < {[main]}}
    derivative(f2, 0)
  }
}
```

Proposed Specification Style

interface I {
  predicate valid(MethodBag depth)
  $\tau\ m(\cdots)$
    req this.valid(d)
    level d
}
static $\tau\ m(I\ o)$
  req o.valid(d)
  level $\{\!\{m\}\!\} \uplus$ d

class $C(I\ f)$ **implements** I {
  predicate valid(MethodBag depth) $=$
    $(f.\text{valid}(\text{df}) \wedge \text{depth} = \{\!\{\text{this}.m\}\!\} \uplus \text{df})$
  $\tau\ m(\cdots)\ \{\ \cdots\ \}$
}

Proposed Specification Style

interface I {
  predicate valid(MethodBag depth)
  τ $m(\cdots)$
    req this.valid(d)
    level d
}

class $C(I\ f)$ **implements** I {
  predicate valid(MethodBag depth)

  static I create($I\ f$)
    req f.valid(df)
    level $\{\!\!\{\text{create}\}\!\!\} \uplus \text{df}$
    ens $\exists\text{d. result.valid(d)} \wedge \text{d} < \{\!\!\{\text{create}\}\!\!\} \uplus \text{df}$
  { $\cdots$ }
}

Contents

Contents

Conclusion

- Contribution:

Conclusion

- Contribution:
  - First approach for modular verification of termination of OO programs

Conclusion

- Contribution:
  - First approach for modular verification of termination of OO programs
- Extras in the paper:

Conclusion

- Contribution:
  - First approach for modular verification of termination of OO programs
- Extras in the paper:
  - Methods taking multiple objects as arguments

Conclusion

- Contribution:
  - First approach for modular verification of termination of OO programs
- Extras in the paper:
  - Methods taking multiple objects as arguments
  - Recursion (direct & mutual)

Conclusion

- Contribution:
  - First approach for modular verification of termination of OO programs
- Extras in the paper:
  - Methods taking multiple objects as arguments
  - Recursion (direct & mutual)
  - Variant based on separation logic

Conclusion

- Contribution:
  - First approach for modular verification of termination of OO programs
- Extras in the paper:
  - Methods taking multiple objects as arguments
  - Recursion (direct & mutual)
  - Variant based on separation logic
    - with first-class call permissions

Conclusion

- Contribution:
  - First approach for modular verification of termination of OO programs
- Extras in the paper:
  - Methods taking multiple objects as arguments
  - Recursion (direct & mutual)
  - Variant based on separation logic
    - with first-class call permissions
    - can be hidden in predicates: useful for continuation-passing style programs

Conclusion

- Contribution:
  - First approach for modular verification of termination of OO programs
- Extras in the paper:
  - Methods taking multiple objects as arguments
  - Recursion (direct & mutual)
  - Variant based on separation logic
    - with first-class call permissions
    - can be hidden in predicates: useful for continuation-passing style programs
    - can be passed between threads

Conclusion

- Contribution:
  - First approach for modular verification of termination of OO programs
- Extras in the paper:
  - Methods taking multiple objects as arguments
  - Recursion (direct & mutual)
  - Variant based on separation logic
    - with first-class call permissions
    - can be hidden in predicates: useful for continuation-passing style programs
    - can be passed between threads
- Extras in the TR:

Conclusion

- Contribution:
  - First approach for modular verification of termination of OO programs
- Extras in the paper:
  - Methods taking multiple objects as arguments
  - Recursion (direct & mutual)
  - Variant based on separation logic
    - with first-class call permissions
    - can be hidden in predicates: useful for continuation-passing style programs
    - can be passed between threads
- Extras in the TR:
  - Combine with deadlock-freedom approach [Leino, Müller, Smans, ESOP 2010] to verify termination of concurrent programs

Conclusion

- Contribution:
  - First approach for modular verification of termination of OO programs
- Extras in the paper:
  - Methods taking multiple objects as arguments
  - Recursion (direct & mutual)
  - Variant based on separation logic
    - with first-class call permissions
    - can be hidden in predicates: useful for continuation-passing style programs
    - can be passed between threads
- Extras in the TR:
  - Combine with deadlock-freedom approach [Leino, Müller, Smans, ESOP 2010] to verify termination of concurrent programs
  - Use levels as wait levels to allow transparent private locks

Conclusion

- Contribution:
  - First approach for modular verification of termination of OO programs
- Extras in the paper:
  - Methods taking multiple objects as arguments
  - Recursion (direct & mutual)
  - Variant based on separation logic
    - with first-class call permissions
    - can be hidden in predicates: useful for continuation-passing style programs
    - can be passed between threads
- Extras in the TR:
  - Combine with deadlock-freedom approach [Leino, Müller, Smans, ESOP 2010] to verify termination of concurrent programs
  - Use levels as wait levels to allow transparent private locks
  - Prove termination of compare-and-swap loops

Conclusion

- Contribution:
  - First approach for modular verification of termination of OO programs
- Extras in the paper:
  - Methods taking multiple objects as arguments
  - Recursion (direct & mutual)
  - Variant based on separation logic
    - with first-class call permissions
    - can be hidden in predicates: useful for continuation-passing style programs
    - can be passed between threads
- Extras in the TR:
  - Combine with deadlock-freedom approach [Leino, Müller, Smans, ESOP 2010] to verify termination of concurrent programs
  - Use levels as wait levels to allow transparent private locks
  - Prove termination of compare-and-swap loops
  - Prove liveness of non-terminating programs

Conclusion

Validation:

- Examples in the paper

Conclusion

Validation:

- Examples in the paper
- Implemented in VeriFast tool; verified some examples

Conclusion

Validation:

- Examples in the paper
- Implemented in VeriFast tool; verified some examples
- More experimentation needed: are all programming patterns supported?

Conclusion

- Contribution:
  - First approach for modular verification of termination of OO programs
- Extras in the paper:
  - Methods taking multiple objects as arguments
  - Recursion (direct & mutual)
  - Variant based on separation logic
    - with first-class call permissions
    - can be hidden in predicates: useful for continuation-passing style programs
    - can be passed between threads
- Extras in the TR:
  - Combine with deadlock-freedom approach [Leino, Müller, Smans, ESOP 2010] to verify termination of concurrent programs
  - Use levels as wait levels to allow transparent private locks
  - Prove termination of compare-and-swap loops
  - Prove liveness of non-terminating programs