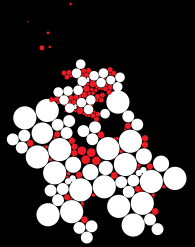
Some Napkin Math

- The day I started on this thesis: June 22th 2019
- Today is April 22th 2020, so that is 305 days (according to Google)
- The summer break was about 30 days, so that is 275 days
- On average, 2 cups of coffee each day
- This thesis equates to about

$$275\ \textit{days} \times\ 2\ \frac{\textit{Cups of Coffee}}{\textit{Day}} \times 0.37\ \frac{\textit{Liters}}{\textit{Cup of Coffee}} = 203.5\ \textit{Liters of Coffee}$$

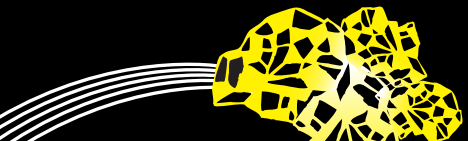
- There are 127 pages in my thesis, so about $\frac{203.5}{127} \approx 1.6\ \frac{\textit{Liters}}{\textit{Page}}$

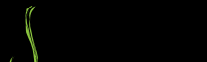
UNIVERSITY OF TWENTE.



Extending Support for Axiomatic Data Types in VerCors

Ömer Faruk Oğuzhan Şakar





Overview

Software Verification

VerCors

Axiomatic Data Types

Maps

Conclusion

Overview

Software Verification

VerCors

Axiomatic Data Types

Maps

Conclusion

Software

Software

Software

Software

Software

- *Software is everywhere*

Software

- *Software is everywhere* is a cliché

Software

- *Software is everywhere* is a cliché, but it's true

Cookie Clicker

Cookie Clicker

(homemade by my sister Nisa)

`Clicks:` 0

Cookie Clicker

Clicks: 1

Cookie Clicker

Clicks: 2

Cookie Clicker

Clicks: 3

Cookie Clicker

Clicks: 4

Cookie Clicker

Clicks: 5

```
method youClicked() {
      Clicks <- Clicks + 1

}
```

Cookie Clicker

Clicks: 6

```
method youClicked() {
      Clicks <- Clicks + 1

}
```

Cookie Clicker

Clicks: 7

4_Cookies: ✗

```
method youClicked() {
     Clicks <- Clicks + 1


}
```

Cookie Clicker

Clicks: 7

4_Cookies: ✗

```
method youClicked() {
     Clicks <- Clicks + 1
     if Clicks is larger than 9,

}
```

Cookie Clicker

Clicks: 7

4_Cookies: ✗

```
method youClicked() {
     Clicks <- Clicks + 1
     if Clicks is larger than 9,
          then 4_Cookies <- ✓
}
```

Cookie Clicker

Clicks: 8

4_Cookies: ✗

```
method youClicked() {
     Clicks <- Clicks + 1
     if Clicks is larger than 9,
         then 4_Cookies <- ✓
}
```

Cookie Clicker

Clicks: 9

4_Cookies: ✗

```
method youClicked() {
     Clicks <- Clicks + 1
     if Clicks is larger than 9,
         then 4_Cookies <- ✓
}
```

Cookie Clicker

Clicks: 10

4_Cookies: ✓

```
method youClicked() {
     Clicks <- Clicks + 1
     if Clicks is larger than 9,
         then 4_Cookies <- ✓
}
```

Cookie Clicker

Clicks: 11

4_Cookies: ✓

```
method youClicked() {
     Clicks <- Clicks + 1
     if Clicks is larger than 9,
         then 4_Cookies <- ✓
}
```

Cookie Clicker

Clicks: 12

4_Cookies: ✓

```
method youClicked() {
     Clicks <- Clicks + 1
     if Clicks is larger than 9,
         then 4_Cookies <- ✓
}
```

Software Verification

Software Verification

- Testing: Click and check the outcome

Software Verification

- Testing: Click and check the outcome
- Verification: Proof that it works

Verifying Cookie Clicker

Clicks: ...
4_Cookies: ...

```
method youClicked() {
     Clicks <- Clicks + 1
     if Clicks is larger than 9,
         then 4_Cookies <- ✓
}
```

Verifying Cookie Clicker

Clicks: ...

4_Cookies: ...

assumes Clicks is positive

```
method youClicked() {
     Clicks <- Clicks + 1
     if Clicks is larger than 9,
         then 4_Cookies <- ✓
}
```

Verifying Cookie Clicker

Clicks: ...

4_Cookies: ...

```
assumes Clicks is positive

method youClicked() {
     Clicks <- Clicks + 1
     if Clicks is larger than 9,
         then 4_Cookies <- ✓
}

ensures Clicks is positive
```

Verifying Cookie Clicker

Clicks: ...

4_Cookies: ...

```
assumes Clicks is positive

method youClicked() {
     Clicks <- Clicks + 1
     if Clicks is larger than 9,
         then 4_Cookies <- ✓
}

ensures Clicks is old Clicks + 1
```

Verifying Cookie Clicker

Clicks: ...
4_Cookies: ...

```
assumes Clicks is positive

method youClicked() {
     Clicks <- Clicks + 1
     if Clicks is larger than 9,
         then 4_Cookies <- ✓
}

ensures Clicks is old Clicks + 1
```

```
ensures if Clicks larger than 9,
     then 4_Cookies is ✓
```

Software Verification in a nutshell

My thesis

My thesis

- Extending Support for Axiomatic Data Types (ADTs) in VerCors

Overview

Software Verification

VerCors

Axiomatic Data Types

Maps

Conclusion

VerCors

VerCors

- Static verification of concurrent/parallel programs.

VerCors

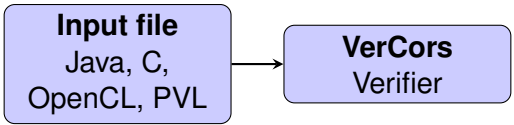
- Static verification of concurrent/parallel programs.
- Developed at the University of Twente

VerCors' Architecture

VerCors' Architecture

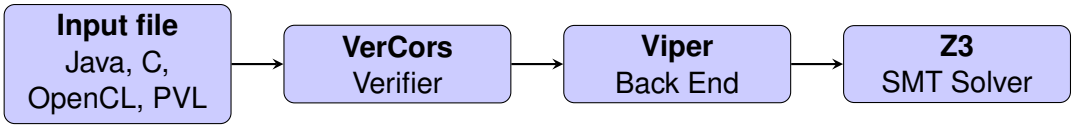
Input file
Java, C,
OpenCL, PVL

VerCors' Architecture



VerCors' Architecture

VerCors' Architecture



Overview

Software Verification

VerCors

Axiomatic Data Types

Maps

Conclusion

Axiomatic Data Types

Axiomatic Data Types

- Lists/sequences, sets, bags, tuples, integers, doubles, floats, etc.

Axiomatic Data Types

- Lists/sequences, sets, bags, tuples, integers, doubles, floats, etc.
- Concrete data types (CDTs)

Axiomatic Data Types

- Lists/sequences, sets, bags, tuples, integers, doubles, floats, etc.
- Concrete data types (CDTs)
- Describe *behavior* instead of *implementation*

Tuple

Tuple

- (`fst`, `snd`)

Tuple

- (`fst`, `snd`)
- Constructor to construct a Tuple

Tuple

- (`fst`, `snd`)
- Constructor to construct a Tuple
- Function to get the first element

Tuple

- (`fst`, `snd`)
- Constructor to construct a Tuple
- Function to get the first element
- Function to get the second element

Tuple CDT (in Java)

```
class Tuple<F,S> {









}
```

Tuple CDT (in Java)

```
class Tuple<F,S> {
    private final F fst;
    private final S snd;









}
```

Tuple CDT (in Java)

```
class Tuple<F,S> {
    private final F fst;
    private final S snd;

    public Tuple(F fstArg, S sndArg) {
        fst = fstArg;
        snd = sndArg;
    }



}
```

Tuple CDT (in Java)

```
class Tuple<F,S> {
    private final F fst;
    private final S snd;

    public Tuple(F fstArg, S sndArg) {
        fst = fstArg;
        snd = sndArg;
    }

    public F getFst() { return fst; }
    public S getSnd() { return snd; }
}
```

Tuple CDT (in Java)

```
class Tuple<F,S> {
    private final F fst;
    private final S snd;

    public Tuple(F fstArg, S sndArg) {
        fst = fstArg;
        snd = sndArg;
    }

    public F getFst() { return fst; }
    public S getSnd() { return snd; }
}
```

Tuple CDT (in Java)

```
class Tuple<F,S> {
    private final F fst;
    private final S snd;

    public Tuple(F fstArg, S sndArg) {
        fst = fstArg;
        snd = sndArg;
    }

    public F getFst() { return fst; }
    public S getSnd() { return snd; }
}
```

Tuple CDT (in Java)

```
class Tuple<F,S> {
    private final F fst;
    private final S snd;

    public Tuple(F fstArg, S sndArg) {
        fst = fstArg;
        snd = sndArg;
    }

    public F getFst() { return fst; }
    public S getSnd() { return snd; }
}
```

Tuple CDT (in Java)

```
class Tuple<F,S> {
    private final F fst;
    private final S snd;

    public Tuple(F fstArg, S sndArg) {
        fst = fstArg;
        snd = sndArg;
    }

    public F getFst() { return fst; }
    public S getSnd() { return snd; }
}
```

Tuple CDT (in Java)

```
class Tuple<F,S> {
    private final F fst;
    private final S snd;

    public Tuple(F fstArg, S sndArg) {
        fst = fstArg;
        snd = sndArg;
    }

    public F getFst() { return fst; }
    public S getSnd() { return snd; }
}
```

Tuple CDT (in Java)

```
class Tuple<F,S> {
    private final F fst;
    private final S snd;

    public Tuple(F fstArg, S sndArg) {
        fst = fstArg;
        snd = sndArg;
    }

    public F getFst() { return fst; }
    public S getSnd() { return snd; }
}
```

Tuple CDT (in Java)

```
class Tuple<F,S> {
    private final F fst;
    private final S snd;

    public Tuple(F fstArg, S sndArg) {
        fst = fstArg;
        snd = sndArg;
    }

    public F getFst() { return fst; }
    public S getSnd() { return snd; }
}
```

Tuple ADT

```
domain Tuple[F,S] {









}
```

Tuple ADT

```
domain Tuple[F,S] {
    function constructor(f:F, s:S): Tuple[F,S]

}
```

Tuple ADT

```
domain Tuple[F,S] {
    function constructor(f:F, s:S): Tuple[F,S]
    function getFst(t:Tuple[F,S]): F
    function getSnd(t:Tuple[F,S]): S









}
```

Tuple ADT

```
domain Tuple[F,S] {
    function constructor(f:F, s:S): Tuple[F,S]
    function getFst(t:Tuple[F,S]): F
    function getSnd(t:Tuple[F,S]): S

    axiom FstAxiom {
        forall f1:F, s1:S :: getFst(constructor(f1,s1)) == f1
    }

}
```

Tuple ADT

```
domain Tuple[F,S] {
    function constructor(f:F, s:S): Tuple[F,S]
    function getFst(t:Tuple[F,S]): F
    function getSnd(t:Tuple[F,S]): S

    axiom FstAxiom {
        forall f1:F, s1:S :: getFst(constructor(f1,s1)) == f1
    }

}
```

Tuple ADT

```
domain Tuple[F,S] {
    function constructor(f:F, s:S): Tuple[F,S]
    function getFst(t:Tuple[F,S]): F
    function getSnd(t:Tuple[F,S]): S

    axiom FstAxiom {
        forall f1:F, s1:S :: getFst(constructor(f1,s1)) == f1
    }

}
```

Tuple ADT

```
domain Tuple[F,S] {
    function constructor(f:F, s:S): Tuple[F,S]
    function getFst(t:Tuple[F,S]): F
    function getSnd(t:Tuple[F,S]): S

    axiom FstAxiom {
        forall f1:F, s1:S :: getFst( constructor(f1,s1) ) == f1
    }

}
```

Tuple ADT

```
domain Tuple[F,S] {
    function constructor(f:F, s:S): Tuple[F,S]
    function getFst(t:Tuple[F,S]): F
    function getSnd(t:Tuple[F,S]): S

    axiom FstAxiom {
        forall f1:F, s1:S :: getFst( constructor(f1,s1) ) == f1
    }

}
```

Tuple ADT

```
domain Tuple[F,S] {
    function constructor(f:F, s:S): Tuple[F,S]
    function getFst(t:Tuple[F,S]): F
    function getSnd(t:Tuple[F,S]): S

    axiom FstAxiom {
        forall f1:F, s1:S :: getFst( constructor(f1,s1) ) == f1
    }

    axiom SndAxiom {
        forall f1:F, s1:S :: getSnd(constructor(f1,s1)) == s1
    }
}
```

Tuple CDT & ADT

```
class Tuple<F, S> {
    private final F fst;
    private final S snd;

    public Tuple(F fstArg, S sndArg) {
        fst = fstArg;
        snd = sndArg;
    }

    public F getFst() { return fst; }
    public S getSnd() { return snd; }
}
```

```
domain Tuple[F,S] {
    function constructor(f:F, s:S): Tuple[F,S]
    function getFst(t:Tuple[F,S]): F
    function getSnd(t:Tuple[F,S]): S

    axiom FstAxiom {
        forall f1:F, s1:S ::
            getFst(constructor(f1,s1)) == f1
    }

    axiom SndAxiom {
        forall f1:F, s1:S ::
            getSnd(constructor(f1,s1)) == s1
    }
}
```

Tuple CDT & ADT

```
class Tuple<F, S> {
    private final F fst;
    private final S snd;

    public Tuple(F fstArg, S sndArg) {
        fst = fstArg;
        snd = sndArg;
    }

    public F getFst() { return fst; }
    public S getSnd() { return snd; }
}
```

```
domain Tuple[F,S] {
    function constructor(f:F, s:S): Tuple[F,S]
    function getFst(t:Tuple[F,S]): F
    function getSnd(t:Tuple[F,S]): S

    axiom FstAxiom {
        forall f1:F, s1:S ::
            getFst(constructor(f1,s1)) == f1
    }

    axiom SndAxiom {
        forall f1:F, s1:S ::
            getSnd(constructor(f1,s1)) == s1
    }
}
```

Tuple CDT & ADT

```
class Tuple<F,S> {
    private final F fst;
    private final S snd;

    public Tuple(F fstArg, S sndArg) {
        fst = fstArg;
        snd = sndArg;
    }

    public F getFst() { return fst; }
    public S getSnd() { return snd; }
}
```

```
domain Tuple[F,S] {
    function constructor(f:F, s:S): Tuple[F,S]
    function getFst(t:Tuple[F,S]): F
    function getSnd(t:Tuple[F,S]): S

    axiom FstAxiom {
        forall f1:F, s1:S ::
            getFst(constructor(f1,s1)) == f1
    }

    axiom SndAxiom {
        forall f1:F, s1:S ::
            getSnd(constructor(f1,s1)) == s1
    }
}
```

Tuple CDT & ADT

```
class Tuple<F,S> {
    private final F fst;
    private final S snd;

    public Tuple(F fstArg, S sndArg) {
        fst = fstArg;
        snd = sndArg;
    }

    public F getFst() { return fst; }
    public S getSnd() { return snd; }
}
```

```
domain Tuple[F,S] {
    function constructor(f:F, s:S) : Tuple[F,S]
    function getFst(t:Tuple[F,S]): F
    function getSnd(t:Tuple[F,S]): S

    axiom FstAxiom {
        forall f1:F, s1:S ::
            getFst(constructor(f1,s1)) == f1
    }

    axiom SndAxiom {
        forall f1:F, s1:S ::
            getSnd(constructor(f1,s1)) == s1
    }
}
```

Tuple CDT & ADT

```
class Tuple<F,S> {
    private final F fst;
    private final S snd;

    public Tuple(F fstArg, S sndArg) {
        fst = fstArg;
        snd = sndArg;
    }

    public F getFst() { return fst; }
    public S getSnd() { return snd; }
}
```

```
domain Tuple[F,S] {
    function constructor(f:F, s:S) : Tuple[F,S]
    function getFst(t:Tuple[F,S]) : F
    function getSnd(t:Tuple[F,S]) : S

    axiom FstAxiom {
        forall f1:F, s1:S ::
            getFst(constructor(f1,s1)) == f1
    }

    axiom SndAxiom {
        forall f1:F, s1:S ::
            getSnd(constructor(f1,s1)) == s1
    }
}
```

Soundness

Soundness

- Every formula that can be proven is logically valid

Unsoundness

Unsoundness

```
domain Tuple[F,S] {
     function constructor(f:F, s:S): Tuple[F,S]
     function getFst(t:Tuple[F,S]): F
     function getSnd(t:Tuple[F,S]): S
     axiom FstAxiom {forall f1:F,s1:S :: getFst(constructor(f1,s1)) == f1}
     axiom SndAxiom {forall f1:F,s1:S :: getSnd(constructor(f1,s1)) == s1}

     axiom UnsoundAxiom {


     }
}
```

Unsoundness

```
domain Tuple[F,S] {
     function constructor(f:F, s:S): Tuple[F,S]
     function getFst(t:Tuple[F,S]): F
     function getSnd(t:Tuple[F,S]): S
     axiom FstAxiom {forall f1:F,s1:S :: getFst(constructor(f1,s1)) == f1}
     axiom SndAxiom {forall f1:F,s1:S :: getSnd(constructor(f1,s1)) == s1}

    axiom UnsoundAxiom {
        forall f1:F, s1:S, s2:S ::

    }
}
```

Unsoundness

```
domain Tuple[F,S] {
     function constructor(f:F, s:S): Tuple[F,S]
     function getFst(t:Tuple[F,S]): F
     function getSnd(t:Tuple[F,S]): S
     axiom FstAxiom {forall f1:F,s1:S :: getFst(constructor(f1,s1)) == f1}
     axiom SndAxiom {forall f1:F,s1:S :: getSnd(constructor(f1,s1)) == s1}

    axiom UnsoundAxiom {
        forall f1:F, s1:S, s2:S ::

    }
}
```

Unsoundness

```
domain Tuple[F,S] {
     function constructor(f:F, s:S): Tuple[F,S]
     function getFst(t:Tuple[F,S]): F
     function getSnd(t:Tuple[F,S]): S
     axiom FstAxiom {forall f1:F,s1:S :: getFst(constructor(f1,s1)) == f1}
     axiom SndAxiom {forall f1:F,s1:S :: getSnd(constructor(f1,s1)) == s1}

    axiom UnsoundAxiom {
        forall f1:F, s1:S, s2:S ::

    }
}
```

Unsoundness

```
domain Tuple[F,S] {
     function constructor(f:F, s:S): Tuple[F,S]
     function getFst(t:Tuple[F,S]): F
     function getSnd(t:Tuple[F,S]): S
     axiom FstAxiom {forall f1:F,s1:S :: getFst(constructor(f1,s1)) == f1}
     axiom SndAxiom {forall f1:F,s1:S :: getSnd(constructor(f1,s1)) == s1}

    axiom UnsoundAxiom {
        forall f1:F, s1:S, s2:S ::
            getFst( constructor(f1,s1) ) != getFst( constructor(f1,s2) )
    }
}
```

Unsoundness

```
domain Tuple[F,S] {
     function constructor(f:F, s:S): Tuple[F,S]
     function getFst(t:Tuple[F,S]): F
     function getSnd(t:Tuple[F,S]): S
     axiom FstAxiom {forall f1:F,s1:S :: getFst(constructor(f1,s1)) == f1}
     axiom SndAxiom {forall f1:F,s1:S :: getSnd(constructor(f1,s1)) == s1}

    axiom UnsoundAxiom {
        forall f1:F, s1:S, s2:S ::
            getFst(constructor(f1,s1)) != getFst(constructor(f1,s2))
    }
}
```

Unsoundness

```
axiom UnsoundAxiom {
    forall f1:F, s1:S, s2:S ::
        getFst(constructor(f1,s1)) != getFst(constructor(f1,s2))
}
```

Unsoundness

```
axiom UnsoundAxiom {
    forall f1:F, s1:S, s2:S ::
        getFst(constructor(f1,s1)) != getFst(constructor(f1,s2))
}

method myMethod() {









}
```

Unsoundness

```
axiom UnsoundAxiom {
    forall f1:F, s1:S, s2:S ::
        getFst(constructor(f1,s1)) != getFst(constructor(f1,s2))
}

method myMethod() {
    var t1 := constructor(1,2)
    var t2 := constructor(1,3)









}
```

Unsoundness

```
axiom UnsoundAxiom {
    forall f1:F, s1:S, s2:S ::
        getFst(constructor(f1,s1)) != getFst(constructor(f1,s2))
}

method myMethod() {
    var t1 := constructor(1,2)
    var t2 := constructor(1,3)

    assert getFst(t1) == getFst(t2)     // 1 == 1, Verifies


}
```

Unsoundness

```
axiom UnsoundAxiom {
    forall f1:F, s1:S, s2:S ::
        getFst(constructor(f1,s1)) != getFst(constructor(f1,s2))
}

method myMethod() {
    var t1 := constructor(1,2)
    var t2 := constructor(1,3)

    assert getFst(t1) == getFst(t2)     // 1 == 1, Verifies
    assert getSnd(t1) == getSnd(t2)     // 2 == 3, Verifies???

}
```

Unsoundness

```
axiom UnsoundAxiom {
    forall f1:F, s1:S, s2:S ::
        getFst(constructor(f1,s1)) != getFst(constructor(f1,s2))
}

method myMethod() {
    var t1 := constructor(1,2)
    var t2 := constructor(1,3)

    assert getFst(t1) == getFst(t2)     // 1 == 1, Verifies
    assert getSnd(t1) == getSnd(t2)     // 2 == 3, Verifies???
    assert false                        // false ,  Verifies???
}
```

My thesis

My thesis

- Extending Support for Axiomatic Data Types (ADTs) in VerCors

My thesis

- Extending Support for Axiomatic Data Types (ADTs) in VerCors
- List of features on ADTs to support

Overview

Software Verification

VerCors

Axiomatic Data Types

Maps

Conclusion

Maps

Maps

collection of key/value pairs with unique keys

Maps

, immutable collection of key/value pairs with unique keys

Maps

, finite, immutable collection of key/value pairs with unique keys

Maps

- Unordered, finite, immutable collection of key/value pairs with unique keys

Maps

- Unordered, finite, immutable collection of key/value pairs with unique keys
- Based on the Dafny map axiomatization

Maps

- Unordered, finite, immutable collection of key/value pairs with unique keys
- Based on the Dafny map axiomatization
- 10 functions and 23 axioms

Modeling a map

Modeling a map

- keys (m: Map[K,V]): Set[K]

Modeling a map

- keys (m: Map[K,V]): Set[K]
- values (m: Map[K,V]): Set[V]

Modeling a map

- keys (m: Map[K,V]): Set[K]
- values (m: Map[K,V]): Set[V]
- items (m: Map[K,V]): Set[Tuple[K,V]]

Modeling a map

- keys (m: Map[K,V]): Set[K]
- values (m: Map[K,V]): Set[V]
- items (m: Map[K,V]): Set[Tuple[K,V]]
- get (m:Map[K,V], k: K): V

Map constructors

Map constructors

- empty(): Map[K,V]

Map constructors

- empty (): Map[K,V]
- build (m: Map[K,V], k: K, v: V): Map[K,V]

Map constructors

- empty (): Map[K,V]
- build (m: Map[K,V], k: K, v: V): Map[K,V]
- A map with pairs $1 \rightarrow 1$, $2 \rightarrow 4$ and $3 \rightarrow 9$

Map constructors

- empty (): Map[K,V]
- build (m: Map[K,V], k: K, v: V): Map[K,V]
- A map with pairs $1 \rightarrow 1$, $2 \rightarrow 4$ and $3 \rightarrow 9$

Map constructors

- empty (): Map[K,V]
- build (m: Map[K,V], k: K, v: V): Map[K,V]
- A map with pairs $1 \rightarrow 1$, $2 \rightarrow 4$ and $3 \rightarrow 9$

  empty()

Map constructors

- empty (): Map[K,V]
- build (m: Map[K,V], k: K, v: V): Map[K,V]
- A map with pairs $1 \rightarrow 1$, $2 \rightarrow 4$ and $3 \rightarrow 9$

  build(empty(), 1, 1)

Map constructors

- empty (): Map[K,V]
- build (m: Map[K,V], k: K, v: V): Map[K,V]
- A map with pairs $1 \rightarrow 1$, $2 \rightarrow 4$ and $3 \rightarrow 9$

  build(build(empty(), 1, 1), 2, 4)

Map constructors

- empty (): Map[K,V]
- build (m: Map[K,V], k: K, v: V): Map[K,V]
- A map with pairs $1 \rightarrow 1$, $2 \rightarrow 4$ and $3 \rightarrow 9$
- build(build(build(empty(), 1, 1), 2, 4), 3, 9)

Axioms on the `empty` function

```
axiom Ax1 {



}
```

Axioms on the `empty` function

```
axiom Ax1 {
     forall k1: K ::


}
```

Axioms on the `empty` function

```
axiom Ax1 {
     forall k1: K ::
          !(k1 in keys(empty())) &&

}
```

Axioms on the `empty` function

```
axiom Ax1 {
     forall k1: K ::
          !(k1 in keys(empty())) &&
          |keys(empty())| == 0
}
```

Axioms on the `empty` function

```
axiom Ax1 {
     forall k1: K ::
         !(k1 in keys(empty())) &&
         |keys(empty())| == 0
}
```

Axioms on the `empty` function

```
axiom Ax1 {
     forall k1: K ::
         !(k1 in keys(empty())) &&
         |keys(empty())| == 0
}

axiom Ax2 {
     forall v1: V ::
         !(v1 in values(empty())) &&
         |values(empty())| == 0
}
```

Axiom on the `build` function

```
axiom Ax3 {

}
```

Axiom on the `build` function

```
axiom Ax3 {
     forall k1: K, v1: V, m1: Map[K,V] ::


}
```

Axiom on the `build` function

```
axiom Ax3 {
     forall k1: K, v1: V, m1: Map[K,V] ::

}
```

Axiom on the `build` function

```
axiom Ax3 {
     forall k1: K, v1: V, m1: Map[K,V] ::
          k1 in keys(build(m1, k1, v1)) &&

}
```

Axiom on the `build` function

```
axiom Ax3 {
     forall k1: K, v1: V, m1: Map[K,V] ::
         k1 in keys(build(m1, k1, v1)) &&
         get(build(m1, k1, v1), k1) == v1
}
```

In practice

In practice

- How does Z3 use these axioms

In practice

- How does Z3 use these axioms
- For Universal Quantifiers:

In practice

- How does Z3 use these axioms
- For Universal Quantifiers: Candidates

Triggers

```
function f(i: Int): Int
```

Triggers

```
function f(i: Int): Int

assume forall k: Int :: f(f(k)) == f(k*k)+1
```

Triggers

```
function f(i: Int): Int

assume forall k: Int :: f(f(k)) == f(k*k)+1
```

Good trigger

```
function f(i: Int): Int

assume forall k: Int :: {f(f(k))} f(f(k)) == f(k*k)+1
```

Good trigger

```
function f(i: Int): Int

assume forall k: Int :: {f(f(k))} f(f(k)) == f(k*k)+1
assert f(f(3)) == f(9)+1 // Verifies
```

Good trigger

```
function f(i: Int): Int

assume forall k: Int :: {f(f(k))} f(f(k)) == f(k*k)+1
assert f(f(3)) == f(9)+1 // Verifies

bound k to 3
```

Good trigger

```
function f(i: Int): Int

assume forall k: Int :: {f(f(k))} f(f(k)) == f(k*k)+1
assert f(f(3)) == f(9)+1 // Verifies

bound k to 3, so f(f(3)) == f(3*3)+1
```

Restrictive trigger

```
function f(i: Int): Int

assume forall k: Int :: {f(f(f(k)))} f(f(k)) == f(k*k)+1
```

Restrictive trigger

```
function f(i: Int): Int

assume forall k: Int :: {f(f(f(k)))} f(f(k)) == f(k*k)+1
assert f(f(3)) == f(9)+1 // Fails
```

Looping trigger

```
function f(i: Int): Int

assume forall k: Int :: {f(k)} f(f(k)) == f(k*k)+1
```

Looping trigger

```
function f(i: Int): Int

assume forall k: Int :: {f(k)} f(f(k)) == f(k*k)+1
assert f(f(3)) == f(9)+1 // Verifies
```

Looping trigger

```
function f(i: Int): Int

assume forall k: Int :: {f(k)} f(f(k)) == f(k*k)+1
assert f(f(3)) == f(9)+1 // Verifies
assert f(f(3)) == f(4)+1 // Times out
```

Looping trigger

```
function f(i: Int): Int

assume forall k: Int :: {f(k)} f(f(k)) == f(k*k)+1
assert f(f(3)) == f(9)+1 // Verifies
assert f(f(3)) == f(4)+1 // Times out

bound k to f(3)
```

Looping trigger

```
function f(i: Int): Int

assume forall k: Int :: {f(k)} f(f(k)) == f(k*k)+1
assert f(f(3)) == f(9)+1 // Verifies
assert f(f(3)) == f(4)+1 // Times out

bound k to f(3), so f(f(f(3))) == f(f(3)*f(3))+1
```

Looping trigger

```
function f(i: Int): Int

assume forall k: Int :: {f(k)} f(f(k)) == f(k*k)+1
assert f(f(3)) == f(9)+1 // Verifies
assert f(f(3)) == f(4)+1 // Times out

bound k to f(3), so f(f(f(3))) == f(f(3)*f(3))+1
bound k to 3
```

Looping trigger

```
function f(i: Int): Int

assume forall k: Int :: {f(k)} f(f(k)) == f(k*k)+1
assert f(f(3)) == f(9)+1 // Verifies
assert f(f(3)) == f(4)+1 // Times out

bound k to f(3), so f(f(f(3))) == f(f(3)*f(3))+1
bound k to 3, so f(f(3)) == f(3*3)+1
```

Looping trigger

```
function f(i: Int): Int

assume forall k: Int :: {f(k)} f(f(k)) == f(k*k)+1
assert f(f(3)) == f(9)+1 // Verifies
assert f(f(3)) == f(4)+1 // Times out

bound k to f(3), so f(f(f(3))) == f(f(3)*f(3))+1
bound k to 3, so f(f(3)) == f(3*3)+1
bound k to 4
```

Looping trigger

```
function f(i: Int): Int

assume forall k: Int :: {f(k)} f(f(k)) == f(k*k)+1
assert f(f(3)) == f(9)+1 // Verifies
assert f(f(3)) == f(4)+1 // Times out

bound k to f(3), so f(f(f(3))) == f(f(3)*f(3))+1
bound k to 3, so f(f(3)) == f(3*3)+1
bound k to 4, so f(f(4)) == f(4*4)+1
```

Looping trigger

```
function f(i: Int): Int

assume forall k: Int :: {f(k)} f(f(k)) == f(k*k)+1
assert f(f(3)) == f(9)+1 // Verifies
assert f(f(3)) == f(4)+1 // Times out

bound k to f(3), so f(f(f(3))) == f(f(3)*f(3))+1
bound k to 3, so f(f(3)) == f(3*3)+1
bound k to 4, so f(f(4)) == f(4*4)+1
bound k to 16
```

Looping trigger

```
function f(i: Int): Int

assume forall k: Int :: {f(k)} f(f(k)) == f(k*k)+1
assert f(f(3)) == f(9)+1 // Verifies
assert f(f(3)) == f(4)+1 // Times out

bound k to f(3), so f(f(f(3))) == f(f(3)*f(3))+1
bound k to 3, so f(f(3)) == f(3*3)+1
bound k to 4, so f(f(4)) == f(4*4)+1
bound k to 16, so f(f(16)) == f(16*16)+1
```

Map triggers

Map triggers

```
void m7() {



}
```

Map triggers

```
void m7() {
     map<int,int> myMap = map<int,int> {1 -> 1, 2 -> 4, 3 -> 9};




}
```

Map triggers

```
void m7() {
     map<int,int> myMap = map<int,int> {1 -> 1, 2 -> 4, 3 -> 9};




}
```

Map triggers

```
void m7() {
     map<int,int> myMap = map<int,int> {1 -> 1, 2 -> 4, 3 -> 9};
     assert !(0 in keys(myMap));



}
```

Map triggers

```
void m7() {
     map<int,int> myMap = map<int,int> {1 -> 1, 2 -> 4, 3 -> 9};
     assert !(0 in keys(myMap));
     assert 1 in keys(myMap);
     assert 2 in keys(myMap);
     assert 3 in keys(myMap);
}
```

Axiom Profiler

Axiom Profiler

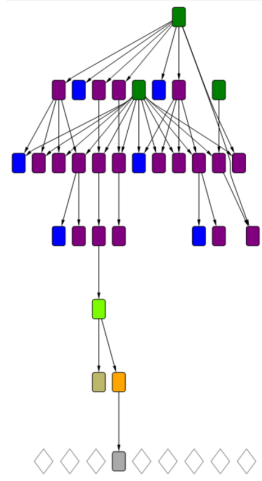
- A tool to debug and understand SMT quantifier instantiation
- Developed at ETH Zürich

Graph for trigger-based proof

```
void m7() {
     map<int,int> myMap =
         map<int,int> {1 -> 1, 2 -> 4, 3 -> 9};
     assert !(0 in keys(myMap));
     assert 1 in keys(myMap);
     assert 2 in keys(myMap);
     assert 3 in keys(myMap);
}
```

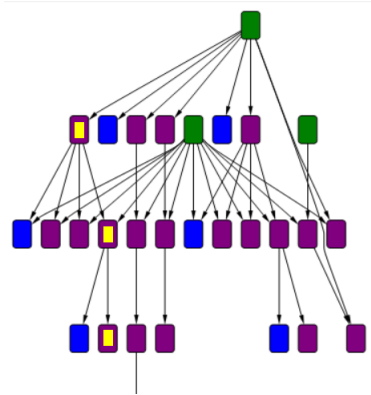
Graph for trigger-based proof

```
void m7() {
     map<int,int> myMap =
         map<int,int> {1 -> 1, 2 -> 4, 3 -> 9};
     assert !(0 in keys(myMap));
     assert 1 in keys(myMap);
     assert 2 in keys(myMap);
     assert 3 in keys(myMap);
}
```



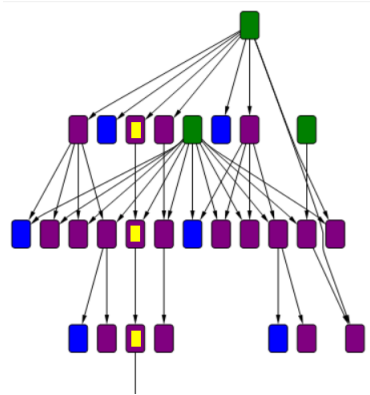
Graph for trigger-based proof

```
void m7() {
     map<int,int> myMap =
         map<int,int> { 1 -> 1, 2 -> 4, 3 -> 9 };
     assert !(0 in keys(myMap));
     assert 1 in keys(myMap);
     assert 2 in keys(myMap);
     assert 3 in keys(myMap);
}
axiom Ax3 {
     forall k1: K, v1: V, m1: Map[K,V] ::
          {build(m1, k1, v1)}
          k1 in keys(build(m1, k1, v1)) &&
          get(build(m1, k1, v1), k1) == v1
}
```



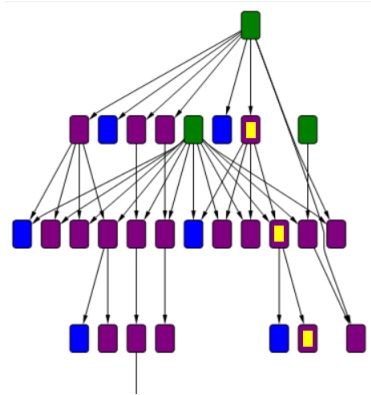
Graph for trigger-based proof

```
void m7() {
     map<int,int> myMap =
         map<int,int> { 1 -> 1, 2 -> 4, 3 -> 9 };
     assert !(0 in keys(myMap));
     assert 1 in keys(myMap);
     assert 2 in keys(myMap);
     assert 3 in keys(myMap);
}
axiom Ax3 {
     forall k1: K, v1: V, m1: Map[K,V] ::
          {build(m1, k1, v1)}
          k1 in keys(build(m1, k1, v1)) &&
          get(build(m1, k1, v1), k1) == v1
}
```



Graph for trigger-based proof

```
void m7() {
     map<int,int> myMap =
         map<int,int> { 1 -> 1, 2 -> 4, 3 -> 9 };
     assert !(0 in keys(myMap));
     assert 1 in keys(myMap);
     assert 2 in keys(myMap);
     assert 3 in keys(myMap);
}
axiom Ax3 {
     forall k1: K, v1: V, m1: Map[K,V] ::
          {build(m1, k1, v1)}
          k1 in keys(build(m1, k1, v1)) &&
          get(build(m1, k1, v1), k1) == v1
}
```



Graph for arith/basic theorem prover

Graph for arith/basic theorem prover

```
requires (\forall int i; 0 <= i && i < 100; i in keysMap(a) && a[i] == i*i);
void m3(map<int, int> a) {
     assert a[20] == 400;
}
```

Graph for arith/basic theorem prover

```
requires (\forall int i; 0 <= i && i < 100; i in keysMap(a) && a[i] == i*i);
void m3(map<int, int> a) {
      assert a[20] == 400;
}
```

Graph for arith/basic theorem prover

```
requires ( \forall int i; 0 <= i && i < 100 ; i in keysMap(a) && a[i] == i*i);
void m3(map<int, int> a) {
     assert a[20] == 400;
}
```

Graph for arith/basic theorem prover

```
requires (\forall int i; 0 <= i && i < 100; i in keysMap(a) && a[i] == i*i );
void m3(map<int, int> a) {
     assert a[20] == 400;
}
```

Graph for arith/basic theorem prover

```
requires (\forall int i; 0 <= i && i < 100;
         i in keysMap(a) && a[i] == i*i);
void m3(map<int, int> a) {
     assert a[20] == 400;
}
```

Graph for arith/basic theorem prover

Knowledge :

```
requires (\forall int i; 0 <= i && i < 100;
         i in keysMap(a) && a[i] == i*i);
void m3(map<int, int> a) {
     assert a[20] == 400;
}
```

Graph for arith/basic theorem prover

```
requires (\forall int i; 0 <= i && i < 100;
         i in keysMap(a) && a[i] == i*i);
void m3(map<int, int> a) {
     assert a[20] == 400;
}
```

Knowledge :

- Introduce new integer i' (introduced by Viper)

Graph for arith/basic theorem prover

```
requires (\forall int i; 0 <= i && i < 100;
         i in keysMap(a) && a[i] == i*i);
void m3(map<int, int> a) {
     assert a[20] == 400;
}
```

Knowledge :

- Introduce new integer i' (introduced by Viper)
- $0 \leq$ i' < 100

Graph for arith/basic theorem prover

```
requires (\forall int i; 0 <= i && i < 100;
         i in keysMap(a) && a[i] == i*i);
void m3(map<int, int> a) {
     assert a[20] == 400;
}
```

Knowledge :

- Introduce new integer i' (introduced by Viper)
- $0 \leq$ i' < 100
- a[i'] == i' $\times$ i'

Graph for arith/basic theorem prover

```
requires (\forall int i; 0 <= i && i < 100;
         i in keysMap(a) && a[i] == i*i);
void m3(map<int, int> a) {
     assert a[20] == 400;
}
```

Knowledge :

- Introduce new integer i' (introduced by Viper)
- $0 \leq i' < 100$
- a[i'] == $i' \times i'$
- $0 \leq 20 < 100$

Graph for arith/basic theorem prover

```
requires (\forall int i; 0 <= i && i < 100;
         i in keysMap(a) && a[i] == i*i);
void m3(map<int, int> a) {
     assert a[20] == 400;
}
```

Knowledge :

- Introduce new integer i' (introduced by Viper)
- $0 \leq$ i' < 100
- a[i'] == i' $\times$ i'
- $0 \leq 20 < 100$
- $20 \times 20 == 400$

Overview

Software Verification

VerCors

Axiomatic Data Types

Maps

Conclusion

Implemented features on ADTs

Implemented features on ADTs

- Checks if its argument is empty

Implemented features on ADTs

- Checks if its argument is empty
- Append/prepend a single value to a sequence

Implemented features on ADTs

- Checks if its argument is empty
- Append/prepend a single value to a sequence
- Take a range from a sequence

Implemented features on ADTs

- Checks if its argument is empty
- Append/prepend a single value to a sequence
- Take a range from a sequence
- Remove an element from a sequence by its index

Implemented features on ADTs

- Checks if its argument is empty
- Append/prepend a single value to a sequence
- Take a range from a sequence
- Remove an element from a sequence by its index
- Simple constructors for sequences, sets and bags which do not need a user-provided type if it has values

Implemented features on ADTs

- Checks if its argument is empty
- Append/prepend a single value to a sequence
- Take a range from a sequence
- Remove an element from a sequence by its index
- Simple constructors for sequences, sets and bags which do not need a user-provided type if it has values
- Subset notation $<=$ and $<$ on sets and bags

Implemented features on ADTs

- Checks if its argument is empty
- Append/prepend a single value to a sequence
- Take a range from a sequence
- Remove an element from a sequence by its index
- Simple constructors for sequences, sets and bags which do not need a user-provided type if it has values
- Subset notation $<=$ and $<$ on sets and bags
- Set comprehension

Implemented features on ADTs

- Checks if its argument is empty
- Append/prepend a single value to a sequence
- Take a range from a sequence
- Remove an element from a sequence by its index
- Simple constructors for sequences, sets and bags which do not need a user-provided type if it has values
- Subset notation $<=$ and $<$ on sets and bags
- Set comprehension
- Maps

Implemented features on ADTs

- Checks if its argument is empty
- Append/prepend a single value to a sequence
- Take a range from a sequence
- Remove an element from a sequence by its index
- Simple constructors for sequences, sets and bags which do not need a user-provided type if it has values
- Subset notation $<=$ and $<$ on sets and bags
- Set comprehension
- Maps
- Generic classes and functions

Questions