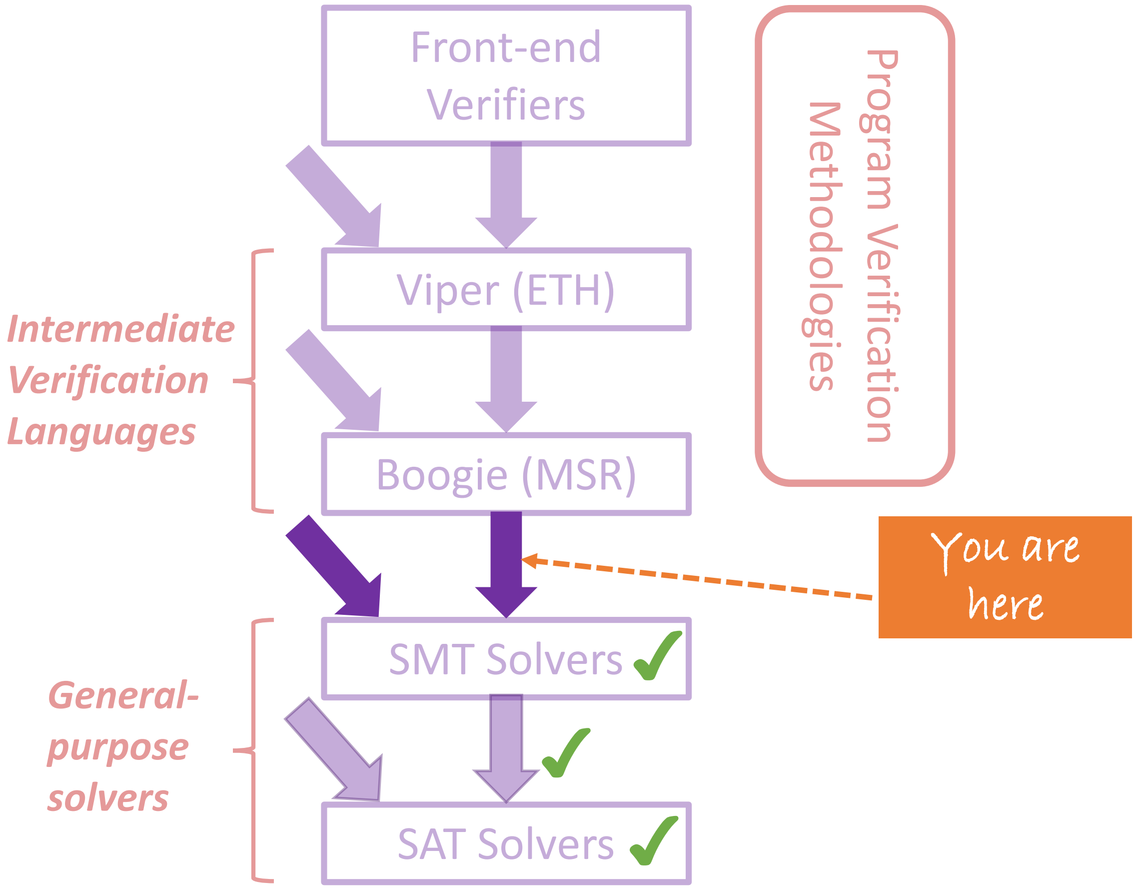
5. Encoding to SMT

Program Verification

ETH Zurich, Spring Semester 2017
Alexander J. Summers

Next up: Encoding to SMT



- Focus: *modelling custom theories and concepts* using SMT
- Expressing additional "types"
  - pairs/tuples, ADTs, maps/arrays, sets, sequences, graphs, heaps
- Typical approach combines:
  - *uninterpreted functions*
  - *quantifiers* (with suitable *triggers*)
  - *modelling tricks during translation*
- We'll develop most examples live in Viper (`viper.ethz.ch`)
  - example code on course website

Viper as a Modelling Language

- We'll use a subset of the *Viper language* for concreteness / checking
- When modelling a new type/concept, we'll typically:
  - Introduce *new types* (uninterpreted sorts), e.g. `Nat`
  - Declare *uninterpreted functions*:

```
function zero() : Nat
function succ(n: Nat) : Nat
function plus(m: Nat, n: Nat) : Nat
```

  - Define *axioms*: extra *assumed properties* (axiom names are not significant)

```
axiom plus_zero {
    forall n: Nat :: {plus(zero(),n)} plus(zero(),n) == n
}
```

  - In Viper, new sorts are declared using *domain* declarations
  - Domain declarations can include function declarations and axiom definitions

```
domain Nat {
  function zero() : Nat
```

- these are not scopes; no restrictions on types used inside
- typically group a bunch of related functions and axioms

Modelling Pairs

- Perhaps the simplest data type: let's consider modelling *pairs*
  - we'll start with pairs of (natively-supported) integers (m,n)
  - In Viper, the type for built-in integers is called `Int`

- What operations do we expect for pairs?
  - *constructing* pairs (m,n)
  - *projection functions* (deconstructing pairs); typically called fst and snd

- What are properties which should be true for pairs?
  - *disequality* properties, e.g. $(1,2) \neq (2,1)$
  - *projection* properties, e.g. $\mathsf{snd}(1,2) = 2$
  - *identity* properties, e.g. $\mathsf{fst}(\mathrm{p}) = \mathsf{fst}(\mathrm{q}) \land \mathsf{snd}(\mathrm{p}) = \mathsf{snd}(\mathrm{q}) \Rightarrow \mathrm{p} = \mathrm{q}$

Pairs: an Axiomatisation

- Example axiomatization in Viper syntax (developed in class)

```
domain IntPair {
  function pair(x: Int, y:Int) : IntPair
  function fst(p: IntPair) : Int
  function snd(p: IntPair) : Int

  axiom fst_defn {
    forall x:Int, y:Int :: {pair(x,y)} (fst(pair(x,y)) == x)
  }
  axiom snd_defn {
    forall x:Int, y:Int :: {pair(x,y)} (snd(pair(x,y)) == y)
  }
  axiom bijection {
      forall p: IntPair :: {fst(p)}{snd(p)} pair(fst(p),snd(p))==p
  }
}
```

- Note: we could easily analogously define pairs of other types

Testing an Axiomatisation

- We write mini "programs" to *test properties of our axiomatization*
- These consist of *Viper methods*, taking arbitrary parameters
  - parameters will be encoded as uninterpreted constants of the correct sort
- Method bodies will be sequence of `assume` and `assert` statements
  - each `assert` $\mathtt{A}$ statement prescribes an entailment to be proved
  - all axioms, and formulas from *prior* `assume` and `assert` statements are *assumed to hold*; the formula $\mathtt{A}$ must be shown to be *entailed by these*
- For example, in a program with axioms $\mathtt{A}$, a method body

  `assume` $\mathtt{A}_1$; `assert` $\mathtt{A}_2$; `assume` $\mathtt{A}_3$; `assert` $\mathtt{A}_4$;

would give rise to the entailments $\mathtt{A} \wedge \mathtt{A}_1 \vDash \mathtt{A}_2$ and $\mathtt{A} \wedge \mathtt{A}_1 \wedge \mathtt{A}_2 \wedge \mathtt{A}_3 \vDash \mathtt{A}_4$

- equivalently, check `unsat` of $\mathtt{A} \wedge \mathtt{A}_1 \wedge \neg \mathtt{A}_2$ and $\mathtt{A} \wedge \mathtt{A}_1 \wedge \mathtt{A}_2 \wedge \mathtt{A}_3 \wedge \neg \mathtt{A}_4$

Pairs: a Test Program

- The properties from slide 107 can be tested e.g. as follows:

```
method test(p : IntPair, q:IntPair) {
  assert pair(1,2) != pair(2,1)
  assert snd(pair(1,2)) == 2
  assert fst(p) == fst(q) && snd(p) == snd(q) ==> p == q
}
```

- All assertions can be proved, using the definitions from slide 108
  - If you run the examples with the Viper verifiers, you should get no errors
- *Very good exercises* (for all examples):
  - check for yourself *how these assertions could be proved* from the axioms
  - which axioms need to be instantiated, and *do the triggers allow this*?
  - experiment with *removing axioms* / *changing triggers*
  - Why do the axioms provided *not give rise to matching loops*?

Peano Natural Numbers : Intro

- Consider a data type `data Nat = Zero | Succ Nat`
- How can we model this in SMT?
- What concepts do we need to model?
  - depends on additional functions over the data type (e.g. `sign`, `+`, etc.)
  - `sign` is meant to return `1` for strictly positive `Nat` values; `0` otherwise
- Some example desired properties (`x`,`y`,`z` uninterpreted `Nat` constants)
  - $\mathtt{sign(Succ(Zero))} = 1$
  - $\mathtt{x} \neq \mathtt{Zero} \Rightarrow \mathtt{sign(x)} = 1$
  - $\forall \mathtt{n:Nat.}\ \mathtt{sign(n)} = 0 \vee \mathtt{sign(n)} = 1$
  - $\mathtt{sign(y)} \neq 0 \Rightarrow \mathtt{sign(x+y)} \neq 0$
  - $\mathtt{succ(Zero)+succ(x)} = \mathtt{succ(succ(x))}$
  - $\mathtt{(x+(y+z))} = \mathtt{((x+y)+z)}$

Peano Natural Numbers: an Axiomatisation

- Modelling `data Nat = Zero | Succ Nat`

```
domain Nat {
  function zero() : Nat
  function succ(n: Nat) : Nat

  function tag(n: Nat) : Int
  axiom tag_zero {
    tag(zero()) == 0
  }
  axiom tag_succ {
    forall n: Nat :: {succ(n)} tag(succ(n)) == 1
  }
  axiom all_tags {
    forall n: Nat :: {tag(n)} (tag(n) == 0 && n == zero()) ||
      (tag(n) == 1 && exists m: Nat :: n==succ(m))
  }
```

- `tag`: which *case of the data type definition* does a value come from?

Peano Natural Numbers : Sign Function I

- Consider modelling a "sign" function defined in pseudo-code as:

```
function sign(n:Nat):Int = n match {
  case Zero => 0
  case Succ m => 1
}
```

- One attempt at a definition:

```
function sign(n: Nat) : Int
axiom sign_zero {
  forall n: Nat :: {sign(zero())} sign(zero()) == 0
}
axiom sign_succ {
  forall m: Nat :: {sign(succ(m))} sign(succ(m)) == 1
}
```

- We can't show e.g. `x != zero() ==> sign(x) == 1` (why?)

Peano Natural Numbers : Sign Function II

- There are *two main problems*:
  - we don't trigger the sign definition axioms, using an arbitrary `Nat` constant
  - we don't find out "these are the only two cases" from anywhere
- We can solve both problems using the `tag` function
  - the `all_tags` axiom (slide 112) says there are just two cases
  - we encode pattern-matching in the definition of `sign` by testing `tag(n)`
  - i.e. `n` matching a `Zero` term is represented by `tag(n)==0`
  - `n` matching a `Succ m` term is represented by `tag(n)==1`

```
axiom sign_def {
  forall n: Nat :: {sign(n)} sign(n) == (tag(n) == 0 ? 0 : 1)
}
```

- Now we obtain `x != zero() ==> sign(x) == 1` (how?)
  - More generally, we can show $\forall$`n:Nat. sign(n) = 0` $\vee$ `sign(n) = 1`

Peano Natural Numbers : Addition I

- Consider modelling `Nat` addition, defined in pseudo-code as:

```
function plus(m:Nat, n:Nat):Nat = m match {
  case Zero => n
  case Succ p => Succ(plus(p,n))
}
```

- Following the same approach, we might start writing:

```
function plus(m:Nat, n: Nat) : Int
axiom plus_def {
  forall m: Nat, n:Nat :: {plus(m,n)} plus(m,n) == (tag(m) == 0 ? n : succ(plus(???,n)))
}
```

- It's not clear how to fill in the ??? for the "recursive" call
  - We could go back to writing individual axioms for the cases of the function
  - Alternatively, we can define a "*destructor*" for the `succ` "*constructor*" …

Peano Natural Numbers : Addition II

- We introduce a destructor (inverse function) for `succ`
  - the idea: for any `n` which is expressible as `succ(m)` for some `m`, we want the `m`
  - we introduce a function `pred` to play the role of mapping such `n` to such `m`
  - for ADTs, sometimes called a *destructor*; in Scala, called an *unapply method*
- Axiomatised as follows:

```
function pred(m : Nat) : Nat
axiom succ_inv {
    forall n: Nat :: {succ(n)} pred(succ(n)) == n
}
axiom pred_inv {
    forall n: Nat :: {pred(n)} tag(n)==1 ==> succ(pred(n)) == n
}
```

- Note we only constrain the definition of `pred` for *non-zero arguments*
  - normally only *total functions* are directly supported in SMT
  - we typically use *under-constrained total functions* to model partial ones

Peano Natural Numbers : Addition III

- We can now define the function

```
function plus(m:Nat, n:Nat):Nat = m match {
   case Zero => n
   case Succ p => Succ(plus(p,n))
}
```

```
function plus(m:Nat, n: Nat) : Int
axiom plus_def {
  forall m: Nat, n:Nat :: {plus(m,n)} plus(m,n) == (tag(m) == 0 ? n : succ(plus(pred(m),n)))
}
```

- The `pred(m)` term works in place of `p` in the above definition
  - This general approach works for *any ADT and simple pattern-matching*
  - If the constructor takes *multiple parameters*, we need *multiple destructors*
  - This is how we handled pairs already (*destructors = projection functions*)

Recursive Definitions and Matching Loops

- The modelling of `plus` shown so far causes potential matching loops

```
function plus(m:Nat, n: Nat) : Int
axiom plus_def {
  forall m: Nat, n:Nat :: {plus(m,n)} plus(m,n) == (tag(m) == 0 ? n : succ(plus(pred(m),n)))
}
```

- An instantiation for `plus(t,s)` will lead to one for `plus(pred(t),s)` , then one for `plus(pred(pred(t)),s)`, etc....

- What policy would we like for choosing *finitely-many* instantiations?
  - one idea: unroll definition just once for each *original occurrence* of `plus`
  - this idea is sometimes called *limited functions* (e.g. in the Dafny verifier)

Limited Functions

- We introduce a second function `plusL` with same signature as `plus`
  - this function is called the *limited version* of `plus`

```
function plusL(m:Nat, n: Nat) : Int
```

- We add an axiom, expressing that the two functions are equal
  - the axiom is triggered on using `plus` but not on using `plusL`

```
axiom plus_limited {
    forall m: Nat, n:Nat :: {plus(m,n)} plus(m,n) == plusL(m,n)
}
```

- In axioms encoding recursive definitions (such as `plus_def`)
  - for the “recursive calls” to the function, we use `plusL` instead of `plus`
  - we trigger the axiom on using `plus` but not on using `plusL`

```
axiom plus_def {
  forall m: Nat, n:Nat :: {plus(m,n)} plus(m,n) == (tag(m) == 0 ? n : succ(plusL(pred(m),n)))
}
```

Beyond Limited Functions

- Using limited functions avoids matching loops, but we still miss some facts which could be proven with finite instantations (e.g. $1+1 = 2$?) :

```
assert plus(succ(zero()), succ(zero())) == succ(succ(zero()))
```

- The assertion fails because we only instantiate once for each `plus`
  - too restrictive when we concretely know the *structure of the argument*
- Instead, we can add specific axioms for these cases
  - we use *knowing the structure of the argument* to govern instantiation

```
axiom plus_zero {
  forall n: Nat :: {plus(zero(),n)} plus(zero(),n) == n
}
axiom plus_succ {
  forall m: Nat, n: Nat :: {plus(succ(m),n)} plus(succ(m),n) == succ(plus(m,n))
}
```

  - note that we *don't use the limited versions*: we unroll for all known structure

(No) Induction

- Some properties don't follow from finitely instantiating the axioms

```
assert forall x:Nat, y:Nat, z:Nat :: plus(x,plus(y,z)) == plus(plus(x,y),z)
```

- Proving this property formally requires an *inductive argument*
  - SMT solvers typically do not natively support/construct inductive proofs
  - i.e. choosing *when to apply* induction, on *which variables*, and *what to prove*
- If we need such properties, one option is to add them as *extra axioms*
- If we *know* how induction needs to be applied, another option is:
  - formulate the property to be proven by induction; e.g. (by induction on `x`):

```
forall y:Nat, z:Nat :: plus(x,plus(y,z)) == plus(plus(x,y),z)
```

  - check instead the *premises of the induction schema* as assertions:

```
assert forall y:Nat, z:Nat :: plus(zero(),plus(y,z)) == plus(plus(zero(),y),z)
assert forall m:Nat, y:Nat, z:Nat :: plus(m,plus(y,z)) == plus(plus(m,y),z)
                                  ==> plus(succ(m),plus(y,z)) == plus(plus(succ(m),y),z)
```

Sequences

- We now consider how to model finite sequences as a type
  - sequences can be *indexed by integers*, and store some (fixed) type T of values
  - for concreteness/simplicity, we'll define sequences of integers as an example
  - we won't enumerate all properties, but will show interesting examples
- What operations might we expect for a sequence type? Examples:
  - sequences `s` have a (finite) *length*: `|s|`
  - it must be possible to *lookup* sequence elements (within bounds): `s(i)`
  - *empty* sequences: `[]` *singleton* sequences: `[v]`
  - *append* operation: s_1`++`s_2, *take*/*drop* `n` elements: `s[n..]` / `s[..n]`
- Note that sequences are not (functional) lists, and *not classical ADTs*
  - there can be many ways to construct the "same" sequence
  - e.g. `[1]++([2]++[3])` vs `([1]++[2])++[3]`
  - there is no canonical "constructor" function for arbitrary-length sequences

Representing Sequences

- Conceptually, sequences *could* be modelled as functions
  - for input problems involving a *statically-known, finite* number of sequence instances, this a possible representation (one function per sequence)
  - all relevant properties would have to be *copied for each sequence*
  - we can't write axioms which quantify over functions (not first-order logic)
- In general, we can instead mimic this idea via *defunctionalisation*
  - represent sequences instead with an *uninterpreted sort* (not functions)
  - add a `lookup` function, parameterised by *both a sequence and an index*
  - `lookup(s,i)` represents looking-up element `i` in the sequence `s`
  - Defunctionalisation is a *general trick* for removing some higher-order features

```
domain Sequence[T] { // note: "Seq" is actually a reserved keyword in Viper
  function lookup(s: Sequence[T], index: Int) : T
  function length(s: Sequence[T]) : Int
```

Length Properties

- Axioms to express how `length` interacts with sequence constructions

```
axiom length_empty {
  length(empty()) == 0
}
axiom length_singleton {
  forall i: Int :: {length(singleton(i))} length(singleton(i)) == 1
}
axiom length_append {
  forall s1: Sequence, s2: Sequence :: length(append(s1,s2)) == length(s1) + length(s2)
}
```

- What would be good trigger(s) for the last axiom?
- Using these axioms, can we prove e.g.

```
assert forall s1: Sequence, s2: Sequence :: length(append(s1,s2)) >= length(s1)
```

  - actually *no*: we are missing one important property:

```
axiom length_nonneg {
  forall s:Sequence :: {length(s)} length(s) >= 0
}
```

Bidirectional Triggering

- Consider appropriate triggers for the following axiom:

```
axiom length_append {
  forall s1: Sequence, s2: Sequence :: length(append(s1,s2)) == length(s1) + length(s2)
}
```

- One obvious choice seems to be `{length(append(s1,s2))}`
  - lets us "unroll" what it means to take the length of two appended sequences
  - on mentioning the length of the appended sequence, we get to unroll
- What about the following test case?

```
assert append(s1,s2) == s1  ==>  length(s2) == 0;
```

  - we can't use the above axiom (with the given trigger) to prove this
  - we also need triggers in *other direction* (from length of the *subsequences*)
  - we add the alternate triggers `{length(s1), append(s1,s2)}` and `{length(s2), append(s1,s2)}` – the latter lets us prove the test case

Extensional Equality

- Are `[1]++([2]++[3])` and `([1]++[2])++[3]` the same sequences?
  - they are constructed differently; nothing so far would tell us so
  - but whenever we apply functions to them, we get the same results
  - in particular, they have the same `length` and `lookup` behaviours
- This idea of “observational equality” is called *extensionality*
  - for ADTs, we get this from the uniqueness of construction (via destructors)
  - for unbounded non-ADT types like sequences, this property is not for free
  - e.g. a similar test case fails: `assert length(s2) == 0  ==>  append(s1,s2) == s1;`
- We could simply omit extensional equality from our type
  - conceptually, the type is then “larger” than the intended mathematical one
  - there are many *not-provably-equal* “copies” of a mathematical object
- Alternatively, we can add an explicit axiom for extensionality …

Axiomatising Extensionality

- The following axiom expresses extensionality for sequences:

```
axiom extensionality {
  forall s1: Sequence, s2: Sequence ::
    (length(s1) == length(s2) &&
     forall i : Int :: 0<= i && i < length(s1) ==> lookup(s1,i) == lookup(s2,i))
        ==> s1 == s2
}
```

- Choosing good triggers for the outer quantifier is very challenging
  - Anything more restrictive than instantiating for every pair of sequences in the input problem is most-likely incomplete for some example test case
  - We can trigger for *every pair of sequences*; see the exercises
  - Unfortunately, this can be very expensive for a problem with many sequences
  - In practice, many tools take some approximation of this complete approach
    - e.g. instantiate only for sequence terms explicitly equated in the original input formula

Encoding to SMT - Summary

- We have covered many issues arising when modelling types in SMT
  - for many (but not all) of these issues, there are *general approaches* to take
- The combination of uninterpreted functions and axioms is powerful
  - often complemented by adding *additional functions* to express key concepts
- Modelling types is an important ingredient in most SMT encodings
  - we will use similar techniques for modelling e.g. *program heaps as maps*
- In the next lecture, we will encode (stateful) program control flow
  - combined with ideas from today, this will let us *build a first program verifier*

Encoding to SMT – Some References

- E-Matching:
  - *Efficient E-Matching for SMT Solvers.* Leonardo de Moura, Nikolaj Bjørner (2007)
  - *Programming with Triggers.* Michał Moskal (2009)
- Encoding Resur:
  - *Efficient E-Matching for SMT Solvers.* Leonardo de Moura, Nikolaj Bjørner (2007)
  - *Programming with Triggers.* Michał Moskal (2009)
- Other teaching material: *Quantifiers.* Leonardo de Moura (SAT/SMT Summer School 2012)
- See also: *Z3 – A Tutorial.* Leonardo de Moura, Nikolaj Bjørner (2011)
  - and `http://rise4fun.com/z3/tutorial`