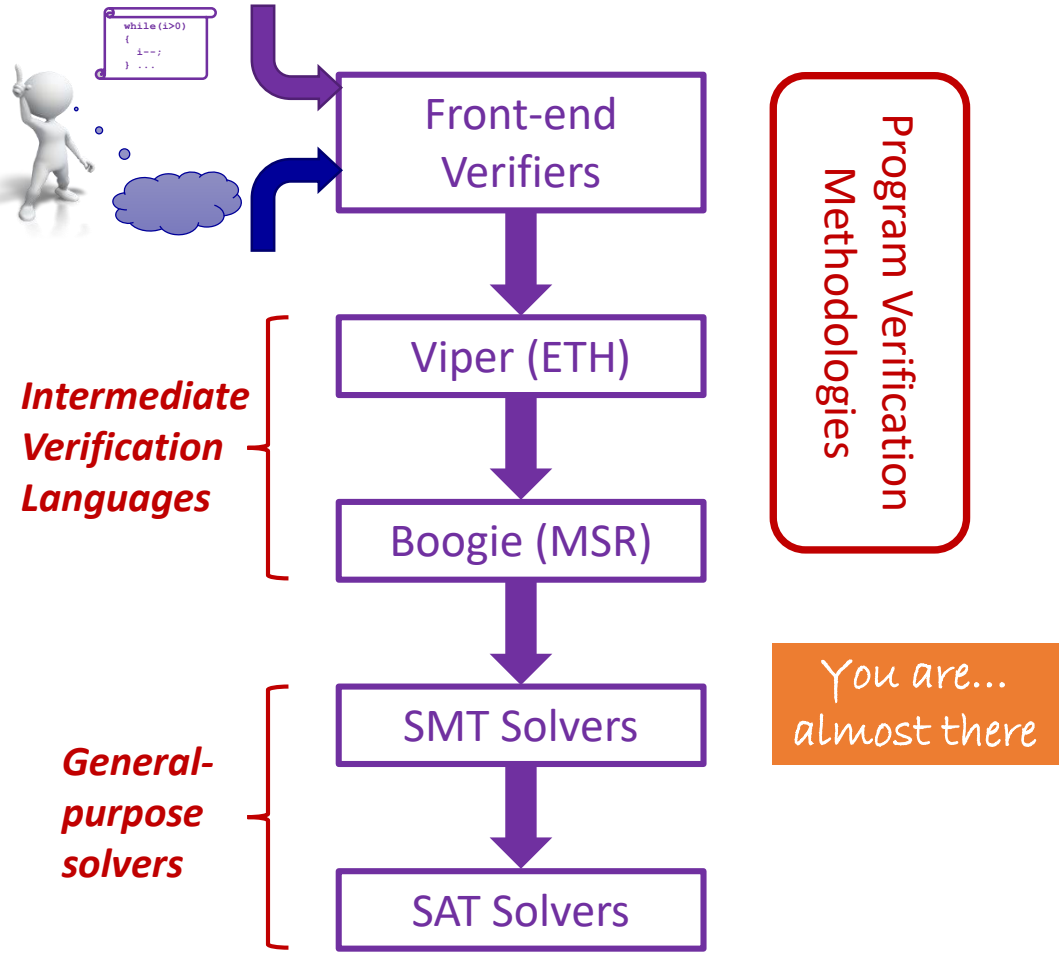
13. Good Abstractions and Research Topics

Program Verification

ETH Zurich, Spring Semester 2017

Alexander J. Summers

A Program Verification Tool Stack



- We've covered a *tool-stack for building program verifiers* (bottom-upwards)
  - Each layer *builds on that (directly) below*
  - User interacts at the *top-level only*
  - Not *all* users will have taken this course ☺
- We'll consider some *design decisions and issues* when developing a layer
  - Focus on *good abstractions:* enable tools/users above to interact without considering the *entire tool stack*
  - Must decide carefully *which abstractions* to aim for, which *details to expose*
- We'll also discuss some *research topics* within the scope of the course

Good Abstractions

- Providing *language abstractions* is a critical aspect of the tool chain
  - Where possible, present the *fiction* of working with higher-level concepts
  - Abstract away how these concepts are *mapped to lower-level tools*
  - For example, Viper's support for *permissions* as a native language feature
- Support for a language abstraction should give *predictable results*
  - Ideally: user should have a clear, high-level view of what they can expect
  - If the support is *unreliable*, a user has to think *in terms of the encoding*
  - Such "*leaky abstractions*" require the user to think many levels downwards
- Unreliability can come in many forms
  - *Expected properties cannot be deduced* by the underlying tools
  - Runtimes are *unpredictable, or too slow* to be usable
  - *Unexpected properties can be deduced* by the tool - being more complete for some examples may *not* necessarily be *more understandable* for the user

10 lines of Simple Viper

- Let's consider how the following example is actually verified:
  - very simple compared to some of the Viper programs we've seen
  - *8 lines* of *Viper* code map down to *414 lines* of *Boogie* code, and *871* lines of *smtlib* for Z3

```
field val : Int

method test(s:Set[Ref], x:Ref, y:Ref) {
  inhale acc(x.val)
  assert perm(x.val) > none
  inhale forall r:Ref :: r in s ==> acc(r.val)
  assert y in s ==> perm(y.val) > none
}
```

- The example makes use of several *language abstractions:*
  - built-in *heap*, *permissions*, mathematical *sets*
- Viper users can (hopefully) think in terms of these native features
  - e.g. encoding sets down to Boogie involves pre-defined *quantified axioms*
  - good triggers for these axioms are *critical for maintaining the abstraction*
- What about support for *permission-related constructs*?

Encoding Permissions in Boogie

- Current field permissions are represented by a `Mask` map in Boogie
  - the map is a `var`: value is updated in the Boogie code
  - used to represent field permissions held in the *current method scope*
- For each Viper method to be verified, we generate a Boogie procedure
  - the `Mask` is initially defined to be `ZeroMask` for each

```
type Field A B; // parameters not relevant here
type Perm = real;
type MaskType = <A, B> [Ref, Field A B]Perm;
var Mask: MaskType;
const ZeroMask: MaskType;
axiom (forall <A, B> o_1: Ref, f_3: (Field A B) ::
  { ZeroMask[o_1, f_3] }
  ZeroMask[o_1, f_3] == NoPerm
);
const NoPerm: Perm;
axiom NoPerm == 0.000000000;
const FullPerm: Perm;
axiom FullPerm == 1.000000000;
```

- The `inhale acc(x.val); assert perm(x.val) > none` statements translate to roughly the following Boogie code:

```
Mask[x, val] := Mask[x, val] + FullPerm;
assert Mask[x, val] > NoPerm;
```

Encoding Quantified Permissions in Boogie I

- What about encoding statement `inhale forall r:Ref :: r in s ==> acc(r.val)`
  - must have effect of updating the `Mask` map for unboundedly many locations
- Idea: recall the "bulk update" operation on maps (ex. sheet 5, Q3)
- We take a *new map variable*, *havoc* it, and *assume* correct properties

```
havoc QPMask;
assume (forall r: Ref ::
  (s[r] ==> QPMask[r, val] == Mask[r, val] + FullPerm)
  && (!s[r] ==> QPMask[r, val] == Mask[r, val])
);
assume (forall <A, B> r: Ref, f: (Field A B) ::
  f != val ==> Mask[r, f] == QPMask[r, f]
);
Mask := QPMask;
```

  - for inhales of *other permission amounts*, we replace `FullPerm` in the above
  - can we subsequently *assert* `s[y] ==> Mask[y, val] > NoPerm`?

Encoding Quantified Permissions in Boogie II

```
havoc QPMask;
assume (forall r: Ref ::
  (s[r] ==> QPMask[r, val] == Mask[r, val] + FullPerm)
  && (!s[r] ==> QPMask[r, val] == Mask[r, val])
);
assume (forall <A, B> r: Ref, f: (Field A B) ::
  f != val ==> Mask[r, f] == QPMask[r, f]
);
Mask := QPMask;
```

- can we subsequently *assert* `s[y] ==> Mask[y, val] > NoPerm`?
  - the answer depends on the *triggers chosen* for these axioms
  - e.g. the trigger `{Mask[r,val]}` on the first axiom will be insufficient
  - the trigger `{QPMask[r,val]}` will let us prove the assertion (why?)
  - by leaving triggers out we leave trigger selection to the underlying tools
  - may be unreliable: *generated axioms should always have triggers*
  - suppose we generate *both triggers* `{Mask[r,val]}` `{QPMask[r,val]}`

Interaction with User Specifications I

- Problems can arise due to *interactions* between tool-generated code and the translation of user-defined specifications
  - e.g. *matching loops* could in theory arise due to a *combination* of *user-defined quantifiers* and *tool-generated quantifiers* (such as those of the last slide)
- A recent (real) example: suppose that we have a Viper function `function f(r:Ref) : Ref` and we want to make a "copy" of the permissions we currently hold to fields `f(r).val` to corresponding permissions to the fields `r.val` (for all `r` in a set `s`)
- We can do this quite easily with a Viper `inhale` statement:

```
inhale forall r:Ref :: r in s ==> acc(r.val, perm(f(r).val))
```

- Unfortunately, this has a bad *potential interaction* with the encoding of inhale statements from the previous slide… *what is it?*

Interaction with User Specifications II

- According to our `inhale forall r:Ref :: r in s ==> acc(r.val, perm(f(r).val))` translation from slide 290, the following axiom will be generated:

```
assume (forall r: Ref :: {Mask[r, val]}{QPMask[r, val]}
  (s[r] ==> QPMask[r, val] == Mask[r, val] + Mask[f(r), val])
  && (!s[r] ==> QPMask[r, val] == Mask[r, val])
);
```

  - note that we've *added the triggers* discussed on slide 291, explicitly
  - the `FullPerm` expression has been replaced with the translation of the Viper expression `perm(f(r).val)` i.e. the Boogie expression `Mask[f(r), val]`
- This axiom has a *simple matching loop*
  - Each instantiation for a particular `r` will yield a new instantiation for `f(r)`
- The cause is the *combination* of our tool-generated axiom (and triggers), and the user-provided expression in the `inhale` statement

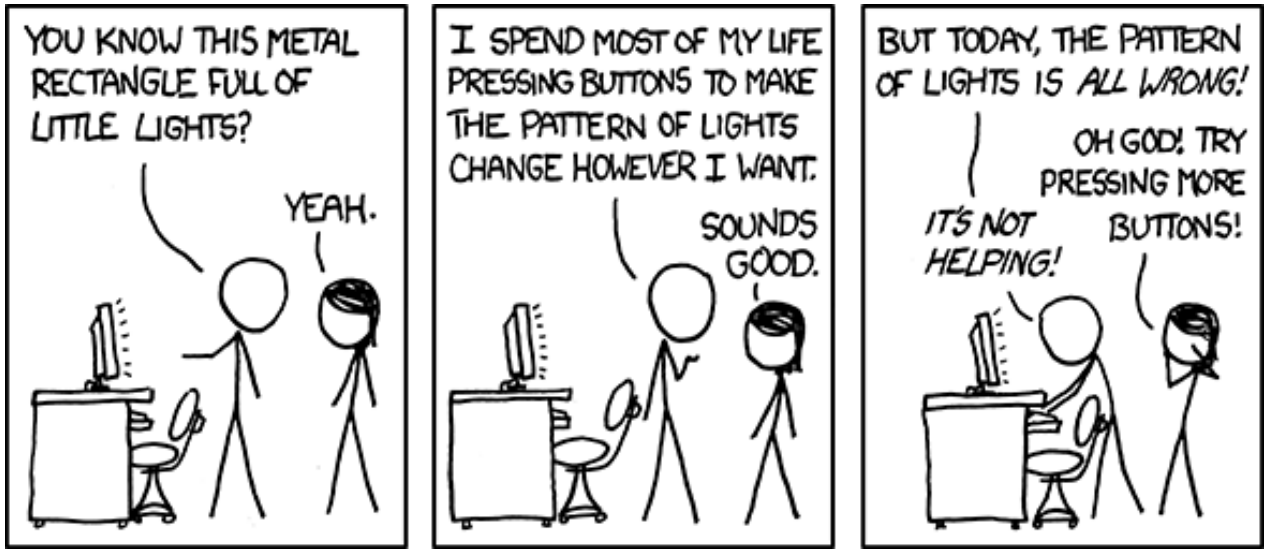
Interaction with User Specifications III

- We can avoid the matching loop by dropping the first trigger:

```
assume (forall r: Ref :: {QPMask[r, val]}
  (s[r] ==> QPMask[r, val] == Mask[r, val] + Mask[f(r), val])
  && (!s[r] ==> QPMask[r, val] == Mask[r, val])
);
```

- This restricts the situations in which the axiom will be triggered
  - only when we make a lookup in the *new* version of the `Mask` variable
- We observe that this is not a problem *given how we use* the `Mask`:
  - We use the `Mask` to check whether we have permissions at certain program points: this will always be determined by the *earlier* updates to the `Mask`
  - triggering the axiom based on lookups in the *new* `Mask` is therefore sufficient
  - in fact, dropping the first trigger will make our encoding *more efficient*
- Same argument doesn't work for general bulk-update axiom on maps
  - we exploit here *knowledge of the role* that `Mask` plays in our encoding

A Further (non-)example of Good Abstractions



(xkcd.com)

- Here's an example from an *old predicates/functions encoding*
- Comes from an early version of the *Chalice* project
  - Concurrency research language using IDF (pre-dates Viper)
  - Also supports predicates and heap-dependent functions, similarly to Viper
  - The latest version of the project *improves on the situation shown here*
  - We use the old encoding to illustrate *potential confusion of leaky abstractions*

Mini example: linked list with sum

- Consider the usual predicate for linked-lists with integer values
  - to make examples shorter, we'll leave out the `fold` and `unfold` statements

```
predicate list(this) {
  … this          list
        next
        val
}
```

- We define a *sum function* over linked-lists

```
function sum(this) : int
  requires list(this)
{
  unfolding list(this) in (this.val +
   (this.next == null ? 0 : sum(this.next)))
}
```

Mini example: adding one

```
method incrementFirst(this)
  requires list(this)
  ensures list(this) && sum(this) == old(sum(this)) + 1
{
  this.val := this.val + 1
}
```

Mini example: swapping two

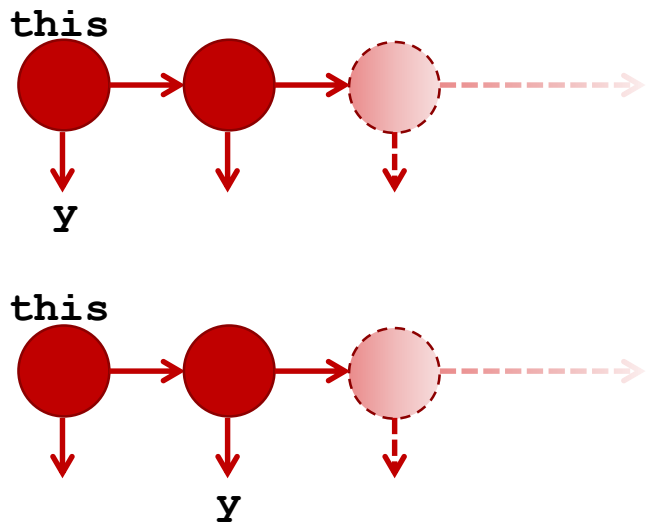
```
method swapFirstTwo(this)
  requires list(this)
  ensures list(this)
{
  if (this.next != null) {

    y := this.val;
    this.val := this.next.val;
    this.next.val := y;

  }
}
```

Mini example: swapping two

```
method swapFirstTwo(this)
  requires list(this)
  ensures list(this) && sum(this) == old(sum(this))
{
  if (this.next != null) {

    y := this.val;
    this.val := this.next.val;
    this.next.val := y;

    assert sum(this) == old(sum(this))
  }
}
```

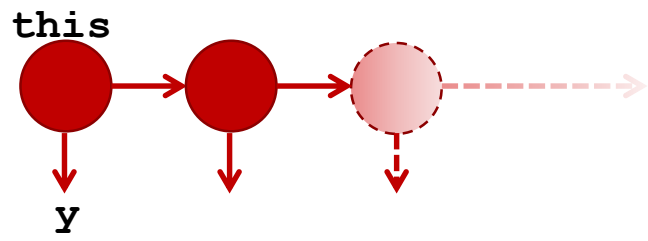
Mini example: swapping two

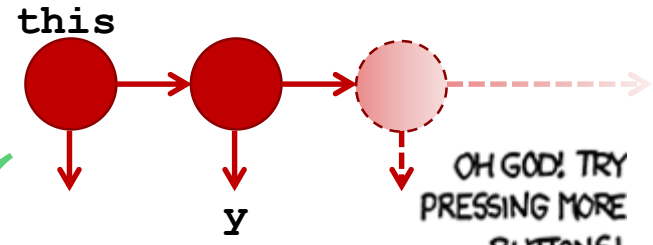
```
method swapFirstTwo(this)
  requires list(this)
  ensures list(this) && sum(this) == old(sum(this))
{
  if (this.next != null) {

    y := this.val;
    this.val := this.next.val;
    this.next.val := y;

    assert sum(this) == old(sum(this))
  }
}
```



Mini example: swapping two

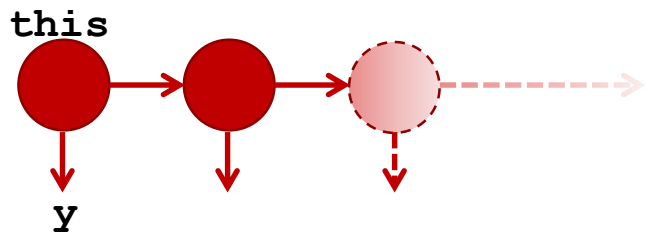
```
method swapFirstTwo(this)
  requires list(this)
  ensures list(this) && sum(this) == old(sum(this))
{
  if (this.next != null) {

    y := this.val;
    this.val := this.next.val;
    this.next.val := y;

    assert  y + old(sum(this.next)) ==
            this.val + sum(this.next)

    assert sum(this) == old(sum(this))
  }
}
```

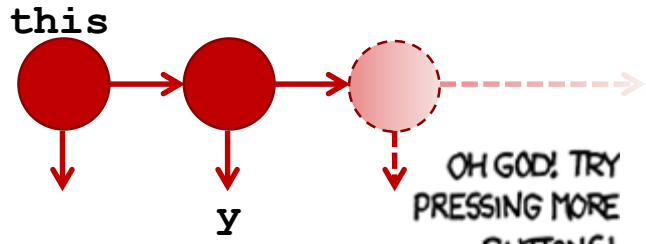




Mini example: swapping two

```
method swapFirstTwo(this)
  requires list(this)
  ensures list(this) && sum(this) == old(sum(this))
{
  if (this.next != null) {

    y := this.val;
    this.val := this.next.val;
    this.next.val := y;

    assert  sum(this.next)
            != old(sum(this.next))

    assert sum(this) == old(sum(this))
  }
}
```





What's Happening Here?

- The issue has to do with when function definitions get unrolled
  - function definitions are *encoded via axioms*, which have *triggers*
  - the only trigger criterion was *occurrences of the function in the program*
  - recursive applications of the function didn't count (avoids matching loops)
  - adding debugging asserts to a program can *trigger additional unrollings*
- The abstraction provided is understandable only if you know the rules
  - If well-documented, the approach is still usable, though not very automatic
- For heap-dependent functions *depending on predicates*, Viper unrolls function definitions based on when the predicates are (un)folded
  - this allows the previous example to work without `assert` statements
- For other function definitions (e.g. those *not using predicates*) the fall-back approach is the one above – *one unrolling per invocation*
  - this will be *improved on in the future* (potential project topic!)

When Good Abstraction is Too Hard

- Choosing to provide a language abstraction brings with it the *responsibility* to guarantee predictable support for that abstraction
- For some verification-related features, this can be *too difficult*
  - e.g. it would be great to abstract away the issues of triggering quantifiers…
  - *without good techniques* for selecting triggers, however this would inevitably sometimes lead to hard-to-debug situations
  - to explain such situations, we'll *need to understand triggering nonetheless*…
- Instead, Boogie and Viper provide *explicit control* over trigger choices
- Debugging tends to become *low-level* for such exposed features
  - e.g. the Axiom Profiler (https://bitbucket.org/viperproject/axiom-profiler)
  - such tools are *very valuable*: hard to find out what the SMT solver is doing
  - better yet would be to map information *back to the high-level* representation
  - lots of research topics here; also *mapping back* / *visualising counter-examples*

Summary: Good Abstractions

- When possible, defining a re-usable abstraction can be very powerful
  - enables users and tools higher up the stack to focus on their own concerns
- Abstractions are only really useful with *predictable tool support*
  - essential that *failing* cases can be debugged at this level of abstraction
  - if this can't be achieved, may be better to expose and give control over details
- Viper provides *good abstractions* for some language features
  - program heap, permission-related constructs, typical function definitions
  - some features need *improved support* (e.g. native sets and sequences)
  - some features (such as *quantifier triggers*) are intentionally exposed
- Helping to *understand failed examples* is an under-explored area
  - low-level tools can help understand lower-level issues (e.g. triggering)
  - mapping information *to higher-levels of abstraction* is an important challenge

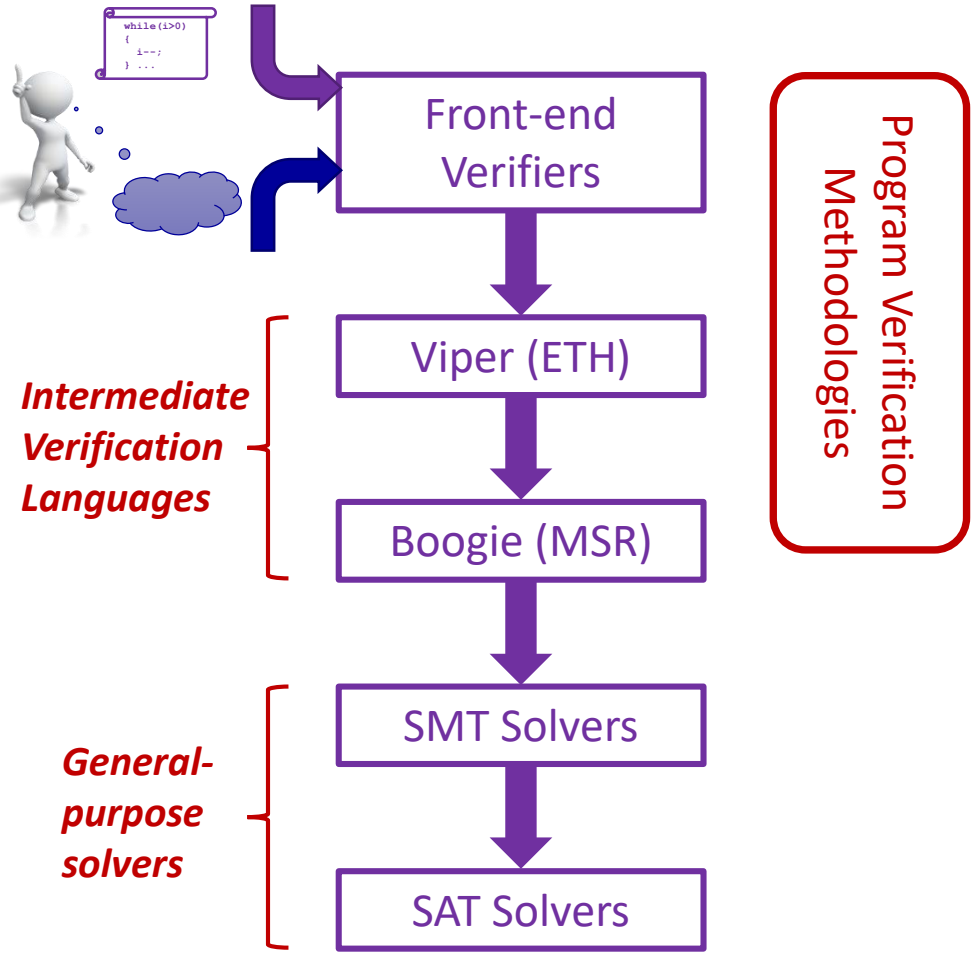
Front-End Specifications

- Verification in tools like Viper requires substantial specifications
  - *permission-related specifications* (the *minimal spec* to make Viper happy)
  - *functional specifications* (what the program actually *does*)
- Front-end tools deal with this need in several different ways
- It could demand rich specifications itself in some annotation language
  - typically: some extension of the *programming language expression syntax*
  - focus on *expert users* (research projects, critical system verification)
  - e.g. *Nagini* (our group) is a verifier for *Python*, with IDF-like specifications
  - e.g. *VerCors* (UTwente) for *Java and OpenCL*, with permission specifications
- Another option is to attempt to *infer* a minimal spec for the program
  - ongoing research on applying *static analysis techniques* to this problem
  - if the permission-related specs can be *generated automatically*, this leaves only the “interesting” (functional) parts of the specification to write

Exploiting Modern Type Systems

- Some modern programming languages (and research papers) propose advanced type systems which prescribe *ownership information*
  - *Rust* is the most popular recent example (sponsored by Mozilla Research)
- Such type systems prescribe *which parts of the heap can be accessed* via *which references* at any given program scopee
  - the compiler uses the information for *memory management*, as well as to guarantee properties such as *race-freedom* for concurrent language features
- Research topic: exploiting this *rich type information for verification*
  - e.g. a type-checked Rust program *implicitly provides* much of the information needed to write *permission-related specifications* for verifiers such as Viper
  - again, if we can *generate* such specifications and, this could enable a lighter-weight program verifier for the Rust language
  - We aim to *hide the permission-related specs entirely* from the programmer

Course Summary - A Good Place to Start From



- We've seen an overview of *techniques and tools for building program verifiers*
  - from *propositional logic* to *program verification*
- A taster: a wide variety of *research topics* connect with this course at all levels, e.g. …
  - *Front-end verifiers* for modern languages
  - *Specification of* challenging language features
  - *Tool automation* for advanced program logics
  - *Intermediate language design* for verification
  - *Static analysis* for specification inference
  - *Advanced debugging tools* to explain failures
  - *Combinations* of alternative program verifiers
- For *potential projects*, just get in touch!

Related Courses

- If you enjoyed this course, in the Autumn Semester you might like…

- *Concepts of Object-Oriented Programming* (Professor Peter Müller)
  - Covers a range of challenging *programming language design* topics, in the broad context of modern object-oriented languages.

- *Program Analysis and Synthesis* (Professor Martin Vechev)
  - On the theory and practice of modern *automated program analysis and synthesis* techniques, applicable to a wide range of programming paradigms.

- *Research Topics in Software Engineering* - Seminar
  - Jointly taught by Professors Peter Müller, Thomas Gross, Markus Püschel, Martin Vechev, on *current research topics* related to all Software groups.

Acknowledgements

- Several people regularly read and commented on these course notes and other material, often at short notice. I very much appreciate all this help and support. Naturally, any remaining errors or moments of confusion are my own work.

- *Thank you very much Lucas Brutschy, Marco Eilers and Malte Schwerhoff*
- *and a special thank you to Uri Juhasz for providing me with extremely generous help before and during the course, despite having left ETH many months ago!*