EtherTrust Developer Guide

EtherTrust is created with HoRSt compiler which allow for creating of analysis tools using high level abstract semantics specifications in form of Horn clauses. These specifications can employ Java functions, i.e., selector functions discussed later that are triggered during the compilation process. It is important to mention that HoRSt not only compiles the semantics specification into working tool, but also performs a number of optimizations on Horn clause improving the analysis results.

HoRSt Glossary

A HoRSt file mainly consists of a collection of named **rules**.

A **rule** has a list of **parameters**, a **selector function** and list of **clauses**.

A **clause** has a list of universally quantified **free variables**, a list of premises consisting of **predicate** applications and Boolean **expressions** and conclusion, which is a single **predicate** application.

A **predicate** has a list of **parameters** and a list of arguments. A predicate application is the name of a predicate followed by list of well-typed **parameters** followed by a list of well-typed arguments. Before being used, a predicate has to be defined with the `pred` keyword.

A **free variable** is a universally quantified variable that is introduced in clause definitions. Free variables are prefixed with `?`.

A **parameter** is a Boolean or integer value, that can be calculated at compile time. Parameters are prefixed with `!`. They are generated by **selector functions**.

A **expression** can either be an integer or Boolean constant, an arithmetic operation on integers, a logic operation on Booleans, a select or store operation on arrays, the construction of a **custom type**, a **custom operation**, the construction of an array from a constant expression, a reference to a variable, free variable or parameter, a conditional expression, a **match expression** or a **sum expression**.

A **selector function** is a Java-defined generator of tuples of integers and Booleans. It has to be declared with the `sel` keyword and provided as Java files to the HoRStCompiler with the `-f` flag. The generated values bind to **parameters**.

A **match expression** consists of a tuple of **expressions** that are compared to a list of tuples of **patterns** each of which is followed by an **expression** called branch. If a pattern matches the expression, the match expression has the value of the matching branch.

A **pattern** looks exactly as a tuple of constructions of values with the additional possibility of using variables that get bound to the matched values and may be used in the corresponding branch. `_` can be used to ignore matched values. As of now, each **match expression** has to end with a `_`-branch.

A **sum expression** is a way to generated iterated sums, products, conjunctions and disjunctions. The collections which are "summed" are generated by **selector functions**.

A **custom type** can be defined by the `datatype` and `eqtype` keywords followed by a name and a list of **constructors**.

Equality types (defined by `eqtype`) may only contain other equality types in their constructors. Integers and Booleans are equality types, arrays are not.

A **constructor** is a name (prefixed by `@`) and a (possibly empty) list of already defined types.

A **custom operation** has a name, a list of **parameters**, a list of arguments, a return type and an *expression* that forms the body of an operation. They have to be declared with the `op` keyword. An application of an operation has the value of the body where all parameters and arguments are substituted by the values given on the call site.

Build

Before building EtherTrust, build Z3.

Get it from https://github.com/Z3Prover/z3. Build with

```
python script/mk_make.py --java
cd build; make
```

Place com.microsoft.z3.jar to %project%/lib.
Then build EtherTrust

```
mvn -Dmaven.test.skip=true package
```

Run

Provide paths to Z3 binary bindings in LD_LIBRARY_PATH (Linux) or DYLD_LIBRARY_PATH (MacOS).

```
LD_LIBRARY_PATH=. java -cp
com.microsoft.z3.jar:EtherTrust-1.0-SNAPSHOT.jar
secpriv.horst.evm.EvmHorstCompiler $file --json-out-dir
$path_to_results -p -b -s $path_to_grammar/evm-abstract-
semantics-partial-standard-calls.txt $path_to_grammar/
queries-reentrancy.txt
```

Selector Functions

What is a selector function

In our specification language HoRSt there are two places where we want to talk about sets (not necessarily sets in the mathematical sense) of values which may depend on the analyzed code:

- in SumExpressions, where we can generate sums/products of sets of integers and conjuctions/disjunctions of sets of booleans
- when generating parameterized rules

These sets are generated by so-called selector functions.

Selector Functions in Horst

Since HoRSt is no general purpose programming language, we can't specify these sets within HoRSt -- we can just declare a signature and then implement the code in Java.
Selector functions in HoRSt are declared like this:

```
sel idsAndPcsForOpcode: int -> [int*int];
//take an integer and generate set of integer-integer-Tuples
```

Selector Functions in Java

The general way of providing an implementation for a selector function to HorstCompiler is proving a Java source code file with the `-f/ --`

`selector-function-provider` command line option. The given file will be loaded and every method which fulfills all of the following requirements will be registered as a selector function:

- the method is public
- the method has a name different from all methods of `java.lang.Object`
- the method's name is different from `"unit"`
- the method's name is different from all previously defined selector functions names
- all of the method's arguments are either of type `BigInteger` or `Boolean`
- the method's return type is either `Iterable<BigInteger>`, `Iterable<Boolean>` or `Iterable<TupleX<T1,T2...TN>>` where `TupleX` is one of the tuple types defined in `secpriv.horst.data.tuples` and `T1...TN` are either `BigInteger` or `Boolean`

Selector functions in Java are implemented like this:

```
  public Iterable<Tuple2<BigInteger, BigInteger>>
idsAndPcsForOpcode(BigInteger opcode) {
              List<Tuple2<BigInteger, BigInteger>> ret = new
ArrayList<>();
        for(Map.Entry<BigInteger, ContractLexer.ContractInfo>
entry : contractInfos.entrySet()) {

CartesianHelper.product(Collections.singletonList(entry.getKe
y()),

entry.getValue().getProgramCountersForOpcode(opcode)).forEach
(ret::add);
        }
        return ret;
    }
```

If a Java file is provided in this way, the class defined in it may not be included in any package (it has to be included in the top level package).

Since the selector function provider has to interact with the analyzed byte code, it has to be possible to provide command line arguments to the selector function provider. Any command line argument which is given immediately after `-f <SelectorFunctionProvider.java>` and does not start with `"-"` or end with `".java"` is forwarded to the selector function provider. To make use of these arguments, the selector function provider may implement a constructor taking a `List<String>` as argument.

Why is there an --evm option now?

We can't put selector function providers in our source tree without "polluting" our project with top level classes (because right now, we can't put the selector function providers that are dynamically compiled in packages).

But if our providers are not in part of our source tree, we do not get nice code completion when working on them (bad) and we can't unit test them easily (worse).

Therefore we moved the `EvmSelectorFunctionProvider` into the source tree, in a special package and provided a command line option to load it from there.

Coding Guide Lines

- the existing code is based on immutable data structures that avoid null. Look at `secpriv.horst.data.Clause` to see how we can implement immutable structures
- these classes do not need getters (and cannot have setters)
- implement complex calculations on these immutable values as Visitors
- strive to keep the nesting level within functions low by return early on errors

```
 if (x == 1) {
        return Optional.empty();
  }

  if (y == 3) {
        return Optional.empty();
  }

  return Optional.of(x+y);

```

is better than

```
if (x !== 1) {
      if (y !== 3) {
           return Optional.of(x+y);
      }
  }
  return Optional.empty();

```

- usually our class hierarchies are implemented in one file (see Expression or Proposition for an example).

Getters are totally fine if

- the value is computed (like `getType()` in `Expression`)
- the value has to be available in all classes implementing an interface or abstract class(also like `getType()` in `Expression`)
- the value is based on a value that may be null (null-lessness beats getter-lessness) or
- is based on a mutable value, that is not arbitrary modifiable (like `definePredicate`/`isPredcateDefined`/`getPredicate` in `VisitorState`)
- ...maybe some other case I forgot

But since most of our simple data structures (e.g. all objects `secpriv.horst.data`) are all immutable and have no behavior except of holding data (and making sure, that they are constructed in a somewhat sane fashion), we don't need getters and setters as they tend to make the code less readable.

How to write selector function providers?

Since we wrote our first selector function providers in the versions of HoRSt we got some new capabilities which may motivate changes in the way we are doing things right now. This is a set of implementation patterns that may help to choose the right approach or may inspire some refactorings.

If selector functions have different data sources, put them in independent selector function providers

`SelectorFunctionHelper::registerProvider` can be called on multiple objects, registering all of their suitable functions in the SelectorFunctionHelper object.

The way IntervalProvider is used now is actually an example for how it should not be done. Instead of forwarding the interval-method to an instance of IntervalProvider, we should just call

```
selectorFunctionHelper.registerProvider(new IntervalProvider());
```

before registering other providers, whenever the the `--evm` command line option is given.

If selector functions have the same data source, but a different purpose, use composition and forwarding

Say you have a data source that provides with a byte code and some metadata. Assume also that there is already a selector function provider `A` that provides the methods `p`, `q` and `r` by using the byte code.

Now you want to make a new selector function provider `B` that uses the data source to provide new methods `x`, `y` and `z` in addition.

The recommended way to achieve this is having `B` hold a instance of `A` and forward the calls to `p`, `q` and `r` to this instance. (In short it makes the code more modular, more arguments in *Effective Java* item 18). Optionally, for modularity, `B` could also hold a instance of a interface implemented by `A`. In future, these interfaces could be generated from HoRSt files.

Alternatively, if the and the data source and selector function provider in instantiated in Java, you can also use the pattern from above like this.

```
DataSource d = new DataSource();
selectorFunctionHelper.registerProvider(new A(d));
selectorFunctionHelper.registerProvider(new B(d)); // B here only has to implement x, y and z
```

The relation of `EvmSelectorFunctionProvider` and `ConstantAnalysis` is actually an example for how it should not be done.

Use small selector functions

We can now use compound invocations of selector functions, which should reduce the number of selector functions we have to define

```
sel idsAndPcsForOpcode: int -> [int*int]; // the set of (id, pc) combinations for a given opcode
sel idsAndPcsAndOffsetsForDup: unit -> [int*int*int]; // the set of (id, pc, os) combinations for DUP opcodes, e.g. contains (1, 4, 3) if at position 4 in contract 1 there is the opcode DUP3
sel idsAndPcsAndOffsetsForSwap: unit -> [int*int*int]; // the set of (id, pc, os) combinations for SWAP opcodes, e.g.
```

```
contains (1, 4, 3) if at position 4 in contract 1 there is
the opcode SWAP3
```

should become

```
sel ids : unit -> [int];
sel pcsForOpcode : int*int -> [int];
sel offsets : int*int -> [int];
sel dups : unit -> [int];
sel swaps : unit -> [int];
```

and invocations like

```
for (!id:int, !pc:int) in idsAndPcsForOpcode(MLOAD)
for (!id: int, !pc: int, !os:int) in
idsAndPcsAndOffsetsForDup()
for (!id: int, !pc: int, !os:int) in
idsAndPcsAndOffsetsForSwap()
```

may become

```
for (!id:int) in ids(), (!pc:int) in pcsForOpcode(!id, MLOAD)
for (!id:int) in ids(), (!op:int) in dups(), (!pc:int) in
pcsForOpcode(!id,!op), (!os:int) in offsets(!id, !pc)
for (!id:int) in ids(), (!op:int) in swaps(), (!pc:int) in
pcsForOpcode(!id,!op), (!os:int) in offsets(!id, !pc)
```

A possible future extension for this may be array literals, such that invocations like this are possible:

```
for (!id:int) in ids(), (!pc:int) in pcsForOpcode(!id, MLOAD)
for (!id:int) in ids(), (!op:int) in [DUP1, DUP2, DUP3, ...],
(!pc:int) in pcsForOpcode(!id,!op), (!os:int) in offsets(!id,
!pc)
for (!id:int) in ids(), (!op:int) in [SWAP1, SWAP2,
SWAP3, ...], (!pc:int) in pcsForOpcode(!id,!op), (!os:int) in
offsets(!id, !pc)
```

Web Interface

EtherTrust is available via an online interface.

Framework

Currently, the web interface accepts contracts with size <=100kb. The web framework works as follows: backend spawns an analysis and updates the results asynchronously. User is free to comeback and check the analysis results anytime using the hash of a task identifier displayed the result is started being processed.

Here is the order of components evoked per each task:

- static web page (backend jar)
- file uploader (backend jar)
- analysis (EtherTrust jar)
- asynchronous results fetcher (backend jar)

Analysis results

Currently analysis results are displayed as a rendered json on a webpage. Please refer to User Guide for Results interpretation discussion.