Singularity Design Note

25

Bytecode Verifier

Extensible Static Analysis of Assemblies

Singularity depends on strong inter-process isolation properties. The Singularity bytecode verifier uses light-weight static analysis to verify these properties for every application installed onto a system. The verifier is designed in a modular way, so that it is easy to extend with new properties to check.

1. Using the Verifier

The current version of the verifier is checked in as `main\base\build\singver.exe` to the Singularity Source Depot repository. The `setenv` script should add this directory to your path when run.

1.1. Paths

The verifier uses the `SINGVER_PATH` and `SINGVER_CORLIB` environment variables to resolve references to DLLs and EXEs.

`SINGVER_PATH` should contain a semicolon-separated list of filesystem paths that should be searched for required assemblies. If this variable is not set, then the verifier constructs a best-effort guess of which paths will be required based on the `SINGULARITY_ROOT` environment variable set by `setenv`, including the current directory "`.`".

`SINGVER_CORLIB` should contain the name of the assembly to be used as a replacement for `mscorlib`, the core standard library. If it is not set, the verifier guesses that it should be `corlib`, which is the correct choice for Singularity applications, but not for the kernel. `sgc` output contains references to bogus assemblies that should all be treated as aliases for this core library.

1.2. Running the Verifier

If you have already run `setenv`, and you have an `sgc`-produced assembly `myapp.exe` in the current directory, then you can verify it by running:

```
singver myapp
```


`singver` checks the `SINGVER_PATH` directories for some EXE or DLL version of the assembly you ask it to verify. If it finds such a version, and if it can verify that that version satisfies the properties it is configured to check, you'll see this happy result:

```
Verification succeeded.
```

If your code has a bug, or if the verifier just isn't smart enough to tell that your code is correct, you'll see some error message instead. To understand what these mean, you'll probably want to read the next section of this SDN. *(Note: In the ideal situation, developers won't have any need to understand error output from the verifier. This might be important for the present because of bugs in the verifier, and because* `sgc` *doesn't yet implement all of the checks that the verifier implements. Eventually* `sgc` *should be responsible for providing developers with readable error messages.)*

2. Algorithmic Overview

The bytecode verifier checks assemblies by performing abstract interpretation on them. The abstract domain consists of **state graphs** that summarize many possible memory configurations. To determine the abstract effects of instructions, the core part of the verifier queries **handlers** registered by **plug-ins** that are concerned with different properties that the verifier should check. Handlers will generally have available to them all of the abstract interpretation state relevant to the entities they deal with, but they can only indirectly modify that state by generating declarative **transformations**.

By running the verifier like this:

```
singver 1 myapp
```

you set the debug output level to 1 (instead of the default of 0). The output will trace every step of finding a fixed point, printing at each stage the state graph, the transformation to be applied, and other information. Reading this output for progressively more complicated inputs can be helpful for understanding how the verifier works.

2.1. State Graphs by Example

State graphs are output in a human-readable ASCII format. In this section, we present some examples of graphs written in this way and their logical meanings.

Here's the simplest example:

```
(local0! => contains{42})
```

This single-node graph corresponds to a state within a method with no parameters and one local variable. That local variable corresponds to the node `local0`, which has the value 42.

The exclamation mark indicates that this node isn't null. Without the exclamation mark, this graph could also describe an empty memory state. Non-null support of this kind is built into the verifier because it affects nearly every kind of specification that we want to generate.

The content of the variable is given in the form of a directed **edge** from `local0` to `42`. Every edge is labeled with a name. In this case, the label is `contains`, indicating that the source of the edge (`local0`) is an object that contains the edge target (`42`). All edges are **injective**, which means that a node can have at most one outgoing edge per label.


```
(local0! => contains{42}) X (arg0! => contains{null})
```

In this graph, we've moved on to multiple local variables. We have one normal local variable and one method argument (`arg0`). The current value of the argument is the null pointer. Notice that the `!` for `arg0` refers to the node itself, not its contents. In other words, a non-null annotation on the node for a variable of pointer type is telling us that this *variable* always exists, not that it does not contain a null pointer.

The two graph nodes are combined with the **tensor** operator `X`, which is lifted from linear logic. It is a generic binary operator over state graphs. The tensor of graphs G_1 and G_2 is a graph describing any state that can be *divided into two disjoint pieces*, one of which is described by G_1 and the other by G_2. The tensor operator is the key to efficient, local reasoning about mutable state, because it allows us to change one part of a state graph without worry that we've "broken" any other part of it, as long as we know that we've only changed objects found in the subgraph that we focus on.

```
(arg0! => contains{init0}) X (init0! => contains{$IN.0})
```

Here we see a graph for the entry of a method with a single, by-reference argument. The special value `init0` denotes the value of the first argument on entry to the method. `$IN.0` is a variable. Graphs with variables can be thought of as standing for any other graphs with specific values substituted consistently for the variables.

```
(arg0! => contains{init0}) X (init0! => own())
```

This is the entry graph for a method with a non-null argument of a tracked type, such as a pointer to a rep struct in the ExHeap. Here we see our first **data fact** in the form of the `own()` annotation for `init0`. Like edges, data facts are associated with graph nodes and have labels. Data facts are more general; they needn't be injective, and they may take any number of arguments, including zero, as in this example. You can distinguish an edge from a unary data fact in debug output, as edges are written with curly braces around their arguments, while data facts use parentheses around theirs.

```
(arg0! => contains{init0}) X (init0 => own())
```

This graph is almost the same as the last one. The difference is that we removed the `!` from `init0`, which indicates that that node may be null. A graph with nullable nodes like this is equivalent logically to the disjunction of all graphs that express the possibilities of making these nodes non-null or removing them from the graph. To use such an object soundly, a program will need to do some runtime non-null testing, which can resolve nullable nodes to one of the mentioned alternatives along some execution paths.


```
(arg0! => contains{init0}) X (init0! => own(); state([Ready]))
```

This could be the entry graph for a method taking a non-null endpoint parameter. The endpoint would belong to some contract `C` containing a `Ready` state, and the argument type would be annotated like `C.Imp:Ready!`. The notation `[Ready]` stands for a symbol. The example shows that nodes can contain multiple data facts and edges, and that they are separated by semicolons in debug output.

```
(arg0! => contains{init0}) X ((init0! => own(); state([Ready])) +
(init0! => own(); state([Waiting])))
```

This state graph could come up later in the execution of a method that started out as in the previous example. Somehow we have reached a point where the state of the endpoint argument isn't uniquely known: it could be `Ready` or `Waiting`. We introduce the **disjunction** operator + to express this. This is the linear disjunction of linear logic, where a disjunctive graph stands for states that satisfy one or the other of the disjuncts.

There a few important things to note about disjunctions.

First, we allow disjunctions at arbitrary levels of nesting in state graphs. This example displays a benefit that this brings: we are able to share the `arg0` node between the two cases.

Second, as in linear logic in general, every resource must be fully accounted for. Adding extra objects to a state that satisfied a disjunction-free state graph *always* breaks that satisfaction relationship, barring special cases like nullable nodes. This means that the nodes mentioned in *either* branch of a disjunctive subgraph of a graph may not be present anywhere else in that graph, though there may be edges to them elsewhere.

2.2. Transformations by Example

Debug output also contains ASCII representations of the transformations applied at each instruction step to model the instruction's symbolic effect. Transformations look very similar to state graphs; they are also graphs built from tensors and disjunctions, but at their nodes they specify changes to edges and data facts instead of whether or not they are present.

```
(stack0!: contains:_->7)
```

This transformation would be applied for a `ldc 7` instruction executed while the stack is empty. It describes an effect on a node `stack0`, the first stack slot. The transformation node given here says that we should add a `contains` edge from `stack0` to `7`. The `_` to the left of the arrow indicates that we should add a new edge here. If an edge with this label is already present in the graph, the transformation will fail. This behavior makes sense because we define edges to be injective.

In general, the verifier models the stack, local variables, and heap in a uniform way, so that effects on all of these can be modeled with transformations. The exception is that it separately tracks the stack length and a single type for each active stack slot.


```
(stack0!: contains:%OLD.10->_)
```

This similar transformation would be generated for a `pop` instruction directly after the `ldc` from the previous example. Here, we want to remove a `contains` edge, no matter what it points to. We use a **unification variable** `%OLD.10` to express this. Unlike the variables we introduced before, unification variables are only used for pattern matching, and they are printed with initial `%`s. Variables printed with initial `$`s stand for persistent objects, and they are the only ones allowed in state graphs.

```
(stack0!: contains:%PTR->_) X (%PTR: own:()->())
```

This transformation gives the effect of a method `void borrow(int* in ExHeap ptr)` when the stack was empty before pushing the actual parameter. It removes the `contains` edge from the active stack slot and requires that the old contents of that slot be some pointer that is owned both before and after the call.

```
(stack0!: contains:%IN->$OUT) X (%IN: own:()->_) X ($OUT: own:_->())
```

Here's a transformation for a similar call to a method `int* in ExHeap identity([Claims] int* in ExHeap ptr)`. The semantics of transformer application are such that it is fine for the unification variable `%IN` to turn out to be the same as `$OUT`, though this isn't required. If this is the case, the `own` capability will be consumed and then recreated harmlessly for a new object. Notice that we use a unification variable for the input object, which should match against any actual value; and a regular variable for the output object, which we are treating as a new symbolic object.

```
(stack0!: contains:%EP->_) X ((%EP: own:()->(); state:([Ready]) ->
([Waiting])) + (%EP: own:()->(); state:([Other1]) -> ([Other2])))
```

This transformation can model the effect of a method `void SendM()` of an endpoint whose contract allows the message `M` to transition from state `Ready` to `Waiting` or from `Other1` to `Other2`. It uses a disjunctive sub-transformation, which is analogous to a disjunctive subgraph of a state graph.

```
(stack0!: contains:%PTR->_) X (%PTR!: own:()->_; exposed:_->$FIELDS) X
($FIELDS!: contains:_->$value) X (($FIELDS + 4)!: contains_->$next) X
($next: own:_->())
```

This transformation expresses the requirements and results of exposing a pointer to a `rep struct S { int value; S* in ExHeap next; }`. Exposing is performed by calling the `Expose` method of the object to be exposed, so we would use this transformation for such a call. It consumes the pointer from the top stack slot and requires that this point be owned. If so, it can consume that ownership and add an `exposed` edge to some new symbolic field nodes. Each of the new fields

contains some initial value, and there is also a node for the fresh value of the `next` field, which has an ownership invariant associated with it. The order of the fields in memory is based on the order in which they are declared in the corresponding struct definition. The verifier must already deal with this order for instructions like those to load the fields of a struct onto the stack, so this `Expose` transformation enables uniform handling of both cases.

2.3. Abstract Interpretation Operations

The interesting abstract interpretation operations that the verifier uses include:

- **Application**. Modifying a state graph to reflect the effect of a transformation. This operation can also fail if the state graph doesn't match up properly with the transformation. The verifier uses this application operation at every instruction, applying a transformation constructed based on queries to the plug-ins.

- **Matching**. Checking two state graphs for compatibility by finding an isomorphism between their nodes that respects the edges and data facts that are present. This is used to check postconditions at method return points. The main operational difference from application is that the two graphs must match up exactly, with no fields left over, while application will leave unmatched state graph nodes unchanged, thanks to linearity. We don't want to allow such behavior when checking postconditions, because this could mask memory leaks and similar errors. We allow the postcondition, but not the return point state graph, to contain unification variables.

- **Subsumption**. This is the abstract interpretation lattice less-than-or-equal operator, where we want to check that every concrete state described by one state graph is also described by another graph. Subsumption checking is used when a program location is visited multiple times and the verifier needs to determine whether it must revisit the following instructions. The abstract interpretation can terminate when every transition from a reachable abstract state leads to a program point whose saved state graph subsumes the graph active coming out of the source state. Subsumption is a thin wrapper around matching, where we modify the possibly-subsuming graph by replacing all of its normal variables with unification variables.

- **Join**. This is a standard abstract interpretation join operation which tries to construct the most specific state graph that subsumes two argument graphs. Repeated use of joining is used to find a fixed point for a mapping from basic blocks to precondition graphs. Often it's not clear how to perform join operations that lead to a terminating verifier run while still preserving enough information to prove safety. Because of this, plug-ins can register **widening hints** that give advice on how to handle the combination of different versions of a node. Each hint can suggest merging the nodes into a fresh variable that we know nothing about, introducing a disjunction to preserve the distinction between the two versions, or give no advice, passing the decision on to the next hint. The verifier uses the first strategy by default, biasing itself towards termination instead of completeness.

- **Merging transformations**. When multiple plug-ins generate transformations that apply to a node, the verifier needs to merge these transformations soundly. For this, it uses the same basic strategy of finding matching nodes as for the join operation, but for

transformations instead of state graphs. Each node in the resulting transformation has all of the edges and data facts of its counterparts in the two input transformations. Since edges are injective, this operation won't succeed if the two versions of a node specify conflicting edges with the same label, though "isomorphic" edges that just differ in variable names are supported.

2.3.1. Matching

Found at the heart of all of these operations is a procedure for matching the nodes of two graphs. There are slight differences in how matching works for different operations, but we'll present a generic matching procedure that should give a good intuition for understanding the operations. It's a very simple procedure based on following chains of injective edges from known equal nodes.

We'll use the algorithm for the actual matching operation to illustrate. For clarity, we omit some details like handling of non-null status. We'll omit handling of data facts, and we'll deal first with cases with disjunction-free actual graphs (but possibly disjunctive postconditions).

The algorithm uses a standard kind of expression **unification** which builds a map σ from unification variables in a postcondition to expressions in the actual graph at a return point:

unify σ `%VAR` *e* = σ[`VAR` → *e*] (if σ doesn't already have a different mapping for `VAR`)

unify σ (p_1 • p_2)(e_1 • e_2) = **unify** (**unify** σ p_1 e_1) p_2 e_2 (where • is some binary expression operator)

The **unify** function can fail if a unification variable would need to be given multiple incompatible instantiations, or if the postcondition and actual expressions differ structurally.

Next, we need a function **match_one** to perform matching for single nodes. It takes as input a graph *G* and a node *n*. If an appropriate counterpart for *n* is found in *G*, then **match_one** returns *G* with the matched part removed, along with an updated map σ containing any new information learned about unification variables. It's important to return the updated graph, because we need to be sure at the end of matching that no obligations have gone unmatched. Otherwise, we could be allowing a memory leak.

match_one σ `(n => edges) (n' => edges')` =

If σ(`n`) ≠ `n'`, then return `None`.

If *Dom*(`edges`) ≠ *Dom*(`edges'`), then return `None`.

For each *`label{target}`* ε `edges`:

Consider the corresponding *`label{target'}`* ε `edges'`.

Set σ to **unify** σ `target target'`, or return `None` if unification fails.

Return `Some (emp,` σ`)`, where `emp` is the empty graph.

match_one σ (*G_1* `X` *G_2*) *n* =

If **match_one** σ *G_1* *n* = `Some (`*G'*`,` σ'`)`, return `Some (`*G'* `X` *G_2*`,` σ'`)`.

Otherwise, if **match_one** σ *G_2* *n* = `Some (`*G'*`,` σ'`)`, return `Some (`*G_1* `X` *G'*`,` σ'`)`.

Otherwise, return `None`.

Now we can use **match_one** to implement our main algorithm **match**:

match σ *G* `(n => edges)` = **match_one** σ *G* `(n => edges)`

match σ *G* `(`G_1 `X` G_2`)` =

If **match** σ *G* G_1 ≠ `Some (`σ', *G*')` for some σ' and *G*', then fail.

Otherwise, return **match** σ' *G*' G_2.

match σ *G* `(`G_1 `+` G_2`)` =

If **match** σ *G* G_1 = `Some (`σ', *G*')`, then return `Some (`σ', *G*')`.

Otherwise, return **match** σ *G* G_2.

To check if graph *G* matches postcondition *P*, we call **match** {} *G P* and verify that the result is `Some (`σ`, emp)` for some σ. This means that the matching succeeded and left no parts of the postcondition unmatched. We consider graphs equivalent based on obvious simplification rules, including the fact that `emp` is a unit value for tensor.

This version of the algorithm is actually simplified significantly. It relies critically on the order in which nodes are represented in the graphs. This is because nodes will only be matched when the unification map generated in previous steps guarantees that they stand for the same object. If the steps execute in the wrong order, some nodes may go unmatched. The implementation handles this problem by executing **match** iteratively until either all obligations are discharged or no new parts of the postcondition are matched.

To handle disjunctions in the actual return point graph, we currently use an inefficient but simple procedure: we expand the actual graph into a set of disjunction-free graphs that cover all of the possible cases, and then we apply **match** for each one.

3. Implementation

3.1. Location

The implementation of the verifier is rooted at `main\base\Windows\Verifier` in the Singularity Source Depot repository.

3.2. Dependencies

The verifier is written with F#, an MSR-developed Caml implementation for .NET, with a home page at:

http://research.microsoft.com/projects/ilx/fsharp.aspx

It also uses the Abstract IL library for parsing .NET bytecode assemblies:

http://research.microsoft.com/projects/ilx/absil.aspx

Both of these are checked into the repository, the compiler executable in `main\base\build\fsc.exe` and library DLLs in `main\base\build\fsc\`. You might

want to download the complete packages from the URLs above if you want to modify the verifier in Visual Studio or do anything else much beyond using it as a client.

3.3. Organization

The verifier source is divided up into modules in the usual ML way. Most modules have both interface and implementation files. For instance, the `Verify` module has an interface in `verify.fsi` and an implementation in `verify.fs`. Filename pairings with `fsi`/`fs` or `mli`/`ml` extensions indicate this interface/implementation relationship.

The code is written to be mostly self-documenting, especially in the case of the interfaces. The types, values, and comments found in those files are the best form of "architecture description."

Here we list the modules of the verifier in dependency order. It should be possible to acquaint oneself with the architecture by reading the corresponding interface files in this order.

Module	Description
`Util`	Utility functions on lists, maps, sets, and strings.
`Debug`	Flags to control debug output.
`Ilbind2`	Modified version of the Abstract IL `Ilbind` module, patched to handle idiosyncrasies of sgc/Bartok assembly references that don't reference real assemblies.
`Ilin2`	Version of Abstract IL `Ilin` that uses `Ilbind2`.
`Loader`	Support for finding assemblies in the filesystem.
`Verify_types`	Definitions of types used throughout the verifier.
`Types`	Operations on the types from `Verify_types`.
`Verify_print`	Pretty-printing.
`Liveness`	Simple liveness analysis on MSIL method bodies. The results are used to forget information about dead variables during the analysis, speeding convergence to a fixed point.
`Abs`	Abstract interpretation operations, including widening hint hooks, transformation application, abstract state join and subsumption tests, and constructors for transformations.
`Register`	Hooks to register handlers for classes, instructions, formal parameters, return values, and method calls.
`Combinators`	Combinators useful for defining handlers of those types, including a general notion of rules and filters, and functions for building handlers from rules.
`Cil`	Plug-in for the standard semantics of MSIL, essentially implementing a standard bytecode verifier.
`Corlib`	Plug-in for additional standard handlers, including support for runtime type tags,

	normal `out` and `ref` parameters, and tracking constant Boolean values.
`System`	Plug-in for `System.Compiler` libraries, currently only including non-null-checking support.
`Singularity`	Plug-in for the basic Singularity environment, including a check to see if the verifier is running on a Windows assembly or a Singularity assembly.
`Ownership`	Plug-in for ownership tracking.
`Contract`	Plug-in for channel operations.
`Rep`	Plug-in for rep structs.
`Config`	Initializes all of the plug-ins that we'd like to use.
`Verify`	Main abstract interpretation engine.
`Test`	Command-line driver for `Verify`.

