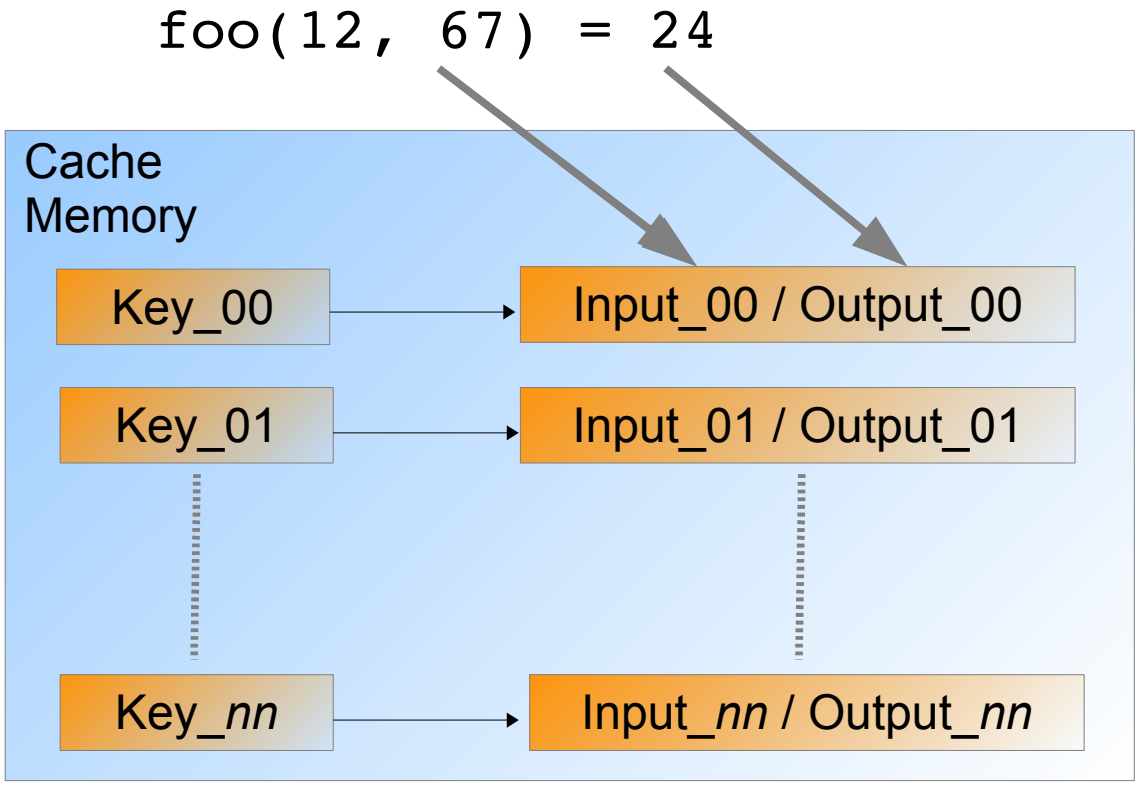
Using the Extension Capability and the Reflection Interface of Specman/*e* for Automatic Memoization

Thorsten Dworzak,
Verilab GmbH

Memoization

Optimization technique to speed up run-time critical methods using caching.



Memoization

Input and output values are saved in a lookup-table (hash table).

- Upon each method call, input values are compared with the cached ones.
- Match: return the cached result instead of recalculating it.
- Else: call original method and save result in cache.

Memoization

Goal: create *e* macro to memoize a method at the place of its definition.

```
MEMOIZE fibonacci(n: uint): uint is {
    result = 1;
    if n >= 2 {
        result = fibonacci(n-2) + fibonacci(n-1);
    }
};
```

- Present some enabling techniques
- Not possible in e.g. SystemVerilog

Memoization

Caveats: can only be applied to *pure* methods.

- Stateless; the result varies only with regard to the inputs.
- No side-effects, like screen output, file update.
- Parameter passed by reference must not be modified.

Three kinds of *e* Macros

- *define* – simple text replacement.

```
define WD 16;
```

```
d_p: simple_port of uint(bits: WD)
is instance;
```

- *define-as* – parameterized macros, syntactically checked match-expression.

```
define <swap_scalar'action> "swap <a'name>
<b'name>" as {
    var x := <a'exp>;
    <a'name> = <b'name>;
    <b'name> = x;
};
```

```
{
    var this: int = 4;
    var that: int = 5;
    swap this that;
};
```

e Macros

- *define-as-computed* – allows the use of *e*-code to parse the macro parameters and generate the macro expansion string.
  - Gives the highest degree of flexibility.
  - Allows to overrule pre-defined syntax or add new constructs.

This will be used for the memoization macro.

The Reflection API

Type introspection

- Allows the type and object properties to be queried at runtime.
- C language – *sizeof(<type>)* operator returns the data size of a given type in bytes.
- Ruby – the *instance_of* and *kind_of* member methods of Ruby classes.

The Reflection API

Reflection – possible to also manipulate objects

- Provides read and write access to the meta-data of programs.
- In *e* this can be done by means of the global *rf_manager* object.

```
type col_t: [RED, GREEN, BLUE];

var t: rf_type =
rf_manager.get_type_by_name("col_t");
if t is a rf_enum (te) {
   outf("n_items=%d", te.get_items.size());
};
```

Key Generation

Cached result must be uniquely identifiable by its input values.

- Each hash table entry is assigned a key.
- In *e*, keyed-list implements hash table.
- Known hashing functions create a 32-bit key from a list of bytes.

Key Generation

Key generation from input values can easily be done in *e*.

- Chose *DJBHash()* method.
- *packing()* method provides list of bytes.
- Example: two *uint* input parameters *a* and *b*:

```
var key: uint = DJBHash(
   pack(a,b).as_a(list of byte)
);
```

- *Note*: key not unique for every set of values.

Key Generation

For compound parameters, the packing method only works on *physical* fields.

- Check this requirement in macro code.
- Implemented general method:

```
deep_is_physical(name: string): bool;
```

- Recursively descend into object tree of struct using the reflection API.

Memoization Macro

Provide some infrastructure via global struct:

```
struct vlab_memoize_manager_s {
   final DJBHash(str: list of byte): uint is ...

   deep_is_physical(name: string): bool is ...
};

extend sn_util {
   !mz_manager: vlab_memoize_manager_s;
   init() is also { mz_manager = new };
};
```

Memoization Macro

Define the cache-entry data struct:

```
struct vlab_memoize_cache_entry_s {
    !hash   : uint;
    !input  : list of byte;
    !output : list of byte;

    set_hash(val: uint) is { hash = val };
};
```

Memoization Macro

Define the macro:

```
define
<vlab_memoize_pure_method_decorator'struct_member>
"MEMOIZE <org'name>[ ]\(<args'name>,...\)[ ]\:[ ]
<ret'type>[ ][@<edge'any>] is <block>"
as computed {
...
}
```

The match-pattern covers also TCMs, but the original method may not contain wait cycles.

```
!foo_memoized_cache: list (key: hash) of
vlab_memoize_cache_entry_s;

final foo (x: vec_s,y: vec_s):vec_s@sys.any is {
   var hit: bool = FALSE;
   var search_input: list of byte = pack(packing.low,x,y);
   var search_key: uint =
util.mz_manager.DJBHash(search_input);
   var key_idx: int =
foo_memoized_cache.key_index(search_key);
   if key_idx != UNDEF {
      var entry: vlab_memoize_cache_entry_s =
foo_memoized_cache[key_idx];
      var x0:vec_s;
      var y0:vec_s;
      var r0:vec_s;
      if entry.input == search_input {
         unpack(packing.low, entry.output, r0);
         result = r0;
         hit    = TRUE;
         util.mz_manager.incr_hit("foo");
      };
   };

   (continues on next slide)
```

```
    (continued)

    if not hit {
        {result = foo_core(x, y)};
        var new_entry: vlab_memoize_cache_entry_s = new with {
            .input  = search_input;
            .output = pack(packing.low, result);
        };
        new_entry.set_hash(search_key);
        if key_idx != UNDEF {
            foo_memoized_cache[key_idx] = new_entry;
        } else {
            if foo_memoized_cache.size() >= 500 { compute
foo_memoized_cache.pop0() };
            foo_memoized_cache.push(new_entry);
        };
        util.mz_manager.incr_miss("foo");
    };
};
```

Summary

The memoization macro can be applied to every method that meets the requirements.

- The *e* package is licensed under GPLv2; it contains some more features not mentioned.
- The source code can be downloaded from www.github.com (archive: *vlab_memoize*).
- Contributions welcome!
- Questions ?

Thorsten Dworzak thorsten.dworzak@verilab.com