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

Thorsten Dworzak

Verilab GmbH, St. Martin-Str. 53-55, D-81669 München, Germany
+49-178-7996134, thorsten.dworzak@verilab.com

ABSTRACT

The *e* hardware verification language provides a powerful macro-definition syntax which can be used to extend the language, i.e. create new constructs that have syntax checking, and the look-and-feel of built-in constructs. Furthermore, it has a reflection API to query and control the properties of simple and compound types and objects.

In this paper we will show how these mechanisms are used to create an automatic memoization macro for pure methods (in *e*, what is generally called a function is called a method, while function arguments are called parameters).

Memoization is an optimization technique that trades run-time for memory space. Run-time critical methods can be sped-up using caching: the input and output parameter values of a method call are saved in a lookup-table. Upon each call of the method, the input values are compared with the cached ones. In case of a match, the function will return the cached result instead of recalculating it. If the cost of the cache-lookup and update is lower than calculating the result, it might be beneficial to memoize the method.

INTRODUCTION

e-Macros

```
define WD 16;
...
wdata: uint(bits: WD);
```

Figure 1: Simple macro

The *e*-language provides three different types of macros. First, there are simple text replacements, most commonly used for named constants such as vector widths (Figure 1). These are similar to the macros known in C and Verilog.

Second, there are parameterized macros which unfold to more complex text and are syntactically checked. These are called *define-as macros* (Figure 2).

However, for true language extension, the *define-as-computed macr*o type allows the use of *e*-code to parse the macro parameters and generate the macro expansion string. This macro type gives the highest degree of flexibility and allows to overrule pre-defined syntax and/or add new constructs. This form of macro will be used for the memoization.

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

Figure 2: Define-as macro

Both types of parameterized macros must expand to syntactically complete *e*-code; partial substitution is not possible. This also means that the expanded code must belong to a syntactic category, like *statement* or *expression*.

The Reflection API

Some computer languages, e.g. Ruby and Perl, support *type introspection*, which allows the type and object properties to be queried at runtime. One simple example from C is the *sizeof(<type>)* operator which returns the data size of a given type in bytes. Other examples are the *instance_of* and *kind_of* member methods of Ruby classes.

```
type col_t: [RED, GREEN, BLUE];
var t: rf_type = rf_manager.get_type_by_name("col_t");
```

Figure 3: rf_manager API

With *reflection* [2], on the other hand, it is possible to also manipulate language objects. This provides read and write access to the meta-data of programs, i.e. to the information about the structs, methods and fields that a program contains. In *e* it is possible by means of the methods of the global *rf_manager* object to get, for example, a list of all fields of a struct and their properties, change a field value, or return the handle of a type using its name as a string (see example Figure 3). Reflection is built into the *e* language. In fact, the Specman data browser is built using this API.

Memoization

Memoization [3] is an optimization technique that trades run-time for memory space. Run-time critical methods can be sped-up using caching: the input and output parameter values of a method call are saved in a lookup-table (Figure 4). Upon each call of the method, the input values are compared with the cached ones. In case of a match, the function will return the cached result instead of recalculating it. If no matching set of input values have been found, the result will be stored in the cache memory. If the cost of the cache lookup and update is lower than calculating the result, and the number of different input value combinations is limited, it is beneficial to memoize the method, i.e. implement the described caching functionality.

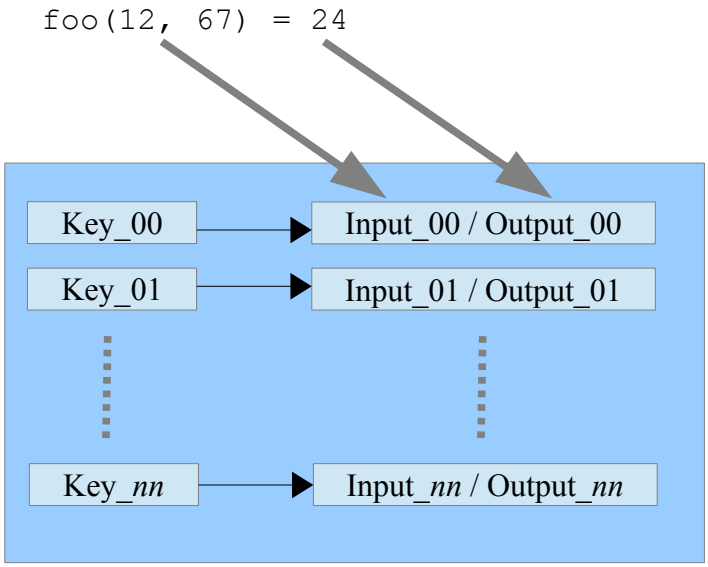

Figure 4: Cache memory

Caveats: Memoization can only be applied to pure methods, i.e. methods where the result depends only on the input parameters. They must be stateless and may not exhibit side-effects such as screen output. To be able to generalize we also specify that the only output of the method is the return value, and no parameters passed by reference are modified.

The run-time saving effect of caching depends on the cost of the caching overhead and hit-rate versus the calculation time for the result in the original method. The caching overhead comprises of the calculation of a key value to be used for the cache-entry lookup and the search of the cache for a given key. The hit-rate is determined by the cache size, the number of different input value combinations and the number of subsequent method calls. The more often the memoized method is called with repeated parameter values, the more speed-up will be measurable.

Methods that lend itself to caching are for example expensive dictionary look-ups, recursive graph/tree searches and analog component models.

Key Generation

A cached result must be uniquely identifiable by its input parameter values. In a one-dimensional cache structure (or hash table, see also [4]), this is done by means of a key

value representing one or more sets of input values. There are many different approaches for generating the key, of which we list just two here.

Key as String A string representation of the inputs of e.g. the method *foo(a: uint, b: string)* can be generated with

```
var key: string = sprintf("%s%s",hex(a),b);
```

For simple input parameters this has the advantage of fast key generation. The key is a simple type and can be used as the key value for a keyed list in *e*, providing us with a fast lookup function. However, for compound or list parameters it would be difficult to create a simple key value; this requires the generation of a string representation for all data types which is too compute-intensive (but could be done using the reflection API).

Key as UINT In this paper we chose to generate a 32-bit key using one of the hashing methods known to the computer science domain. Usually these hashing methods take a list of bytes and generate a 32-bit value out of it that is very distinctive for similar lists of bytes. How do we get a list of bytes from our set of input values? The *pack()* method can be used.

For example two input parameters *a: uint* and *b: my_struct* we can compute the key (using the default packing option) as

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

The packing method for structs requires that the struct's members are physical fields which is a restriction of the type of functions that can be used for memoization. Here we can use the reflection API to check that this restriction has been honoured (it is possible to create a new packing method using the reflection API that is not restricted to physical fields, but this has been left to further work).

This list can then be passed to the hash function for key generation. Note that the hash key is theoretically not unique for every set of input parameters, so every cache lookup has to make a final check for hash-collision: if the hash key exists, we must also check if the stored input parameters match with the function call.

THE MEMOIZATION MACRO

The goal is to create a memoized method from every method that fulfills the required conditions for memoization and key generation. These requirements are met by the implementation of the memoization macro which is described in detail below.

Infrastructure

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

Figure 5: The vlab_memoize_manager

We require some global methods and objects that will be implemented in a dedicated struct, the *vlab_memoize_manager_s*. This itself is instantiated in the global unit *sn_util*, see Figure 5.

Cache Memory

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

Figure 6: Cache-entry

Each memoized function gets its own cache memory, which is a keyed-list of the struct *vlab_memoize_cache_entries_s* as in Figure 6. It contains the hash value and the packed lists of input and output values.

Hash Methods

```
extend vlab_memoize_manager_s {
    final DJBHash(str: list of byte) : uint is { ... }
};
extend vlab_memoize_cache_entry_s {
   set_hash(val: uint) is { hash = val                              };
   calc_hash()         is { hash = util.mz_manager.DJBHash(input) };
};
```

Figure 7: Hash Accessor Methods

As mentioned, there are a number of known cache methods. We chose the *DJBHash()* method developed by DJ Bernstein [5] which is simple enough to not sacrifice run-time and is good at generating distinctive keys. Its implementation is not shown here. We can extend the cache-entry struct to manipulate the hash key, see Figure 7.

Checking the Input/Output Types

```
extend vlab_memoize_manager_s {
    deep_is_physical(name: string): bool is { ... };
};
```

Figure 8: Method to Check Packability

The packing method we use for the key-generation requires that the input and output parameters are either scalars, or structs with physical fields, or lists thereof. To check this requirement we use the reflection API. A method *deep_is_physical()* has been implemented (see Figure 8) that checks each parameter during the macro expansion: if the parameter is a struct or a list, check whether it is packable (i.e. has only physical members) or recursively descend into the object tree of the struct/list until that decision can be made.

Macro Match Pattern

Given the example below for a method, we want to memoize it at the place of its definition using the simple syntax

```
MEMOIZE bar(a: uint, b: my_struct):uint { <method-body> }
```

which can be matched using the macro-definition:

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

This covers both TCMs (time-consuming method) and non-TCMs. Note: we support the syntax for a TCM, but it may not contain any trigger or wait statements. The cached method does not consume time. A possible extension of the macro would allow to specify a constant number of cycles.

The memoized method may not be extended. This is enforced by the macro through the *final* attribute added to the method definition.

RESULTS

The source-code is licensed under GPLv2 and is available at github.com (archive *vlab_memoize*). It contains some more features which have not been explained here, such as the optional hit/miss statistics and configurable cache-memory size.

SUMMARY

We have created a memoization macro for *e* methods using the advanced techniques of *define-as-computed* macros and the Reflection API. This can be used for compute-intensive methods that are suitable for caching.

REFERENCES

[1] Dominus, MJ. 2005. Higher Order Perl. Elsevier Inc., San Francisco, CA.

[2] Wikipedia, The Free Encyclopedia. 20 March 2032. “Reflection (computer programming)”. Wikimedia Foundation, Inc. Web. 31 March 2013. <http://en.wikipedia.org/wiki/Reflection_(computer_programming)>

[3] Wikipedia, The Free Encyclopedia. 14 July 2012. “Memoization”. Wikimedia Foundation, Inc. Web. 16 August 2012. <http://en.wikipedia.org/wiki/Memoization>

[4] Wikipedia, The Free Encyclopedia. 14 July 2012. “Hash Table”. Wikimedia Foundation, Inc. Web. 19 August 2012. <http://en.wikipedia.org/wiki/Hash_table>

[5] “General Purpose Hash Function Algorithms”, Arash Partow. Web. 03 Apr 2013. <http://www.partow.net/programming/hashfunctions/index.html>.