16.4 ASSERT: Dynamic Verification of Assertions on Function Types

ASSERT admits to add to symbolic mode RLISP code assertions (partly) specifying *types* of the arguments and results of RLISP expr procedures. These types can be associated with functions testing the validity of the respective arguments during runtime.

Author: Thomas Sturm.

16.4.1 Loading and Using

The package is loaded using `load_package` or `load!-package` in algebraic or symbolic mode, resp. There is a central switch `assertcheck`, which is off by default. With `assertcheck` off, all type definitions and assertions described in the sequel are ignored and have the status of comments. For verification of the assertions it most be turned on (dynamically) before the first relevant type definition or assertion.

ASSERT aims at the dynamic analysis of RLISP expr procedure in symbolic mode. All uses of `typedef` and `assert` discussed in the following have to take place in symbolic mode. There is, in contrast, a final print routine `assert_analyze` that is available in both symbolic and algebraic mode.

16.4.2 Type Definitions

Here are some examples for definitions of types:

```
typedef any;
typedef number checked by numberp;
typedef sf checked by sfpx;
typedef sq checked by sqp;
```

The first one defines a type `any`, which is not possibly checked by any function. This is useful, e.g., for functions which admit any argument at one position but at others rely on certain types or guarantee certain result types, e.g.,

```
procedure cellcnt(a);
   % a is any, returns a number.
   if not pairp a then 0 else cellcnt car a + cellcnt cdr a + 1;
```

The other ones define a type `number`, which can be checked by the RLISP function `numberp`, a type `sf` for standard forms, which can be checked by the function

`sfpx` provided by ASSERT, and similarly a type for standard quotients.

All type checking functions take one argument and return extended Boolean, i.e., non-nil iff their argument is of the corresponding type.

16.4.3 Assertions

Having defined types, we can formulate assertions on expr procedures in terms of these types:

```
assert cellcnt: (any) -> number;
assert addsq: (sq,sq) -> sq;
```

Note that on the argument side parenthesis are mandatory also with only one argument. This notation is inspired by Haskell but avoids the intuition of currying.[1]

Assertions can be dynamically checked only for expr procedures. When making assertions for other types of procedures, a warning is issued and the assertion has the status of a comment.

It is important that assertions via assert come after the definitions of the used types via `typedef` and also after the definition of the procedures they make assertions on.

A natural order for adding type definitions and assertions to the source code files would be to have all typedefs at the beginning of a module and assertions immediately after the respective functions. Fig. 16.1 illustrates this. Note that for dynamic checking of the assertions the switch `assertcheck` has to be on during the translation of the module; i.e., either when reading it with `in` or during compilation. For compilation this can be achieved by commenting in the `on assertcheck` at the beginning or by parameterizing the Lisp-specific compilation scripts in a suitable way.

An alternative option is to have type definitions and assertions for specific packages right after `load_package` in batch files as illustrated in Fig. 16.2.

16.4.4 Dynamic Checking of Assertions

Recall that with the switch `assertcheck` off at translation time, all type definitions and assertions have the status of comments. We are now going to discuss how these statements are processed with `assertcheck` on.

`typedef` marks the type identifier as a valid type and possibly associates the given typechecking function with it. Technically, the property list of the type identifier is

[1] This notation has benn suggested by C. Zengler

```
module sizetools;

load!-package 'assert;

% on assertcheck;

typedef any;
typedef number checked by number;

procedure cellcnt(a);
   % a is any, returns a number.
   if not pairp a then 0 else cellcnt car a + cellcnt cdr a + 1;

assert cellcnt: (any) -> number;

% ...

endmodule;

end;  % of file
```

Figure 16.1: Assertions in the source code.

```
load_package sizetools;
load_package assert;

on assertcheck;

lisp <<
   typedef any;
   typedef number checked by numberp;

   assert cellcnt: (any) -> number
>>;

% ... computations ...

assert_analyze();

end;  % of file
```

Figure 16.2: Assertions in a batch file.

used for both purposes.

`assert` encapsulates the procedure that it asserts on into another one, which checks the types of the arguments and of the result to the extent that there are typechecking functions given. Whenever some argument does not pass the test by the typechecking function, there is a warning message issued. Furthermore, the following numbers are counted for each asserted function:

1. The number of overall calls,
2. the number of calls with at least one assertion violation,
3. the number of assertion violations.

These numbers can be printed anytime in either symbolic or algebraic mode using the command `assert_analyze()`. This command at the same time resets all the counters.

Fig. 16.3 shows an interactive sample session.

16.4.5 Switches

As discussed above, the switch `assertcheck` controls at translation time whether or not assertions are dynamically checked.

There is a switch `assertbreak`, which is off by default. When on, there are not only warnings issued for assertion violations but the computations is interrupted with a corresponding error.

The statistical counting of procedure calls and assertion violations is toggled by the switch `assertstatistics`, which is on by default.

16.4.6 Efficiency

The encapsulating functions introduced with assertions are automatically compiled.

We have experimentally checked assertions on the standard quotient arithmetic `addsq`, `multsq`, `quotsq`, `invsq`, `negsq` for the test file `taylor.tst` of the TAYLOR package. For CSL we observe a slowdown of factor 3.2, and for PSL we observe a slowdown of factor 1.8 in this particular example, where there are 323 750 function calls checked altogether.

The ASSERT package is considered an analysis and debugging tool. Production system should as a rule not run with dynamic assertion checking. For critical applications, however, the slowdown might be even acceptable.

```
1: symbolic$

2* load_package assert$

3* on assertcheck$

4* typedef sq checked by sqp;
sqp

5* assert negsq: (sq) -> sq;
+++ negsq compiled, 13 + 20 bytes

(negsq)

6* assert addsq: (sq,sq) -> sq;
+++ addsq compiled, 14 + 20 bytes

(addsq)

7* addsq(simp 'x,negsq simp 'y);

((((x . 1) . 1) ((y . 1) . -1)) . 1)

8* addsq(simp 'x,negsq numr simp 'y);

*** assertion negsq: (sq) -> sq violated by arg1 (((y . 1) . 1))

*** assertion negsq: (sq) -> sq violated by result (((y . -1) . -1))

*** assertion addsq: (sq,sq) -> sq violated by arg2 (((y . -1) . -1))

*** assertion addsq: (sq,sq) -> sq violated by result (((y . -1) . -1))

(((y . -1) . -1))

9* assert_analyze()$
------------------------------------------------------------------------
function          #calls              #bad calls   #assertion violations
------------------------------------------------------------------------
addsq                  2                       1                      2
negsq                  2                       1                      2
------------------------------------------------------------------------
sum                    4                       2                      4
------------------------------------------------------------------------
```

Figure 16.3: An interactive sample session.

16.4.7 Possible Extensions

Our assertions could be used also for a static type analysis of source code. In that case, the type checking functions become irrelevant. On the other hand, the introduction of variouse unchecked types becomes meaningful.

In a model, where the source code is systematically annotated with assertions, it is technically no problem to generalize the specification of procedure definitions such that assertions become implicit. For instance, one could *optionally* admit procedure definitions like the following:

```
procedure cellcnt(a:any):number;
   if not pairp a then 0 else cellcnt car a + cellcnt cdr a + 1;
```