HOL-TestGen Version 1.8
USER GUIDE

BRUCKER A D / BRUGGER L / FELIACHI A / KELLER C / KRIEGER M P / LONGUET D / NEMOUCHI Y / TUONG F / WOLFF B

Unité Mixte de Recherche 8623
CNRS-Université Paris Sud-LRI

04/2016

Rapport de Recherche N° 1586
Copyright © 2003–2012 ETH Zurich, Switzerland
Copyright © 2007–2015 Achim D. Brucker, Germany
Copyright © 2008–2016 University Paris-Sud, France
Copyright © 2016 The University of Sheffield, UK

All rights reserved.
Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the copyright holders nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Note:
This manual describes HOL-TestGen version 1.8.0 (r12601). The manual of version 1.8.0 is also available as technical report number 1586 from the Laboratoire en Recherche en Informatique (LRI), Université Paris-Sud 11, France.
1. Introduction

Today, essentially two validation techniques for software are used: software verification and software testing. Whereas verification is rarely used in “real” software development, testing is widely-used, but normally in an ad-hoc manner. Therefore, the attitude towards testing has been predominantly negative in the formal methods community, following what we call Dijkstra’s verdict [13, p.6]:

“Program testing can be used to show the presence of bugs, but never to show their absence!”

More recently, three research areas, albeit driven by different motivations, converge and result in a renewed interest in testing techniques:

**Abstraction Techniques:** model-checking raised interest in techniques to abstract infinite to finite models. Provided that the abstraction has been proven sound, testing may be sufficient for establishing correctness [3, 12].

**Systematic Testing:** the discussion over test adequacy criteria [26], i.e. criteria solving the question “when did we test enough to meet a given test hypothesis,” led to more systematic approaches for partitioning the space of possible test data and the choice of representatives. New systematic testing methods and abstraction techniques can be found in [16, 14].

**Specification Animation:** constructing counter-examples has raised interest also in the theorem proving community, since combined with animations of evaluations, they may help to find modelling errors early and to increase the overall productivity [2, 17, 11].

The first two areas are motivated by the question “are we building the program right?” the latter is focused on the question “are we specifying the right program?” While the first area shows that Dijkstra’s Verdict is no longer true under all circumstances, the latter area shows, that it simply does not apply in practically important situations. In particular, if a formal model of the environment of a software system (e.g. based among others on the operation system, middleware or external libraries) must be reverse-engineered, testing (“experimenting”) is without alternative (see [7]).

Following standard terminology [26], our approach is a specification-based unit test. In general, a test procedure for such an approach can be divided into:

**Test Case Generation:** for each operation the pre/postcondition relation is divided into sub-relations. It assumes that all members of a sub-relation lead to a similar behavior of the implementation.

**Test Data Generation:** (also: Test Data Selection) for each test case (at least) one representative is chosen so that coverage of all test cases is achieved. From the resulting test data, test input data processable by the implementation is extracted.
**Test Execution**: the implementation is run with the selected test input data in order to determine the test output data.

**Test Result Verification**: the pair of input/output data is checked against the specification of the test case.

The development of HOL-TestGen [8] has been inspired by [15], which follows the line of specification animation works. In contrast, we see our contribution in the development of techniques mostly on the first and to a minor extent on the second phase.

Building on QuickCheck [11], the work presented in [15] performs essentially random test, potentially improved by hand-programmed external test data generators. Nevertheless, this work also inspired the development of a random testing tool for Isabelle [2]. It is well-known that random test can be ineffective in many cases; in particular, if preconditions of a program based on recursive predicates like “input tree must be balanced” or “input must be a typable abstract syntax tree” rule out most of randomly generated data. HOL-TestGen exploits these predicates and other specification data in order to produce adequate data, combining automatic data splitting, automatic constraint solving, and manual deduction.

As a particular feature, the automated deduction-based process can log the underlying test hypothesis made during the test; provided that the test hypothesis is valid for the program and provided the program passes the test successfully, the program must guarantee correctness with respect to the test specification, see [6, 9] for details.
2. Preliminary Notes on Isabelle/HOL

2.1. Higher-order logic — HOL

Higher-order logic (HOL) \[10, 1\] is a classical logic with equality enriched by total polymorphic higher-order functions. It is more expressive than first-order logic, since e.g. induction schemes can be expressed inside the logic. Pragmatically, HOL can be viewed as a combination of a typed functional programming language like Standard ML (SML) or Haskell extended by logical quantifiers. Thus, it often allows a very natural way of specification.

2.2. Isabelle

Isabelle \[21, 18\] is a generic theorem prover. New object logics can be introduced by specifying their syntax and inference rules. Among other logics, Isabelle supports first order logic (constructive and classical), Zermelo-Fränkel set theory and HOL, which we chose as the basis for the development of HOL-TestGen.

Isabelle consists of a logical engine encapsulated in an abstract data type \texttt{thm} in Standard ML; any \texttt{thm} object has been constructed by trusted elementary rules in the kernel. Thus Isabelle supports user-programmable extensions in a logically safe way. A number of generic proof procedures (tactics) have been developed; namely a simplifier based on higher-order rewriting and proof-search procedures based on higher-order resolution.

We use the possibility to build on top of the logical core engine own programs performing symbolic computations over formulae in a logically safe (conservative) way: this is what HOL-TestGen technically is.

\(\text{to be more specific: parametric polymorphism}\)
3. Installation

3.1. Prerequisites

HOL-TestGen is built on top of Isabelle/HOL, version 2013-2, thus you need a working installation of *Isabelle 2013-2*. To install Isabelle, follow the instructions on the Isabelle web-site:


3.2. Installing HOL-TestGen

In the following we assume that you have a running Isabelle 2013-2 environment. The installation of HOL-TestGen requires the following steps:

1. Unpack the HOL-TestGen distribution, e.g.:

   ```
tar zxvf hol-testgen-1.8.0.tar.gz
   ```

   This will create a directory `hol-testgen-1.8.0` containing the HOL-TestGen distribution.

   ```
cd hol-testgen-1.8.0
   ```

   and build the HOL-TestGen heap image for Isabelle by calling

   ```
isabelle build -d . -b HOL-TestGen
   ```

3.3. Starting HOL-TestGen

HOL-TestGen can now be started using the `isabelle` command:

```isabelle jedit -d . -l HOL-TestGen "examples/unit/List/List_test.thy"
```  
After a few seconds you should see a jEdit window similar to the one shown in Figure 3.1. Alternatively, the example can be run in batch mode, e.g.,

```
isabelle build -d . HOL-TestGen-List
```  

\footnote{Note that the `isabelle` command must be provided by Isabelle 2013-2.}
Figure 3.1.: A HOL-TestGen session Using the jEdit Interface of Isabelle
4. Using HOL-TestGen

4.1. HOL-TestGen: An Overview

HOL-TestGen allows one to automate the interactive development of test cases, refine them to concrete test data, and generate a test script that can be used for test execution and test result verification. The test case generation and test data generation (selection) is done in an Isar-based [25] environment (see Figure 4.1 for details). The test executable (and the generated test script) can be built with any SML-system.

4.2. Test Case and Test Data Generation

In this section we give a brief overview of HOL-TestGen related extension of the Isar [25] proof language. We use a presentation similar to the one in the Isar Reference Manual [25], e.g. “missing” non-terminals of our syntax diagrams are defined in [25]. We introduce the HOL-TestGen syntax by a (very small) running example: assume we want to test a function that computes the maximum of two integers.

Starting your own theory for testing: For using HOL-TestGen you have to build your Isabelle theories (i.e. test specifications) on top of the theory Testing instead of Main. A sample theory is shown in Table 4.1.

Defining a test specification: Test specifications are defined similar to theorems in Isabelle, e.g.,

```
test_spec "prog a b = max a b"
```

would be the test specification for testing a simple program computing the maximum value of two integers. The syntax of the keyword `test_spec : theory \rightarrow proof(prove)` is given by:

```
\langle goal \rangle \::= \langle props \rangle \quad \langle longgoal \rangle \::= \langle thmdecl \rangle \langle contextelem \rangle\quad shows \langle goal \rangle
```

Please look into the Isar Reference Manual [25] for the remaining details, e.g. a description of `\langle contextelem \rangle`.
theory max_test
imports Testing
begin

test_spec "prog a b = max a b"
  apply (gen_test_cases "prog" simp: max_def)
  mk_test_suite "max_test"

gen_test_data "max_test"

thm max_test.concrete_tests

generate_test_script "max_test"
thm max_test.test_script

text {"Testing an SML implementation:"}
export_code max_test.test_script in SML module_name TestScript file "impl/sml/max_test_script.sml"

text {"Finally, we export the raw test data in an XML-like format:"}
export_test_data "impl/data/max_data.dat" max_test

end

Table 4.1.: A simple Testing Theory
Generating symbolic test cases: Now, abstract test cases for our test specification can (automatically) be generated, e.g. by issuing

\texttt{apply(gen_test_cases "prog" simp: max_def)}

The \texttt{gen_test_cases} : \emph{method} tactic allows to control the test case generation in a fine-granular manner:

\begin{verbatim}
  gen_test_cases ⟨depth⟩ ⟨breadth⟩ ⟨progname⟩ ⟨clasimpmod⟩
\end{verbatim}

where \(⟨depth⟩\) is a natural number describing the depth of the generated test cases and \(⟨breadth⟩\) is a natural number describing their breadth. Roughly speaking, the \(⟨depth⟩\) controls the term size in data separation lemmas in order to establish a regularity hypothesis (see \[6\] for details), while the \(⟨breadth⟩\) controls the number of variables occurring in the test specification for which regularity hypotheses are generated. The default for \(⟨depth⟩\) and \(⟨breadth⟩\) is 3 resp. 1. \(⟨progname⟩\) denotes the name of the program under test. Further, one can control the classifier and simplifier sets used internally in the \texttt{gen_test_cases} tactic using the optional \(⟨clasimpmod⟩\) option:

\begin{verbatim}
  ⟨clasimpmod⟩ ::= simp add del only cong split iff add del iff add del intro elim dest del?
\end{verbatim}

The generated test cases can be further processed, e.g., simplified using the usual Isabelle/HOL tactics.

Creating a test suite: HOL-TestGen provides a kind of container, called \textit{test-suites}, which store all relevant logical and configuration information related to a particular test-scenario. Test-suites were initially created after generating the test cases (and test hypotheses); you should store your result of the derivation, usually the test-theorem which is the output of the test-generation phase, in a test suite by:

\texttt{mk_test_suite "max_test"}

for further processing. This is done using the \texttt{mk_test_suite : proof(prove) \rightarrow proof(prove) \mid theory} command which also closes the actual “proof state” (or test state). Its syntax is given by:

\begin{verbatim}
  mk_test_suite ⟨name⟩
\end{verbatim}

where \(⟨name⟩\) is a fresh identifier which is later used to refer to this test state. This name is even used at the very end of the test driver generation phase, when test-executions are performed (externally to HOL-TestGen in a shell). Isabelle/HOL can access the corresponding test theorem using the identifier \(⟨name⟩\).test_thm, e.g.:
Generating test data: In a next step, the test cases can be refined to concrete test data:

`gen_test_data "max_test"`

The `gen_test_data : theory|proof → theory|proof` command takes only one parameter, the name of the test suite for which the test data should be generated:

```
--- gen_test_data - ⟨name⟩ -----------------------------------------------
```

After the successful execution of this command Isabelle can access the test hypothesis using the identifier `⟨name⟩.test_hyps` and the test data using the identifier `⟨name⟩.test_data`

```
thm max_test.test_hyps
thm max_test.concrete_test
```

In our concrete example, we get the output:

```
THYP ((∃ x xa. x ≤ xa ∧ prog x xa = xa) −→ (∀ xa. x ≤ xa → prog x xa = xa))
THYP ((∃ x xa. ¬ x ≤ xa ∧ prog x xa = x) −→ (∀ xa. ¬ x ≤ xa → prog x xa = x))
```

as well as:

```
prog −9 −3 = −3
prog −5 −8 = −5
```

By default, generating test data is done by calling the random solver. This is fine for such a simple example, but as explained in the introduction, this is far incomplete when the involved data-structures become more complex. To handle them, HOL-TestGen also comes with a more advanced data generator based on SMT solvers (using their integration in Isabelle, see e.g. [4]).

To turn on SMT-based data generation, use the following option:

```
declare [[ testgen_SMT]]
```

(which is thus set to `false` by default). It is also recommended to turn off the random solver:

```
declare [[ testgen_iterations =0]]
```

In order for the SMT solver to know about constant definitions and properties, one needs to feed it with these definitions and lemmas. For instance, if the test case involves some inductive function `foo`, you can provide its definition to the solver using:

```
declare foo.simps [testgen_smt_facts]
```

as well as related properties (if needed).

A complete description of the configuration options can be found below.

Exporting test data: After the test data generation, HOL-TestGen is able to export the test data into an external file, e.g.:
**export_test_data** "test_max.dat" "max_test"

exports the generated test data into a file `text_max.dat`. The generation of a test data file is done using the `export_test_data : theory|proof → theory|proof` command:

```
export_test_data (filename) (name) ⟨smlprogname⟩
```

where ⟨filename⟩ is the name of the file in which the test data is stored and ⟨name⟩ is the name of a collection of test data in the test environment.

**Generating test scripts:** After the test data generation, HOL-TestGen is able to generate a test script, e.g.:

```
gen_test_script "test_max.sml" "max_test" "prog"
"myMax.max"
```

produces the test script shown in Table 4.2 that (together with the provided test harness) can be used to test real implementations. The generation of test scripts is done using the `generate_test_script : theory|proof → theory|proof` command:

```
gen_test_script (filename) (name) (progname) ⟨smlprogname⟩
```

where ⟨filename⟩ is the name of the file in which the test script is stored, and ⟨name⟩ is the name of a collection of test data in the test environment, and ⟨progname⟩ the name of the program under test. The optional parameter ⟨smlprogname⟩ allows for the configuration of different names of the program under test that is used within the test script for calling the implementation.

Alternatively, the code-generator can be configured to generate test-driver code in other programming languages, see below.

**Configure HOL-TestGen:** The overall behavior of test data and test script generation can be configured, e.g.

```
declare [[ testgen_iterations =15]]
```

The parameters (all prefixed with `testgen_`) have the following meaning:

- **depth:** Test-case generation depth. Default: 3.
- **breadth:** Test-case generation breadth. Default: 1.
- **bound:** Global bound for data statements. Default: 200.
- **case_breadth:** Number of test data per case, weakening uniformity. Default: 1.
- **iterations:** Number of attempts during random solving phase. Default: 25. Set to 0 to turn off the random solver.
- **gen_prelude:** Generate datatype specific prelude. Default: true.
- **gen_wrapper:** Generate wrapper/logging-facility (increases verbosity of the generated test script). Default: true.
- **SMT:** If set to “true” external SMT solvers (e.g., Z3) are used during test-case generation. Default: false.
Table 4.2.: Test Script

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>smt_facts:</td>
<td>Add a theorem to the SMT-based data generator basis.</td>
</tr>
<tr>
<td>toString:</td>
<td>Type-specific SML-function for converting literals into strings (e.g., Int.toString), used for generating verbose output while executing the generated test script. Default: &quot;&quot;.</td>
</tr>
<tr>
<td>setup_code:</td>
<td>Customized setup-initialization code (copied verbatim to generated test script). Default: &quot;&quot;.</td>
</tr>
<tr>
<td>dataconv_code:</td>
<td>Customized code for converting datatypes (copied verbatim to generated test script). Default: &quot;&quot;.</td>
</tr>
<tr>
<td>type_range_bound:</td>
<td>Bound for choosing type instantiation (effectively used elements type grounding list). Default: 1.</td>
</tr>
<tr>
<td>type_candidates:</td>
<td>List of types that are used, during test script generation, for instantiating type variables (e.g., α list). The ordering of the types determines their likelihood of being used for instantiating a polymorphic type. Default: [int, unit, bool, int set, int list]</td>
</tr>
</tbody>
</table>

**Configuring the test data generation:** Further, an attribute test : attribute is provided, i.e.:  

**lemma** max_abs_case [test "maxtest"]: max 4 7 = 7

or
**Table 4.3.:** Implementation in SML of max

```
structure myMax = struct
  fun max x y = if (x < y) then y else x
end
```

4.3. Test Execution and Result Verification

In principle, any SML-system, e.g. [24, 22, 23, 19, 20], should be able to run the provided test-harness and generated test-script. Using their specific facilities for calling foreign code, testing of non-SML programs is possible. For example, one could test

- implementations using the .Net platform (more specific: CLR IL), e.g. written in C# using smlnet [23],
- implementations written in C using, e.g. the foreign language interface of sml/NJ [23] or MLton [20],
- implementations written in Java using mlj [19].

Also, depending on the SML-system, the test execution can be done within an interpreter (it is even possible to execute the test script within HOL-TestGen) or using a compiled test executable. In this section, we will demonstrate the test of SML programs (using SML/NJ or MLton) and ANSI C programs.

4.3.1. Testing an SML-Implementation

Assume we have written a max-function in SML (see Table 4.3) stored in the file `max.sml` and we want to test it using the test script generated by HOL-TestGen. Following Figure 4.1 we have to build a test executable based on our implementation, the generic test harness (harness.sml) provided by HOL-TestGen, and the generated test script (test_max.sml), shown in Table 4.2.

If we want to run our test interactively in the shell provided by sml/NJ, we just have to issue the following commands:

```
use "harness.sml"
use "max.sml"
use "test_max.sml"
```

After the last command, sml/NJ will automatically execute our test and you will see a output similar to the one shown in Table 4.4.

If we prefer to use the compilation manager of sml/NJ, or compile our test to a single test executable using MLton, we just write a (simple) file for the compilation manager of sml/NJ (which is understood both, by MLton and sml/NJ) with the following content:
Test Results:
=============
Test 0 - SUCCESS, result: 69
Test 1 - SUCCESS, result: ~11

Summary:
--------
Number successful tests cases: 2 of 2 (ca. 100%)
Number of warnings: 0 of 2 (ca. 0%)
Number of errors: 0 of 2 (ca. 0%)
Number of failures: 0 of 2 (ca. 0%)
Number of fatal errors: 0 of 2 (ca. 0%)

Overall result: success
===============

Table 4.4.: Test Trace

Group is
harness.sml
max.sml
test_max.sml

#if(defined(SMLNJ_VERSION))
  $/basis.cm
  $smlnj/compiler/compiler.cm
#else
#endif

and store it as test.cm. We have two options, we can

- use sml/NJ: we can start the sml/NJ interpreter and just enter

  CM.make("test.cm")

  which will build a test setup and run our test.

- use MLton to compile a single test executable by executing

  mlton test.cm

  on the system shell. This will result in a test executable called test which can be
directly executed.

In both cases, we will get a test output (test trace) similar to the one presented in Table 4.4.
Table 4.5.: Implementation in ANSI C of max

### 4.3.2. Testing Non-SML Implementations

Suppose we have an ANSI C implementation of max (see Table 4.5) that we want to test using the foreign language interface provided by MLton. First we have to import the max method written in C using the _import keyword of MLton. Further, we provide a “wrapper” function doing the pairing of the curried arguments:

```sml
cstructure myMax = struct
val cmax = _import "max": int * int -> int ;
fun max a b = cmax(a,b);
end
```

We store this file as max.sml and write a small configuration file for the compilation manager:

```
Group is
harness.sml
max.sml
test_max.sml
```

We can compile a test executable by the command

```
mlton -default-ann 'allowFFI true' test.cm max.c
```

on the system shell. Again, we end up with an test executable test which can be called directly. Running our test executable will result in trace similar to the one presented in Table 4.4.

### 4.4. Profiling Test Generation

HOL-TestGen includes support for profiling the test procedure. By default, profiling is turned off. Profiling can be turned on by issuing the command

```
--- profiling_on -------------------------------
```

Profiling can be turned off again with the command

```
--- profiling_off -----------------------------
```

When profiling is turned on, the time consumed by gen_test_cases and gen_test_data is recorded and associated with the test theorem. The profiling results can be printed by

```
--- print_clocks -------------------------------
```
A LaTeX version of the profiling results can be written to a file with the command
\begin{verbatim}
\texttt{write_clocks \langle filename \rangle}
\end{verbatim}

Users can also record the runtime of their own code. A time measurement can be started by issuing
\begin{verbatim}
\texttt{start_clock \langle name \rangle}
\end{verbatim}

where \langle name \rangle is a name for identifying the time measured. The time measurement is completed by
\begin{verbatim}
\texttt{stop_clock \langle name \rangle}
\end{verbatim}

where \langle name \rangle has to be the name used for the preceding start\_clock. If the names do not match, the profiling results are marked as erroneous. If several measurements are performed using the same name, the times measured are added. The command
\begin{verbatim}
\texttt{next_clock}
\end{verbatim}

proceeds to a new time measurement using a variant of the last name used.

These profiling instructions can be nested, which causes the names used to be combined to a path. The \texttt{Clocks} structure provides the tactic analogues \texttt{start\_clock\_tac}, \texttt{stop\_clock\_tac} and \texttt{next\_clock\_tac} to these commands. The profiling features available to the user are independent of HOL-TestGen’s profiling flag controlled by \texttt{profiling\_on} and \texttt{profiling\_off}. 

20
5. Examples

5.1. List

5.1.1. Testing List Properties

```ml
theory List-test
imports List
../../..src/codegen-fsharp/Code-Integer-Fsharp
../../..src/Testing
begin

In this example we present the current main application of HOL-TestGen: generating test data for black box testing of functional programs within a specification based unit test. We use a simple scenario, developing the test theory for testing sorting algorithms over lists, develop test specifications (elsewhere called test targets or test goals), and explore the different possibilities.

A First Model and a Quick Walk Through
In the following we give a first impression of how the testing process using HOL-TestGen looks like. For brevity we stick to default parameters and explain possible decision points and parameters where the testing can be improved in the next section.

Writing the Test Specification
We start by specifying a primitive recursive predicate describing sorted lists:

```ml
primrec is-sorted:: int list ⇒ bool
  where is-sorted [] = True |
  is-sorted (x#xs) = (case xs of
      [] ⇒ True
      | y#ys ⇒ x ≤ y ∧ is-sorted xs)
```

We will use this HOL predicate for describing our test specification, i.e. the properties our implementation should fulfill:

```ml
test-spec is-sorted(PUT l)
```

where `prog` is a “placeholder” for our program under test.

However, for the code-generation necessary to generate a test-driver and actually run the test of an external program, the `program under test` or `PUT` for short, it is sensible to represent the latter as an un-interpreted constant; the code-generation will later on tweaked such that
the place-holder in the test-driver code is actually linked to the real, external program which is a black box from the point of view of this model (the testing procedure needs actually only executable code).

consts SUT :: 'a list ⇒ 'a list

Note that any other name would do the trick as well.

Generating test cases

Now we can automatically generate test cases. Using the default setup, we just apply our gen-test-cases:

declare PO-def [simp del] apply (gen-test-cases 3 1 SUT)

which leads to the test partitioning one would expect:

1. is-sorted (SUT [])
2. \( \text{THYP} \) (is-sorted (SUT []) \( \rightarrow \) is-sorted (SUT [ ]))
3. is-sorted (SUT [??X8X31])
4. \( \text{THYP} \) ((\( \exists x \). is-sorted (SUT [x])) \( \rightarrow \) (\( \forall x \). is-sorted (SUT [x])))
5. is-sorted (SUT [??X6X25, ??X5X24])
6. \( \text{THYP} \) ((\( \exists x xa \). is-sorted (SUT [xa, x])) \( \rightarrow \) (\( \forall x xa \). is-sorted (SUT [xa, x])))
7. is-sorted (SUT [??X3X17, ??X2X16, ??X1X15])
8. \( \text{THYP} \) ((\( \exists x xa xb \). is-sorted (SUT [xb, xa, x])) \( \rightarrow \)
\( (\forall x xa xb \). is-sorted (SUT [xb, xa, x])))
9. \( \text{THYP} \) (3 < length l \( \rightarrow \) is-sorted (SUT l))

Now we bind the test theorem to a particular named test environment.

mk-test-suite is-sorted-result

The current test theorem contains holes, that correspond to the concrete data of the test that have not been generated yet

thm is-sorted-result.test-thm

Generating test data

Now we want to generate concrete test data, i.e. all variables in the test cases must be instantiated with concrete values. This involves a random solver which tries to solve the constraints by randomly choosing values.

thm is-sorted-result.test-thm

gen-test-data is-sorted-result

thm is-sorted-result.test-inst-thm

Which leads to the following test data:

is-sorted (SUT [])

is-sorted (SUT [10])

is-sorted (SUT [3, 10])

is-sorted (SUT [−8, −3, −3])

Note that by the following statements, the test data, the test hypotheses and the test theorem can be inspected interactively.
The generated test data can be exported to an external file:

\texttt{export-test-data impl/data/test-data.data is-sorted-result}

**Test Execution and Result Verification**  In principle, any SML-system should be able to run the provided test-harness and generated test-script. Using their specific facilities for calling foreign code, testing of non-SML programs is possible. For example, one could test implementations written

- for the .Net platform, e.g., written in C\# using sml.net [23],
- in C using, e.g. the foreign language interface of sml/NJ [24] or MLton [20],
- in Java using MLj [19].

Depending on the SML-system, the test execution can be done within an interpreter or using a compiled test executable. Testing implementations written in SML is straight-forward, based on automatically generated test scripts. This generation is based on the internal code generator of Isabelle and must be set up accordingly.

The following, we show the general generation of test-scripts (part of the finally generated test-driver) in different languages; finally, we will concentrate on the test-generation scenario for C.

code-printing
code-printing
\begin{verbatim}
constant SUT => (Fsharp) myList.sort
   and (SML)  myList.sort
   and (Scala) myList.sort
\end{verbatim}
generate-test-script is-sorted-result

\textbf{testing an SML implementation:}

\texttt{export-code is-sorted-result.test-script in SML module-name TestScript file impl/sml/is-sorted-test-script.sml}

We use the SML test script also for testing an implementation written in C:

\texttt{export-code is-sorted-result.test-script in SML module-name TestScript file impl/c/is-sorted-test-script.sml}

Testing an F\# implementation:

\texttt{export-code is-sorted-result.test-script in Fsharp module-name TestScript file impl/fsharp/is-sorted-test-script.fs}

We use the F\# test script also for testing an implementation written in C\#:

\texttt{export-code is-sorted-result.test-script in Fsharp module-name TestScript file impl/csharp/is-sorted-test-script.fs}

Testing a Scala implementation:

\texttt{export-code is-sorted-result.test-script in Scala}
module-name TestScript file impl/scala/is-sorted-test-script.scala

We use the Scala script also for testing an implementation written in Java:

export-code is-sorted-result.test-script in Scala
module-name TestScript file impl/java/is-sorted-test-script.scala

Finally, we export the raw test data in an XML-like format:

export-test-data impl/data/is-sorted-test-data.dat is-sorted-result

which generates the following test harness:

In the following, we assume an ANSI C implementation of our sorting method for sorting C arrays that we want to test. (In our example setup, it is contained in the file impl/c/sort.c.)

Using the foreign language interface provided by the SML compiler MLton we first have to import the sort method written in C using the _import keyword of MLton and further, we provide a “wrapper” doing some datatype conversion, e.g. converting lists to arrays and vice versa:

structure myList = struct
  val csort = _import "sort": int array * int -> int array;
  (* this is the link to the external, "black-box" program *)
  fun ArrayToList a = Array.foldl (op ::) [] a;
  fun sort_list list = ArrayToList (csort(Array.fromList(list),(length list)));
  fun sort list = map IntInf.fromInt (sort_list (map IntInf.toInt list))
end

That’s all, now we can build the test executable using MLton and end up with a test executable which can be called directly. In impl/c, the process of

1. compiling the generated is_sorted_test_script.sml, the test harness (harness.sml), a main routine (main.sml) and this wrapper myList (contained in the generated List.sml) to to a combined test-driver in C,

2. compiling the C test-driver and linking it to the program under test sort.c, and

3. executing the test

is captured in a Makefile. So: executes the test and displays a test-statistic as shown in Table 5.1 on the facing page

A Refined Model and Improved Test-Results

Obviously, in reality one would not be satisfied with the test cases generated in the previous section: for testing sorting algorithms one would expect that the test data somehow represents the set of permutations of the list elements. We have already seen that the test specification used in the last section “only” enumerates lists up to a specific length without any ordering constraints on their elements. What is missing, is a test that input and output sequence are in fact permutations of each other. We could state for example:

fun del-member :: 'a ⇒ 'a list ⇒ 'a list option
where del-member x [] = None
>make
mlton -default-ann 'allowFFI true' is_sorted_test.mlb sort.c
./is_sorted_test

Test Results:
==============
Test 0 - SUCCESS
Test 1 - SUCCESS
Test 2 - SUCCESS
Test 3 - SUCCESS
Test 4 - SUCCESS
Test 5 - SUCCESS
Test 6 - SUCCESS

Summary:
--------
Number successful tests cases: 7 of 7 (ca. 100%)
Number of warnings: 0 of 7 (ca. 0%)
Number of errors: 0 of 7 (ca. 0%)
Number of failures: 0 of 7 (ca. 0%)
Number of fatal errors: 0 of 7 (ca. 0%)

Overall result: success
==============

Table 5.1.: A Sample Test Trace: The ascending property tested.
\[
del-member x (y \neq S) = (\text{if } x = y \text{ then } \text{Some } S \\
\text{else case del-member } x S \text{ of} \\
\text{None } \Rightarrow \text{None} \\
\text{Some } S' \Rightarrow \text{Some}(y \neq S'))
\]

fun is-permutation :: 'a list ⇒ 'a list ⇒ bool
where is-permutation [] [] = True
|is-permutation (a#S)(a'#S') = (if a = a' then is-permutation S S' \\
else case del-member a S' of \\
None ⇒ False \\
| Some S'' ⇒ is-permutation S (a'#S''))

fun is-perm :: 'a list ⇒ 'a list ⇒ bool
where is-perm [] [] = True
|is-perm [] T = False \\
is-perm (a#S) T = (if length T = length S + 1 \\
then is-perm S (remove1 a T) \\
else False)

value is-perm [1,2,3::int] [3,1,2]

A test for permutation, that is not hopelessly non-constructive like "the existence of a bijection on the indexes \([0 .. n-1]\), that is pairwise mapped to the list" or the like, is obviously quite complex; the apparent "mathematical specification" is not always the easiest. We convince ourselves that the predicate \text{is-permutation} indeed captures our intuition by animations of the definition:

value is-permutation [1,2,3] [3,2,1::nat]
value ¬ is-permutation [1,2,3] [3,1::nat]
value ¬ is-permutation [2,3] [3,2,1::nat]
value ¬ is-permutation [1,2,1,3] [3,2,1::nat]
value is-permutation [2,1,3] [1::nat,3,2]

value is-perm [1,2,3] [3,2,1::nat]
value ¬ is-perm [1,2,3] [3,1::nat]
value ¬ is-perm [2,3] [3,2,1::nat]
value ¬ is-perm [1,2,1,3] [3,2,1::nat]
value is-perm [2,1,3] [1::nat,3,2]

... which are all executable and thus were compiled and all evaluated to true.

Based on these concepts, a test-specification is straight-forward and easy:

\text{declare} [[goals-limit=5]]
\text{apply} [gen-test-cases 5 1 SUT]
\text{mk-test-suite ascending-permutation-test}

A quick inspection of the test theorem reveals that there are in fact no relevant constraints to solve, so test-data selection is easy:

\text{declare} [[testgen-iterations=100]]
\text{gen-test-data ascending-permutation-test}
\text{thm ascending-permutation-test.concrete-tests}
Again, we convert this into test-scripts that can be compiled to a test-driver.

generate-test-script ascending-permutation-test

thm ascending-permutation-test-test-script

We use the SML implementation also for testing an implementation written in C:

export-code ascending-permutation-test-test-script in SML
module-name TestScript file impl/c/ascending-permutation-test-script.sml

Try make run_ascending_permutation in directory impl/c to compile and execute the generated test-driver.

A Test-Specification based on a Comparison with a Reference Implementation

We might opt for an alternative modeling approach: Thus we decide to try a more “descrip-
tive” test specification that is based on the behavior of an insertion sort algorithm:

fun ins :: ('a:linorder) ⇒ 'a list ⇒ 'a list
where ins x [] = [x]
    | ins x (y#ys) = (if (x < y) then x#y#ys else (y#(ins x ys))

fun sort :: ('a:linorder) list ⇒ 'a list
where sort [] = []
    | sort (x#xs) = ins x (sort xs)

Now we state our test specification by requiring that the behavior of the program under
test PUT is identical to the behavior of our specified sorting algorithm sort:

Based on this specification gen_test_cases produces test cases representing all permu-
tations of lists up to a fixed length n. Normally, we also want to configure up to which length
lists should be generated (we call this the depth of test case), e.g. we decide to generate lists
up to length 3. Our standard setup

declare [[goals-limit=100]]
test-spec sort l = PUT l
apply(gen-test-cases PUT)

mk-test-suite is-sorting-algorithm

generates 9 test cases describing all permutations of lists of length 1, 2 and 3. "Permu-
tation" means here that not only test cases (i.e. I/O-partitions) are generated for lists of
length 0, 1, 2 and 3; the partitioning is actually finer: for two-elementary lists, for example,
the case of a list with the first element larger or equal and the dual case are distinguished.

The entire test-theorem looks as follows:

[] = PUT []; THYP ([] = PUT [] → [] = PUT []); [??X31X177] = PUT [??X31X177];
THYP ((∃x. [x] = PUT [x]) → (∀x. [x] = PUT [x])); PO (??X29X169 < ??X28X168);
[??X29X169, ??X28X168] = PUT [??X29X169, ??X28X168]; THYP ((∃x xa. xa < x ∧
[xa, x] = PUT [xa, x]) → (∀x xa. xa < x → [xa, x] = PUT [xa, x])); PO (¬ ??X26X158 < ??X25X157); [??X25X157, ??X26X158] = PUT [??X26X158, ??X25X157]; THYP ((∃x xa. ¬ xa < x ∧
[xa, x] = PUT [xa, x]) → (∀x xa. ¬ xa < x → [xa, x] = PUT [xa, x])); PO
((??X22X144 < ??X21X143 ∧ ??X23X145 < ??X21X143) ∧ ??X23X145 < ??X22X144);
[??X23X145, ??X22X144, ??X21X143] = PUT [??X23X145, ??X22X144, ??X21X143];
THYP ((∃x xa xb. xa < x ∧ xb < x ∧ xb < xa ∧ [xb, xa, x] = PUT [xb, xa, x]) →
(∀x xa xb. xa < x → xb < x → xb < xa → [xb, xa, x] = PUT [xb, xa, x])); PO (¬

27
**permutation-test**

In this scenario, 39 test cases are generated describing all permutations of lists of length 1, 2, 3 and 4. "Permutation" means here that not only test cases (i.e. I/O-partitions) are
generated for lists of length 0, 1, 2, 3, 4; the partitioning is actually finer: for two-elementary lists, take one case for the lists with the first element larger or equal.

The case for all lists of depth 5 is feasible, however, it will already take 8 minutes. The resulting constraints for the test cases are complex and require more intensive effort in resolving.

There are several options for the test-data selection. On can either use the (very old) random solver or the more modern smt interface. (One day, we would also have a nitpick-interface to constraint solving via bitblasting sub-models of the constraints to SAT.) The random solver, however, finds only 67 instances out of 150 abstract test cases, while smt instantiates all of them:

Test theorem (gen_test_data) 'permutation_test': 67 test cases in 2.951 seconds

declare [[testgen-iterations=0]]
declare [[testgen-SMT]]
gen-test-data  permutation-test
thm  permutation-test.concrete-tests

generate-test-script  permutation-test
thm  permutation-test.test-script

We use the SML implementation also for testing an implementation written in C:

export-code  permutation-test.test-script in SML
module-name TestScript file impl/c/permutation-test-script.sml

We obtain test cases like:

\[
\begin{align*}
\text{\texttt{SUT}} &= \text{\texttt{SUT}} [-3] \\
\text{\texttt{SUT[0, 1, 0]}} &= \text{\texttt{SUT}} [-3, -1, 0] \\
\text{\texttt{SUT[0, 0, 1]}} &= \text{\texttt{SUT}} [-2, -1, 0] \\
\text{\texttt{SUT[0, 0, 1, 0]}} &= \text{\texttt{SUT}} [-2, -1, 0] \\
\text{\texttt{SUT[0, 0, 0, 1]}} &= \text{\texttt{SUT}} [-1, -3, -1, 0] \\
\text{\texttt{SUT[0, 0, 1, 0]}} &= \text{\texttt{SUT}} [-1, -2, -1, 0] \\
\text{\texttt{SUT[0, 0, 2, 0]}} &= \text{\texttt{SUT}} [-1, -2, -1, 0] \\
\text{\texttt{SUT[0, 0, 0, 1, 0]}} &= \text{\texttt{SUT}} [-1, -2, -1, 0] \\
\text{\texttt{SUT[0, 0, 1, 0, 0]}} &= \text{\texttt{SUT}} [-1, -2, -1, 0] \\
\text{\texttt{SUT[0, 0, 0, 1, 0, 0]}} &= \text{\texttt{SUT}} [-1, -2, -1, 0] \\
\end{align*}
\]
\[-2, -1, 0, 0\] = SUT \[0, -2, -1, 0\]
\[0, 1, 1, 1\] = SUT \[1, 0, 1, 1\]
\[0, 0, 1, 1\] = SUT \[1, 0, 1, 0\]
\[-2, -2, -1, 0\] = SUT \[-2, -1, -2, 0\]
\[-1, -1, 0, 0\] = SUT \[-1, -1, -1, 0\]
\[-1, 0, 0, 0\] = SUT \[0, 0, 0, 0\]
\[-2, -3, -2, -1, 0\] = SUT \[-4, -3, -2, -1, 0\]
\[-2, -1, 0, 1, 1\] = SUT \[-2, -1, 0, 1, 1\]
\[0, 1, 1, 2\] = SUT \[0, 1, 2, 1\]
\[0, 0, 0, 1, 2\] = SUT \[0, 0, 1, 2, 0\]
\[-3, -3, -2, -1, 0\] = SUT \[-3, -3, -2, -1, 0\]
\[-1, -1, 0, 1, 1\] = SUT \[-1, -1, 0, 1, 1\]
\[0, 0, 1, 1, 2\] = SUT \[0, 0, 1, 2, 1\]
\[0, 0, 0, 1, 2\] = SUT \[0, 0, 1, 2, 0\]
\[-1, -3, -2, -2, -1, 0\] = SUT \[-3, -3, -2, -2, -1, 0\]
\[-1, 0, 0, 0, 1, 1\] = SUT \[-1, 0, 0, 1, 1\]
\[-1, 0, 0, 0, 1, 1\] = SUT \[-1, 0, 0, 2, 1\]
\[0, 0, 1, 1, 2\] = SUT \[0, 1, 1, 2, 1\]
\[0, 0, 1, 1, 2\] = SUT \[0, 1, 1, 2, 0\]
\[-2, -3, -3, -2, -1, 0\] = SUT \[-2, -3, -2, -3, -1, 0\]
\[0, 0, 1, 2, 2\] = SUT \[0, 1, 0, 2, 2\]
\[0, 0, 0, 0, 1, 1\] = SUT \[0, 0, 0, 0, 1, 0\]
\[-3, -3, -2, -1, -1, 0\] = SUT \[-3, -2, -1, -1, 0\]
\[-3, -2, -1, 0, 0\] = SUT \[-3, -2, -1, 0, 0\]
\[-1, 1, 1, 1, 1\] = SUT \[-1, 0, 1, 1, 1\]
\[0, 0, 0, 1, 2\] = SUT \[0, 1, 0, 2, 0\]
\[0, 0, 0, 0, 1, 1\] = SUT \[0, 0, 0, 0, 1, 0\]
\[-3, -3, -2, -1, -1, 0\] = SUT \[-3, -2, -1, -1, 0\]
\[-3, -2, -1, 0, 0\] = SUT \[-3, -2, -1, 0, 0\]
\[-1, 1, 1, 1, 1\] = SUT \[-1, 0, 1, 1, 1\]
\[0, 0, 1, 2, 2\] = SUT \[0, 1, 2, 2, 0\]
\[0, 0, 0, 1, 1\] = SUT \[0, 0, 0, 1, 1\]
\[0, 0, 0, 1, 1\] = SUT \[0, 0, 0, 1, 0\]
\[0, 1, 1, 2, 2\] = SUT \[0, 1, 2, 2, 1\]
\[0, 0, 1, 2, 0\] = SUT \[0, 1, 2, 0, 0\]
\[0, 0, 0, 1, 1\] = SUT \[0, 0, 0, 1, 1\]
\[0, 1, 1, 2, 2\] = SUT \[0, 1, 2, 2, 1\]
\[0, 0, 0, 0, 1\] = SUT \[0, 0, 0, 0, 1\]
\[0, 1, 1, 2, 2\] = SUT \[0, 1, 2, 2, 1\]
\[0, 0, 0, 1, 0\] = SUT \[0, 0, 1, 0, 0\]
\[-3, -3, -2, -2, -2, -1, 0\] = SUT \[-3, -3, -2, -2, -2, -1, 0\]
\[-3, -2, -1, 0, 0\] = SUT \[-3, 0, -2, -1, 0\]
\[-1, 0, 1, 1, 1\] = SUT \[-1, 1, 0, 1, 1\]
\[0, 1, 1, 1, 2\] = SUT \[0, 2, 1, 2, 1\]
\[0, 0, 1, 2, 2\] = SUT \[0, 2, 1, 2, 0\]
\[-2, -2, -1, -1, 0\] = SUT \[-2, -1, -2, -1, 0\]
\[-2, -2, -1, 0, 0\] = SUT \[-2, 0, -2, -1, 0\]
\[0, 0, 1, 1, 1\] = SUT \[0, 1, 0, 1, 1\]
\[0, 1, 1, 1, 2\] = SUT \[1, 2, 0, 2, 1\]
\[0, 0, 1, 2, 2\] = SUT \[1, 2, 0, 2, 0\]
\[-3, -2, -2, -1, 0\] = SUT \[-3, -2, -1, -2, 0\]
\[-2, -1, 0, 0, 1\] = SUT \[-2, 0, 1, -1, 0\]
\[-2, -1, 0, 0, 0\] = SUT \[-2, 0, 0, -1, 0\]
\[0, 0, 1, 1, 1\] = SUT \[0, 1, 1, 1, 0\]
\[-3, -2, -1, 1, 1\] = SUT \[-3, -2, -1, 1, 0\]
\[-2, -2, -1, 0, 0\] = SUT \[-2, -2, -1, 0, 0\]
\[0, 0, 1, 1, 1\] = SUT \[0, 1, 0, 1, 1\]
\[0, 1, 1, 1, 2\] = SUT \[2, 0, 1, 2, 1\]
\[0, 0, 1, 2, 2\] = SUT \[2, 0, 1, 2, 0\]
\[-3, -2, -1, 1, 1\] = SUT \[-3, -2, -1, 1, 0\]
\[-2, -2, -1, 0, 0\] = SUT \[-2, -2, -1, 0, 0\]
\[0, 0, 1, 1, 1\] = SUT \[1, 0, 0, 1, 1\]
\[0, 1, 1, 1, 2\] = SUT \[2, 1, 0, 2, 1\]
\[0, 0, 1, 2, 2\] = SUT \[1, 0, 0, 1, 0\]
\[-3, -2, -2, -1, 0\] = SUT \[-3, -2, -1, -2, 0\]
\[-2, -1, 0, 0, 1\] = SUT \[-2, -1, 0, -1, 0\]
\[-2, -1, 0, 0, 0\] = SUT \[-2, -1, 0, 0, 0\]
\[0, 0, 1, 1, 1\] = SUT \[0, 1, 0, 1, 1\]
\[-3, -2, -2, -1, 1\] = SUT \[-3, -2, -1, 1, 0\]
\[-2, -1, 0, 0, 1\] = SUT \[-2, -1, 0, -1, 0\]
\[-2, -1, 0, 0, 0\] = SUT \[-2, -1, 0, 0, 0\]
\[0, 0, 1, 1, 1\] = SUT \[1, 2, 0, 1, 1\]
\[0, 0, 1, 1, 1\] = SUT \[1, 1, 0, 1, 1\]
\[-3, -3, -2, -1, 0\] = SUT \[-3, -3, -1, -2, 0\]
\[-2, -1, 0, 0, 1\] = SUT \[-2, -1, 0, -1, 0\]
\[-1, 1, 0, 0, 1\] = SUT \[-1, 1, 0, -1, 1\]
obtain many test cases with unresolved constraints where number of iterations which reveals that we need to set iterations to 100 to find all solutions 24 cases for lists of length 4 only 9 could be solved by the random solver (thus, overall 19 reveals that all cases for lists with length up to (and including) 3 could be solved. From the\[
SUT = \begin{bmatrix}
  [0, 0, 0, 1, 0] \\
  [0, 0, 0, 1, 2] \\
  [0, 0, 0, 1, 2] \\
  [0, 0, 0, 1, 2] \\
  [...] \\
  [0, 0, 0, 0, 0, 0] \\
  [0, 0, 0, 0, 0, 0] \\
  [...]
\end{bmatrix}
\]
Instead of increasing the number of iterations one could also add other techniques such as
If we scale down to only 10 iterations, this is not sufficient to solve all conditions, i.e. we obtain many test cases with unresolved constraints where RSF marks unsolved cases. In these cases, it is unclear if the test partition is empty. Analyzing the generated test data reveals that all cases for lists with length up to (and including) 3 could be solved. From the 24 cases for lists of length 4 only 9 could be solved by the random solver (thus, overall 19 of the 34 cases were solved). To achieve better results, we could interactively increase the number of iterations which reveals that we need to set iterations to 100 to find all solutions reliably.

\[
\begin{array}{c|ccc|c|cc|c|cc|c|cc|c|cc}
\text{iterations} & 5 & 10 & 20 & 25 & 30 & 40 & 50 & 75 & 100 \\
\hline
\text{solved goals (of 34)} & 13 & 19 & 23 & 24 & 25 & 29 & 33 & 33 & 34 \\
\end{array}
\]

Instead of increasing the number of iterations one could also add other techniques such as
1. deriving new rules that allow for the generation of a simplified test theorem,
2. introducing abstract test cases or
3. supporting the solving process by derived rules.

Running the test (in the current setup: make run_permutation_test) against our sample C-program under impl/c yields the following result:
Test Results:
=============
Test 0 - SUCCESS
Test 1 - SUCCESS
Test 2 - *** FAILURE: post-condition false
Test 3 - *** FAILURE: post-condition false
Test 4 - *** FAILURE: post-condition false
Test 5 - *** FAILURE: post-condition false
Test 6 - *** FAILURE: post-condition false
Test 7 - *** FAILURE: post-condition false
Test 8 - *** FAILURE: post-condition false
Test 9 - *** FAILURE: post-condition false
Test 10 - *** FAILURE: post-condition false
Test 11 - *** FAILURE: post-condition false
Test 12 - *** FAILURE: post-condition false
Test 13 - *** FAILURE: post-condition false
Test 14 - *** FAILURE: post-condition false
Test 15 - *** FAILURE: post-condition false
Test 16 - *** FAILURE: post-condition false
Test 17 - *** FAILURE: post-condition false
Test 18 - *** FAILURE: post-condition false
Test 19 - *** FAILURE: post-condition false
Test 20 - *** FAILURE: post-condition false
Test 21 - *** FAILURE: post-condition false
Test 22 - SUCCESS
Test 23 - SUCCESS
Test 24 - *** FAILURE: post-condition false
Test 25 - *** FAILURE: post-condition false
Test 26 - *** FAILURE: post-condition false
Test 27 - *** FAILURE: post-condition false
Test 28 - *** FAILURE: post-condition false
Test 29 - *** FAILURE: post-condition false
Test 30 - *** FAILURE: post-condition false
Test 31 - *** FAILURE: post-condition false
Test 32 - *** FAILURE: post-condition false

Summary:
--------
Number successful tests cases: 4 of 33 (ca. 12%)
Number of warnings: 0 of 33 (ca. 0%)
Number of errors: 0 of 33 (ca. 0%)
Number of failures: 29 of 33 (ca. 87%)
Number of fatal errors: 0 of 33 (ca. 0%)

Overall result: failed

Table 5.2.: A Sample Test Trace for the Permutation Test Scenario
Summary  A comparison of the three scenarios reveals that albeit a reasonable degree of automation in the test generation process, the essence of model-based test case generation remains an interactive process that is worth to be documented in a formal test-plan with respect to various aspects: the concrete modeling that is chosen, the precise formulation of the test-specifications (or: test-goals), the configuration and instrumentation of the test-data selection process, the test-driver synthesis and execution. This process can be complemented by proofs establishing equivalences allowing to convert initial test-specifications into more executable ones, or more 'symbolically evaluable’ ones, or that help to reduce the complexity of the constraint-resolution in the test-data selection process.

But the most important aspect remains: what is a good testing model ? Besides the possibility that the test specification simply does not test what the tester had in mind, the test theory and test-specification have a crucial importance on the quality of the generated test data that seems to be impossible to capture automatically.

Non-Inherent Higher-order Testing

HOL-TestGen can use test specifications that contain higher-order operators — although we would not claim that the test case generation is actually higher-order (there are no enumeration schemes for the function space, so function variables are untreated by the test case generation procedure so far).

Just for fun, we reformulate the problem of finding the maximal number in a list as a higher-order problem:

```plaintext
test-spec foldr max l (0::int) = PUT2 l
apply(gen-test-cases PUT2 simp:max-def)
mk-test-suite maximal-number

declare [[testgen-iterations = 200]]
gen-test-data maximal-number

thm maximal-number.concrete-tests
```

5.2. Bank

5.2.1. A Simple Deterministic Bank Model

```plaintext
theory Bank
imports

..//..//src/codegen-fsharp/Code-Integer-Fsharp
..//..//src/Testing
begin
```

34
The Bank Example: Test of a Distributed Transaction Machine

declare [[testgen-profiling]]

The intent of this little example is to model deposit, check and withdraw operations of a little Bank model in pre-postcondition style, formalize them in a setup for HOL-TestGen test sequence generation and to generate elementary test cases for it. The test scenarios will be restricted to strict sequence checking; this excludes aspects of account creation which will give the entire model a protocol character (a create-operation would create an account number, and then all later operations are just referring to this number; thus there would be a dependence between system output and input as in reactive sequence test scenarios.).

Moreover, in this scenario, we assume that the system under test is deterministic.

The theory of Proof-based Sequence Test Methodology can be found in [9].

The state of our bank is just modeled by a map from client/account information to the balance.

type-synonym client = string

type-synonym account-no = int

type-synonym data-base = (client × account-no) → int

Operation definitions: Concept A standard, JML or OCL or VCC like interface specification might look like:

Init: forall (c,no) : dom(data_base). data_base(c,no)>=0

op deposit (c : client, no : account_no, amount:nat) : unit
pre (c,no) : dom(data_base)
post data_base'\'=data_base[(c,no) := data_base(c,no) + amount]

op balance (c : client, no : account_no) : int
pre (c,no) : dom(data_base)
post data_base'\'=data_base and result = data_base(c,no)

op withdraw(c : client, no : account_no, amount:nat) : unit
pre (c,no) : dom(data_base) and data_base(c,no) >= amount
post data_base'\'=data_base[(c,no) := data_base(c,no) - amount]

Operation definitions: The model as ESFM Interface normalization turns this interface into the following input type:

datatype in-c = deposit client account-no nat
| withdraw client account-no nat
| balance client account-no

typ Bank.in-c
datatype out-c = depositO | balanceO nat | withdrawO

fun precond :: data-base ⇒ in-c ⇒ bool
where precond σ (deposit c no m) = ((c, no) ∈ dom σ)
  | precond σ (balance c no) = ((c, no) ∈ dom σ)
  | precond σ (withdraw c no m) = ((c, no) ∈ dom σ ∧ (int m) ≤ the(σ(c, no)))

fun postcond :: in-c ⇒ data-base ⇒ (out-c × data-base) set
where postcond (deposit c no m) σ = { (n, σ') . (n = depositO ∧ σ' = σ((c, no)→ the(σ(c, no)) + int m))}
  | postcond (balance c no) σ = { (n, σ') . (σ = σ' ∧ (∃ x. balanceO x = n ∧ x = nat(the(σ(c, no)))))}
  | postcond (withdraw c no m) σ = { (n, σ') . (n = withdrawO ∧ σ' = σ((c, no)→ the(σ(c, no)) - int m))}

definition init :: data-base ⇒ bool
where init σ ≡ ∀ x ∈ dom σ. the(σ x) ≥ 0

Constructing an Abstract Program Using the Operators_impl and strong_impl, we can synthesize an abstract program right away from the specification, i.e. the pair of pre- and post-condition defined above. Since this program is even deterministic, we will derive a set of symbolic execution rules used in the test case generation process which will produce symbolic results against which the PUT can be compared in the test driver.

lemma precond-postcond-implementable: implementable precond postcond
apply(auto simp: implementable-def)
apply(case-tac i, simp-all)
done

Based on this input-output specification, we construct the system model as the canonical completion of the (functional) specification consisting of pre- and post-conditions. Canonical completion means that the step function explicitly fails (returns None) if the precondition fails; this makes it possible to treat sequential execution failures in a uniform way. The system SYS can be seen as the step function in an input-output automata or, alternatively, a kind of Mealy machine over symbolic states, or, as an extended finite state machine.

definition SYS :: in-c ⇒ (out-c, data-base)MON_SE
where SYS = (strong-impl precond postcond)

The combinator strong-impl turns the pre-post pair in a suitable step functions with the aforementioned characteristics for failing pre-conditions.

Prerequisites

Proving Symbolic Execution Rules for the Abstractly Program The following lemmas reveal that this "constructed" program is actually (due to determinism of the spec):

lemma Eps-split-eq' : (SOME (x', y'). x'= x ∧ y'= y) = (SOME (x', y'). x = x' ∧ y = y')
by(rule arg-cong[of - - Eps], auto)
interpretation deposit : efsm-det
  precond postcond SYS (deposit c no m) λ. depositO
  λ σ. σ((c, no) ↦ (the(σ(c, no)) + int m)) λ σ. ((c, no) ∈ dom σ)
  by unfold-locales (auto simp: SYS-def Eps-split-eq')

find-theorems name:deposit
withdraw

interpretation withdraw : efsm-det
  precond postcond SYS (withdraw c no m) λ. withdrawO
  λ σ. σ((c, no) ↦ (the(σ(c, no)) - int m)) λ σ. ((c, no) ∈ dom σ)
  by unfold-locales (auto simp: SYS-def Eps-split-eq')

balance

interpretation balance : efsm-det
  precond postcond SYS (balance c no) λσ. balanceO (nat(the(σ(c, no))))
  λ σ. σ λ σ. ((c, no) ∈ dom σ)
  by unfold-locales (auto simp: SYS-def Eps-split-eq')

Now we close the theory of symbolic execution by excluding elementary rewrite steps on mbind FailSave, i.e. the rules mbind FailSave [] ?iostep ?σ = Some ([]), ?σ) mbind FailSave (?a ≠ ?S) ?iostep ?σ = (case ?iostep ?a ?σ of None ⇒ Some ([]), ?σ) | Some (out, σ') ⇒ case mbind FailSave ?S ?iostep σ' of None ⇒ Some ([out], σ') | Some (outs, σ'') ⇒ Some (out ≠ outs, σ'')

declare mbind.simps(1) [simp del]
mbind.simps(2) [simp del]

Here comes an interesting detail revealing the power of the approach: The generated sequences still respect the preconditions imposed by the specification - in this case, where we are talking about a client for which a defined account exists and for which we will never produce traces in which we withdraw more money than available on it.

Restricting the Test-Space by Test Purposes  We introduce a constraint on the input sequence, in order to limit the test-space a little and eliminate logically possible, but irrelevant test-sequences for a specific test-purpose. In this case, we narrow down on test-sequences concerning a specific client c with a specific bank-account number no.

We make the (in this case implicit, but as constraint explicitly stated) test hypothesis, that the SUT is correct if it behaves correct for a single client. This boils down to the assumption that they are implemented as atomic transactions and interleaved processing does not interfere with a single thread.

fun test-purpose :: [client, account-no, in-c list] ⇒ bool
where
  test-purpose c no [balance c' no'] = (c=c' ∧ no=no')
| test-purpose c no ((deposit c' no' m)#R) = (c=c' ∧ no=no' ∧ test-purpose c no R)
| test-purpose c no ((withdraw c' no' m)#R) = (c=c' ∧ no=no' ∧ test-purpose c no R)
| test-purpose c no - = False

lemma [simp] : test-purpose c no [a] = (a = balance c no)
by(cases a, auto)
lemma [simp] : \( R \neq \[] \Rightarrow \text{test-purpose } c\ no\ (a \# R) = \left( (\exists m. a = (\text{deposit}\ c\ no\ m)) \lor (\exists m. a = (\text{withdraw}\ c\ no\ m)) \right) \land \text{test-purpose } c\ no\ R \)

apply simp add: List.neq-Nil-conv, elim_eqE, simp
by (cases a, auto)

The TestGen Setup  The default configuration of gen_test_cases does not descend into sub-type expressions of type constructors (since this is not always desirable, the choice for the default had been for "non-descent"). This case is relevant here since in-c list has just this structure but we need ways to explore the input sequence type further. Thus, we need configure, for all test cases, and derivation descendants of the restulting clauses during splitting, again splitting for all parameters of input type in-c:

set-pre-safe-tac
\[
\langle\langle\ (\text{fn}}\ \text{ctxt} =\times\ \text{TestGen.ALLCASES}(\\text{TestGen.CLOSURE}\ (\text{TestGen.case-tac-typ}}\ \text{ctxt} [\text{Bank.in-c}]))\rangle\rangle
\]

Preparation: Miscellaneous  We construct test-sequences for a concrete client (implicitly assuming that interleaving actions with other clients will not influence the system behaviour. In order to prevent HOL-TestGen to perform case-splits over names, i.e., list of characters—we define it as constant.

definition \( c_0 ::\ \text{string}\ \text{where}\ \ c_0 = "meyer"

cconsts \( \text{PUT} :: (\text{in-c} \Rightarrow (\text{out-c}, '\sigma)\text{MON}_{SE})

lemma HH : \((A \land (A \rightarrow B)) = (A \land B)\) by auto

Small, rewriting based Scenarios including standard code-generation

Exists in two formats : General Fail-Safe Tests (which allows for scenarios with normal and exceptional behaviour; and Fail-Stop Tests, which generates Tests only for normal behaviour and correspond to inclusion test refinement.

In the following, we discuss a test-scenario with failsafe error semantics; i.e. in each test-case, a sequence may be chosen (by the test data selection) where the client has several accounts. In other words, tests were generated for both standard and exceptional behaviour. The splitting technique is general exploration of the type in-c list.

test-spec test-balance:
assumes account-def : \((c_0,\text{no}) \in \text{dom } \sigma_0\)
and accounts-pos : \(\text{init } \sigma_0\)
and test-purpose : test-purpose \(c_0\ no\ S\)
and sym-exec-spec :
\(\sigma_0 \models (s \leftarrow \text{mbind}_{\text{FailSave}} S\ \text{SYS}; \text{return} (s = x))\)
shows \(\sigma_0 \models (s \leftarrow \text{mbind}_{\text{FailSave}} S\ \text{PUT}; \text{return} (s = x))\)

Prelude: Massage of the test-theorem — representing the assumptions of the test explicitly in HOL and blocking \(x\) from being case-splitted (which complicates the process).
apply (rule rev-mp OF sym-exec-spec)
apply (rule rev-mp OF account-def)
apply (rule rev-mp OF accounts-pos)
apply (rule rev-mp OF test-purpose)
apply (rule-tac x=x in spec[OF allI])

Starting the test generation process.
apply (gen-test-cases 5 1 PUT)
apply (simp-all add: init-def HH split: HOL.split-if-asm)

mk-test-suite bank-simpleSNXB
thm bank-simpleSNXB.test-thm

And now the Fail-Stop scenario — this corresponds exactly to inclusion tests for normal-behaviour tests: any transition in the model is only possible iff the pre-conditions of the transitions in the model were respected.
declare Monads.mbind\'=bind [simp def]
test-spec test-balance2:
assumes account-def : (c0, no) ∈ dom σ0
and accounts-pos : init σ0
and test-purpose : test-purpose c0 no S
and sym-exec-spec :
s0 |= (s ← mbind\_FailStop S SYS; return (s = x))
shows σ0 |= (s ← mbind\_FailStop S PUT; return (s = x))

Prelude: Massage of the test-theorem — representing the assumptions of the test explicitly in HOL and blocking x from being case-splitted (which complicates the process).
apply (rule rev-mp OF sym-exec-spec)
apply (rule rev-mp OF account-def)
apply (rule rev-mp OF accounts-pos)
apply (rule rev-mp OF test-purpose)
apply (rule-tac x=x in spec[OF allI])

Starting the test generation process - variant without uniformity generation.
using [[no-uniformity]]
apply (gen-test-cases 4 1 PUT)

So lets go for a more non-destructive approach:
using [[goals-limit=20]]
apply (simp-all add: init-def HH split: HOL.split-if-asm)

using [[no-uniformity=false]]
apply (tactic TestGen.ALLCASES(TestGen.uniformityI-tac @{context} [PUT]))

mk-test-suite bank-simpleNB
thm bank-simpleNB.test-thm

Test-Data Generation
Configuration
declare [[testgen-iterations=0]]
Test Data Selection for the Normal and Exceptional Behaviour Test Scenario

gen-test-data bank-simpleSNXB
thm bank-simpleSNXB.test-thm
thm bank-simpleSNXB.test-inst-thm
thm bank-simpleSNXB.concrete-tests

Test Data Selection for the Normal Behaviour Test Scenario

declare []

declare c0-def [testgen-smt-facts]
declare mem-Collect-eq [testgen-smt-facts]
declare Collect-mem-eq [testgen-smt-facts]
declare dom-def [testgen-smt-facts]
declare the.simps [testgen-smt-facts]

Generating the Test-Driver for an SML and C implementation

The generation of the test-driver is non-trivial in this exercise since it is essentially two-staged: Firstly, we chose to generate an SML test-driver, which is then secondly, compiled to a C program that is linked to the actual program under test. Recall that a test-driver consists of four components:

- ../..//..//..//harness/sml/main.sml the global controller (a fixed element in the library),
- ../..//..//..//harness/sml/main.sml a statistic evaluation library (a fixed element in the library),
- bank_simple_test_script.sml the test-script that corresponds merely one-to-one to the generated test-data (generated)
- bank_adapter.sml a hand-written program; in our scenario, it replaces the usual (black-box) program under test by SML code, that calls the external C-functions via a foreign-language interface.

On all three levels, the HOL-level, the SML-level, and the C-level, there are different representations of basic data-types possible; the translation process of data to and from the C-code under test has therefore to be carefully designed (and the sheer space of options is sometimes a pain in the neck). Integers, for example, are represented in two ways inside Isabelle/HOL; there is the mathematical quotient construction and a "numerals" representation providing 'bit-string-representation-behind-the-scene" enabling relatively efficient symbolic computation. Both representations can be compiled "natively" to data types in the SML level. By
an appropriate configuration, the code-generator can map "int" of HOL to three different implementations: the SML standard library \texttt{Int.int}, the native-C interfaced by \texttt{Int32.int}, and the \texttt{IntInf.int} from the multi-precision library \texttt{gmp} underneath the polyml-compiler.

We do a three-step compilation of data-representations model-to-model, model-to-SML, SML-to-C.

A basic preparatory step for the initializing the test-environment to enable code-generation is:

\begin{verbatim}
generate-test-script bank-simpleSNXB
thm bank-simpleSNXB.test-script
\end{verbatim}

\begin{verbatim}
generate-test-script bank-simpleNB
thm bank-simpleNB.test-script
\end{verbatim}

In the following, we describe the interface of the SML-program under test, which is in our scenario an \textit{adapter} to the C code under test. This is the heart of the model-to-SML translation. The the SML-level stubs for the program under test are declared as follows:

\begin{verbatim}
consts balance-stub :: string ⇒ int ⇒ (int, 'σ)MONSE
code-printing
  constant balance-stub => (SML) BankAdapter.balance

consts deposit-stub :: string ⇒ int ⇒ int ⇒ (unit, 'σ)MONSE
code-printing
  constant deposit-stub => (SML) BankAdapter.deposit

consts withdraw-stub:: string ⇒ int ⇒ int ⇒ (unit, 'σ)MONSE
code-printing
  constant withdraw-stub => (SML) BankAdapter.withdraw
\end{verbatim}

Note that this translation step prepares already the data-adaption; the type \texttt{nat} is seen as a predicative constraint on integer (which is actually not tested). On the model-to-model level, we provide a global step function that distributes to individual interface functions via stubs (mapped via the code generation to SML ...). This translation also represents uniformly \texttt{nat} by \texttt{int}’s.

\begin{verbatim}
fun my-nat-conv :: int ⇒ nat
where my-nat-conv x = (if x <= 0 then 0 else Suc (my-nat-conv(x - 1)))
\end{verbatim}

\begin{verbatim}
fun stepAdapter :: (in-c ⇒ (out-c, 'σ)MONSE)
where
  stepAdapter(balance name no) =
    (x ← balance-stub name no; return(balanceO (my-nat-conv x)))
  | stepAdapter(deposit name no amount) =
    (- ← deposit-stub name no (int amount); return(depositO))
  | stepAdapter(withdraw name no amount)=
    (- ← withdraw-stub name no (int amount); return(withdrawO))
\end{verbatim}

The \texttt{stepAdapter} function links the HOL-world and establishes the logical link to HOL stubs which were mapped by the code-generator to adapter functions in SML (which call internally to C-code inside \texttt{bank_adapter.sml} via a foreign language interface)
... We configure the code-generator to identify the PUT with the generated SML code implicitly defined by the above stepAdapter definition.

code-printing
constant PUT => (SML) stepAdapter

And there we go and generate the bank_simple_test_script.sml:

export-code stepAdapter bank-simpleSNXB.test-script in SML
module-name TestScript file impl/c/bank-simpleSNXB-test-script.sml

export-code stepAdapter bank-simpleNB.test-script in SML
module-name TestScript file impl/c/bank-simpleNB-test-script.sml

More advanced Test-Case Generation Scenarios

Exploring a bit the limits ...

Rewriting based approach of symbolic execution ... FailSave Scenario

test-spec test-balance:
assumes account-def : (c0,no) ∈ dom σ0
and accounts-pos : init σ0
and test-purpose : test-purpose c0 no S
and sym-exec-spec :
σ0 |= (s ← mbind_FailSave S SYS; return (s = x))
shows σ0 |= (s ← mbind_FailSave S PUT; return (s = x))

Prelude: Massage of the test-theorem — representing the assumptions of the test explicitly in HOL and blocking x from being case-splitted (which complicates the process).

apply(insert account-def test-purpose sym-exec-spec)
apply(tactic TestGen.mp-fy 1,rule-tac x=x in spec[OF allI])

Starting the test generation process.

apply(gen-test-cases 5 1 PUT)

Symbolic Execution:

apply(simp-all add: HH split: HOL.split-if-asm)

mk-test-suite bank-large

gen-test-data bank-large

thm bank-large.concrete-tests

Rewriting based approach of symbolic execution ... FailSave Scenario

test-spec test-balance:
assumes account-def : (c0,no) ∈ dom σ0
and accounts-pos : init σ0
and test-purpose : test-purpose c0 no S
and sym-exec-spec :
σ0 |= (s ← mbind_FailStop S SYS; return (s = x))
shows σ0 |= (s ← mbind_FailStop S PUT; return (s = x))

Prelude: Massage of the test-theorem — representing the assumptions of the test explicitly in HOL and blocking x from being case-splitted (which complicates the process).
apply(insert account-def test-purpose sym-exec-spec)
apply(tactic TestGen.mp.fy 1, rule-tac x=x in spec[OF allI])

Starting the test generation process.
apply(gen-test-cases 3 1 PUT)

Symbolic Execution:
apply(simp-all add: HH split: HOL.split-if-asm)

mk-test-suite bank-large'
gen-test-data bank-large'

thm bank-large'.concrete-tests

And now, to compare, elimination based procedures ...
declare deposit.exec-mbindFSave-If[simp del]
declare balance.exec-mbindFSave-If[simp del]
declare withdraw.exec-mbindFSave-If[simp del]
declare deposit.exec-mbindFStop[simp del]
declare balance.exec-mbindFStop[simp del]
declare withdraw.exec-mbindFStop[simp del]

thm deposit.exec-mbindFSave-E withdraw.exec-mbindFSave-E balance.exec-mbindFSave-E

test-spec test-balance:
assumes account-defined: (c0,no) ∈ dom σ0
and accounts-pos : init σ0
and test-purpose : test-purpose c0 no S
and sym-exec-spec :
  σ0 |= (s ← mbindF failStop S SYS; return (s = x))
shows σ0 |= (s ← mbindF failStop S PUT; return (s = x))
apply(insert account-def test-purpose sym-exec-spec)
apply(tactic TestGen.mp.fy 1, rule-tac x=x in spec[OF allI])
using [[no-uniformity]]
apply(gen-test-cases 3 1 PUT)

apply(tactic ALLGOALS(TestGen.REPEAT' (ematch-tac [@{thm balance.exec-mbindFStop-E}],
  @{thm withdraw.exec-mbindFStop-E},
  @{thm deposit.exec-mbindFStop-E},
  @{thm valid-mbind'-mt}
  )))
apply(simp-all)
using([[no-uniformity=false]]
apply(tactic TestGen.ALLCASES(TestGen.unifomityI-tac @{context} [PUT]))

mk-test-suite bank-large-very

Yet another technique: "deep" symbolic execution rules involving knowledge from the
model domain. Here: input alphabet must be case-split over deposit, withdraw and balance.
This avoids that gen_test_cases has to do deep splitting.
theorem hulk:
assumes redex : σ ⊢ (s ← (mbind_FailStop (a ≠ S) SYS); return (P s))
and case-deposit : ∀c no m. a = deposit c no m ⇒ (c, no) ∈ dom σ ⇒
σ((c, no) ↦ the(σ(c, no))) + int m ⊢
(s ← mbind_FailStop S SYS; return P (depositO ≠ s)) ⇒
and case-withdraw : ∀c no m. a = withdraw c no m ⇒ (c, no) ∈ dom σ ⇒
int m ≤ the(σ(c, no)) ⇒
σ((c, no) ↦ the(σ(c, no))− int m) ⊢
(s ← mbind_FailStop S SYS; return P (withdrawO ≠ s)) ⇒
and case-balance : ∀c no. (c, no) ∈ dom σ ⇒
σ ⊢ (s ← mbind_FailStop S SYS;
return P (balanceO (nat (the(σ(c, no)))) ≠ s)) ⇒
shows Q
proof(cases a) print-cases
  case (deposit c no m) assume hyp : a = deposit c no m show Q
    using hyp redex
    apply(simp only: deposit.exec-mbindFStop)
    apply(rule case-deposit, auto)
    done
next
  case (withdraw c no m) assume hyp : a = withdraw c no m show Q
    using hyp redex
    apply(simp only: withdraw.exec-mbindFStop)
    apply(rule case-withdraw, auto)
    done
next
  case (balance c no) assume hyp : a = balance c no show Q
    using hyp redex
    apply(simp only: balance.exec-mbindFStop)
    apply(rule case-balance, auto)
    done
qed
Experimental Space
declare[[testgen-trace]]
ML⟨⟨prune-params-tac; Drule.triv forall equality⟩⟩
end

5.2.2. A Simple Non-Deterministic Bank Model

theory NonDetBank
imports
begin

declare [[testgen-profiling]]

This testing scenario is a modification of the Bank example. The purpose is to explore specifications which are nondeterministic, but at least \(\sigma\)-deterministic, i.e. from the observable output, the internal state can be constructed (which paves the way for symbolic executions based on the specification).

The state of our bank is just modeled by a map from client/account information to the balance.

\[
type-synonym \textit{client} = \text{string}
\]

\[
type-synonym \textit{account-no} = \text{int}
\]

\[
type-synonym \textit{register} = (\textit{client} \times \textit{account-no}) \rightarrow \text{int}
\]

**Operation definitions**  
We use a similar setting as for the Bank example — with one minor modification: the withdraw operation gets a non-deterministic behaviour: it may withdraw any amount between 1 and the demanded amount.

\[
op \text{deposit} (c : \textit{client}, no : \textit{account-no}, amount : \text{nat}) : \text{unit}
\]

\[
\text{pre} \ (c, no) : \text{dom(\textit{register})}
\]

\[
\text{post} \ \text{register}' = \text{register}[(c, no) := \text{register}(c, no) + amount]
\]

\[
op \text{balance} (c : \textit{client}, no : \textit{account-no}) : \text{int}
\]

\[
\text{pre} \ (c, no) : \text{dom(\textit{register})}
\]

\[
\text{post} \ \text{register}' = \text{register} \text{ and result} = \text{register}(c, no)
\]

\[
op \text{withdraw}(c : \textit{client}, no : \textit{account-no}, amount : \text{nat}) : \text{nat}
\]

\[
\text{pre} \ (c, no) : \text{dom(\textit{register}) and register}(c, no) >= amount
\]

\[
\text{post} \ \text{result} <= \text{amount} \text{ and}
\]

\[
\text{register}' = \text{register}[(c, no) := \text{register}(c, no) - \text{result}]
\]

Interface normalization turns this interface into the following input type:

\[
\text{datatype} \ \textit{in-c} = \text{deposit} \ \textit{client} \ \textit{account-no} \ \text{nat}
\]

\[
| \ \text{withdraw} \ \textit{client} \ \textit{account-no} \ \text{nat}
\]

\[
| \ \text{balance} \ \textit{client} \ \textit{account-no}
\]

\[
\text{datatype} \ \textit{out-c} = \text{depositO} | \ \text{balanceO} \ \text{nat} \ | \ \text{withdrawO} \ \text{nat}
\]

\[
\text{fun} \ \text{precond} :: \text{register} \Rightarrow \textit{in-c} \Rightarrow \text{bool}
\]

\[
\text{where} \ \text{precond} \ \sigma \ (\text{deposit} c \ no \ m) = ((c, no) \in \text{dom} \ \sigma)
\]

\[
| \ \text{precond} \ \sigma \ (\text{balance} c \ no) = ((c, no) \in \text{dom} \ \sigma)
\]

\[
| \ \text{precond} \ \sigma \ (\text{withdraw} c \ no \ m) = ((c, no) \in \text{dom} \ \sigma \ \land \ (\text{int} \ m) \leq \text{the}(\sigma(c, no)))
\]

\[
\text{fun} \ \text{postcond} :: \textit{in-c} \Rightarrow \text{register} \Rightarrow (\textit{out-c} \times \text{register}) \ \text{set}
\]

45
where postcond (deposit c no m) σ =
    (\{ (n,σ'). (n = depositO ∧ σ' = σ((c,σ)→ the(σ(c,σ)) + int m))\})
| postcond (balance c no m) σ =
    (\{ (n,σ'). (σ = σ' ∧ (\exists x. balanceO x = n ∧ x = nat(the(σ(c,σ))))))\})
| postcond (withdraw c no m) σ =
    (\{ (n,σ'). (\exists x≤m. n = withdrawO x ∧ σ' = σ((c,σ)→ the(σ(c,σ)) − int x))\})

Proving Symbolic Execution Rules for the Abstractly Constructed Program  Using the Operators impl and strong_impl, we can synthesize an abstract program right away from the specification, i.e. the pair of pre and postcondition defined above. Since this program is even deterministic, we derive a set of symbolic execution rules used in the test case generation process which will produce symbolic results against which the PUT can be compared in the test driver.

definition implementable :: [\σ ⇒ \ι ⇒ bool, \ι ⇒ (\o ⇒ \o, \σ)MON_SB] ⇒ bool
where implementable pre post = (\ ∀ σ ι. pre σ ι → (\∃ out σ'. pre out σ' σ ∈ post ι σ ))

lemma precond-postcond-implementable:
    implementable precond postcond
apply(auto simp: implementable-def)
apply(case-simp l, simp-all)
apply auto
done

The following lemmas reveal that this "constructed" program is actually (due to determinism of the spec)

lemma impl-1:
    strong-impl precond postcond (deposit c no m) =
    (\ λ σ . if (c, no) ∈ dom σ
        then Some(depositO,σ((c, no) → the (σ (c, no)) + int m))
        else None)
by(rule ext, auto simp: strong-impl-def )

lemma valid-both-spec1[simp]:
(\σ = (s ← mbind ((deposit c no m)\#S) (strong-impl precond postcond); return (P s))) =
    (if (c, no) ∈ dom σ
        then (σ((c, no) → the (σ (c, no)) + int m)) = (s ← mbind S (strong-impl precond postcond); return (P (depositO\#s)))
        else σ = (return (P [])))
by(auto simp: exec-mbindFSave impl-1)

lemma impl-2:
    strong-impl precond postcond (balance c no) =
    (\ λ σ . if (c, no) ∈ dom σ
        then Some(balanceO(nat(the (σ (c, no)))),σ)
        else None)
by (rule ext, auto simp: strong-impl-def Eps-split)

**Lemma valid-both-spec2 [simp]:**

\[
(\sigma \models (s \leftarrow \text{mbind } ((\text{balance } c \text{ no})\#S) \text{ (strong-impl precond postcond)}; \\
\quad \text{return } (P s))) = \\
(\text{if } (c, \text{no}) \in \text{dom } \sigma \\
\quad \text{then } (\sigma \models (s \leftarrow \text{mbind } S \text{ (strong-impl precond postcond)}; \\
\quad \text{return } (P (\text{balanceO(nat(the } (\sigma (c, \text{no}))))\#s)))) \\
\quad \text{else } (\sigma \models (\text{return } (P [])))))
\]

by (auto simp: exec-mbindFSave impl-2)

So far, no problem; however, so far, everything was deterministic. The following key-theorem does not hold:

**Lemma impl-3:**

\[
\text{strong-impl precond postcond} (\text{withdraw } c \text{ no } m) = \\
(\lambda \sigma. \text{if } (c, \text{no}) \in \text{dom } \sigma \land (\text{int } m) \leq \text{the}(\sigma(c,\text{no})) \land x \leq m \\
\quad \text{then } \text{Some}(\text{withdrawO } x,\sigma((c, \text{no}) \mapsto \text{the } (\sigma (c, \text{no})))) - \text{int } x) \\
\quad \text{else } \text{None})
\]

This also breaks our deterministic approach to compute the sequence beforehand and to run the test of PUT against this sequence.

However, we can give an acceptance predicate (an automaton) for correct behaviour of our PUT:

**Fun accept :: (in-c list × out-c list × int) ⇒ bool**

**Where**

\[
\text{accept}((\text{deposit } c \text{ no } n)\#S, \text{depositO}\#S', m) = \text{accept} (S, S', m + (\text{int } n)) \\
\text{accept}((\text{withdraw } c \text{ no } n)\#S, (\text{withdrawO } k)\#S', m) = (k \leq n \land \text{accept} (S, S', m - (\text{int } k))) \\
\text{accept}(\text{[balance } c \text{ no}], \text{[balanceO } n]), m) = (\text{int } n = m) \\
\text{accept}(a,b,c) = \text{False}
\]

This format has the advantage

TODO: Work out foundation. accept works on an abstract state (just one single balance of a user), while PUT works on the (invisible) concrete state. A data-refinement is involved, and it has to be established why it is correct.

**Test Specifications**

**Fun** test-purpose :: [client, account-no, in-c list] ⇒ bool

**Where**

\[
\text{test-purpose } c \text{ no } [] = \text{False} \\
\text{test-purpose } c \text{ no } (a\#R) = (\text{case } R \text{ of} \\
\quad [] \Rightarrow a = \text{balance } c \text{ no} \\
\quad a'\#R' \Rightarrow ((\exists m. a = \text{deposit } c \text{ no } m) \lor \\
\quad (\exists m. a = \text{withdraw } c \text{ no } m)) \land \\
\quad \text{test-purpose } c \text{ no } R)
\]

**Test-spec** test-balance:

assumes account-defined; (c, no) ∈ dom σ₀

and test-purpose : test-purpose c no is

shows σ₀ \models (os ← mbind is PUT; return (accept(is, os, the(σ₀ (c, no))))))
apply (insert account-defined test-purpose)
apply (gen-test-cases PUT split: HOL.split-if-asm)

mk-test-suite nbank
declare [[testgen-iterations = 0]]
gen-test-data nbank

thm nbank.concrete-tests
end

5.3. MyKeOS

5.3.1. A Shared-Memory-Model

theory SharedMemory
imports Main
begin

Shared Memory Model

Prerequisites
Prerequisite: a generalization of fun-upd-def: \( ?f(\langle a := b \rangle) \equiv \lambda x. \text{if } x = a \text{ then } b \text{ else } f x \). It represents updating modulo a sharing equivalence, i.e. an equivalence relation on parts of the domain of a memory.

definition fun-upd-equivp :: ('a ⇒ 'a ⇒ bool) ⇒ ('a ⇒ 'b) ⇒ ('a ⇒ 'b ⇒ ('a ⇒ 'b)) where
  fun-upd-equivp eq f a b = (λx. if eq x a then b else f x)

— This lemma is the same as Fun.fun-upd-same: \( (?f(\langle x := y \rangle) \text{ if } x = y) \); applied on our generalization fun-upd-equivp ?eq ?f ?a ?b = (λx. if ?eq x ?a then ?b else ?f x) of \( ?f(\langle a := b \rangle) \equiv \lambda x. \text{if } x = a \text{ then } b \text{ else } ?f x \). This proof tell if our function fun-upd-equivp op = f x y is equal to f this is equivalent to the fact that \( f x = y \)

lemma fun-upd-equivp-iff: ((fun-upd-equivp (op =) f x y) = f) = (f x = y)
  by (simp add :fun-upd-equivp-def, safe, erule subst, auto)

— Now we try to proof the same lemma applied on any equivalent relation equiv eqv instead of the equivalent relation op =. For this case, we had split the lemma to 2 parts. the lemma fun-upd-equivp-iff-part1 to proof the case when eq (f a) b \( \rightarrow \) eq (fun-upd-equivp eqv f a b z) (f z), and the second part is the lemma fun-upd-equivp-iff-part2 to proof the case equiv eqv \( \Rightarrow \) fun-upd-equivp eqv f a b = f \( \rightarrow \) f a = b.
lemma fun-upd-equivp-iff-part1:
  equivp R \Rightarrow (\forall z. R x z \Rightarrow R (f z) y) \Rightarrow R (fun-upd-equivp R f x y z) (f z)
by (auto simp: fun-upd-equivp-def Equiv-Relations.equivp-reflp Equiv-Relations.equivp-symp)

lemma fun-upd-equivp-iff-part2: equivp R \Rightarrow fun-upd-equivp R f x y y = f \rightarrow f x = y
apply (simp add :fun-upd-equivp-def, safe)
apply (erule subst, auto simp: Equiv-Relations.equivp-reflp)
done


lemma equivp R \Rightarrow (\forall z. R x z \Rightarrow R (fun-upd-equivp R f x y z) (f z)) \Rightarrow R (\forall f y) (f x)
by (simp add: fun-upd-equivp-def Equiv-Relations.equivp-reflp Equiv-Relations.equivp-symp)

this lemma is the same in [equivp ?R; \forall z. ?R \forall x \Rightarrow \forall R (?f z) ?y] \Rightarrow ?R (fun-upd-equivp ?R ?f ?x ?y ?z) (?f ?z) where op = is generalized by another equivalence relation

lemma fun-upd-equivp-idem: f x = y \Rightarrow (fun-upd-equivp (op =) f x y) = f
by (simp only: fun-upd-equivp-def)

lemma fun-upd-equivp-triv : fun-upd-equivp (op =) f x (f x) = f
by (simp only: fun-upd-equivp-def)

— This is the generalization of fun-upd-equivp op = ?f ?x ?y = ?f on a given equivalence relation

lemma fun-upd-equivp-triv-part1 :
  equivp R \Rightarrow (\forall z. R x z \Rightarrow fun-upd-equivp (R') f x (f z) x) \Rightarrow f x
apply (auto simp:fun-upd-equivp-def)
apply (metis equivp-reflp)
done

lemma fun-upd-equivp-triv-part2 :
  equivp R \Rightarrow (\forall z. R x z \Rightarrow f z) \Rightarrow fun-upd-equivp (R') f x (f x) x
by (simp add:fun-upd-equivp-def simp: split-if)

lemma fun-upd-equivp-apply [simp]:
  (fun-upd-equivp (op =) f x y) z = (if z = x then y else f z)
by (simp only: fun-upd-equivp-def)

— This is the generalization of fun-upd-equivp op = ?f ?x ?y ?z = (if ?z = ?x then ?y else ?f ?z) with e given equivalence relation and not only with op =

lemma fun-upd-equivp-apply1 [simp]:
  equivp R \Rightarrow (fun-upd-equivp R f x y) z = (if R x x then y else f z)
by (simp add: fun-upd-equivp-def)

lemma fun-upd-equivp-same: (fun-upd-equivp (op =) f x y) x = y
by (simp only: fun-upd-equivp-def)
This is the generalization of `fun-upd-equivp op = ?f ?x ?y ?x = ?y` with a given equivalence relation.

**Lemma** `fun-upd-equivp-same1`: `equivp R ==> (fun-upd-equivp R f x y) x = y`  
by (simp add: `fun-upd-equivp-def equivp-reflp`)

For the special case that `term eq` is just the equality `term "op ="`, sharing update and classical update are identical.

**Lemma** `fun-upd-equivp-vs-fun-upd`: `(fun-upd-equivp (op =)) = fun-upd`  
by (rule ext, rule ext, rule ext,simp add:fun-upd-def fun-upd-equivp-def)

**Definition of the shared-memory type** `typedef` `(α, β) memory = {σ :: ′α ⇀ ′β, R}.`  
equivp R ∧ (∀ x y. R x y −→ σ x = σ y)`

**Proof**
- show `(Map.empty, (op =)) ∈ ?memory`  
  by (auto simp: `identity-equivp`)

**Qed**

**Definition** `memory-inv :: (′a ⇒ ′b option) × (′a ⇒ ′a ⇒ bool) ⇒ bool`  
where `memory-inv (Pair f R) = (equivp R ∧ (∀ x y. R x y −→ f x = f y))`

**Lemma** `Abs-Rep-memory [simp]: Abs-memory (Rep-memory σ) = σ`  
by (simp add: `Rep-memory-inverse`)

**Lemma** `memory-invariant [simp]:`  
`memory-inv σ-rep = (Rep-memory (Abs-memory σ-rep) = σ-rep)`  
by `smt`

**Lemma** `Pair-code-eq`:  
`Rep-memory σ = Pair (fst (Rep-memory σ)) (snd (Rep-memory σ))`  
by (simp add: `Product-Type.surjective-pairing`)

**Lemma** `snd-memory-eqv [simp]: equivp(snd(Rep-memory σ))`  
by (insert `Rep-memory [of σ]`, auto)

**Operations on Shared-Memory**  
**Definition** `init :: (′a, β) memory`  
where  
`init = Abs-memory (Map.empty, op =)`

**Value** `init::(nat,int)memory`  
**Value** `map (λx. the (fst (Rep-memory init)x)) [1 .. 10]`  
**Value** `take (10) (map (Pair Map.empty) [(op =) ])`

**Value** `replicate 10 init`  
**Term** `Rep-memory σ`  
**Term** `[(σ::nat ⇒ int, R) <−> xs . equivp R ∧ (∀ x y. R x y −→ σ x = σ y)]`

**Definition** `init-mem-list :: ′α list ⇒ (nat, ′a) memory`  
where  
`init-mem-list s = Abs-memory (let h = zip (map nat [0 .. int(length s)]) s)`
in foldl (λx (y,z). fun-upd x y (Some z))
         Map.empty h,
         op =)

value init-mem-list [−22, 2, −3]

Memory Read Operation  definition lookup :: (α, β) memory ⇒ α ⇒ β (infixl $ 100)
where  σ $ x = the (fst (Rep-memory σ) x)

setup-lifting type-definition-memory

Memory Update Operation  fun Pair-upd-lifter:: (α ⇒ β option) × (α ⇒ α ⇒ bool) ⇒
                   (α ⇒ β option) × (α ⇒ α ⇒ bool)
where  Pair-upd-lifter (f, R) x y = (fun-upd-equivp R f x (Some y), R)

lemma update-sound'::
  assumes σ ∈{(σ, R). equivp R ∧ (∀x y. R x y → σ x = σ y)}
  shows  Pair-upd-lifter σ x y ∈{(σ, R). equivp R ∧ (∀x y. R x y → σ x = σ y)}
proof –
  obtain mem and R
  where Pair: (mem, R) = σ and Eq: equivp R and Mem: ∀ x y . R x y → mem x = mem y
       using assms equivpE by auto
  obtain mem' and R'
    where Pair': (mem', R') = Pair-upd-lifter σ x y
          using surjective-pairing by metis
  have Def1: mem' = fun-upd-equivp R mem x (Some y)
    and Def2: R' = R
          using Pair Pair' by auto
  have Eq': equivp R'
    using Def2 Eq by auto
  moreover have ∀ y z . R' y z → mem' y = mem' z
    using Mem equivp-symp equivp-transp
    unfolding Def1 Def2 by (metis Eq fun-upd-equivp-def)
  ultimately show ?thesis
    using Pair' by auto
qed

lift-definition update :: (α, β) memory ⇒ α ⇒ β ⇒ (α, β) memory (- (- := $ -') 100)
is Pair-upd-lifter
using update-sound'
by simp

lemma update': σ (x := y) = Abs-memory (fun-upd-equivp (snd (Rep-memory σ))
        (fst (Rep-memory σ)) x (Some y), (snd (Rep-memory σ)))
using Rep-memory-inverse surjective-pairing Pair-upd-lifter.simps update.rep-eq
by metis

fun update-list-rep :: (α ⇒ β) × (α ⇒ α ⇒ bool) ⇒ (α × β )list ⇒
\((\alpha \to '\beta) \times (\alpha \Rightarrow '\alpha \Rightarrow \text{bool})\)

**where** update-list-rep \((f, R)\ nlist = (\text{foldl} \ (\lambda(f, R)(\text{addr},\text{val})\ (f, R)\ nlist)\)

\[
\text{lemma update-list-rep-p:}
\]
assumes 1: \(P\ \sigma\)
and 2: \(\forall \ src\ dst\ \sigma.\ P\ \sigma\ \Rightarrow\ P\ (\text{Pair-upd-lifter}\ \sigma\ src\ dst)\)
shows \(P\ (\text{update-list-rep}\ \sigma\ \text{list})\)
using 1 2
apply (induct list arbitrary: \(\sigma\))
apply (force, safe)
apply (simp del: Pair-upd-lifter.simps)
using surjective-pairing
apply mesi
done

**lemma update-list-rep-sound:**
assumes 1: \(\sigma\ \in\ \{(\sigma, R).\ \text{equivp} R\ \land\ (\forall x y.\ R\ x\ y\ \rightarrow\ \sigma\ x = \sigma\ y)\}\)
shows \(\text{update-list-rep}\ \sigma\ (\text{nlist})\ \in\ \{(\sigma, R).\ \text{equivp} R\ \land\ (\forall x y.\ R\ x\ y\ \rightarrow\ \sigma\ x = \sigma\ y)\}\)
using 1
apply (elim update-list-rep-p)
apply (erule update-sound')
done

**lift-definition** update-list : \((\alpha, '\beta)\ \text{memory} \Rightarrow (\alpha \times '\beta)\ \text{list} \Rightarrow (\alpha, '\beta)\ \text{memory}\ (**infixed** '/:= 30)
is update-list-rep
using update-list-rep-sound by simp

**lemma update-list-Nil[simp]:** \((\sigma /:= []) = \sigma\)
unfolding update-list-def
by(simp, subst surjective-pairing[of Rep-memory \(\sigma\)], subst update-list-rep.simps, simp)

**lemma update-list-Cons[simp]:** \((\sigma /:= ((a, b)#S)) = (\sigma(a :=_S b) /:= S)\)
unfolding update-list-def
apply(simp, subst surjective-pairing[of Rep-memory \(\sigma\)], subst update-list-rep.simps, simp)
apply(subst surjective-pairing[of Rep-memory \(\sigma (a :=_S b)\)], subst update-list-rep.simps, simp)
apply(simp add: update-def)
apply(subst Abs-memory-inverse)
apply (metis (lifting, mono-tags) Rep-memory update-sound')
by simp

**Type-invariant:**

**lemma update_sound:**
assumes Rep-memory \(\sigma = (\sigma', \text{eq})\)
shows \((\text{fun-upd-equiv} \ \text{eq} \ \sigma'\ x\ (\text{Some} y), \text{eq})\ \in\ \{(\sigma, R).\ \text{equivp} R\ \land\ (\forall x y.\ R\ x\ y\ \rightarrow\ \sigma\ x = \sigma\ y)\}\)
using assms insert Rep-memory[of \(\sigma\)]
apply(auto simp: fun-upd-equiv-def)
apply(rename-tac xa xb, erule contrapos-np)
apply(rule-tac \(R = \text{eq and y = xa in equivp-transp,simp})
apply(erule equivp-symp, simp-all)
apply(rename-tac xa xb, erule contrapos-np)
apply(rule-tac R=eq and y=xb in equivp-transp,simp-all)
done

Memory Transfer Based on Sharing Transformation  

fun transfer-rep :: ('α -> 'β) 
\times ('α\Rightarrow\alpha \Rightarrow \text{bool}) \Rightarrow \alpha \Rightarrow ('α\Rightarrow'β) 
\times ('α\Rightarrow\alpha \Rightarrow \text{bool})

where transfer-rep (m, r) src dst = (m o (id (dst := src)), 
(\lambda x y . r ((id (dst := src)) x) ((id (dst := src)) y)))

lemma transfer-rep-simp :
transfer-rep X src dst = ((fst X) o (id (dst := src)), 
(\lambda x y . (snd X) ((id (dst := src)) x) ((id (dst := src)) y)))
by(subst surjective-pairing[of X],subst transfer-rep,simps,simp)

lemma transfer-rep-sound:
assumes \sigma \in \{\sigma, R). equivp R \land (\forall x y. R x y \rightarrow \sigma x = \sigma y}\}
supports transfer-rep \sigma src dst \in \{\sigma, R). equivp R \land (\forall x y. R x y \rightarrow \sigma x = \sigma y}\}
proof
obtain mem and R 
where P: (mem, R) = \sigma and E: equivp R and M: \forall x y . R x y \rightarrow mem x = mem y
using assms equivpE by auto
obtain mem' and R'
where P': (mem', R') = transfer-rep \sigma src dst
by (metis surj-pair)
have D1: mem' = (mem o (id (dst := src)))
and D2: R' = (\lambda x y . R ((id (dst := src)) x) ((id (dst := src)) y))
using P P' by auto
have equivp R'
using E unfolding D2 equivp-def by metis
moreover have \forall y z . R' y z \rightarrow mem' y = mem' z
using M unfolding D1 D2 by auto
ultimately show \?thesis
using P' by auto
qed

lift-definition transfer :: ('\alpha,'\beta)memory \Rightarrow '\alpha \Rightarrow ('\alpha, '\beta)memory (- ('\cdot \cdot \cdot') [0,111,111]110)
is transfer-rep
using transfer-rep-sound
by simp

lemma transfer-rep-sound2 :
transfer-rep (Rep-memory \sigma) a b \in \{\sigma, R). equivp R \land (\forall x y. R x y \rightarrow \sigma x = \sigma y}\}
by (metis (lifting, mono-tags) Rep-memory transfer-rep-sound)
fun share-list2 :: ('α, 'β) memory ⇒ ('α × 'α) list ⇒ ('α, 'β) memory (infix ‘/\ 60)
where σ /\ S = (foldl (λ σ (a, b), (σ (a×b)))) σ S

lemma sharelist2-Nil[simp] : σ /\ [] = σ by simp

lemma sharelist2-Cons[simp] : σ /\ ((a,b)#S) = (σ(a×b) /\ S) by simp

fun share-list-rep :: ('α ⇀ 'β) × ('α ⇒ 'α ⇒ bool) ⇒ ('α × 'α) list ⇒ ('α ⇀ 'β) × ('α ⇒ 'α ⇒ bool)
where share-list-rep (f, R) nlist = (foldl (λ (f, R) (src, dst). transfer-rep (f, R src dst)) (f, R) nlist)

fun share-list-rep' :: ('α ⇒ 'β) ⇒ ('α ⇒ 'α ⇒ bool) ⇒ ('α × 'α) list ⇒ ('α ⇒ 'β) ⇒ ('α ⇒ bool)
where share-list-rep' (f, R) [] = (f, R)
     share-list-rep' (f, R) (n#nlist) = share-list-rep' (transfer-rep(f,R)(fst n)(snd n)) nlist

lemma share-list-rep'-p:
  assumes 1: P σ
  and  2: \!\!σ = (f mem y z) mem list
  shows P (share-list-rep' σ list)
  using 1 2
  apply (induct list arbitrary: σ P)
  apply force
  apply simp del: transfer-rep.simps
  using surjective-pairing
  apply force
  done

lemma foldl-preserve-p:
  assumes 1: P mem
  and  2: \!\!y z mem . P mem ⇒ P (f mem y z)
  shows P (foldl (λa (y, z), f mem y z) mem list)
  using 1 2
  apply (induct list arbitrary: f mem , auto)
  apply force
  done

lemma share-list-rep-p:
  assumes 1: P σ
  and  2: \!\!σ = (f mem y z) mem list
  shows P (share-list-rep σ list)
  using 1 2
  apply (induct list arbitrary: σ)
  apply force
  apply simp del: transfer-rep.simps
using surjective-pairing
apply metis
done

The modification of the underlying equivalence relation on addresses is only defined on very strong conditions — which are fulfilled for the empty memory, but difficult to establish on a non-empty one. And of course, the given relation must be proven to be an equivalence relation. So, the case is geared towards shared-memory scenarios where the sharing is defined initially once and for all.

definition update_R :: ('α, 'β)memory ⇒ ('α ⇒ 'α ⇒ bool) ⇒ ('α, 'β)memory (σ := R - 100)
where σ := R σ ≡ Abs-memory (fst(Rep-memory σ), R)
definition lookup_R :: ('α, 'β)memory ⇒ ('α ⇒ 'α ⇒ bool) ($R - 100)
where $R σ ≡ (snd(Rep-memory σ))

Sharing Relation Definition

lemma update_R-comp-lookup_R:
assumes equiv : equiv σ
and sharing-conform : ∀ x y. R x y ⇒ fst(Rep-memory σ) x = fst(Rep-memory σ) y
shows ($R (σ := R)) = R
unfolding lookup_R-def update_R-def
by(auto)

Properties on Sharing Relation

lemma sharing-charn:
equivp (snd (Rep-memory σ))
using Rep-memory[of σ]
unfolding sharing-def
by auto

lemma sharing-charn':
assumes 1: (x shares_σ y)
shows (∃R. equiv R ∧ R x y)
by (auto simp add: sharing-def snd-def equivp-def)

lemma sharing-charn2:
shows ∃ x y. (equivp (snd (Rep-memory σ)) ∧ (snd (Rep-memory σ)) x y)
using sharing-charn [THEN equivp-reflp]
by (simp)fast

— Lemma to show that (?x shares_σ ?y ≡ snd (Rep-memory ?σ) ?x ?y is reflexive
lemma sharing-refl: (x shares_σ x)
using insert Rep-memory[of σ]
— Lemma to show that \(?x \text{ shares} \sigma \ ?y \equiv \text{snd}(\text{Rep-memory} \sigma)\) \(?x \ ?y\) is symmetric

\begin{verbatim}
lemma sharing-sym [sym]
  assumes x shares_\sigma y
  shows y shares_\sigma x
  using assms Rep-memory[of \sigma]
  by (auto simp: sharing-def elim: equivp-symp)
\end{verbatim}

— Lemma to show that \(?x \text{ shares} \sigma \ ?y \equiv \text{snd}(\text{Rep-memory} \sigma)\) \(?x \ ?y\) is transitive

\begin{verbatim}
lemma sharing-commute : x shares_\sigma y = (y shares_\sigma x)
  by(auto intro: sharing-sym)
\end{verbatim}

— Lemma to show that \(?x \text{ shares} \sigma \ ?y \equiv \text{snd}(\text{Rep-memory} \sigma)\) \(?x \ ?y\) is symmetric

\begin{verbatim}
lemma sharing-trans [trans]
  assumes x shares_\sigma y
  and y shares_\sigma z
  shows x shares_\sigma z
  using assms insert Rep-memory[of \sigma]
  by(auto simp: sharing-def elim: equivp-transp)
\end{verbatim}

\begin{verbatim}
lemma shares-result:
  assumes x shares_\sigma y
  shows \text{fst}(\text{Rep-memory} \sigma) \ x = \text{fst}(\text{Rep-memory} \sigma) \ y
  using assms
  unfolding sharing-def
  using Rep-memory[of \sigma]
  by auto
\end{verbatim}

\begin{verbatim}
lemma sharing-init:
  assumes 1: i \neq k
  shows \neg(i \text{ shares-init} k)
  unfolding sharing-def init-def
  using 1
  by (auto simp: Abs-memory-inverse identity-equivp)
\end{verbatim}

\begin{verbatim}
lemma shares-init[simp]: (x \text{ shares-init} y) = (x=y)
  unfolding sharing-def init-def
  by (metis init-def sharing-init sharing-def sharing-refl)
\end{verbatim}

\begin{verbatim}
lemma sharing-init-mem-list:
  assumes 1: i \neq k
  shows \neg(i \text{ shares-init-mem-list} S \ k)
  unfolding sharing-def init-mem-list-def
  using 1
  by (auto simp: Abs-memory-inverse identity-equivp)
\end{verbatim}
**Definition** reset :: ('α, 'β) memory ⇒ 'α set ⇒ ('α, 'β) memory (- (reset -) 100)

where

\[ \sigma (\text{reset } X) = (\text{let } (\sigma', eq) = \text{Rep-memory } \sigma; \]
\[ eq' = \lambda a b. eq a b \lor (\exists x \in X. eq a x \lor eq b x) \]
\[ \text{in } \begin{cases} \sigma & \text{if } X = \{\} \text{ then } \sigma \\ \text{else Abs-memory (fun-upd-equivp eq'} & \sigma' (\text{SOME } x \in X) \text{ None, eq}) \end{cases} \]

**Lemma** reset-mt : σ (reset {}) = σ

**Unfolding** reset-def Let-def
by simp

**Lemma** reset-sh :
assumes * : (x shares_σ y)
and **: x ∈ X
shows σ (reset X) $ y = None

**Oops**

**Memory Domain Definition**

**Definition** Domain :: ('α, 'β) memory ⇒ 'α set

where

\[ \text{Domain } \sigma = \text{dom (fst (Rep-memory } \sigma)) \]

**Properties on Memory Domain**

**Lemma** Domain-charn :

assumes 1: x ∈ Domain σ
shows \( \exists y. \) Some y = fst (Rep-memory σ) x
using 1
by (auto simp: Domain-def)

**Lemma** Domain-charn1 :

assumes 1: x ∈ Domain σ
shows \( \exists y. \) the (Some y) = σ $ x
using 1
by (auto simp: Domain-def lookup-def)

— This lemma says that if x and y are equivalent this means that they are in the same set of equivalent classes

**Lemma** shares-dom [code-unfold, intro]:

assumes x shares_σ y
shows \( x \in \text{Domain } \sigma \) = \( y \in \text{Domain } \sigma \)
using insert Rep-memory[of σ] assms
by (auto simp: sharing-def Domain-def)

**Lemma** Domain-mono :

assumes 1: x ∈ Domain σ
and 2: (x shares_σ y)
shows y ∈ Domain σ
using 1 2 Rep-memory[of σ]
by (auto simp add: sharing-def Domain-def )

**Corollary** Domain-nonshares :

assumes 1: x ∈ Domain σ
and 2: $y \notin \text{Domain } \sigma$
shows $\neg (x \text{ shares}_\sigma y)$
using 1 2 Domain-mono
by (fast)

lemma Domain-init[simp]: Domain init = {}
unfolding init-def Domain-def
by (simp-all add: identity-equivp Abs-memory-inverse)

lemma Domain-update[simp]: Domain $(\sigma(x := y)) = (\text{Domain } \sigma) \cup \{y . y \text{ shares}_\sigma x\}$
unfolding update-def Domain-def sharing-def
proof (simp-all)

have *: Pair-upd-lifter (Rep-memory $\sigma$) $x y \in \{(\sigma, R). \text{ equivp } R \land (\forall x y. R \ x \ y \rightarrow \sigma \ x = \sigma \ y)\}$
by (simp, metis (lifting, mono-tags) Rep-memory mem-Collect-eq update-sound)

have **: snd (Rep-memory $\sigma$) $x x$
by (metis equivp-reflp sharing-charn2)

show dom (fst (Rep-memory (Abs-memory (Pair-upd-lifter (Rep-memory $\sigma$) $x y$)))) =
     dom (fst (Rep-memory $\sigma$)) \cup \{y. snd (Rep-memory $\sigma$) $y x\}$
apply (simp-all add: Abs-memory-inverse[OF *])
apply (subt surjective-pairing[of (Rep-memory $\sigma$)])
apply (subt Pair-upd-lifter.simps, simp)
apply (auto simp: ** fun-upd-equivp-def)
done
qed

lemma Domain-share1:
assumes 1: $a \in \text{Domain } \sigma$
and 2: $b \in \text{Domain } \sigma$
shows $\text{Domain } (\sigma(a \ast b)) = \text{Domain } \sigma$
proof (simp-all add: Set.set-eq-iff, tactic ALLGOALS (rtac @{thm allI}))
fix $x$

have ***: transfer-rep (Rep-memory $\sigma$) (id $a$) (id $b$) \in \{(\sigma, R). \text{ equivp } R \land (\forall x y. R \ x \ y \rightarrow \sigma \ x = \sigma \ y)\}$
by (metis (lifting, mono-tags) Rep-memory transfer-rep-sound)

show $(x \in \text{Domain } (\sigma(a \ast b))) = (x \in \text{Domain } \sigma)$
unfolding sharing-def Domain-def transfer-def map-fun-def o-def
apply (subt Abs-memory-inverse[OF ***])
apply (insert 1 2, simp add: o-def transfer-rep-simp Domain-def)
done
qed

lemma Domain-share-tgt: $a \in \text{Domain } \sigma \Rightarrow b \in \text{Domain } (\sigma(a \ast b))$
unfolding sharing-def Domain-def transfer-def map-fun-def o-def id-def
apply (subt Abs-memory-inverse[OF transfer-rep-sound2])
unfolding sharing-def Domain-def transfer-def map-fun-def o-def id-def
apply (simp add: o-def transfer-rep-simp Domain-def)

58
lemma Domain-share2 :
assumes 1 : a ∈ Domain σ
and 2 : b ∉ Domain σ
shows  Domain (σ(a×b)) = (Domain σ - {x, x sharesσ b} ∪ {b})
proof(simp-all add: Set.set-eq-iff, auto)

fix x
assume 3 : x ∈ SharedMemory.Domin (σ (a × b))
and 4 : x ≠ b
show x ∈ SharedMemory.Domin σ
  apply(insert 3 4)
  unfolding sharing-def Domain-def transfer-def map-fun-def o-def id-def
  apply(subst (asm) Abs-memory-inverse[OF transfer-rep-sound2])
  apply(insert 1, simp add: o-def transfer-rep-simp Domain-def)
  apply(auto split: split-if split-if-asm)
  done

next
fix x
assume 3 : x ∈ Domain (σ (a × b))
and 4 : x ≠ b
and 5 : x sharesσ b
have ** : x ∉ Domain σ using 2 5 Domain-mono by (fast)
show False
  apply(insert 3 4 5, erule contrapos-pp, simp)
  unfolding sharing-def Domain-def transfer-def map-fun-def o-def id-def
  apply(subst Abs-memory-inverse[OF transfer-rep-sound2])
  apply(insert 1, simp add: o-def transfer-rep-simp Domain-def)
  apply(auto split: split-if split-if-asm)
  using ** SharedMemory.Domin-def domI apply fast
  done

next
show b ∈ Domain (σ (a × b))
  using 1 Domain-share-tgt by fast

next
fix x
assume 3 : x ∈ Domain σ
and 4 : ¬ x sharesσ b
show x ∈ Domain (σ (a × b))
  unfolding sharing-def Domain-def transfer-def map-fun-def o-def id-def
  apply(subst Abs-memory-inverse[OF transfer-rep-sound2])
  apply(insert 1, simp add: o-def transfer-rep-simp Domain-def)
  apply(auto split: split-if split-if-asm)
  using 3 SharedMemory.Domin-def domD
  apply fast
  done

qed
lemma Domain-share3:
  assumes 1: a $\notin$ Domain $\sigma$
  shows Domain ($\sigma(a \times b)$) = (Domain $\sigma$ - {b})
proof(simp-all add:Set.set-eq-iff, auto)
  fix x
  assume 3: x $\in$ Domain ($\sigma (a \times b)$)
  show x $\in$ Domain $\sigma$
    apply(insert 3)
    unfolding sharing-def Domain-def transfer-def map-fun-def o-def id-def
    apply(subst (asm) Abs-memory-inverse[OF transfer-rep-sound2])
    apply(insert 1 , simp add: o-def transfer-rep-simp Domain-def )
    apply(auto split: split-if split-if-asm )
  done
next
  assume 3: b $\in$ Domain ($\sigma (a \times b)$)
  show False
    apply(insert 1 3)
    apply(erule contrapos-pp[of b $\in$ SharedMemory.Domain ($\sigma (a \times b)$)], simp)
    unfolding sharing-def Domain-def transfer-def map-fun-def o-def id-def
    apply(subst Abs-memory-inverse[OF transfer-rep-sound2])
    apply(insert 1 , simp add: o-def transfer-rep-simp Domain-def )
    apply(auto split: split-if )
  done
next
  fix x
  assume 3: x $\in$ Domain $\sigma$
  and 4: x $\neq$ b
  show x $\in$ Domain ($\sigma (a \times b)$)
    apply(insert 3 4)
    unfolding sharing-def Domain-def transfer-def map-fun-def o-def id-def
    apply(subst Abs-memory-inverse[OF transfer-rep-sound2])
    apply(insert 1 , simp add: o-def transfer-rep-simp Domain-def )
    apply(auto split: split-if split-if-asm )
  done
qed

lemma Domain-transfer :
Domain ($\sigma(a \times b)$) = (if a $\notin$ Domain $\sigma$
then (Domain $\sigma$ - {b})
else if b $\notin$ Domain $\sigma$
then (Domain $\sigma$ - {x. x shares$_\sigma$ b} $\cup$ {b})
else Domain $\sigma$ )
using Domain-share1 Domain-share2 Domain-share3
by metis

lemma Domain-transfer-approx : Domain ($\sigma(a \times b)$) $\subseteq$ Domain ($\sigma$) $\cup$ {b}
by(auto simp: Domain-transfer)

Sharing Relation and Memory Update  lemma sharing-upd: x shares$_\sigma$ (a :=$_\sigma$ b) $\Longleftrightarrow$ x shares$_\sigma$ y
using insert Rep-memory[of σ]
by(auto simp: sharing-def update-def Abs-memory-inverse[OF update_sound])

— this lemma says that if we do an update on an address \( x \) all the elements that are equivalent of \( x \) are updated

\textbf{lemma} \( \text{update}'' \):

\[
\sigma (x := y) = \text{Abs-memory}(\text{fun-upd-equivp (\& x y. x shares}_{\sigma} y) \ (\text{fst (Rep-memory \sigma)}) \ x \ (\text{Some y}),
\]

\[
\text{snd (Rep-memory \sigma)}
\]

\textbf{unfolding} \text{update-def} \text{sharing-def}
\by\text{metis update'' update-def}

\textbf{theorem} \text{update-cancel}:
\textbf{assumes} \( x \text{ shares}_{\sigma} x' \)
\textbf{shows} \( \sigma(x := y)(x' := z) = (\sigma(x' := z)) \)
\textbf{proof –}
\textbf{have} \( * :\ (\text{fun-upd-equivp (\text{snd(Rep-memory \sigma)}) (\text{fst(Rep-memory \sigma)}) \ x \ (\text{Some y}),}\text{snd (Rep-memory \sigma)}) \in \{ (\sigma, R). \text{equivp R \& (\forall x y. R x y \rightarrow \sigma x = \sigma y) } \} \)
\textbf{unfolding} \text{fun-upd-equivp-def}
\by\text{rule update_sound[simplified fun-upd-equivp-def], simp}
\textbf{have} \( ** : \bigwedge R \sigma. \text{equivp R = \rightarrow \neg R x x' } \rightarrow \text{fun-upd-equivp R (fun-upd-equivp R x (Some y)) x' (Some z) = fun-upd-equivp R x (Some y)} \)
\textbf{unfolding} \text{fun-upd-equivp-def}
\apply\text{rule ext}
\apply\text{auto}
\apply\text{erule contrapos-np, erule equivp-transp, simp-all}
\done
\textbf{show} \ ?thesis
\apply\text{simp add: update''}
\apply\text{insert sharing-charn assms[simplified sharing-def]}
\apply\text{simp add: Abs-memory-inverse [OF *] **}
\done

\textbf{qed}

\textbf{theorem} \text{update-commute}:
\textbf{assumes} \( \sim 1: (x \text{ shares}_{\sigma} x') \)
\textbf{shows} \( (\sigma(x := y))(x' := z)(x := y)) \)
\textbf{proof –}
\textbf{have} \( * : \bigwedge x y. (\text{fun-upd-equivp (\text{snd(Rep-memory \sigma)}) (\text{fst(Rep-memory \sigma)}) \ x \ (\text{Some y}),}\text{snd (Rep-memory \sigma)}) \in \{ (\sigma, R). \text{equivp R \& (\forall x y. R x y \rightarrow \sigma x = \sigma y) } \} \)
\textbf{unfolding} \text{fun-upd-equivp-def}
\by\text{rule update_sound[simplified fun-upd-equivp-def], simp}
\textbf{have} \( ** : \bigwedge R \sigma. \text{equivp R = \rightarrow \neg R x x' } \rightarrow \text{fun-upd-equivp R (fun-upd-equivp R x (Some y)) x' (Some z) = fun-upd-equivp R x (Some y)} \)
\textbf{unfolding} \text{fun-upd-equivp-def}
\apply\text{rule ext}

61
apply (case-tac R xa x', auto)
apply (erule contrapos-np)
apply (frule equivp-transp, simp-all)
apply (erule equivp-symp, simp-all)
done

show ?thesis
apply (simp add: update')
apply (insert assms[simplified sharing-def])
apply (simp add: Abs-memory-inverse [OF *] **) 
done

qed

Properties on lookup and update wrt the Sharing Relation

lemma update-triv:
assumes 1: \( x \text{ shares}_\sigma y \)
and 2: \( y \in \text{Domain } \sigma \)
shows \( \sigma (x :=_s (\sigma \lhd y)) = \sigma \)

proof -
{
  fix z
  assume zx: z shares_\sigma x
  then have zy: z shares_\sigma y
    using 1 by (rule sharing-trans)
  have F: \( y \in \text{Domain } \sigma \implies x \text{ shares}_\sigma y \)
    by (auto simp: Domain-def dest: shares-result)
  have Some (the (fst (Rep-memory \sigma x))) = fst (Rep-memory \sigma y)
    by (auto simp: Domain-def dest: shares-result)
  have Some (the (fst (Rep-memory \sigma y))) = fst (Rep-memory \sigma z)
    using zx and shares-result [OF zy] shares-result [OF zx]
  have F [OF 2 1]
    by simp
}

note 3 = this

show ?thesis
  unfolding update'' lookup-def fun-upd-equivp-def
  by (simp add: 3 Rep-memory-inverse if-cong)

qed

lemma update-idem':
assumes 1: \( x \text{ shares}_\sigma y \)
and 2: \( x \in \text{Domain } \sigma \)
and 3: \( \sigma \lhd x = z \)
shows \( \sigma(y:=_s z) = \sigma \)

proof -
  have *: \( y \in \text{Domain } \sigma \)
    by (simp add: shares-dom[OF 1, symmetric] 2)
  have **: \( \sigma (x :=_s (\sigma \lhd y)) = \sigma \)
    using 1 2 *
    by (simp add: update-triv)
  also have \( (\sigma \lhd y) = \sigma \lhd x \)
    by (simp only: lookup-def shares-result [OF 1])
  finally show ?thesis
    using 1 2 3 sharing-sym update-triv
    by fast
lemma update-idem:
  assumes 2: \( x \in \text{Domain} \sigma \)
  and 3: \( \sigma \$ x = z \)
  shows \( \sigma(x :=_\$ z) = \sigma \)
proof
  show \(?thesis\)
  using 2 3 sharing-refl update-triv
  by fast
qed

lemma update-apply: \((\sigma(x :=_\$ y)) \$ z = (if z \text{shares}_\sigma x \text{ then } y \text{ else } \sigma \$ z)\)
proof
  have \( \ast: (\lambda z. \text{if } z \text{ shares}_\sigma x \text{ then } \text{Some } y \text{ else } \text{fst} (\text{Rep-memory } \sigma) z, \text{snd} (\text{Rep-memory } \sigma)) \in \{(\sigma, R). \text{equivp } R \land (\forall x y. R x y \rightarrow \sigma x = \sigma y)\}\)
    unfolding sharing-def
    by(rule update\_sound[\text{simplified fun-upd-equivp-def}], simp)
  show \(?thesis\)
  proof (cases z \text{shares}_\sigma x)
    case True
    assume A: \( z \text{ shares}_\sigma x \)
    show \( \sigma(x :=_\$ y) \$ z = (if z \text{shares}_\sigma x \text{ then } y \text{ else } \sigma \$ z) \)
      unfolding update\_\prime\prime lookup-def fun-upd-equivp-def
      by(simp add: Abs-memory-inverse \( \{OF \ast\}\))
  next
    case False
    assume A: \( \neg z \text{ shares}_\sigma x \)
    show \( \sigma(x :=_\$ y) \$ z = (if z \text{shares}_\sigma x \text{ then } y \text{ else } \sigma \$ z) \)
      unfolding update\_\prime\prime lookup-def fun-upd-equivp-def
      by(simp add: Abs-memory-inverse \( \{OF \ast\}\))
  qed
qed

lemma update-share:
  assumes \( z \text{ shares}_\sigma x \)
  shows \( \sigma(x :=_\$ a) \$ z = a \)
  using assms
  by (simp only: update-apply if-True)

lemma update-other:
  assumes \( \neg(z \text{ shares}_\sigma x) \)
  shows \( \sigma(x :=_\$ a) \$ z = \sigma \$ z \)
  using assms
  by (simp only: update-apply if-False)

lemma lookup-update-rep:
  assumes 1: \( (\text{snd} (\text{Rep-memory } \sigma')) x y \)
  shows \( (\text{fst} (\text{Pair-upd-lifter} (\text{Rep-memory } \sigma') \text{ src dst})) x = (\text{fst} (\text{Pair-upd-lifter} (\text{Rep-memory } \sigma') \text{ src dst})) y \)
  using 1 \text{shares-result} sharing-def sharing-upd update\_rep-eq
  by (metis \text{hide-lams, no-types})
lemma lookup-update-rep":
assumes 1: x shares σ y
shows (σ (src :=ₜ dst)) $ x = (σ (src :=ₜ dst)) $ y
using 1 lookup-def lookup-update-rep sharing-def update rep-eq
by metis

theorem memory-ext :
assumes * : ∀ x y. (x shares σ y) = (x shares σ' y)
and ** : Domain σ = Domain σ'
and *** : ∀ x. σ $ x = σ' $ x
shows σ = σ'
apply (subst Rep-memory-inverse [symmetric])
apply (subst (3) Rep-memory-inverse [symmetric])
apply (rule arg-cong [of - Abs-memory])
apply (auto simp: Product-Type prod-eq-iff)
proof –
  show fst (Rep-memory σ) = fst (Rep-memory σ')
    apply (rule ext, insert ** ** *, simp add: SharedMemory.lookup-def Domain-def)
    apply (metis (lifting, no-types) domD domIff the.simps)
    done
next
  show snd (Rep-memory σ) = snd (Rep-memory σ')
    by (rule ext, rule ext, insert *, simp add: sharing-def)
qed

Nice connection between sharing relation, domain of the memory and content equality on
the one hand and equality on the other; this proves that our memory model is fully abstract
in these three operations.

corollary memory-ext2: (σ = σ') = ((∀ x y. (x shares σ y) = (x shares σ' y))
  ∧ Domain σ = Domain σ'
  ∧ (∀ x. σ $ x = σ' $ x))
by (auto intro: memory-ext)

Rules On Sharing and Memory Transfer  lemma transfer-rep-inv-E:
assumes 1: σ ∈ {σ, R}. equiv R ∧ (∀ x y. R x y → σ x = σ y)
and 2: memory-inv (transfer-rep σ src dst) ⇒ Q
shows Q
using assms transfer-rep-sound[of σ]
by (auto simp: Abs-memory-inverse)

lemma transfer-rep-fst1:
assumes 1: σ = fst (transfer-rep (Rep-memory σ') src dst)
shows ∀ x. x = dst ⇒ σ x = (fst (Rep-memory σ')) src
using 1 unfolding transfer-rep-simp
by simp

lemma transfer-rep-fst2:
assumes 1: $\sigma = \text{fst}(\text{transfer-rep} (\text{Rep-memory} \ \sigma') \ src \ dst)$
shows $\forall x. \ x \neq \ dst \implies \sigma \ x = (\text{fst} (\text{Rep-memory} \ \sigma')) (\text{id} \ x)$
using 1 unfolding transfer-rep-simp
by simp

lemma lookup-transfer-rep':
  $(\text{fst} (\text{transfer-rep} (\text{Rep-memory} \ \sigma') \ src \ dst)) \ src =$
  $(\text{fst} (\text{transfer-rep} (\text{Rep-memory} \ \sigma') \ src \ dst)) \ dst$
using Rep-memory [of $\sigma'$]
apply (erule-tac src= src and dst = dst in transfer-rep-inv-E)
apply (rotate-tac 1)
apply (subt (asm) surjective-pairing[of (transfer-rep (Rep-memory $\sigma'$) src dst)])
unfolding memory-inv.simps
apply (erule conjE)
apply (erule allE)+
apply (erule impE)
unfolding transfer-rep-simp
apply auto
using equivp-reflp snd-memory-equivp
apply metis
done

theorem share-transfer:
  $x \ shares_{\sigma(a \land b)} y = (y = b \land (x = b$
  $\lor (x \neq b \land x \ shares_{\sigma} a))) \lor$
  $(y \neq b \land ((x = b \land a \ shares_{\sigma} y)$
  $\lor (x \neq b \land x \ shares_{\sigma} y)))$
unfolding sharing-def transfer-def
unfolding transfer-def map-fun-def o-def id-def
apply(subt Abs-memory-inverse[OF transfer-rep-sound2], simp add: transfer-rep-simp)
by (metis equivp-reflp sharing-charn2)

lemma transfer-share:a shares_{\sigma(a \land b)} b by(simp add: share-transfer sharing-refl)
lemma transfer-share-sym:a shares_{\sigma(b \land a)} b by(simp add: share-transfer sharing-refl)
lemma transfer-share-mono:x shares_{\sigma} y \implies \neg(x \ shares_{\sigma} b) \implies (x \ shares_{\sigma} (a \land b) \ y)
  by(auto simp: share-transfer sharing-refl)
lemma transfer-share-charn:
  $\neg(x \ shares_{\sigma} b) \implies \neg(y \ shares_{\sigma} b) \implies x \ shares_{\sigma(a \land b)} y = x \ shares_{\sigma} y$
  by(auto simp: share-transfer sharing-refl)
lemma transfer-share-trans:(a shares_{\sigma} x) \implies (x \ shares_{\sigma(a \land b)} b)
  by(auto simp: share-transfer sharing-refl sharing-sym)
lemma transfer-share-trans-sym: (a \ shares_{\sigma} y) \Longrightarrow (b \ shares_{(\sigma(a \otimes b))} y)
using transfer-share-trans sharing-sym by fast

lemma transfer-share-trans': (a \ shares_{(\sigma(a \otimes b))} z) \Longrightarrow (b \ shares_{(\sigma(a \otimes b))} z)
using transfer-share sharing-sym by fast

lemma transfer-tri : x \ shares_{\sigma} (a \otimes b) y \Longrightarrow x \ shares_{\sigma} b \lor b \ shares_{\sigma} y \lor x \ shares_{\sigma} y
by (metis sharing-sym transfer-share-charn)

lemma transfer-tri' : \lnot x \ shares_{\sigma} (a \otimes b) y \Longrightarrow y \ shares_{\sigma} b \lor \lnot x \ shares_{\sigma} y
by (metis sharing-sym sharing-trans transfer-share-mono)

lemma transfer-dest'
assumes 1 : a \ shares_{\sigma} (a \otimes b) y
and 2 : b \neq y
shows a \ shares_{\sigma} y
using assms by(auto simp: share-transfer sharing-refl sharing-sym)

lemma transfer-dest :
assumes 1 : \lnot (x \ shares_{\sigma} a)
and 2 : x \neq b
and 3 : x \ shares_{\sigma} b
shows \lnot (x \ shares_{\sigma} (a \otimes b) b)
using assms by(auto simp: share-transfer sharing-refl sharing-sym)

lemma transfer-dest'': x = b \Longrightarrow y \ shares_{\sigma} a \Longrightarrow x \ shares_{\sigma(a \otimes b)} y
by (metis sharing-sym transfer-share-trans-sym)

thm share-transfer
  transfer-share
  transfer-share-sym
  sharing-sym [THEN transfer-share-trans]
  sharing-sym [THEN transfer-share-trans-sym]
  transfer-share-trans'
  transfer-dest''
  transfer-dest'
  transfer-tri'
  transfer-share-mono
  transfer-tri
  transfer-share-charn

66
Properties on Memory Transfer and Lookup  

**Lemma** `transfer-share-lookup1`: \((\sigma(x \star y)) \, \sigma x = \sigma x\)

- **Using** `lookup-transfer-rep' transfer-rep-fst1`
- **Unfolding** `lookup-def transfer.rep-eq`
- **By** `metis`

**Lemma** `transfer-share-lookup2`:

\((\sigma(x \star y)) \, \sigma y = \sigma y\)

- **Using** `transfer-rep-fst1`
- **Unfolding** `transfer.rep-eq lookup-def`
- **By** `metis`

**Lemma** `add_e_not-share-lookup`:

- **Assumes** 1: \(\neg(x \text{ shares}_\sigma z)\)
- **And** 2: \(\neg(y \text{ shares}_\sigma z)\)
- **Shows** \((x \star y) \, z = \sigma z\)

- **Using** `assms`
- **Unfolding** `sharing-def lookup-def transfer.rep-eq`
- **Using** `id-def sharing-def sharing-refl transfer-rep-fst2`
- **By** `metis`

**Lemma** `transfer-share-dom`:

- **Assumes** 1: \(z \in \text{Domain } \sigma\)
- **And** 2: \(\neg(y \text{ shares}_\sigma z)\)
- **Shows** \((\sigma(x \star y)) \, z = \sigma z\)

- **Using** `assms`
- **Unfolding** `Domain-def sharing-def lookup-def`
- **Using** 2 `transfer.rep-eq id-apply sharing-refl transfer-rep-fst2`
- **By** `metis`

**Lemma** `shares-result'`:

- **Assumes** 1: \((x \text{ shares}_\sigma y)\)
- **Shows** \(\sigma x = \sigma y\)

- **Using** `assms lookup-def shares-result`
- **By** `metis`

**Lemma** `transfer-share-cancel1`:

- **Assumes** 1: \((x \text{ shares}_\sigma z)\)
- **Shows** \((\sigma(x \star y)) \, z = \sigma x\)

- **Using** 1 `transfer.rep-eq transfer-share-trans lookup-def transfer-rep-fst1 shares-result`
- **By** `(metis)`

**Lemmas** `sharing-refl-smt = sharing-refl`
An Infrastructure for Global Memory Spaces Memory spaces are common concepts in Operating System (OS) design since it is a major objective of OS kernels to separate logical, linear memory spaces belonging to different processes (or in other terminologies such as PiKeOS: tasks) from each other. We achieve this goal by modeling the addresses of memory spaces as a pair of a subject (e.g. process or task, denominated by a process-id or task-id) and a location (a conventional address).

Our model is still generic - we do not impose a particular type for subjects or locations (which could be modeled in a concrete context by an enumeration type as well as integers of bitvector representations); for the latter, however, we require that they are instances of the type class ‘a assuring that there is a minimum of infrastructure for address calculation: there must exist a 0::<‘a-element, a distinct 1::<‘a-element and an addition operation with the usual properties.

fun initglobalmem :: (('subx'loc::comm-semiring-1), 'β) memory ⇒ ('subx'loc) ⇒ 'β list
where σ | start < | [] = σ
| σ | > (sub,loc) < | (a ≠ S) = (((σ((sub,loc):=₄ a)) | > (sub,loc+1)< | S)

lemma Domain-mem-init-Nil : Domain(σ | > start < | []) = Domain σ
by simp

Example type-synonym task-id = int
type-synonym loc = int
type-synonym global-mem = ((task-idxloc), int)memory
definition σ₀ :: global-mem
where σ₀ ≡ init | > (0,0) < | [0,0,0,0]
| > (2,0) < | [0,0]
| > (4,0) < | [2,0]

lemma σ₀-Domain: Domain σ₀ = {(4, 1), (4, 0), (2, 1), (2, 0), (0, 3), (0, 2), (0, 1), (0, 0)}
unfolding σ₀-def
by (simp add: sharing-upd)

Memory Transfer Based on Sharing Closure (Experimental) One might have a fundamentally different understanding on memory transfer — at least as far as the sharing relation is concerned. The prior definition of sharing is based on the idea that the overridden part is “carved out” of the prior equivalence. Instead of transforming the equivalence relation, one might think of transfer as an operation where the to be shared memory is synchronized and then the equivalence relation closed via reflexive-transitive closure.

definition transfer’ :: ('a,'b)memory ⇒ 'a ⇒ 'a ⇒ ('a, 'b)memory (- '(- |x| |-') [0,111,111]110)
where σ(i |x| k) = (σ(i :=₄ (σ $ k)) :=₄ rtranclp(λx y. ($R σ) x y ∨ (x=i ∧ y = k) ∨ (x=k ∧ y = i))))

lemma transfer’-rep-sound:
Framing Conditions on Shared Memories (Experimental)

The Frame of an action — or a monadic operation — is the smallest possible subset of the domain of a memory, in which the action has effect, i.e., it modifies only locations in this set.

**Definition** Frame :: ((α, β)memory ⇒ (α, β)memory) ⇒ 'α set

**Where** Frame A ≡ Least(λX. ∀ σ. (σ(reset X)) = ((A σ)(reset X)))

**Lemma** Frame-update : Frame (λσ. σ(x :=ₗ y)) = {x}

**Oops**

**Lemma** Frame-compose : Frame (A o B) ⊆ Frame A ∪ Frame B

---

(fst(Rep-memory (σ(i:=ₗ(σ $ k))))) (λxa ya. ($R σ) xa ya ∨ xa = x ∧ ya = y ∨ xa = y ∧ ya = x)**

∈

{⟨(σ, R). equivp R ∧ (∀ x y. R x y → σ x = σ y)⟩}

**Unfolding** update-def

**Proof** (auto)

let ?R' = (⟨λxa ya. ($R σ) xa ya ∨ xa = x ∧ ya = y ∨ xa = y ∧ ya = x⟩)** xa ya

have E : equivp ($R σ) unfolding lookup_R-def by (metis snd-memory-equivp)

have fact1 : symp ?R'

unfolding symp-def

apply (auto)

apply (erule Transitive-Closure.rtranclp-induct,auto)

apply (erule E[THEN equivp-synth])

by (metis (lifting, full-types) converse-rtranclp-into-rtranclp+)

have fact2 : transp ?R'

unfolding transp-def

by (metis (lifting, no-types) rtranclp-trans)

have fact3 : reflp ?R'

unfolding reflp-def

by (metis (lifting) rtranclp.rtrancl-refl)

show equivp (λxa ya. ($R σ) xa ya ∨ xa = x ∧ ya = y ∨ xa = y ∧ ya = x)***

using fact1 fact2 fact3 equivpI by auto

next

fix xa ya

assume H : (λxa ya. ($R σ) xa ya ∨ xa = x ∧ ya = y ∨ xa = y ∧ ya = x)** xa ya

have * : (fun-upd-equivp (snd (Rep-memory σ)) (fst (Rep-memory σ)) i (Some (σ $ k)))

σ

∈ {⟨(σ, R). equivp R ∧ (∀ x y. R x y → σ x = σ y)⟩} sorry

show fst (Rep-memory (Abs-memory (Pair-upd-lifter (Rep-memory σ) i (σ $ k)))) xa =

fst (Rep-memory (Abs-memory (Pair-upd-lifter (Rep-memory σ) i (σ $ k))))) ya

apply[simp add: SharedMemory.lookup-def]

apply(insert H, simp add: SharedMemory.lookup_R-def)

oops
theory SharedMemory-test
imports ../../../src/Testing
  SharedMemory
begin

Our Local Instance of a Global memory Model
type-synonym task-id = int
type-synonym loc = int
type-synonym global-mem = ((task-id × loc), int)memory
definition σ₀ :: global-mem
where  σ₀ ≡ init |> (0,0) <| (0,0,0,0) |
       |> (2,0) <| (0,0) |
       |> (4,0) <| (2,0) |

find-theorems sharing

lemma σ₀-Domain: Domain σ₀ = \{(4, 1), (4, 0), (2, 1), (2, 0), (0, 3), (0, 2), (0, 1), (0, 0)\}
unfolding σ₀-def
by(simp add: σ₀-Domain sharing-upd)

datatype in-c = load  task-id loc |
  store  task-id loc int |
  share  task-id loc task-id loc

thm in-c.split

datatype out = load-ok  int |
  store-ok |
  share-ok
fun precond :: global-mem ⇒ in-c ⇒ bool
where precond σ (load tid addr) = (((tid,addr) ∈ Domain σ) | precond σ (store tid addr n) = True | precond σ (share tid addr tid’ addr’) = (((tid,addr) ∈ Domain σ ∧ (tid’,addr’) ∈ Domain σ)

term load-ok (σ₀ $ ((tid,addr))

fun postcond :: in-c ⇒ global-mem ⇒ (out × global-mem) set
where postcond (load tid addr) σ = {((n,σ'), (n = load-ok (σ $ (tid,addr)))) ∧ σ' = σ} | postcond (store tid addr m) σ = {((n,σ'), (n = store-ok ∧ σ' = σ((tid,addr)::=₄ m))} | postcond (share tid addr tid’ addr’) σ = { (n,σ'). n = share-ok ∧ σ' = σ((tid,addr) ≺ (tid’,addr’))}

definition SYS = (strong-impl precond postcond)

lemma SYS-is-strong-impl : is-strong-impl precond postcond SYS
by(simp add: SYS-def is-strong-impl)

lemma precond-postcond-implementable:
  implementable precond postcond
apply(auto simp: implementable-def)
apply(case-tac 1, simp-all)
done

thm SYS-is-strong-impl[simplified is-strong-impl-def,THEN spec,of (alloc c no m), simplified, standard]

lemma Eps-split-eq' : (SOME (x', y'). x' = x ∧ y' = y) = (SOME (x', y'). x = x' ∧ y = y')
by(rule arg-cong[of - - Eps], auto)

consts PUT :: in-c ⇒ 'σ ⇒ (out × 'σ) option

interpretation load : fsm-det
  precond postcond SYS (load tid addr)
  λσ. load-ok (σ $ (tid,addr))
  λ σ. σ
  λ σ. (tid,addr) ∈ Domain σ
  by unfold-locales (auto simp: SYS-def Eps-split-eq')

interpretation store : fsm-det
  precond postcond SYS (store tid addr m)
  λ-. store-ok
  λ σ. σ((tid, addr)::=₄ m)
  λ σ.(True)
  by unfold-locales (auto simp: SYS-def Eps-split-eq')

interpretation share : fsm-det
  precond postcond SYS (share tid addr tid’ addr’)

71
\[ \text{\(\lambda\). share-ok} \]
\[ \text{\(\lambda\) \(\sigma\). \((\text{tid}, \text{addr}) \times (\text{tid}', \text{addr}')\) } \]
\[ \text{\(\lambda\) \(\sigma\). \((\text{tid}, \text{addr}) \in \text{Domain} \sigma \land (\text{tid}', \text{addr}') \in \text{Domain} \sigma\) } \]

by unfold-locales (auto simp: SYS-def Eps-split-eq')

The TestGen Setup 
\[
\text{set-pre-safe-tac} (fn \text{ctxt} => \text{TestGen.ALLCASES( TestGen.CLOSURE ( TestGen.case-tac-typ ctxt [SharedMemory-test.in-c])))} \]
\]

declare Monads.mbind'-bind[simp del]

lemmas update-simps = update-share sharing-upd update-apply update-other update-cancel update-triv update-commute

shares-dom Domain-transfer Domain-update

shares-init sharing-refl sharing-upd transfer-share share-transfer sharing-commute

thm update-simps

An Abstract Test-Case Generation Scenario

Scenario with tests starting on an fixed initialized memory. This corresponds roughly to checking that all inductively defined shared memories build over store, load and transfer reveal the behaviour rescribed by the model.

test-spec test-status:
assumes sym-exec-spec:

\[ \text{init} \mid (s \leftarrow \text{mbind}\_\text{FailStop} S \text{SYS}; \text{return} (s = x)) \]

shows \[ \text{init} \mid (s \leftarrow \text{mbind}\_\text{FailStop} S \text{PUT}; \text{return} (s = x)) \]

apply(insert assms)
apply(tactic TestGen.mp-fy 1,rule-tac \text {x=x in spec[OF allI]})
apply(tactic asm-full-simp-tac @(context \text {I} )
using [[no-uniformity]]]
apply(gen-test-cases 5 1 \text {PUT})
apply(tactic ALLGOALS(TestGen.REPEAT'(\text {ematch-tac [\{thm load.exec-mbindFStop-E\}, \{thm store.exec-mbindFStop-E\}, \{thm share.exec-mbindFStop-E\}, \{thm valid-mbind'-mtE\}]})]

Normalization

apply(simp-all add: update-simps)
apply(tactic TestGen.ALLCASES(TestGen.TRY'(fn n => REPEAT-DETERM1 ( (safe-steps-tac @(\{context \text {I} \}) n)))))
apply(simp-all add: update-simps)

Closing : Extracting PO’s
using [[no-uniformity=false]]
apply(tactic TestGen.ALLCASES(TestGen.uniformityI-tac @{context} [PUT]))

mk-test-suite SharedMemoryNB

Concrete Test Data Selection

declare [[testgen-iterations=0]]
declare [[testgen-SMT]]

gen-test-data SharedMemoryNB
thm SharedMemoryNB.concrete-tests
thm SharedMemoryNB.test-inst-thm

end

5.3.2. The MyKeOS Case Study

theory MyKeOS
imports
   ../../../src/Testing
begin

This example is drawn from the operating system testing domain; it is a rough abstraction of PiKeOS and explains the underlying techniques of this particular case study on a small example. The full paper can be found under [5].

This is a fun-operating system — closely following the Bank example — intended to explain the principles of symbolic execution used in our PikeOS study.

Moreover, in this scenario, we assume that the system under test is deterministic.

The state of a thread (belonging to a task, i.e. a Unix/PosiX like “process” just modeled by a map from task-id/thread-id information to the number of a resource (a communication channel descriptor, for example) that was allocated to a thread.

type-synonym task-id = int
type-synonym thread-id = int
type-synonym thread-local-var-tab = (task-id × thread-id) → int

Operation definitions A standard, JML or OCL or VCC like interface specification might look like:

Init: forall (c,no) : dom(data_base::thread_local_var_tab). data_base(c,no)>=0

op alloc (c : task_id, no : thread_id, amount:nat) : unit
pre (c,no) : dom(data_base)
post data_base' = data_base[(c,no) := data_base(c,no) + amount]

op release(c : task_id, no : thread_id, amount:nat) : unit
pre (c,no) : dom(data_base) and data_base(c,no) >= amount
post data_base' = data_base[(c,no) := data_base(c,no) - amount]

op status (c : task_id, no : thread_id) : int
pre (c,no) : dom(data_base)
post data_base' = data_base and result = data_base(c,no)

Interface normalization turns this interface into the following input type:

datatype
  in-c = alloc task-id thread-id nat
  | release task-id thread-id nat
  | status task-id thread-id

typ MyKeOS.in-c

datatype out-c = alloc-ok | release-ok | status-ok nat

fun precond :: thread-local-var-tab ⇒ in-c ⇒ bool
where precond σ (alloc c no m) = ((c,no) ∈ dom σ)
  | precond σ (release c no m) = ((c,no) ∈ dom σ ∧ (int m) ≤ the(σ(c,no)))
  | precond σ (status c no) = ((c,no) ∈ dom σ)

fun postcond :: in-c ⇒ thread-local-var-tab ⇒ (out-c × thread-local-var-tab) set
where postcond (alloc c no m) σ =
  { (n,σ'). (n = alloc-ok ∧ σ'=σ((c,no)→ the(σ(c,no)) + int m))}
| postcond (release c no m) σ =
  { (n,σ'). (n = release-ok ∧ σ'=σ((c,no)→ the(σ(c,no)) − int m))}
| postcond (status c no) σ =
  { (n,σ'). (σ=σ' ∧ (∃ x. status-ok x = n ∧ x = nat(∀ c(no)))))}

Constructing an Abstract Program Using the Operators impl and strong_impl, we can synthesize an abstract program right away from the specification, i.e. the pair of pre- and postcondition defined above. Since this program is even deterministic, we derive a set of symbolic execution rules used in the test case generation process which will produce symbolic results against which the PUT can be compared in the test driver.

lemma precond-postcond-implementable:
  implementable precond postcond
apply(auto simp: implementable-def)
apply(case-tac i, simp-all)
done

Based on this machinery, it is now possible to construct the system model as the canonical completion of the (functional) specification consisting of pre- and post-conditions.
definition \( SYS = (\text{strong-impl precond postcond}) \)

lemma \( SYS-is-strong-impl : \text{is-strong-impl precond postcond SYS} \)
by (simp add: SYS-def is-strong-impl)

thm \( SYS-is-strong-impl[\text{simplified is-strong-impl-def, THEN spec, of (alloc c no m), simplified, standard}] \)

Proving Symbolic Execution Rules for the Abstractly Program

The following lemmas reveal that this "constructed" program is actually (due to determinism of the spec):

lemma \( Eps-split-eq' : (\text{SOME } (x', y'). \ x'= x \land y'= y) = (\text{SOME } (x', y'). \ x = x' \land y = y') \)
by (rule arg-cong[of - - Eps], auto)

interpretation alloc : \( \text{efsm-det precond postcond SYS (alloc c no m) \lambda- alloc-ok} \)
\( \lambda \sigma. \sigma((c, no) \mapsto (\sigma(c, no)) + \text{int m})) \lambda \sigma. ((c, no) \in \text{dom } \sigma) \)
by unfold-locales (auto simp: SYS-def Eps-split-eq')

interpretation release : \( \text{efsm-det precond postcond SYS (release c no m) release-ok} \)
\( \lambda \sigma. \sigma((c, no) \mapsto (\sigma(c, no)) - \text{int m})) \lambda \sigma. ((c, no) \in \text{dom } \sigma) \land (\text{int m}) \leq (\sigma(c, no)) \)
by unfold-locales (auto simp: SYS-def Eps-split-eq')

interpretation status : \( \text{efsm-det precond postcond SYS (status c no)} \)
\( \lambda \sigma. \sigma((\text{status-ok } (\text{nat}(\sigma(c, no)))))) \)
\( \lambda \sigma. \lambda \sigma. ((c, no) \in \text{dom } \sigma) \) by unfold-locales (auto simp: SYS-def Eps-split-eq')

Setup

Now we close the theory of symbolic execution by excluding elementary rewrite steps on \( \text{mbind_FailSave} \), i.e. the rules \( \text{mbind_FailSave } [] ?iostep ?\sigma \Rightarrow \text{Some } (\[], \ ?\sigma) \)
\( \text{mbind_FailSave } (?a # ?S) ?iostep ?\sigma = (\text{case } ?iostep ?a ?\sigma \text{ of None } \Rightarrow \text{Some } (\[], ?\sigma) \mid \text{Some } (\text{out}, ?\sigma') \Rightarrow \text{case } \text{mbind_FailSave } ?S ?iostep ?\sigma' \text{ of None } \Rightarrow \text{Some } ([\text{out}], ?\sigma') \mid \text{Some } (\text{outs}, ?\sigma'') \Rightarrow \text{Some } (\text{out } # \text{outs}, ?\sigma'')) \)

declare mbind.simps(1) [simp del]
mbind.simps(2) [simp del]

Here comes an interesting detail revealing the power of the approach: The generated sequences still respect the preconditions imposed by the specification - in this case, where we are talking about a \text{task_id} for which a defined account exists and for which we will never produce traces in which we release more money than available on it.

Restricting the Test-Space by Test Purposes

We introduce a constraint on the input sequence, in order to limit the test-space a little and eliminate logically possible, but irrelevant test-sequences for a specific test-purpose. In this case, we narrow down on test-sequences concerning a specific \text{task_id} \( c \) with a specific bank-account number \( no \).
We make the (in this case implicit, but as constraint explicitly stated) test hypothesis, that the SUT is correct if it behaves correct for a single task_id. This boils down to the assumption that they are implemented as atomic transactions and interleaved processing does not interfere with a single thread.

fun test-purpose :: [(task-id × thread-id) list, in-c list] ⇒ bool
where
  test-purpose ((c,no)#R) [status c’ no’] = ((c=c’ ∧ no=no’) ∨ test-purpose R [status c’ no’])
| test-purpose ((c,no)#R) ((alloc c’ no’ m)#S) = ((c=c’ ∧ no=no’ ∧ test-purpose ((c,no)#R) S) ∨ test-purpose R ((alloc c’ no’ m)#S))
| test-purpose ((c,no)#R) ((release c’ no’ m)#S) = ((c=c’ ∧ no=no’ ∧ test-purpose ((c,no)#R) S) ∨ test-purpose R ((release c’ no’ m)#S))
| test-purpose - - = False

lemma [simp] : test-purpose [] a = False by simp
lemma [simp] : test-purpose r [] = False by simp

lemma [simp] : test-purpose ((c,no)#R) [a] = ((a = status c no) ∨ test-purpose R [a])
proof (induct R)
  case Nil show ?case by (cases a, auto)
next
  case (Cons a’ R’) then show ?case
    apply (cases a, simp-all)
    apply (cases a’, simp)
    apply (cases a’, simp)
    apply (cases a’, simp)
    apply (rename-tac int1 int2 int1 int2 a b)
    apply (case-tac c = int1 ∧ no = int2, auto)
  done

qed

find-theorems name:in-c name :split
lemma [simp] : R#[] ⇒ test-purpose [[c,no),(c’,no’)] (a#R) = (((∃ m. a = (alloc c no m)) ∨ (∃ m. a = (release c no m))) ∨ (∃ m. a = (release c’ no’ m))) ∧ test-purpose [[c,no),(c’,no’)] R)
apply (simp add: List.neq-nil-conv, elim exE, simp)
apply (auto split: in-c.split in-c.split-asm)
apply (cases a, auto)
sorry

consts PUT :: in-c ⇒ ‘σ ⇒ (out-c × ‘σ) option
5.3.3. The MyKeOS "Classic" Data-sequence enumeration approach

theory MyKeOS-test
imports MyKeOS
  ../../../src/codegen-fsharp/Code-Integer-Fsharp
begin

The purpose of these test-scenarios is to apply the brute-force data-exploration approach to a little operation system example. It is conceptually very close to the Bank-example, essentially a renaming. However, the present "data-exploration" based approach is an interesting intermediate step to the subsequently shown scenarios based on:

1. exploration if the interleaving space

2. optimized exploration if the interleaving space, including theory for partial-order reduction.

declare [[testgen-profiling]]

The TestGen Setup The default configuration of gen_test_cases does not descend into sub-type expressions of type constructors (since this is not always desirable, the choice for the default had been for "non-descent"). This case is relevant here since in-c list has just this structure but we need ways to explore the input sequence type further. Thus, we need configure, for all test cases, and derivation descendants of the resulting clauses during splitting, again splitting for all parameters of input type in-c:

set-pre-safe-tac
  (fn ctxt => TestGen.ALLCASES(
    TestGen.CLOSURE (
      TestGen.case-tac-typ ctxt [MyKeOS.in-c])))

The Scenario We construct test-sequences for a concrete task_id (implicitly assuming that interleaving actions with other task_id's will not influence the system behaviour. In order to prevent HOL-TestGen to perform case-splits over names — i.e. list of characters — we define it as constant.

definition tid0 :: task-id where tid0 = 0
definition tid1 :: task-id where tid1 = 1
definition thid0 :: thread-id where thid0 = 4
definition thid1 :: thread-id where thid1 = 6

declare[[goals-limit = 500]]

Making my own test-data generation — temporarily lemma HH : (A ∧ (A → B)) = (A ∧ B) by auto
Some Experiments with nitpick as Testdata Selection Machine. Exists in two formats: General Fail-Safe Tests (which allows for scenarios with normal and exceptional behaviour; and Fail-Stop Tests, which generates Tests only for normal behaviour and correspond to inclusion test refinement.

**Lemma H:**
\[
\begin{align*}
&(((X586X11506, X587X11507) \in \text{dom } X588X11508 \rightarrow \\
&\quad \text{status-ok(nat(the((X586X11506, X588X11508) (X586X11506, X587X11507))))} = X590X11510 \land \\
&\quad ((X586X11506, X587X11507) \notin \text{dom } X588X11508 \rightarrow \\
&\quad \text{[]} = X590X11510 \land X588X11508 (X586X11506, X587X11507) = Some X589X11509))
\end{align*}
\]

\text{nitpick[satisfy,debug]}

**OOPS**

In the following, we discuss a test-scenario with error-abort semantics; i.e. in each test-case, a sequence may be chosen (by the test data selection) where the \texttt{task_id} has several accounts.

**Test-spec test-status:**

\text{assumes account-def} : (c0, no) \in \text{dom } \sigma_0 \land (c0, no') \in \text{dom } \sigma_0

\text{and test-purpose} : \text{test-purpose } [(c0, no), (c0, no')] S

\text{and sym-exec-spec} : \sigma_0 \models (s \leftarrow \text{mbind}_Fai\text{lS}\text{ave} S SYS; \text{return } (s = x))

**Shows**

\sigma_0 \models (s \leftarrow \text{mbind}_Fai\text{lS}\text{ave} S PUT; \text{return } (s = x))

**Prelude:** Massage of the test-theorem — representing the assumptions of the test explicitly in HOL and blocking \(x\) from being case-split (which complicates the process).

\text{apply(rule rev-mp[OF sym-exec-spec])}
\text{apply(rule rev-mp[OF account-def])}
\text{apply(rule rev-mp[OF test-purpose])}
\text{apply(rule-tac } x=x \text{ in spec[OF allI])}

**Starting the test generation process.**

\text{apply(gen-test-cases 4 1 PUT)}
\text{apply(simp-all add: HH split: HOL.split-if-asm)}
\text{mk-test-suite mykeos-simpleSNXB}

And now the Fail-Stop scenario — this corresponds exactly to inclusion test.

\text{declare Monads.mbind'-bind [simp def]}
test-spec test-status2:
assumes system-def : \((c_0, no) \in \text{dom } \sigma_0 \land (c_0, no') \in \text{dom } \sigma_0\)
and test-purpose : test-purpose \[\[(c_0, no), (c_0, no')\]\] \(S\)
and sym-exec-spec :
\[\sigma_0 \models (s \leftarrow \text{mbind}_{\text{FailStop} S \text{SYS}; \text{return } (s = x)})\]
shows \[\sigma_0 \models (s \leftarrow \text{mbind}_{\text{FailStop} S \text{PUT}; \text{return } (s = x)})\]

Prelude: Massage of the test-theorem — representing the assumptions of the test explicitly in HOL and blocking \(x\) from being case-splitted (which complicates the process).

apply (rule rev-mp [OF sym-exec-spec])
apply (rule rev-mp [OF system-def])
apply (rule rev-mp [OF test-purpose])
apply (rule-tac \(x = x\) in spec [OF allI])

Starting the test generation process.

using [[no-uniformity]]
apply (gen-test-cases 3 1 PUT)

So lets go for a more non-destructive approach:

apply simp-all

using [[no-uniformity = false]]
apply (tactic TestGen.ALLCASES (TestGen.uniformityI-tac @{context} [PUT]))

mk-test-suite mykeos-simpleNB

Test-Data Generation thm mykeos-simpleSNXB.test-thm

declare [[testgen-iterations = 0]]
declare [[testgen-SMT]]

declare tid_0-def [testgen-smt-facts]
declare tid_1-def [testgen-smt-facts]
declare thid_0-def [testgen-smt-facts]
declare thid_1-def [testgen-smt-facts]

declare mem-Collect-eq [testgen-smt-facts]
declare Collect-mem-eq [testgen-smt-facts]
declare dom-def [testgen-smt-facts]
declare the.simps [testgen-smt-facts]

gen-test-data mykeos-simpleSNXB
thm mykeos-simpleSNXB.concrete-tests

gen-test-data mykeos-simpleNB
thm mykeos-simpleNB.concrete-tests
Generating the Test-Driver for an SML and C implementation  The generation of the test-driver is non-trivial in this exercise since it is essentially two-staged: Firstly, we chose to generate an SML test-driver, which is then secondly, compiled to a C program that is linked to the actual program under test. Recall that a test-driver consists of four components:

- ../../../../../harness/sml/main.sml the global controller (a fixed element in the library),
- ../../../../../harness/sml/main.sml a statistic evaluation library (a fixed element in the library),
- bank_simple_test_script.sml the test-script that corresponds merely one-to-one to the generated test-data (generated)
- bank_adapter.sml a hand-written program; in our scenario, it replaces the usual (black-box) program under test by SML code, that calls the external C-functions via a foreign-language interface.

On all three levels, the HOL-level, the SML-level, and the C-level, there are different representations of basic data-types possible; the translation process of data to and from the C-code under test has therefore to be carefully designed (and the sheer space of options is sometimes a pain in the neck). Integers, for example, are represented in two ways inside Isabelle/HOL; there is the mathematical quotient construction and a "numerals" representation providing "bit-string-representation-behind-the-scene" enabling relatively efficient symbolic computation. Both representations can be compiled "natively" to data types in the SML level. By an appropriate configuration, the code-generator can map "int" of HOL to three different implementations; the SML standard library `Int.int`, the native-C interfaced by `Int32.int`, and the `IntInf.int` from the multi-precision library `gmp` underneath the polyml-compiler.

We do a three-step compilation of data-representations model-to-model, model-to-SML, SML-to-C.

A basic preparatory step for the initializing the test-environment to enable code-generation is:

```plaintext
generate-test-script mykeos-simpleNB
thm          mykeos-simpleNB.test-script
generate-test-script mykeos-simpleSNXB
```

In the following, we describe the interface of the SML-program under test, which is in our scenario an adapter to the C code under test. This is the heart of the model-to-SML translation. The the SML-level stubs for the program under test are declared as follows:

```plaintext
consts status-stub :: task-id ⇒ thread-id ⇒ (int, 'σ)MONSE
code-const status-stub (SML MyKeOSAdapter.status)
consts alloc-stub :: task-id ⇒ thread-id ⇒ int ⇒ (unit, 'σ)MONSE
code-const alloc-stub (SML MyKeOSAdapter.alloc)
consts release-stub:: task-id ⇒ thread-id ⇒ int ⇒ (unit, 'σ)MONSE
code-const release-stub (SML MyKeOSAdapter.release)
```
Note that this translation step prepares already the data-adaption; the type \texttt{nat} is seen as an predicative constraint on integer (which is actually not tested). On the model-to-model level, we provide a global step function that distributes to individual interface functions via stubs (mapped via the code generation to SML ...). This translation also represents uniformly nat by int’s.

\begin{verbatim}
fun my-nat-conv :: int ⇒ nat
where my-nat-conv x = (if x <= 0 then 0 else Suc (my-nat-conv (x - 1)))
\end{verbatim}

\begin{verbatim}
fun stepAdapter :: (in-c ⇒ (out-c, 'σ)MON_SE)
where stepAdapter (status tid thid) =
  (x ← status-stub tid thid; return (status-ok (my-nat-conv x)))
| stepAdapter (alloc tid thid amount) =
  (- ← alloc-stub tid thid (int amount); return (alloc-ok))
| stepAdapter (release tid thid amount) =
  (- ← release-stub tid thid (int amount); return (release-ok))
\end{verbatim}

The \texttt{stepAdapter} function links the HOL-world and establishes the logical link to HOL stubs which were mapped by the code-generator to adapter functions in SML (which call internally to C-code inside \texttt{bank_adapter.sml} via a foreign language interface) ... We configure the code-generator to identify the \texttt{PUT} with the generated SML code implicitly defined by the above \texttt{stepAdapter} definition.

code-const \texttt{PUT} (SML stepAdapter)

And there we go and generate the \texttt{mykeos_simpleNB_test_script.sml} as well as the \texttt{mykeos_simpleSNXB_test_script.sml}:

\begin{verbatim}
export-code stepAdapter mykeos-simpleSNXB_test_script in SML
module-name TestScript file impl/c/mykeos-simpleSNXB-test-script.sml
\end{verbatim}

\begin{verbatim}
export-code stepAdapter mykeos-simpleNB_test_script in SML
module-name TestScript file impl/c/mykeos-simpleNB-test-script.sml
\end{verbatim}

\section*{More advanced Test-Case Generation Scenarios}
Exploring a bit the limits ...

Rewriting based approach of symbolic execution ... FailSave Scenario

test-spec test-status:
assumes account-def : (c_0, no) ∈ dom σ_0 ∧ (c_0, no') ∈ dom σ_0
and test-purpose : test-purpose [(c_0, no), (c_0, no')] S
and sym-exec-spec :
  σ_0 ≡ (s ← mbind_{FailSave} S SYS; return (s = x))
shows σ_0 ≡ (s ← mbind_{FailSave} S PUT; return (s = x))

Prelude: Massage of the test-theorem — representing the assumptions of the test explicitly in HOL and blocking x from being case-splitted (which complicates the process).

apply(insert account-def test-purpose sym-exec-spec)
apply(tactic TestGen.mp-fy 1,rule-tac x=x in spec[OF allI])

Starting the test generation process.

apply(gen-test-cases 3 1 PUT)

Symbolic Execution:

apply(simp-all add: HH split: HOL.split-if-asm)
Rewriting based approach of symbolic execution ... FailSave Scenario

test-spec test-status:
assumes account-def : \((c_0, no) \in \text{dom } \sigma_0 \land (c_0, no') \in \text{dom } \sigma_0\)
and test-purpose : test-purpose \([(c_0, no), (c_0, no')] S\)
and sym-exec-spec :
\[\sigma_0 \models (s \leftarrow \text{mbind}_{\text{FailStop}} S \text{SYS}; \text{return } (s = x))\]
shows \[\sigma_0 \models (s \leftarrow \text{mbind}_{\text{FailStop}} S \text{PUT}; \text{return } (s = x))\]

Prelude: Massage of the test-theorem — representing the assumptions of the test explicitly in HOL and blocking \(x\) from being case-splitted (which complicates the process).

apply(insert account-def test-purpose sym-exec-spec)
apply(tactic TestGen.mp-fy 1,rule-tac x=x in spec[OF allI])

Starting the test generation process.

using [[no-uniformity]]
apply(gen-test-cases 3 1 PUT)

Symbolic Execution:

apply(simp-all add: HH split: HOL.split-if-asm)
apply(auto)

And now, to compare, elimination based procedures ...

declare alloc.exec-mbindFSave-If [simp del]
status.exec-mbindFSave-If [simp del]
release.exec-mbindFSave-If [simp del]
alloc.exec-mbindFSave-St [simp del]
status.exec-mbindFSave-St [simp del]
release.exec-mbindFSave-St [simp del]

thm alloc.exec-mbindFSave-E release.exec-mbindFSave-E status.exec-mbindFSave-E

ML(open Tactical; )
thest-spec test-status:
assumes account-defined; \((c_0, no) \in \text{dom } \sigma_0 \land (c_0, no') \in \text{dom } \sigma_0\)
and test-purpose : test-purpose \([(c_0, no), (c_0, no')] S\)
and sym-exec-spec :
\[\sigma_0 \models (s \leftarrow \text{mbind}_{\text{FailStop}} S \text{SYS}; \text{return } (s = x))\]
shows \[\sigma_0 \models (s \leftarrow \text{mbind}_{\text{FailStop}} S \text{PUT}; \text{return } (s = x))\]
apply(insert account-defined test-purpose sym-exec-spec)
apply(tactic TestGen.mp-fy 1,rule-tac x=x in spec[OF allI])
apply(tactic asm-full-simp-tac @{context} 1)
using [[no-uniformity]]
apply(gen-test-cases 3 1 PUT)

apply(tactic ALLGOALS(Ematch-tac \[@\{thm status.exec-mbindFStop-E\},
  \{@\{thm release.exec-mbindFStop-E\},
  \{@\{thm alloc.exec-mbindFStop-E\},
  \{@\{thm valid-mbind'-mtE\}
  \})))
apply(simp-all)
mk-test-suite mykeos-very-large

end

5.3.4. Implementation of integer numbers by target-language integers

theory Code-Target-Int
imports Main
begin

code-datatype int-of-integer

lemma [code, code del]:
  integer-of-int = integer-of-int ..

lemma [code]:
  integer-of-int (int-of-integer k) = k
  by transfer rule

lemma [code]:
  Int.Pos = int-of-integer ◦ integer-of-num
  by transfer (simp add: fun-eq-iff)

lemma [code]:
  Int.Neg = int-of-integer ◦ uminus ◦ integer-of-num
  by transfer (simp add: fun-eq-iff)

lemma [code-abbrev]:
  int-of-integer (numeral k) = Int.Pos k
  by transfer simp

lemma [code-abbrev]:
  int-of-integer (neg-numeral k) = Int.Neg k
  by transfer simp

lemma [code, symmetric, code-post]:
  0 = int-of-integer 0
  by transfer simp

lemma [code, symmetric, code-post]:
  1 = int-of-integer 1
by transfer simp

lemma [code]:
k + l = int-of-integer (of-int k + of-int l)
by transfer simp

lemma [code]:
− k = int-of-integer (− of-int k)
by transfer simp

lemma [code]:
k − l = int-of-integer (of-int k − of-int l)
by transfer simp

lemma [code]:
Int.dup k = int-of-integer (Code-Numeral.dup (of-int k))
by transfer simp

lemma [code, code del]:
Int.sub = Int.sub ..

lemma [code]:
k * l = int-of-integer (of-int k * of-int l)
by simp

lemma [code]:
Divides.divmod-abs k l = map-pair int-of-integer int-of-integer
(Code-Numeral.divmod-abs (of-int k) (of-int l))
by (simp add: prod-eq-iff)

lemma [code]:
k div l = int-of-integer (of-int k div of-int l)
by simp

lemma [code]:
k mod l = int-of-integer (of-int k mod of-int l)
by simp

lemma [code]:
HOL.equal k l = HOL.equal (of-int k :: integer) (of-int l)
by transfer (simp add: equal)

lemma [code]:
k ≤ l ←→ (of-int k :: integer) ≤ of-int l
by transfer rule

lemma [code]:
k < l ←→ (of-int k :: integer) < of-int l
by transfer rule

lemma (in ring-1) of-int-code:
of-int k = (if k = 0 then 0
else if \( k < 0 \) then \(-\)of-int \((- k)\)
else let
\((l, j) = \text{divmod-int } k 2\)
\(l' = 2 \times \text{of-int } l\)
in if \( j = 0 \) then \( l' \) else \( l' + 1 \))

proof –
from \text{mod-div-equality} have \(*: \text{of-int } k = \text{of-int } (k \div 2 \times 2 + k \mod 2)\) by simp
show \(?\text{thesis}\)
by (simp add: Let-def divmod-int-div mod-2-not-eq-zero-eq-one-int
\text{of-int-add} [\text{symmetric}] ) (simp add: * mult-commute)
qed

declare \text{of-int-code} [\text{code}]

lemma [\text{code}]:
\text{nat} = \text{nat-of-integer} \circ \text{of-int}
by transfer (simp add: fun-eq-iff)

code-identifier
\text{code-module} \text{Code-Target-Int} \rightarrow
(SML) \text{Arith} and (OCaml) \text{Arith} and (Haskell) \text{Arith}

end

5.3.5. Avoidance of pattern matching on natural numbers

theory \text{Code-Abstract-Nat}
imports \text{Main}
begin

When natural numbers are implemented in another than the conventional inductive \( 0 / \text{Suc} \)
representation, it is necessary to avoid all pattern matching on natural numbers altogether.
This is accomplished by this theory (up to a certain extent).

Case analysis  Case analysis on natural numbers is rephrased using a conditional expression:

lemma [\text{code}, \text{code-unfold}]:
\text{nat-case} = (\lambda f \ g \ n. \text{if } n = 0 \text{ then } f \text{ else } g \ (n - 1))
by (auto simp add: fun-eq-iff dest!: gr0-implies-Suc)

Preprocessors  The term \text{Suc } n \text{ is no longer a valid pattern}. Therefore, all occurrences
of this term in a position where a pattern is expected (i.e. on the left-hand side of a code
equation) must be eliminated. This can be accomplished – as far as possible – by applying
the following transformation rule:

lemma \text{Suc-if-eq}: (\forall n. \text{f } \text{(Suc } n) \equiv h \ n) \implies \text{f } 0 \equiv g \implies
\text{f } n \equiv \text{if } n = 0 \text{ then } g \text{ else } h \ (n - 1)
by (rule eq-reflection) (cases \ n, simp-all)
The rule above is built into a preprocessor that is plugged into the code generator.

```fun remove-suc thy thms =
  let
    val vname = singleton (Name.variant-list (map fst
      (fold (Term.add-var-names o Thm.full-prop-of) thms []))) n;
    val cv = cterm-of thy (Var ((vname, 0), HOLogic.natT));
    fun lhs-of th = snd (Thm.dest-comb
      (fst (Thm.dest-comb (cprop-of th))));
    fun rhs-of th = snd (Thm.dest-comb (cprop-of th));
    fun find-vars ct = (case term-of ct of
      (Const (@{const-name Suc}, -) $ Var -) =>
        ((cv, snd (Thm.dest-comb ct)) | - $ - =>
          let val (ct1, ct2) = Thm.dest-comb ct
            in
              map (apfst (fn ct => Thm.apply ct ct2)) (find-vars ct1) @
              map (apfst (Thm.apply ct1)) (find-vars ct2)
            end | - => []);
    val eqs = maps
      (fn th => map (pair th) (find-vars (lhs-of th))) thms;
    fun mk-thms (th, (ct, cv′)) =
      let
        val th′ =
          Thm.implies-elim
            (Conv.fconv-rule (Thm.beta-conversion true)
              (Drule.instantiate′
                (SOME (ctyp-of-term ct)] [SOME (Thm.lambda cv ct),
                  SOME (Thm.lambda cv′ (rhs-of th)), NONE, SOME cv′]
                @{thm Suc-if-eq}) (Thm.forall-intr cv′ th))
          in
            case map-filter (fn th'' =>
              SOME (th'', singleton
                (Variable.trade (K (fn [th''' => [th''′ RS th′′′]])
                  (Variable.global-thm-context th′′′)) th′′′)
                handle THM - => NONE) thms of
              [] => NONE |
              thps =>
                let val (ths1, ths2) = split-list thps
                  in SOME (subtract Thm.eq-thm (th :: ths1) thms @ ths2) end
            end
          in get-first mk-thms eqs end;

    fun eqn-suc-base-preproc thy thms =
      let
        val dest = fst o Logic.dest-equals o prop-of;
        val contains-suc = exists-Const (fn (e, -) => e = @{const-name Suc});
      in
        if forall (can dest) thms andalso exists (contains-suc o dest) thms
```
then thms |> perhaps-loop (remove-suc thy) |> (Option.map o map) Drule.zero-var-indexes else NONE
end;

val eqn-suc-preproc = Code-Preproc.simple-functrans eqn-suc-base-preproc;
in
Code-Preproc.add-functrans (eqn-Suc, eqn-suc-preproc)
end;
⟩⟩
end

5.3.6. Implementation of natural numbers by target-language integers

theory Code-Target-Nat
imports Code-Abstract-Nat
begin

Implementation for nat lift-definition Nat :: integer ⇒ nat
is nat .

lemma [code-post]:
Nat 0 = 0
Nat 1 = 1
Nat (numeral k) = numeral k
by (transfer, simp)+

lemma [code-abbrev]:
integer-of-nat = of-nat
by transfer rule

lemma [code-unfold]:
Int.nat (int-of-integer k) = nat-of-integer k
by transfer rule

lemma [code abstype]:
Code-Target-Nat.Nat (integer-of-nat n) = n
by transfer simp

lemma [code abstract]:
integer-of-nat (nat-of-integer k) = max 0 k
by transfer auto

lemma [code-abbrev]:
nat-of-integer (numeral k) = nat-of-num k
by transfer (simp add: nat-of-num-numeral)
lemma [code abstract]:
  integer-of-nat (nat-of-num n) = integer-of-num n
by transfer (simp add: nat-of-num-numeral)

lemma [code abstract]:
  integer-of-nat 0 = 0
by transfer simp

lemma [code abstract]:
  integer-of-nat 1 = 1
by transfer simp

lemma [code]:
  Suc n = n + 1
by simp

lemma [code abstract]:
  integer-of-nat (m + n) = of-nat m + of-nat n
by transfer simp

lemma [code abstract]:
  integer-of-nat (m - n) = max 0 (of-nat m - of-nat n)
by transfer simp

lemma [code abstract]:
  integer-of-nat (m * n) = of-nat m * of-nat n
by transfer (simp add: of-nat-mult)

lemma [code abstract]:
  integer-of-nat (m div n) = of-nat m div of-nat n
by transfer (simp add: zdiv-int)

lemma [code abstract]:
  integer-of-nat (m mod n) = of-nat m mod of-nat n
by transfer (simp add: zmod-int)

lemma [code]:
  Divides.divmod-nat m n = (m div n, m mod n)
by (simp add: prod-eq-iff)

lemma [code]:
  HOL.equal m n = HOL.equal (of-nat m :: integer) (of-nat n)
by transfer (simp add: equal)

lemma [code]:
  m ≤ n ↔ (of-nat m :: integer) ≤ af-nat n
by simp

lemma [code]:
  m < n ↔ (of-nat m :: integer) < af-nat n
by simp
lemma num-of-nat-code [code]:
num-of-nat = num-of-integer o of-nat
by transfer (simp add: fun-eq-iff)

lemma (in semiring-1) of-nat-code:
of-nat n = (if n = 0 then 0
else let
  (m, q) = divmod-nat n 2;
  m’ = 2 * of-nat m
in if q = 0 then m’ else m’ + 1)

proof
from mod-div-equality have *: of-nat n = of-nat (n div 2 * 2 + n mod 2) by simp
of-nat-add [symmetric])
qed

declare of-nat-code [code]

definition int-of-nat :: nat ⇒ int where
[code-abbrev]: int-of-nat = of-nat

lemma [code]:
int-of-nat n = int-of-integer (of-nat n)
by (simp add: int-of-nat-def)

lemma [code abstract]:
integer-of-nat (nat k) = max 0 (integer-of-int k)
by transfer auto

lemma term-of-nat-code [code]:
— Use nat-of-integer in term reconstruction instead of Code-Target-Nat.Nat such that reconstructed terms can be fed back to the code generator
term-of-class.term-of n =
  Code-Evaluation.App
  (typerep.Typerep (STR "fun")
   [(typerep.Typerep (STR "Code-Numeral.integer") []),
    typerep.Typerep (STR "Nat.nat") []])
  (term-of-class.term-of (integer-of-nat n))
by(simp add: term-of-anything)

lemma nat-of-integer-code-post [code-post]:
nat-of-integer 0 = 0
nat-of-integer 1 = 1
nat-of-integer (numeral k) = numeral k
by(transfer, simp)+

code-identifier

code-module Code-Target-Nat →
(SML) Arith and (OCaml) Arith and (Haskell) Arith
5.3.7. Implementation of natural and integer numbers by target-language integers

theory Code-Target-Numeral
imports Code-Target-Int Code-Target-Nat
begin

end

theory Code-gdb-script
imports Main ../TestLib
begin

datatype gdb-command =
  break string gdb-command
| commands gdb-command
| silent gdb-command
| continue gdb-command
| thread gdb-command
| end gdb-command
| sharp string
| start

datatype gdb-option =
  logging gdb-option
| on
| off
| pagination gdb-option
| file string
| print gdb-option

fun writeFiles - - [] = []
| writeFiles filePath fileExtension (gdb-script :: gdb-script-list) =
ML `< Thy-Load.master-directory @{theory};

fun masterPath-add theory Path = Path
|> Path.explode
|> Path.append (Thy-Load.master-directory theory)
|> Path.implode;

Printing a list of terms in column using Pretty  ML`<

fun pretty-terms' context terms = terms |> (Syntax.pretty-term context
|> List.map)
|> Pretty.chunks;

Pretty.writeln (pretty-terms' @{context} [@{term 2::int}, @{term 2::int}]);

Going from a list of terms to ASCII string  ML`<(*fun render-thm ctxt thm =
Print-Mode.setmp [xsymbols]
(fn - => Display.pretty-thm ctxt thm
|> Pretty.str-of
|> YXML.parse-body
|> XML.content-of) ();
render-thm @{context} @{thm conjI};*)

fun render-term ctxt term =
Print-Mode.setmp [xsymbols]
(fn - => Syntax.pretty-term ctxt term
|> Pretty.str-of
|> YXML.parse-body
|> XML.content-of) ();

render-term @{context} @{term 1::int};

fun render-term-list ctxt term =
Print-Mode.setmp [xsymbols]
(fn - => pretty-terms' ctxt term
|> Pretty.str-of
|> YXML.parse-body
|> XML.content-of) ();
render-term-list @{context} [@{term 1::int}, @{term 1::int}];`

GDB terms script to control scheduler  ML `< val gdb-header =
@{term "#setting gdb options"} $ @ {term ""} $
@{term set} $ @ {term logging (file "Example-sequential.log")} $ @ {term ""} $
@{term set} $ @ {term logging on} $ @ {term ""} $
@{term set} $ @ {term pagination off} $ @ {term ""} $

```
fun gdb-break-point-entry fun-nam-term thread-id-term =
   @ { term "input" } $ @ { term "["" ] } $
   @ { term break } $ @ { term fun-nam-term } $ @ { term "["" ] } $
   @ { term commands } $ @ { term "["" ] } $
   @ { term silent } $ @ { term "["" ] } $
   @ { term thread } $ @ { term thread-id-term } $ @ { term "["" ] } $
   @ { term continue } $ @ { term "["" ] } $
   @ { term end } $ @ { term "["" ] } $ @ { term "["" ] } ;

fun gdb-break-point-exist line-number-term thread-id-term =
   @ { term "input" } $ @ { term "["" ] } $
   @ { term break } $ @ { line-number-term } $ @ { term "["" ] } $
   @ { term commands } $ @ { term "["" ] } $
   @ { term silent } $ @ { term "["" ] } $
   @ { term thread } $ @ { term thread-id-term } $ @ { term "["" ] } $
   @ { term continue } $ @ { term "["" ] } $
   @ { term end } $ @ { term "["" ] } $ @ { term "["" ] } ;

fun gdb-break-main-exit line-number-term thread-id-term =
   @ { term "input" } $ @ { term "["" ] } $
   @ { term break } $ @ { line-number-term } $ @ { term "["" ] } $
   @ { term commands } $ @ { term "["" ] } $
   @ { term silent } $ @ { term "["" ] } $
   @ { term thread } $ @ { term thread-id-term } $ @ { term "["" ] } $
   @ { term continue } $ @ { term "["" ] } $
   @ { term end } $ @ { term "["" ] } $ @ { term "["" ] } ;

fun gdb-break-main-entry fun-nam-term =
   @ { term "input" } $ @ { term "["" ] } $
   @ { term break } $ @ { fun-nam-term } $ @ { term "["" ] } $
   @ { term commands } $ @ { term "["" ] } $
   @ { term silent } $ @ { term "["" ] } $
   @ { term set } $ @ { term "["" ] } $
   @ { term continue } $ @ { term "["" ] } $
   @ { term on } $ @ { term "["" ] } $
   @ { term end } $ @ { term "["" ] } $ @ { term "["" ] } ;

fun gdb-break-main-exit line-number-term thread-id-term =
   @ { term "input" } $ @ { term "["" ] } $
   @ { term break } $ @ { line-number-term } $ @ { term "["" ] } $
   @ { term commands } $ @ { term "["" ] } $
   @ { term silent } $ @ { term "["" ] } $
   @ { term set } $ @ { term "["" ] } $
   @ { term continue } $ @ { term "["" ] } $
   @ { term on } $ @ { term "["" ] } $
   @ { term end } $ @ { term "["" ] } $ @ { term "["" ] } ;

val gdb-start-term = @ { term "input" } $ @ { term "["" ] } $

val gdb-endFile = @ { term "input" } $ @ { term "["" ] } $

ML ""gdb-header"""
removing quotes and parentheses from ASCII string  ML \(\langle\langle\)  
\[
\text{fun remove-char nil = []} \\
\text{remove-char (x::xs) = (if (x = # (orelse x = #)) orelse x = #' \\
\text{then remove-char xs \\
\text{else x::remove-char xs);}}\]
\(\rangle\rangle\)

Jump to the next line  ML \(\langle\langle\)  
\[
\text{fun next-line nil = []} \\
\text{next-line (x::xs) = (if x = #{ \\
\text{then next-line (#\n::xs) \\
\text{else x::next-line xs);}}\]
\(\rangle\rangle\)

Going from a simple list to a list of terms  ML \(\langle\langle\)  
\[
\text{fun thm-to-term thm = thm} \\
\text{concl-of |> HO Logic.dest-Trueprop;} \\
\text{thms-to-terms thms = thms} \\
\text{map} ; \\
\text{fun dest-valid-SE-term terms = terms |> ((fn term => case term of} \\
\text{(Const(@{const-name valid-SE}, -) $ -) \\
\text{Const(@{const-name bind-SE}, -) $ T $ -)) => T} \\
\text{| - => term} \\
\text{map);} \\
\text{fun dest-mbind-term terms = terms |> ((fn term => case term of} \\
\text{Const (@{const-name mbind}, -) $ LIST $ - => LIST} \\
\text{| - => term )} \\
\text{map);} \\
\text{fun dest-mbind-term' terms = terms |> ((fn term => case term of} \\
\text{Const (@{const-name mbind'}, -) $ LIST $ - => LIST} \\
\text{| - => term )} \\
\text{map);} \\
\text{fun dest-List-term terms = terms |> ((fn term => HO Logic.dest-list term) => map);} \\
\text{)}\]

From a test thm to terms of input sequences  ML \(\langle\langle\)  
\[
\text{fun thm-to-inputSeqTerms test-facts = test-facts} \\
\text{thms-to-terms |> dest-valid-SE-term} \\
\text{dest-mbind-term |> dest-List-term;} \\
\text{thm-to-inputSeqTerms' test-facts =} \]

93
from input sequences to strings  ML ⟨⟨ fun inputSeq-to-gdbStrings actTerm-to-gdbTerm inputSeqTerms = inputSeqTerms ⟩⟩

‖ from sequences of strings to a gdb script  ML ⟨⟨ fun gdbStrings-to-gdbScripts gdbStrings = gdbStrings ⟩⟩

‖ concat terms  ML ⟨⟨ fun add-entry-exist-terms [] [] = [] | add-entry-exist-terms terms [] = terms | add-entry-exist-terms [] terms = terms | add-entry-exist-terms (term :: terms) (term' :: terms') = term $ term' :: add-entry-exist-terms terms terms'  ‖

‖ add-entry-exist-termsS [] [] = [] | add-entry-exist-termsS termsS [] = termsS | add-entry-exist-termsS [] termsS = termsS | add-entry-exist-termsS (terms :: termsS) (terms' :: termsS') = add-entry-exist-termsS terms terms' :: add-entry-exist-termsS termsS termsS' | add-entry-exist-termsS' [] [] = [] | add-entry-exist-termsS' termsS [] = termsS | add-entry-exist-termsS' [] termsS = termsS | add-entry-exist-termsS' (terms :: termsS) (terms' :: termsS') = (terms @ terms') :: add-entry-exist-termsS' termsS termsS'  ‖
from thms to gdb scripts  ML

fun thms-to-gdbScripts inputSeq-to-gdbEn inputSeq-to-gdbEx inputSeq-to-gdbMain infos thms =
  thms
  |> thm-to-inputSeqTerms
  |> ((fn terms => inputSeq-to-gdbMain infos terms) |> map)
  |> add-entry-exist-termsS'
    (thms |> thm-to-inputSeqTerms |> ((fn terms => inputSeq-to-gdbEx infos terms) |> map))
  |> add-entry-exist-termsS
    (thms |> thm-to-inputSeqTerms |> ((fn terms => inputSeq-to-gdbEn infos terms) |> map))
  |> inputSeq-to-gdbStrings (fn term => term)
    |> gdbStrings-to-gdbScripts;

fun thms-to-gdbScripts' inputSeq-to-gdbEn inputSeq-to-gdbEx inputSeq-to-gdbMain infos thms =
  thms
  |> thm-to-inputSeqTerms'
  |> ((fn terms => inputSeq-to-gdbMain infos terms) |> map)
  |> add-entry-exist-termsS'
    (thms |> thm-to-inputSeqTerms' |> ((fn terms => inputSeq-to-gdbEx infos terms) |> map))
  |> add-entry-exist-termsS
    (thms |> thm-to-inputSeqTerms' |> ((fn terms => inputSeq-to-gdbEn infos terms) |> map))
  |> inputSeq-to-gdbStrings (fn term => term)
    |> gdbStrings-to-gdbScripts;

isa markup  ML

fun gen-gdb-scripts
  inputSeq-to-gdbEn inputSeq-to-gdbEx inputSeq-to-gdbMain infos theory path thms =
  thms
  |> thms-to-gdbScripts inputSeq-to-gdbEn inputSeq-to-gdbEx inputSeq-to-gdbMain infos
  |> writeFiles (path |> masterPath-add theory) .gdb;

(*For mbind')
fun gen-gdb-scripts'
  inputSeq-to-gdbEn inputSeq-to-gdbEx inputSeq-to-gdbMain infos theory path thms =
  thms
  |> thms-to-gdbScripts' inputSeq-to-gdbEn inputSeq-to-gdbEx inputSeq-to-gdbMain infos
  |> writeFiles (path |> masterPath-add theory) .gdb;

(* val _ = Outer-Syntax.command
  @{command-spec gen-gdb-script}
  store test state (theorem)
  ;(*)

(*For mbind*)

(*val gen-gdb-script = @{thms mykeos-simple.test-data}
  |> thm-to-inputSeqTerms

95
theory MyKeOS-test-conc
imports MyKeOS

～~/src/HOL/Library/Code-Target-Numeral
..../..~/src/codegen-gdb/Code-gdb-script

begin

declare [[testgen-profiling]]

Interleaving

The purpose of this example is to model system calls that consists of a number of (internal) atomic actions; the global behavior is presented by the interleaving of the actions actions

\[ \text{definition } \text{syscall } \text{tid } \text{thid } \text{m } \text{m}' = [\text{alloc } \text{tid } \text{thid } \text{m}, \text{release } \text{tid } \text{thid } \text{m}', \text{status } \text{tid } \text{thid}] \]

\[ \text{value } \text{interleave } (\text{syscall } 5 \ 0 \ \text{m } \text{m}') (\text{syscall } 5 \ 1 \ \text{m } \text{m}') \]

In the following, we do a predicate abstraction on the interleace language, leading to an automaton represented as a set of rewrites ...

fun \text{Interleave} :: in-c list \Rightarrow nat \times nat \Rightarrow nat \Rightarrow nat \Rightarrow nat \Rightarrow bool (infixl \bowtie100)

where \quad S \bowtie (a, b) = (\lambda \text{tid } m \ m' m'' m'''. (S \in \text{interleave} (\text{drop a (syscall } \text{tid } 0 \ \text{m } \text{m}'))) 

\quad (\text{drop b (syscall } \text{tid } 1 \ m'' m''')))

lemma \text{init-Interleave} : (S \bowtie (0, 0)) \text{tid } m \ m' m'' m''' = 

\quad (S \in \text{interleave} (\text{syscall } \text{tid } 0 \ \text{m } \text{m}')) 

\quad (\text{syscall } \text{tid } 1 \ m'' m'''))

by simp

value \text{interleave} (\text{syscall } 5 \ 0 \ \text{m } \text{m}') (\text{syscall } 5 \ 1 \ \text{m } \text{m}')

find-theorems name:Interleave

lemma \text{ref-mt} [simp]: \neg [(\bowtie (0, 0)) \text{tid } m \ m' m'' m''']

by (simp add: syscall-def)

lemma \text{ref-0-0} [simp]: \neg ((\text{status } a \ b) \# R) \bowtie (0, 0)) \text{tid } m \ m' m'' m'''

by (simp add: syscall-def)

lemma \text{ref-1-0} [simp]: \neg ((\text{status } a \ b) \# R) \bowtie (1, 0)) \text{tid } m \ m' m'' m'''

by (simp add: syscall-def)

lemma \text{ref-0-1} [simp]: \neg ((\text{status } a \ b) \# R) \bowtie (0, 1)) \text{tid } m \ m' m'' m'''

by (simp add: syscall-def)

lemma \text{ref-1-1} [simp]: \neg ((\text{status } a \ b) \# R) \bowtie (1, 1)) \text{tid } m \ m' m'' m'''

by (simp add: syscall-def)

lemma \text{ref-3-1} [simp]: \neg ((\text{status } a \ b) \# R) \bowtie (3, 1)) \text{tid } m \ m' m'' m'''

96
by (simp add: syscall-def)
lemma ref-1-3 [simp]: \neg((\text{status } a \ b) \neq R) \Rightarrow (1, 3)) \; \text{tid } m \ m'' \ m'''
by (simp add: syscall-def)

lemma trans-0-0 [simp]: (((a \neq R) \Rightarrow (0, 0)) \; \text{tid } m \ m'' \ m''') = 
((a = \text{alloc } tid 0 \; \text{m} \wedge (R \Rightarrow (1, 0)) \; \text{tid } m \ m'' \ m''') \vee 
(a = \text{alloc } tid 1 \; m'' \wedge (R \Rightarrow (0, 1)) \; \text{tid } m \ m'' \ m'''))
by (simp add: syscall-def, rule iffI, metis, metis)

lemma trans-1-0 [simp]: (((a \neq R) \Rightarrow (1, 0)) \; \text{tid } m \ m'' \ m''') = 
((a = \text{release } tid 0 \; m' \wedge (R \Rightarrow (2, 0)) \; \text{tid } m \ m'' \ m''') \vee 
(a = \text{alloc } tid 1 \; m'' \wedge (R \Rightarrow (1, 1)) \; \text{tid } m \ m'' \ m'''))
by (simp add: syscall-def, rule iffI, metis, metis)

lemma trans-2-0 [simp]: (((a \neq R) \Rightarrow (2, 0)) \; \text{tid } m \ m'' \ m''') = 
((a = \text{status } tid 0 \wedge R = [\text{alloc } tid 1 \; m'', \text{release } tid 1 \; m'', \text{status } tid 1]) \vee 
(a = \text{alloc } tid 1 \; m'' \wedge (R \Rightarrow (2, 1)) \; \text{tid } m \ m'' \ m'''))
by (simp add: syscall-def, rule iffI, metis, metis)

lemma trans-2-1 [simp]: (((a \neq R) \Rightarrow (2, 1)) \; \text{tid } m \ m'' \ m''') = 
((a = \text{status } tid 0 \wedge R = [\text{release } tid 1 \; m'', \text{status } tid 1]) \vee 
(a = \text{release } tid 1 \; m'' \wedge (R \Rightarrow (2, 2)) \; \text{tid } m \ m'' \ m'''))
by (simp add: syscall-def, rule iffI, metis, metis)

lemma trans-2-2 [simp]: (((a \neq R) \Rightarrow (2, 2)) \; \text{tid } m \ m'' \ m''') = 
((a = \text{status } tid 0 \wedge R = [\text{status } tid 1]) \vee 
(a = \text{status } tid 1 \wedge R = [\text{status } tid 0]))
by (simp add: syscall-def, rule iffI, metis, metis)

value interleave (drop 0 (syscall tid 0 \; m \; m'))(drop 0 (syscall tid 1 \; m'' \; m'''))

TestData Hack:

lemma PO-norm0 [simp]: PO True by(simp add: PO-def)

The following scenario is meant to describe the symbolic execution step by step.

declare Monads.mbind' -bind [simp def]

find-theorems mbindF\text{FailStop} []

lemma example-symbolic-execution-simulation :
assumes H: S = [\text{alloc } tid 1 \; m'', \text{release } tid 0 \; m', \text{release } tid 1 \; m'', \text{status } tid 1]
assumes SE: σ₀ |= (s \leftarrow mbindF\text{FailStop} S SYS; return (x = s))
shows P
apply(insert SE H)
apply(hypsubst)
apply(tactic ematch-tac [\{thm status.exec-mbindFStop-E\}, \{thm release.exec-mbindFStop-E\},
\{thm alloc.exec-mbindFStop-E\}] 1)
apply(tactic ematch-tac [\{thm status.exec-mbindFStop-E\}, \{thm release.exec-mbindFStop-E\},
\{thm alloc.exec-mbindFStop-E\}] 1)
apply(tactic ematch-tac [\{thm status.exec-mbindFStop-E\}, \{thm release.exec-mbindFStop-E\},
\{thm alloc.exec-mbindFStop-E\}] 1)
apply(tactic ematch-tac [\{thm status.exec-mbindFStop-E\}, \{thm release.exec-mbindFStop-E\},
\{thm alloc.exec-mbindFStop-E\}] 1)

97
apply {tactic ematch-tac [@[thm status.exec-mbindFStop-E], @[thm release.exec-mbindFStop-E], @[thm alloc.exec-mbindFStop-E], @[thm valid-mbind΄-mtE] ] I}
apply simp
oops

lemma
assumes valid: (σ |= ( s ← mbind_FAILSave (alloc c no m ≠ S) SYS; unitSE (P s)))
and case1: (c, no) ∈ dom σ ⇒
  σ((c, no) ↦ the (σ (c, no)) + int m) |=
  (s ← mbind_FAILSave S SYS; unitSE (P (alloc-ok # s)) ⇒ Q
and case2: (c, no) /∈ dom σ ⇒ σ |= unitSE (P []) ⇒ Q
shows Q
apply (insert assms)
apply (erule MyKeOS.alloc.exec-mbindFSave-E)
apply metis
by metis

thm MyKeOS.alloc.exec-mbindFSave-E

ML ⟨⟨ hyp-subst-tac ⟩⟩
test-spec test-status:
assumes account-defined: (tid,0) ∈ dom σ₀ ∧ (tid,1) ∈ dom σ₀
and test-purpose : S ∈ interleave (syscall tid 0 m m' (syscall tid 1 m'' m'''))
and sym-exec-spec :
  σ₀ |= (s ← mbind_FAILStop S SYS; return (x = s))
shows σ₀ |= (s ← mbind_FAILStop S PUT; return (s = x))
apply(insert account-defined test-purpose sym-exec-spec)
apply(frule length-interleave)
apply(simp add: syscall-def)
apply(tactic TestGen.mp.fy 1,rule-tac x=x in spec[OF allI])
  just case elaboration of test-cases
apply (clarify, elim disjE)
apply (tactic ALLGOALS(hyp-subst-tac @[context]))
symbolic execution
apply(tactic ALLGOALS(TestGen.REPEAT"(ematch-tac [@[thm status.exec-mbindFStop-E],
  @[thm release.exec-mbindFStop-E],
  @[thm alloc.exec-mbindFStop-E],
  @[thm valid-mbind΄-mtE] ]))
  elimination of infeasible executions
apply(simp-all)
apply(tactic ALLGOALS(hyp-subst-tac @[context]))
apply(tactic ALLGOALS(TestGen.uniformityf-tac @[context] [PUT]))

mk-test-suite mykos-interleave

declare [[testgen-iterations=0]]
declare [[testgen-SMT]]
gen-test-data mykeos-interleave
thm mykeos-interleave.concrete-tests
gen-test-script mykeos-interleave
thm mykeos-interleave.test-script

Generation of a gdb file  ML ⟨⟨ (*building the gdb term*)
fun actTerm-to-gdbTerm (Const(C{const-name alloc}, typ) $ - $ B $ -) =
gdb-break-point-entry (Const(C{const-name alloc}, typ)) B $
gdb-break-point-exist C{term 0} B
| actTerm-to-gdbTerm (Const(C{const-name release}, typ) $ - $ B $ -) =
gdb-break-point-entry (Const(C{const-name release}, typ)) B $
gdb-break-point-exist C{term 0} B
| actTerm-to-gdbTerm (Const(C{const-name status}, typ) $ - $ B) =
gdb-break-point-entry (Const(C{const-name status}, typ)) B $
gdb-break-point-exist C{term 0} B
| actTerm-to-gdbTerm (Const(C{const-name end}, -) $ -) =
gdb-start-term $ gdb-endFile;
⟩⟩

ML ⟨⟨
⟩⟩

ML ⟨⟨
type info-threads-configure = {input-type : typ, get-task-id : term -> int, (* precond: term must be of type typ*) get-thread-id : term -> int, (* precond: term must be of type typ*) config-atomic-actions : (string -> int * int) -> unit, set-break-main : (int * int) -> unit};
⟩⟩

(* So, the package gdb-script-generator can provide a function: *)

fun generate-gdb-script-config (X: info-threads-configure) (testenv: string) (* in this case: mykeos-interleave *)
    = []: (string * int * int) list

⟩⟩

ML ⟨⟨ @{thms mykeos-interleave.concrete-tests}}⟩⟩
ML

val thread-info1 = {task-id = 5 , th-id = 1 , order = 2,
    break-alloc = (50, 59), break-release= (59, 61), break-status = (61, 63),
    break-main = (123, 137)};

val thread-info12 = {task-id = 5 , th-id = 0 , order = 3,
    break-alloc = (68, 77), break-release= (77, 79), break-status = (79, 81),
    break-main = (123, 137)};

val thread-info2 = {task-id = 3 , th-id = 1 , order = 2,
    break-alloc = (94, 96), break-release= (96, 98), break-status = (98, 100),
    break-main = (123, 137)};

val thread-info31 = {task-id = 3 , th-id = 0 , order = 3,
    break-alloc = (112, 114), break-release= (114, 116), break-status = (116, 118),
    break-main = (123, 137)};

val needed-informations = [thread-info1, thread-info12, thread-info2, thread-info31, thread-info2, thread-info31]


ML

fun check-identity (info:info-threads) task-id th-id =
  (task-id > Const(100)) andalso (th-id > Const(100))
  (info |> #task-id) = (task-id |> HOLogic.dest-number) andalso
  (info |> #th-id) = (th-id |> HOLogic.dest-number)

fun get-successor-order [] - = 99 |> mk-number @(typ int)

| get-successor-order ((info:info-threads)::infos) (Const@[{const-name alloc}, typ]) $ task-id $ th-id $ value =
  if check-identity info task-id th-id
  then info |> #order |> mk-number @(typ int)
  else get-successor-order infos (Const@[{const-name alloc}, typ]) $ task-id $ th-id $ value

| get-successor-order ((info:info-threads)::infos) (Const@[{const-name release}, typ]) $ task-id $ th-id $ value =
  if check-identity info task-id th-id
  then info |> #order |> mk-number @(typ int)
  else get-successor-order infos (Const@[{const-name release}, typ]) $ task-id $ th-id $ value

| get-successor-order ((info:info-threads)::infos) (Const@[{const-name status}, typ]) $ task-id $ th-id =
  if check-identity info task-id th-id
  then info |> #order |> mk-number @(typ int)
  else get-successor-order infos (Const@[{const-name status}, typ]) $ task-id $ th-id

| get-successor-order - (- $ - $ - $ - $ - $ - $ - $ - $ term11 $ - $ - $ - $ - $ - $ - $ - $ - )
  = term11;

fun inputSeq-to-gdbEn [] [] = []
inputSeq-to-gdbEn - [] = []
inputSeq-to-gdbEn [] terms = terms

inputSeq-to-gdbEn ((info::info-threads):::infos)
  ([(Const(®{const-name alloc}, typ) $ task-id $ th-id $ value)]) =
   if check-identity info task-id th-id
   then gdb-break-point-entry (info $ #break-alloc $ fst $ mk-number @{typ int})
   (info $ #order $ mk-number @{typ int}) :: []
   else inputSeq-to-gdbEn infos [(Const(®{const-name alloc}, typ) $ task-id $ th-id $ value)]

inputSeq-to-gdbEn ((info::info-threads):::infos)
  ([(Const(®{const-name release}, typ) $ task-id $ th-id $ value)]) =
   if check-identity info task-id th-id
   then gdb-break-point-entry (info $ #break-release $ fst $ mk-number @{typ int})
   (info $ #order $ mk-number @{typ int}) :: []
   else inputSeq-to-gdbEn infos [(Const(®{const-name release}, typ) $ task-id $ th-id $ value)]

inputSeq-to-gdbEn ((info::info-threads):::infos)
  ([(Const(®{const-name status}, typ) $ task-id $ th-id)]) =
   if check-identity info task-id th-id
   then gdb-break-point-entry (info $ #break-status $ fst $ mk-number @{typ int})
   (info $ #order $ mk-number @{typ int}) :: []
   else inputSeq-to-gdbEn infos [(Const(®{const-name status}, typ) $ task-id $ th-id)]

inputSeq-to-gdbEn ((info::info-threads):::infos)
  ([(Const(®{const-name alloc}, typ) $ task-id $ th-id $ value):::terms) =
   if check-identity info task-id th-id
   then (gdb-break-point-entry (info $ #break-alloc $ fst $ mk-number @{typ int})
   (info $ #order $ mk-number @{typ int}):: terms
   else inputSeq-to-gdbEn (info:::infos) terms
inputSeq-to-gdbEn ((info:::infos)
  ([(Const(®{const-name alloc}, typ) $ task-id $ th-id $ value)):::terms)

inputSeq-to-gdbEn ((info::info-threads):::infos)
  ([(Const(®{const-name release}, typ) $ task-id $ th-id $ value):::terms) =
   if check-identity info task-id th-id
   then (gdb-break-point-entry (info $ #break-release $ fst $ mk-number @{typ int})
   (info $ #order $ mk-number @{typ int}):: terms
   else inputSeq-to-gdbEn (info:::infos) terms
inputSeq-to-gdbEn (info:::infos)
  ([(Const(®{const-name release}, typ) $ task-id $ th-id $ value)):::terms)

inputSeq-to-gdbEn ((info::info-threads):::infos)
  ([(Const(®{const-name status}, typ) $ task-id $ th-id):::terms) =
   if check-identity info task-id th-id
   then (gdb-break-point-entry (info $ #break-status $ fst $ mk-number @{typ int})
   (info $ #order $ mk-number @{typ int}):: terms
   else inputSeq-to-gdbEn (info:::infos) terms
inputSeq-to-gdbEn (info:::infos)
  ([(Const(®{const-name status}, typ) $ task-id $ th-id)):::terms)
fun inputSeq-to-gdbEx ([ ] [ ] = [])
| inputSeq-to-gdbEx - [] = []
| inputSeq-to-gdbEx [] terms = terms

| inputSeq-to-gdbEx ((info::info-threads)::infos)
  ([([Const(@{const-name alloc}, typ) $ task-id $ th-id $ value])]) =
  if check-identity info task-id th-id
  then gdb-break-point-exist (info | > #break-alloc | > snd | > mk-number @{typ int})
  (info | > #order | > mk-number @{typ int}): []
  else inputSeq-to-gdbEx infos ((Const(@{const-name alloc}, typ) $ task-id $ th-id $ value))

| inputSeq-to-gdbEx ((info::info-threads)::infos)
  ([([Const(@{const-name release}, typ) $ task-id $ th-id $ value])]) =
  if check-identity info task-id th-id
  then gdb-break-point-exist (info | > #break-release | > snd | > mk-number @{typ int})
  (info | > #order | > mk-number @{typ int}): []
  else inputSeq-to-gdbEx infos ((Const(@{const-name release}, typ) $ task-id $ th-id $ value))

| inputSeq-to-gdbEx ((info::info-threads)::infos)
  ([([Const(@{const-name status}, typ) $ task-id $ th-id])]) =
  if check-identity info task-id th-id
  then gdb-break-point-exist (info | > #break-status | > snd | > mk-number @{typ int})
  (info | > #order | > mk-number @{typ int}): []
  else inputSeq-to-gdbEx infos ((Const(@{const-name status}, typ) $ task-id $ th-id)]

| inputSeq-to-gdbEx ((info::info-threads)::infos)
  ([([Const(@{const-name alloc}, typ) $ task-id $ th-id $ value]):terms) =
  if check-identity info task-id th-id
  then gdb-break-point-exist (info | > #break-alloc | > snd | > mk-number @{typ int})
  (if terms = []
  then (info | > #order | > mk-number @{typ int})
  else get-successor-order (info::infos)
  (hd terms):[])
  else inputSeq-to-gdbEx (info::infos) terms
  (info::info-threads)::infos)}

| inputSeq-to-gdbEx ((info::info-threads)::infos)

102
fun add-gdb-main [] terms = terms

  | add-gdb-main ((info:info-threads)::infos)
    (Const(@{const-name alloc}, typ) $ task-id $ th-id $ value::terms) =
    if check-identity info task-id th-id
    then gdb-break-main-entry (info | #break-main | fst | mk-number @{typ int}$)
    gdb-break-main-exit (info | #break-main | snd | mk-number @{typ int})
    else add-gdb-main infos (Const(@{const-name alloc}, typ) $ task-id $ th-id $ value::terms)

  | add-gdb-main ((info:info-threads)::infos)
    (Const(@{const-name release}, typ) $ task-id $ th-id $ value::terms) =
    if check-identity info task-id th-id
    then gdb-break-main-entry (info | #break-main | fst | mk-number @{typ int}$)
    gdb-break-main-exit (info | #break-main | snd | mk-number @{typ int})
    else add-gdb-main infos (Const(@{const-name release}, typ) $ task-id $ th-id $ value::terms)
(info | #order | mk-number @{typ int})]
else add-gdb-main infos (Const(@{const-name release}, typ) $ task-id $ th-id $ value::terms)

| add-gdb-main ((info::info-threads):::infos)
| (Const(@{const-name status}, typ) $ task-id $ th-id::terms) =
| if check-identity info task-id th-id
| then
| [gdb-break-main-entry (info | #break-main | fst | mk-number @{typ int})]
| gdb-break-main-exit (info | #break-main | snd | mk-number @{typ int})
| (info | #order | mk-number @{typ int})]
| else add-gdb-main infos (Const(@{const-name status}, typ) $ task-id $ th-id::terms)

| add-gdb-main - - = [];

⟩⟩
ML ⟨⟨

gen-gdb-scripts'
inputSeq-to-gdbEn inputSeq-to-gdbEx add-gdb-main needed-informations
@{theory} impl/c-conc/MyKeOS (@{thms mykeos-interleave.concrete-tests});
⟩⟩

Experimental Space

declare[[testgen-trace]]

Code Generation Setup For Concurrent Scenario

Generation of an SML file to put datatypes  definition program-dum-conc
::(int × int ⇒ int option) ⇒ in-c ⇒ out-c ⇒ (int × int ⇒ int option)
where  program-dum-conc σ a outs = [(0, 0)⇒ 0]

export-code program-dum-conc in SML
module-name Datatypes file impl/c-conc/datatypes.sml

Code Setup for Datatypes  code-printing
type-constructor in-c => (SML) Datatypes.in’-c
  | constant alloc => (SML) !(Datatypes.Alloc (-, -, -))
  | constant release => (SML) !(Datatypes.Release (-, -, -))
  | constant status => (SML) !(Datatypes.Status (-, -))

code-printing
type-constructor out-c => (SML) Datatypes.out’-c
  | constant alloc-ok => (SML) Datatypes.Alloc’-ok
  | constant release-ok => (SML) Datatypes.Release’-ok
  | constant status-ok => (SML) !(Datatypes.Status’-ok (-, -))

code-printing
type-constructor int =>

104
(SML) Datatypes.int
| constant int-of-integer =>
(SML) !(Datatypes.Int'-of'-integer (-))

code-printing
  type-constructor nat =>
  (SML) Datatypes.nat
  | constant Nat => (SML) !(Datatypes.Nat (-))

HOL to SML adapter

Constant definitions: stubs  consts MyKeOS-conc1:: int ⇒ int ⇒ int ⇒ (int, 'σ)MONSE

Conversion: Integer to Action Output  fun  my-nat-conv :: int ⇒ nat
  where  my-nat-conv x = (if x <= 0 then 0 else Suc (my-nat-conv(x - 1)))

fun stubs-to-out-conc::in-c ⇒ (int × int ⇒ int option) ⇒ int ⇒ out-c
  where  stubs-to-out-conc (alloc task-id th-id res) σ σ-impl =
  (if (σ-impl = (plus ((the o σ)(task-id,th-id)) (int res)))
   then alloc-ok
   else release-ok)
  | stubs-to-out-conc (release task-id th-id res) σ σ-impl =
  (if (σ-impl = (minus ((the o σ)(task-id,th-id)) (int res)))
   then release-ok
   else alloc-ok)
  | stubs-to-out-conc (status task-id th-id) σ σ-impl =
  (if (σ-impl = ((the o σ)(task-id,th-id)))
   then status-ok (my-nat-conv σ-impl)
   else release-ok)

fun mykeAdapter-con::in-c ⇒ (int × int ⇒ int option) ⇒ (out-c × (int × int ⇒ int option))
  option
  where  mykeAdapter-con (alloc task-id th-id res) σ =
  (out ← MyKeOS-conc1 task-id th-id res;
   return(stubs-to-out-conc (alloc task-id th-id res) σ
   ((fst o the) (MyKeOS-conc1 task-id th-id res σ)))); σ)
  | mykeAdapter-con (release task-id th-id res) σ =
  (out ← MyKeOS-conc1 task-id th-id res;
   return(stubs-to-out-conc (alloc task-id th-id res) σ
   ((fst o the) (MyKeOS-conc1 task-id th-id res σ)))); σ)
  | mykeAdapter-con (status task-id th-id) σ =
  (out ← MyKeOS-conc1 task-id th-id (the (σ (task-id, th-id)));
   return(status-ok ((my-nat-conv o the) (σ (task-id, th-id)))); σ)
Serialisation: semantics of conc stubs  code-printing
  constant MyKeOS-conc1 => (SML)! (MyKeOSAdapter.get'-state ( - ) ( - ) ( - ) ( - ))

export-code  mykeAdapter-con in SML
  module-name MykeAdapter file impl/c-conc/mykeAdapter.sml

Serialisation: semantics of SUT  code-printing
  constant PUT => (SML)! (MykeAdapter.mykeAdapter'-con ( - ) ( - ))

export-code  mykeos-interleave.test-script in SML
  module-name TestScript file  impl/c-conc/mykeos-test-script.sml

end
A. Glossary

Abstract test data: In contrast to pure ground terms over constants (like integers 1, 2, 3, or lists over them, or strings ...) abstract test data contain arbitrary predicate symbols (like triangle 3 4 5).

Regression testing: Repeating of tests after addition/bug fixes have been introduced into the code and checking that behavior of unchanged portions has not changed.

Stub: Stubs are “simulated” implementations of functions, they are used to simulate functionality that does not yet exist or cannot be run in the test environment.

Test case: An abstract test stimuli that tests some aspects of the implementation and validates the result.

Test case generation: For each operation the pre/postcondition relation is divided into sub-relations. It assumes that all members of a sub-relation lead to a similar behavior of the implementation.

Test data: One or more representative for a given test case.

Test data generation (Test data selection): For each test case (at least) one representative is chosen so that coverage of all test cases is achieved. From the resulting test data, test input data processable by the implementation is extracted.

Test execution: The implementation is run with the selected test input data in order to determine the test output data.

Test executable: An executable program that consists of a test harness, the test script and the program under test. The Test executable executes the test and writes a test trace documenting the events and the outcome of the test.

Test harness: When doing unit testing the program under test is not a runnable program in itself. The test harness or test driver is a main program that initiates test calls (controlled by the test script), i.e. drives the method under test and constitutes a test executable together with the test script and the program under test.

Test hypothesis: The hypothesis underlying a test that makes a successful test equivalent to the validity of the tested property, the test specification. The current implementation of HOL-TestGen only supports uniformity and regularity hypotheses, which are generated “on-the-fly” according to certain parameters given by the user like depth and breadth.

Test specification: The property the program under test is required to have.
**Test result verification:** The pair of input/output data is checked against the specification of the test case.

**Test script:** The test program containing the control logic that drives the test using the test harness. HOL-TestGen can automatically generate the test script for you based on the generated test data.

**Test theorem:** The test data together with the test hypothesis will imply the test specification. HOL-TestGen conservatively computes a theorem of this form that relates testing explicitly with verification.

**Test trace:** Output made by a test executable.
Bibliography


[23] sml.net.


Index

abstract test data, 107
breath, 107
(breadth), 13
⟨clasimpmod⟩, 13
data separation lemma, 13
depth, 107
⟨depth⟩, 13
export_test_data (command), 15
gen_test_cases (method), 13
gen_test_data (command), 14
generate_test_script (command), 15

higher-order logic, see HOL

HOL, 7

Isabelle, 6, 7, 9
Main (theory), 11
mk_test_suite (command), 13
⟨name⟩, 13

program under test, 13, 15
random solver, 14
regression testing, 107
regularity hypothesis, 13

SML, 7
software
testing, 5
validation, 5
verification, 5

Standard ML, see SML
stub, 107

test, 6
test (attribute), 16
test specification, 11
test theorem, 13
test case, 11
test data generation, 11
test executable, 11
test specification, 6
test case, 5, 107
test case generation, 5, 11, 13, 17, 107
test data, 5, 11, 14, 107
test data generation, 5, 107
test data selection, see test data generation

regression testing, 107
regularity hypothesis, 13

unit test
specification-based, 5