Clean - An Abstract Imperative Programming Language and its Theory

Frédéric Tuong Burkhart Wolff
(with Contributions by Chantal Keller)

October 4, 2019

LRI, Univ. Paris-Sud, CNRS, Université Paris-Saclay
bât. 650 Ada Lovelace, 91405 Orsay, France

Abstract

Clean is based on a simple, abstract execution model for an imperative target language. "Abstract" is understood as contrast to "Concrete Semantics"; alternatively, the term "shallow-style embedding" could be used. It strives for a type-safe notation of program-variables, an incremental construction of the typed state-space, support of incremental verification, and open-world extensibility of new type definitions being intertwined with the program definitions.

Clean is based on a "no-frills" state-exception monad with the usual definitions of *bind* and *unit* for the compositional glue of state-based computations. Clean offers conditionals and loops supporting C-like control-flow operators such as *break* and *return*. The state-space construction is based on the extensible record package. Direct recursion of procedures is supported.

Clean's design strives for extreme simplicity. It is geared towards symbolic execution and proven correct verification tools. The underlying libraries of this package, however, deliberately restrict themselves to the most elementary infrastructure for these tasks. The package is intended to serve as demonstrator semantic backend for Isabelle/C [5], or for the test-generation techniques described in [4].

Contents

1 The Clean Language

theory *Clean*
  imports *Symbex-MonadSE*
  keywords *global-vars local-vars-test* :: *thy-decl*
    and *returns pre post local-vars variant*
    and *function-spec* :: *thy-decl*
    and *rec-function-spec* :: *thy-decl*

begin

Clean (pronounced as: “C lean” or “Céline” [selin]) is a minimalistic imperative language with C-like control-flow operators based on a shallow embedding into the “State Exception Monads” theory formalized in `MonadSE.thy`. It strives for a type-safe notation of program-variables, an incremental construction of the typed state-space in order to facilitate incremental verification and open-world extensibility to new type definitions intertwined with the program definition.

It comprises:

- C-like control flow with *break* and *return*,
- global variables,
- function calls (seen as monadic executions) with side-effects, recursion and local variables,
- parameters are modeled via functional abstractions (functions are monads); a passing of parameters to local variables might be added later,
- direct recursive function calls,
- cartouche syntax for λ-lifted update operations supporting global and local variables.

Note that Clean in its current version is restricted to *monomorphic* global and local variables as well as function parameters. This limitation will be overcome at a later stage. The construction in itself, however, is deeply based on parametric polymorphism (enabling structured proofs over extensible records as used in languages of the ML family http://www.cs.ioc.ee/tfp-icfp-gpce05/tfp-proc/21num.pdf and Haskell https://www.schoolofhaskell.com/user/fumieval/extensible-records).

1.1 A High-level Description of the Clean Memory Model

1.1.1 A Simple Typed Memory Model of Clean: An Introduction

Clean is based on a "no-frills" state-exception monad **type-synonym** $('o, '\sigma)\ MON_{SE} = \langle '\sigma \rightharpoonup ('o \times '\sigma)\rangle$ with the usual definitions of *bind* and *unit*. In this language, sequence operators, conditionals and loops can be integrated.

From a concrete program, the underlying state $'\sigma$ is *incrementally* constructed by a sequence of extensible record definitions:

1. Initially, an internal control state is defined to give semantics to *break* and *return* statements:

   ```
   record control_state = break_val :: bool return_val :: bool
   ```

   control-state represents the σ_0 state.
2. Any global variable definition block with definitions $a_1 : \tau_1 \ldots a_n : \tau_n$ is translated into a record extension:

   ```
   record σ_{n+1} = σ_n + a_1 :: τ_1; ...; a_n :: τ_n
   ```

3. Any local variable definition block (as part of a procedure declaration) with definitions $a_1 : \tau_1 \ldots a_n : \tau_n$ is translated into the record extension:

   ```
   record σ_{n+1} = σ_n + a_1 :: τ_1 list; ...; a_n :: τ_n list; result :: τ_{result-type} list;
   ```

   where the - *list*-lifting is used to model a *stack* of local variable instances in case of direct recursions and the *result-value* used for the value of the *return* statement.

The **record** package creates an $'\sigma$ extensible record type $'\sigma$ *control-state-ext* where the $'\sigma$ stands for extensions that are subsequently "stuffed" in them. Furthermore, it generates definitions for the constructor, accessor and update functions and automatically derives a number of theorems over them (e.g., "updates on different fields commute", "accessors on a record are surjective", "accessors yield the value of the last update"). The collection of these theorems constitutes the *memory model* of Clean, providing an incrementally extensible state-space for global and local program variables. In contrast to axiomatizations of memory models, our generated state-spaces might be "wrong" in the sense that they do not reflect the operational behaviour of a particular compiler or a sufficiently large portion of the C language; however, it is by construction *logically consistent* since it is impossible to derive falsity from the entire set of conservative extension schemes used in their construction. A particular advantage of the incremental state-space construction is that it supports incremental verification and interleaving of program definitions with theory development.

1.1.2 Formally Modeling Control-States

The control state is the "root" of all extensions for local and global variable spaces in Clean. It contains just the information of the current control-flow: a *break* occurred (meaning all commands till the end of the control block will be skipped) or a *return* occurred (meaning all commands till the end of the current function body will be skipped).

record *control-state* =
    break-status :: *bool*
    return-status :: *bool*

definition *break* :: $(unit, ('\sigma\text{-}ext)\ control\text{-}state\text{-}ext)\ MON_{SE}$
  where $break \equiv (\lambda\ \sigma.\ Some((), \sigma\ (\!|\ break\text{-}status := True\ |\!)))$

definition *unset-break-status* :: $(unit, ('\sigma\text{-}ext)\ control\text{-}state\text{-}ext)\ MON_{SE}$
  where $unset\text{-}break\text{-}status \equiv (\lambda\ \sigma.\ Some((), \sigma\ (\!|\ break\text{-}status := False\ |\!)))$

definition *set-return-status* :: $(unit, ('\sigma\text{-}ext)\ control\text{-}state\text{-}ext)\ MON_{SE}$
  where $set\text{-}return\text{-}status = (\lambda\ \sigma.\ Some((), \sigma\ (\!|\ return\text{-}status := True\ |\!)))$

definition *unset-return-status* :: $(unit, ('\sigma\text{-}ext)\ control\text{-}state\text{-}ext)\ MON_{SE}$
  where $unset\text{-}return\text{-}status = (\lambda\ \sigma.\ Some((), \sigma\ (\!|\ return\text{-}status := False\ |\!)))$

definition *exec-stop* :: $('\sigma\text{-}ext)\ control\text{-}state\text{-}ext \Rightarrow bool$
  where $exec\text{-}stop = (\lambda\ \sigma.\ break\text{-}status\ \sigma \lor return\text{-}status\ \sigma\ )$

lemma *exec-stop1* [*simp*] : $break\text{-}status\ \sigma \Longrightarrow exec\text{-}stop\ \sigma$
  unfolding *exec-stop-def* **by** *simp*

lemma *exec-stop2* [*simp*] : $return\text{-}status\ \sigma \Longrightarrow exec\text{-}stop\ \sigma$
  unfolding *exec-stop-def* **by** *simp*

On the basis of the control-state, assignments, conditionals and loops are reformulated into *break*-aware and *return*-aware versions as shown in the definitions of *assign* and *if-C* (in this theory file, see below).

For Reasoning over Clean programs, we need the notion of independance of an update from the control-block:

definition *control-independence* ::
$$(('b \Rightarrow 'b) \Rightarrow 'a\ control\text{-}state\text{-}scheme \Rightarrow\ 'a\ control\text{-}state\text{-}scheme) \Rightarrow bool \qquad (\sharp)$$
where $\sharp\ upd \equiv (\forall \sigma\ \ T\ b.\ break\text{-}status\ (upd\ T\ \sigma) = break\text{-}status\ \sigma$
$\land\ return\text{-}status\ (upd\ T\ \sigma) = return\text{-}status\ \sigma$
$\land\ upd\ T\ (\sigma(\!|\ return\text{-}status := b\ |\!)) = (upd\ T\ \sigma)(\!|\ return\text{-}status := b\ |\!)$
$\land\ upd\ T\ (\sigma(\!|\ break\text{-}status := b\ |\!)) = (upd\ T\ \sigma)(\!|\ break\text{-}status := b\ |\!))$

lemma *exec-stop-vs-control-independence* [*simp*]:
  $\sharp\ upd \Longrightarrow exec\text{-}stop\ (upd\ f\ \sigma) = exec\text{-}stop\ \sigma$
  unfolding *control-independence-def exec-stop-def* **by** *simp*

lemma *exec-stop-vs-control-independence'* [*simp*]:
$\sharp$ *upd* $\Longrightarrow$ (*upd f* (σ (| *return-status* := *b* |))) = (*upd f* σ)(| *return-status* := *b* |)
unfolding *control-independence-def exec-stop-def* **by** *simp*

lemma *exec-stop-vs-control-independence''* [*simp*]:
$\sharp$ *upd* $\Longrightarrow$ (*upd f* (σ (| *break-status* := *b* |))) = (*upd f* σ) (| *break-status* := *b* |)
unfolding *control-independence-def exec-stop-def* **by** *simp*

1.1.3 An Example for Global Variable Declarations.

We present the above definition of the incremental construction of the state-space in more detail via an example construction.

Consider a global variable *A* representing an array of integer. This *global variable declaration* corresponds to the effect of the following record declaration:

record *state0* = *control-state* + *A* :: *int list*

which is later extended by another global variable, say, *B* representing a real described in the Cauchy Sequence form *nat* $\Rightarrow$ *int* $\times$ *int* as follows:

record *state1* = *state0* + *B* :: *nat* $\Rightarrow$ (*int* $\times$ *int*).

A further extension would be needed if a (potentially recursive) function *f* with some local variable *tmp* is defined: **record** *state2* = *state1* + *tmp* :: *nat stack result-value* :: *nat stack*, where the *stack* needed for modeling recursive instances is just a synonym for *list*.

1.1.4 The Assignment Operations (embedded in State-Exception Monad)

Based on the global variable states, we define *break*-aware and *return*-aware version of the assignment. The trick to do this in a generic *and* type-safe way is to provide the generated accessor and update functions (the "lens" representing this global variable, cf. [1–3]) to the generic assign operators. This pair of accessor and update carries all relevant semantic and type information of this particular variable and *characterizes* this variable semantically. Specific syntactic support [1] will hide away the syntactic overhead and permit a human-readable form of assignments or expressions accessing the underlying state.

consts *syntax-assign* :: ($'\alpha$ $\Rightarrow$ *int*) $\Rightarrow$ *int* $\Rightarrow$ *term* (**infix** := *60*)

definition *assign* :: (($'\sigma$*-ext*) *control-state-scheme* $\Rightarrow$
($'\sigma$*-ext*) *control-state-scheme*) $\Rightarrow$
(*unit*,($'\sigma$*-ext*) *control-state-scheme*)MON_{SE}
where *assign f* = ($\lambda\sigma$. *if exec-stop* σ *then Some*((), σ) *else Some*((), *f* σ))

definition *assign-global* :: (($'a$ $\Rightarrow$ $'a$) $\Rightarrow$ $'\sigma$*-ext control-state-scheme* $\Rightarrow$ $'\sigma$*-ext control-state-scheme*)
$\Rightarrow$ ($'\sigma$*-ext control-state-scheme* $\Rightarrow$ $'a$)

[1] via the Isabelle concept of cartouche: https://isabelle.in.tum.de/doc/isar-ref.pdf

$\Rightarrow (unit,'\sigma\text{-}ext\ control\text{-}state\text{-}scheme)\ MON_{SE}$
where *assign-global upd rhs* = *assign*($\lambda\sigma$. ((*upd*) (λ-. *rhs* σ)) σ)

An update of the variable A based on the state of the previous example is done by *assign-global A-upd* ($\lambda\sigma$. *list-update* (A σ) (i) (A σ ! j)) representing $A[i] = A[j]$; arbitrary nested updates can be constructed accordingly.

Local variable spaces work analogously; except that they are represented by a stack in order to support individual instances in case of function recursion. This requires automated generation of specific push- and pop operations used to model the effect of entering or leaving a function block (to be discussed later).

fun *map-hd* :: ($'a \Rightarrow 'a$) $\Rightarrow$ $'a$ *list* $\Rightarrow$ $'a$ *list*
where *map-hd f* [] = []
| *map-hd f* ($a\#S$) = f a # S

lemma *tl-map-hd* [*simp*] :*tl* (*map-hd f S*) = *tl S* **by** (*metis list.sel*(*3*) *map-hd.elims*)

definition *map-nth* = ($\lambda i\ f\ l$. *list-update l i* (f (l ! i)))

definition *assign-local* :: (($'a$ *list* $\Rightarrow$ $'a$ *list*)
$\Rightarrow '\sigma\text{-}ext\ control\text{-}state\text{-}scheme \Rightarrow '\sigma\text{-}ext\ control\text{-}state\text{-}scheme)$
$\Rightarrow ('\sigma\text{-}ext\ control\text{-}state\text{-}scheme \Rightarrow\ 'a)$
$\Rightarrow (unit,'\sigma\text{-}ext\ control\text{-}state\text{-}scheme)\ MON_{SE}$
where *assign-local upd rhs* = *assign*($\lambda\sigma$. ((*upd o map-hd*) (%-. *rhs* σ)) σ)

Semantically, the difference between *global* and *local* is rather unimpressive as the following lemma shows. However, the distinction matters for the pretty-printing setup of Clean.

lemma *assign-local upd rhs* = *assign-global* (*upd o map-hd*) *rhs*
unfolding *assign-local-def assign-global-def* **by** *simp*

The *return* command in C-like languages is represented basically by an assignment to a local variable *result-value* (see below in the Clean-package generation), plus some setup of *return-status*. Note that a *return* may appear after a *break* and should have no effect in this case.

definition $return_C$ :: (($'a$ *list* $\Rightarrow$ $'a$ *list*) $\Rightarrow '\sigma\text{-}ext\ control\text{-}state\text{-}scheme \Rightarrow '\sigma\text{-}ext\ control\text{-}state\text{-}scheme)$
$\Rightarrow ('\sigma\text{-}ext\ control\text{-}state\text{-}scheme \Rightarrow\ 'a)$
$\Rightarrow (unit,'\sigma\text{-}ext\ control\text{-}state\text{-}scheme)\ MON_{SE}$
where $return_C$ *upd rhs* =($\lambda\sigma$. *if exec-stop* σ *then Some*((), σ)
else (*assign-local upd rhs* ;− *set-return-status*) σ)

1.1.5 Example for a Local Variable Space

Consider the usual operation *swap* defined in some free-style syntax as follows:

function-spec swap (*i*::*nat*,*j*::*nat*)

local-vars *tmp* :: *int*
defines ‹ *tmp* := *A* ! *i*› ;−
‹ *A*[*i*] := *A* ! *j*› ;−
‹ *A*[*j*] := *tmp*›

For the fantasy syntax *tmp* := *A* ! *i*, we can construct the following semantic code: *assign-local tmp-update* $(\lambda\sigma.\ (A\ \sigma)\ !\ i\ )$ where *tmp-update* is the update operation generated by the **record**-package, which is generated while treating local variables of *swap*. By the way, a stack for *return*-values is also generated in order to give semantics to a *return* operation: it is syntactically equivalent to the assignment of the result variable in the local state (stack). It sets the *return-val* flag.

The management of the local state space requires function-specific *push* and *pop* operations, for which suitable definitions are generated as well:

definition push-local-swap-state :: (*unit*,′*a local-swap-state-scheme*) MON_{SE}
where *push-local-swap-state* σ =
Some((),σ⦇*local-swap-state.tmp* := *undefined* # *local-swap-state.tmp* σ,
local-swap-state.result-value := *undefined* #
local-swap-state.result-value σ ⦈)

definition pop-local-swap-state :: (*unit*,′*a local-swap-state-scheme*) MON_{SE}
where *pop-local-swap-state* σ =
Some(*hd*(*local-swap-state.result-value* σ),
σ⦇*local-swap-state.tmp*:= *tl*(*local-swap-state.tmp* σ) ⦈)

where *result-value* is the stack for potential result values (not needed in the concrete example *swap*).

1.2 Global and Local State Management via Extensible Records

In the sequel, we present the automation of the state-management as schematically discussed in the previous section; the declarations of global and local variable blocks are constructed by subsequent extensions of ′*a control-state-scheme*, defined above.

ML‹

```
structure StateMgt-core =
struct

val control-stateT = Syntax.parse-typ @{context} control-state
val control-stateS = @{typ ('a)control-state-scheme};

fun optionT t = Type(@{type-name Option.option},[t]);
fun MON-SE-T res state = state --> optionT(HOLogic.mk-prodT(res,state));
```

```
fun merge-control-stateS (@{typ ('a)control-state-scheme},t) = t
   |merge-control-stateS (t, @{typ ('a)control-state-scheme}) = t
   |merge-control-stateS (t, t') = if (t = t') then t else errorcan not merge Clean state

datatype var-kind = global-var of typ | local-var of typ

fun type-of(global-var t) = t | type-of(local-var t) = t

type state-field-tab = var-kind Symtab.table

structure Data = Generic-Data
(
  type T                    = (state-field-tab * typ (* current extensible record *))
  val  empty                = (Symtab.empty,control-stateS)
  val  extend               = I
  fun  merge((s1,t1),(s2,t2)) = (Symtab.merge (op =)(s1,s2),merge-control-stateS(t1,t2))
);


val get-data                  = Data.get o Context.Proof;
val map-data                  = Data.map;
val get-data-global           = Data.get o Context.Theory;
val map-data-global           = Context.theory-map o map-data;

val get-state-type            = snd o get-data
val get-state-type-global     = snd o get-data-global
val get-state-field-tab       = fst o get-data
val get-state-field-tab-global = fst o get-data-global
fun upd-state-type f          = map-data (fn (tab,t) => (tab, f t))
fun upd-state-type-global f   = map-data-global (fn (tab,t) => (tab, f t))

fun fetch-state-field (ln,X)  = let val a::b:: - = rev (Long-Name.explode ln) in ((b,a),X) end;

fun filter-name name ln       = let val ((a,b),X) = fetch-state-field ln
                                in  if a = name then SOME((a,b),X) else NONE end;

fun filter-attr-of name thy   = let val tabs = get-state-field-tab-global thy
                                in  map-filter (filter-name name) (Symtab.dest tabs) end;

fun is-program-variable name thy = Symtab.defined((fst o get-data-global) thy) name

fun is-global-program-variable name thy = case Symtab.lookup((fst o get-data-global) thy) name
of
                                            SOME(global-var -) => true
                                          | - => false

fun is-local-program-variable name thy = case Symtab.lookup((fst o get-data-global) thy) name
of
                                            SOME(local-var -) => true
```

| - => *false*

fun declare-state-variable-global f field thy =
let val Const(*name*,*ty*) = *Syntax.read-term-global thy field*
in (*map-data-global* (*apfst* (*Symtab.update-new*(*name*,*f ty*))) (*thy*)
handle Symtab.DUP - => *error*(*multiple declaration of global var*))
end;

fun declare-state-variable-local f field ctxt =
let val Const(*name*,*ty*) = *Syntax.read-term-global* (*Context.theory-of ctxt*) *field*
in (*map-data* (*apfst* (*Symtab.update-new*(*name*,*f ty*)))(*ctxt*)
handle Symtab.DUP - => *error*(*multiple declaration of global var*))
end;

end⟩

1.2.1 Block-Structures

On the managed local state-spaces, it is now straight-forward to define the semantics for a *block* representing the necessary management of local variable instances:

definition $block_C$ :: (*unit*, (′σ-*ext*) *control-state-ext*)MON_{SE}
⇒ (*unit*, (′σ-*ext*) *control-state-ext*)MON_{SE}
⇒ (′α, (′σ-*ext*) *control-state-ext*)MON_{SE}
⇒ (′α, (′σ-*ext*) *control-state-ext*)MON_{SE}
where $block_C$ *push core pop* ≡ (— assumes break and return unset
push ;− — create new instances of local variables
core ;− — execute the body
unset-break-status ;− — unset a potential break
unset-return-status;− — unset a potential return break
(*x* ← *pop*; — restore previous local var instances
$unit_{SE}$(*x*))) — yield the return value

Based on this definition, the running *swap* example is represented as follows:

definition swap-core :: *nat* × *nat* ⇒ (*unit*,′*a local-swap-state-scheme*) MON_{SE}
where swap-core ≡ (λ(*i*,*j*). ((*assign-local tmp-update* (λσ. *A* σ ! *i*)) ;−
(*assign-global A-update* (λσ. *list-update* (*A* σ) (*i*) (*A* σ ! *j*))) ;−
(*assign-global A-update* (λσ. *list-update* (*A* σ) (*j*) ((*hd o tmp*) σ)))))

definition swap :: *nat* × *nat* ⇒ (*unit*,′*a local-swap-state-scheme*) MON_{SE}
where swap ≡ λ(*i*,*j*). $block_C$ *push-local-swap-state* (*swap-core* (*i*,*j*)) *pop-local-swap-state*

1.2.2 Call Semantics

It is now straight-forward to define the semantics of a generic call — which is simply a monad execution that is *break*-aware and *return*-aware.

definition $call_C$ :: ($'\alpha \Rightarrow ('\varrho, ('\sigma\text{-}ext)\ control\text{-}state\text{-}ext)MON_{SE}) \Rightarrow$
$((('\sigma\text{-}ext)\ control\text{-}state\text{-}ext) \Rightarrow '\alpha) \Rightarrow$
$('\varrho, ('\sigma\text{-}ext)\ control\text{-}state\text{-}ext)MON_{SE}$
where $call_C\ M\ A_1 = (\lambda\sigma.\ if\ exec\text{-}stop\ \sigma\ then\ Some(undefined, \sigma)\ else\ M\ (A_1\ \sigma)\ \sigma)$

Note that this presentation assumes a uncurried format of the arguments. The question arises if this is the right approach to handle calls of operation with multiple arguments. Is it better to go for an some appropriate currying principle? Here are some more experimental variants for curried operations...

definition $call\text{-}0_C$ :: $('\varrho, ('\sigma\text{-}ext)\ control\text{-}state\text{-}ext)MON_{SE} \Rightarrow ('\varrho, ('\sigma\text{-}ext)\ control\text{-}state\text{-}ext)MON_{SE}$
where $call\text{-}0_C\ M = (\lambda\sigma.\ if\ exec\text{-}stop\ \sigma\ then\ Some(undefined, \sigma)\ else\ M\ \sigma)$

The generic version using tuples is identical with $call\text{-}1_C$.

definition $call\text{-}1_C$ :: ($'\alpha \Rightarrow ('\varrho, ('\sigma\text{-}ext)\ control\text{-}state\text{-}ext)MON_{SE}) \Rightarrow$
$((('\sigma\text{-}ext)\ control\text{-}state\text{-}ext) \Rightarrow '\alpha) \Rightarrow$
$('\varrho, ('\sigma\text{-}ext)\ control\text{-}state\text{-}ext)MON_{SE}$
where $call\text{-}1_C = call_C$

definition $call\text{-}2_C$ :: ($'\alpha \Rightarrow '\beta \Rightarrow ('\varrho, ('\sigma\text{-}ext)\ control\text{-}state\text{-}ext)MON_{SE}) \Rightarrow$
$((('\sigma\text{-}ext)\ control\text{-}state\text{-}ext) \Rightarrow '\alpha) \Rightarrow$
$((('\sigma\text{-}ext)\ control\text{-}state\text{-}ext) \Rightarrow '\beta) \Rightarrow$
$('\varrho, ('\sigma\text{-}ext)\ control\text{-}state\text{-}ext)MON_{SE}$
where $call\text{-}2_C\ M\ A_1\ A_2 = (\lambda\sigma.\ if\ exec\text{-}stop\ \sigma\ then\ Some(undefined, \sigma)\ else\ M\ (A_1\ \sigma)\ (A_2\ \sigma)\ \sigma)$

definition $call\text{-}3_C$ :: ($'\alpha \Rightarrow '\beta \Rightarrow '\gamma \Rightarrow ('\varrho, ('\sigma\text{-}ext)\ control\text{-}state\text{-}ext)MON_{SE}) \Rightarrow$
$((('\sigma\text{-}ext)\ control\text{-}state\text{-}ext) \Rightarrow '\alpha) \Rightarrow$
$((('\sigma\text{-}ext)\ control\text{-}state\text{-}ext) \Rightarrow '\beta) \Rightarrow$
$((('\sigma\text{-}ext)\ control\text{-}state\text{-}ext) \Rightarrow '\gamma) \Rightarrow$
$('\varrho, ('\sigma\text{-}ext)\ control\text{-}state\text{-}ext)MON_{SE}$
where $call\text{-}3_C\ M\ A_1\ A_2\ A_3 = (\lambda\sigma.\ if\ exec\text{-}stop\ \sigma\ then\ Some(undefined, \sigma)$
$else\ M\ (A_1\ \sigma)\ (A_2\ \sigma)\ (A_3\ \sigma)\ \sigma)$

1.3 Some Term-Coding Functions

In the following, we add a number of advanced HOL-term constructors in the style of `HOLogic` from the Isabelle/HOL libraries. They incorporate the construction of types during term construction in a bottom-up manner. Consequently, the leafs of such terms should always be typed, and anonymous loose-`Bound` variables avoided.

ML‹

```
(* HOLogic extended *)

fun mk-None ty = let val none = const-name ‹Option.option.None›
                     val none-ty = ty --> Type(type-name ‹option›,[ty])
                 in  Const(none, none-ty)
                 end;
```

```
fun mk-Some t = let val some = const-name ‹Option.option.Some›
                    val ty = fastype-of t
                    val some-ty = ty --> Type(type-name ‹option›,[ty])
                in  Const(some, some-ty) $ t
                end;

fun dest-listTy (Type(type-name ‹List.list›, [T])) = T;

fun mk-hdT t = let val ty = fastype-of t
               in  Const(const-name ‹List.hd›, ty --> (dest-listTy ty)) $ t end

fun mk-tlT t = let val ty = fastype-of t
               in  Const(const-name ‹List.tl›, ty --> ty) $ t end


fun  mk-undefined (@{typ unit}) = Const (const-name ‹Product-Type.Unity›, typ ‹unit›)
    |mk-undefined t             = Const (const-name ‹HOL.undefined›, t)

fun meta-eq-const T = Const (const-name ‹Pure.eq›, T --> T --> propT);

fun mk-meta-eq (t, u) = meta-eq-const (fastype-of t) $ t $ u;

fun   mk-pat-tupleabs [] t = t
    | mk-pat-tupleabs [(s,ty)] t = absfree(s,ty)(t)
    | mk-pat-tupleabs ((s,ty)::R) t = HOLogic.mk-case-prod(absfree(s,ty)(mk-pat-tupleabs R t));

fun read-constname ctxt n = fst(dest-Const(Syntax.read-term ctxt n))

fun wfrecT order recs =
    let val funT = domain-type (fastype-of recs)
        val aTy  = domain-type funT
        val ordTy = HOLogic.mk-setT(HOLogic.mk-prodT (aTy,aTy))
    in Const(const-name ‹Wfrec.wfrec›, ordTy --> (funT --> funT) --> funT) $ order $
recs end


›
```

And here comes the core of the *Clean*-State-Management: the module that provides the functionality for the commands keywords **global-vars**, **local-vars** and **local-vars-test**. Note that the difference between **local-vars** and **local-vars-test** is just a technical one: **local-vars** can only be used inside a Clean function specification, made with the **function-spec** command. On the other hand, **local-vars-test** is defined as a global Isar command for test purposes.

A particular feature of the local-variable management is the provision of definitions for *push* and *pop* operations — encoded as $('o, '\sigma)\ MON_{SE}$ operations — which are vital for the function specifications defined below.

ML‹

```
structure StateMgt =
struct

open StateMgt-core

val result-name = result-value

fun get-result-value-conf name thy =
        let val  S = filter-attr-of name thy
        in  hd(filter (fn ((-,b),-) => b = result-name) S)
            handle Empty => error internal error: get-result-value-conf  end;


fun mk-lookup-result-value-term name sty thy =
    let val ((prefix,name),local-var(Type(fun, [-,ty]))) = get-result-value-conf name thy;
        val long-name = Sign.intern-const thy (prefix^.^name)
        val term = Const(long-name, sty --> ty)
    in  mk-hdT (term $ Free(σ,sty)) end


fun  map-to-update sty is-pop thy ((struct-name, attr-name), local-var (Type(fun,[-,ty]))) term
=
       let val tlT = if is-pop then Const(const-name‹List.tl›, ty --> ty)
                     else Const(const-name‹List.Cons›, dest-listTy ty --> ty --> ty)
                          $ mk-undefined (dest-listTy ty)
           val update-name = Sign.intern-const  thy (struct-name^.^attr-name^-update)
       in (Const(update-name, (ty --> ty) --> sty --> sty) $ tlT) $ term end
   | map-to-update - - - ((-, -),-) - = error(internal error map-to-update)

fun mk-local-state-name binding =
       Binding.prefix-name local- (Binding.suffix-name -state binding)
fun mk-global-state-name binding =
       Binding.prefix-name global- (Binding.suffix-name -state binding)

fun construct-update is-pop binding sty thy =
       let val long-name = Binding.name-of( binding)
           val attrS = StateMgt-core.filter-attr-of long-name thy
       in  fold (map-to-update sty is-pop thy) (attrS) (Free(σ,sty)) end

fun cmd (decl, spec, prems, params) = #2 oo Specification.definition' decl params prems spec

fun mk-push-name binding = Binding.prefix-name push- binding

fun push-eq binding  name-op rty sty lthy =
         let val mty = MON-SE-T rty sty
             val thy = Proof-Context.theory-of lthy
             val term = construct-update false binding sty thy
         in  mk-meta-eq((Free(name-op, mty) $ Free(σ,sty)),
                        mk-Some ( HOLogic.mk-prod (mk-undefined rty,term)))
```

```
        end;

fun mk-push-def binding sty lthy =
    let val name-pushop =  mk-push-name binding
        val rty = typ ‹unit›
        val eq = push-eq binding  (Binding.name-of name-pushop) rty sty lthy
        val mty = StateMgt-core.MON-SE-T rty sty
        val args = (SOME(name-pushop, SOME mty, NoSyn), (Binding.empty-atts,eq),[],[])
    in cmd args true lthy  end;

fun mk-pop-name binding = Binding.prefix-name pop-  binding

fun pop-eq  binding name-op rty sty lthy =
        let val mty = MON-SE-T rty sty
            val thy = Proof-Context.theory-of lthy
            val res-access = mk-lookup-result-value-term (Binding.name-of binding) sty thy
            val term = construct-update true binding  sty thy
        in  mk-meta-eq((Free(name-op, mty) $ Free(σ,sty)),
                       mk-Some ( HOLogic.mk-prod (res-access,term)))
        end;


fun mk-pop-def binding rty sty lthy =
    let val mty = StateMgt-core.MON-SE-T rty sty
        val name-op =  mk-pop-name binding
        val eq = pop-eq binding (Binding.name-of name-op) rty sty lthy
        val args = (SOME(name-op, SOME mty, NoSyn),(Binding.empty-atts,eq),[],[])
    in cmd args true lthy
    end;


fun read-parent NONE ctxt = (NONE, ctxt)
  | read-parent (SOME raw-T) ctxt =
      (case Proof-Context.read-typ-abbrev ctxt raw-T of
        Type (name, Ts) => (SOME (Ts, name), fold Variable.declare-typ Ts ctxt)
      | T => error (Bad parent record specification:  ^ Syntax.string-of-typ ctxt T));


fun read-fields raw-fields ctxt =
  let
    val Ts = Syntax.read-typs ctxt (map (fn (-, raw-T, -) => raw-T) raw-fields);
    val fields = map2 (fn (x, -, mx) => fn T => (x, T, mx)) raw-fields Ts;
    val ctxt' = fold Variable.declare-typ Ts ctxt;
  in (fields, ctxt') end;

fun parse-typ-'a ctxt binding =
  let val ty-bind =  Binding.prefix-name 'a  (Binding.suffix-name -scheme binding)
  in case Syntax.parse-typ ctxt (Binding.name-of ty-bind) of
```

```
      Type (s, -) => Type (s, [@{typ 'a::type}])
    | - => error (Unexpected type ^ Position.here here)
  end

fun add-record-cmd0 read-fields overloaded is-global-kind raw-params binding raw-parent raw-fields
thy =
  let
    val ctxt = Proof-Context.init-global thy;
    val params = map (apsnd (Typedecl.read-constraint ctxt)) raw-params;
    val ctxt1 = fold (Variable.declare-typ o TFree) params ctxt;
    val (parent, ctxt2) = read-parent raw-parent ctxt1;
    val (fields, ctxt3) = read-fields raw-fields ctxt2;
    fun lift (a,b,c) =  (a, HOLogic.listT b, c)
    val fields' = if is-global-kind then fields else map lift fields
    val params' = map (Proof-Context.check-tfree ctxt3) params;
    val declare = StateMgt-core.declare-state-variable-global
    fun upd-state-typ thy = let val ctxt = Proof-Context.init-global thy
                                val ty = Syntax.parse-typ ctxt (Binding.name-of binding)
                            in  StateMgt-core.upd-state-type-global(K ty)(thy) end
    fun insert-var ((f,-,-), thy) =
            if is-global-kind
            then declare StateMgt-core.global-var (Binding.name-of f) thy
            else declare StateMgt-core.local-var  (Binding.name-of f) thy
    fun define-push-pop thy =
            if not is-global-kind
            then let val sty = parse-typ-'a (Proof-Context.init-global thy) binding;
                     val rty = dest-listTy (#2(hd(rev fields')))
                 in thy

                    |> Named-Target.theory-map (mk-push-def binding sty)
                    |> Named-Target.theory-map (mk-pop-def  binding rty sty)

                 end
            else thy
  in thy |> Record.add-record overloaded (params', binding) parent fields'
         |> (fn thy =>  List.foldr insert-var (thy) (fields'))
         |> upd-state-typ
         |> define-push-pop
  end;



fun typ-2-string-raw (Type(s,[TFree -])) = if String.isSuffix -scheme s
                                           then Long-Name.base-name(unsuffix -scheme s)
                                           else Long-Name.base-name(unsuffix -ext s)

   |typ-2-string-raw (Type(s,-)) =
                      error (Illegal parameterized state type − not allowed in Clean:  ^ s)
   |typ-2-string-raw - = error  Illegal state type − not allowed in Clean.
```

```
fun new-state-record0 add-record-cmd is-global-kind (((raw-params, binding), res-ty), raw-fields)
thy =
    let val binding = if is-global-kind
                      then mk-global-state-name binding
                      else mk-local-state-name binding
        val raw-parent = SOME(typ-2-string-raw (StateMgt-core.get-state-type-global thy))
        val pos = Binding.pos-of binding
        fun upd-state-typ thy =
              StateMgt-core.upd-state-type-global (K (parse-typ-'a (Proof-Context.init-global thy)
binding)) thy
        val result-binding = Binding.make(result-name,pos)
        val raw-fields' = case res-ty of
                            NONE => raw-fields
                          | SOME res-ty => raw-fields @ [(result-binding,res-ty, NoSyn)]
    in  thy |> add-record-cmd {overloaded = false} is-global-kind
                             raw-params binding raw-parent raw-fields'
            |> upd-state-typ

    end

val add-record-cmd    = add-record-cmd0 read-fields;
val add-record-cmd'   = add-record-cmd0 pair;

val new-state-record  = new-state-record0 add-record-cmd
val new-state-record' = new-state-record0 add-record-cmd'

val - =
  Outer-Syntax.command
      command-keyword ‹global-vars›
      define global state record
      ((Parse.type-args-constrained -- Parse.binding)
    -- Scan.succeed NONE
    -- Scan.repeat1 Parse.const-binding
    >> (Toplevel.theory o new-state-record true));
;

val - =
  Outer-Syntax.command
      command-keyword ‹local-vars-test›
      define local state record
      ((Parse.type-args-constrained -- Parse.binding)
    -- (Parse.typ >> SOME)
    -- Scan.repeat1 Parse.const-binding
    >> (Toplevel.theory o new-state-record false))
;
end
›
```

1.4 Syntactic Sugar supporting λ-lifting for Global and Local Variables

ML ‹

```
structure Clean-Syntax-Lift =
struct
  local
    fun mk-local-access X = Const (@{const-name Fun.comp}, dummyT)
                          $ Const (@{const-name List.list.hd}, dummyT) $ X
  in
    fun app-sigma db tm ctxt = case tm of
      Const(name, -) => if StateMgt-core.is-global-program-variable name (Proof-Context.theory-of
ctxt)
                         then tm $ (Bound db) (* lambda lifting *)
                       else if StateMgt-core.is-local-program-variable name (Proof-Context.theory-of
ctxt)
                            then (mk-local-access tm) $ (Bound db) (* lambda lifting local *)
                            else tm                (* no lifting *)
      | Free - => tm
      | Var - => tm
      | Bound n => if n > db then Bound(n + 1) else Bound n
      | Abs (x, ty, tm') => Abs(x, ty, app-sigma (db+1) tm' ctxt)
      | t1 $ t2 => (app-sigma db t1 ctxt) $ (app-sigma db t2 ctxt)

    fun scope-var name =
      Proof-Context.theory-of
      #> (fn thy =>
            if StateMgt-core.is-global-program-variable name thy then SOME true
            else if StateMgt-core.is-local-program-variable name thy then SOME false
            else NONE)

    fun assign-update var = var ^ Record.updateN

    fun transform-term0 abs scope-var tm =
      case tm of
          Const (@{const-name Clean.syntax-assign}, -)
          $ (t1 as Const (-type-constraint-, -) $ Const (name, ty))
          $ t2 =>
            Const ( case scope-var name of
                      SOME true => @{const-name assign-global}
                    | SOME false => @{const-name assign-local}
                    | NONE => raise TERM (mk-assign, [t1])
                  , dummyT)
            $ Const(assign-update name, ty)
            $ abs t2
        | - => abs tm

    fun transform-term ctxt sty =
      transform-term0
```

```
      (fn tm => Abs (σ, sty, app-sigma 0 tm ctxt))
      (fn name => scope-var name ctxt)

  fun transform-term' ctxt = transform-term ctxt dummyT

  fun string-tr ctxt content args =
    let fun err () = raise TERM (string-tr, args)
    in
      (case args of
        [(Const (@{syntax-const -constrain}, -)) $ (Free (s, -)) $ p] =>
          (case Term-Position.decode-position p of
            SOME (pos, -) => Symbol-Pos.implode (content (s, pos))
                      |> Syntax.parse-term ctxt
                      |> transform-term ctxt (StateMgt-core.get-state-type ctxt)
                      |> Syntax.check-term ctxt
          | NONE => err ())
        | - => err ())
    end
  end
end
⟩
```

syntax *-cartouche-string* :: *cartouche-position* ⇒ *string* (-)

parse-translation ⟨
```
  [(@{syntax-const -cartouche-string},
   (fn ctxt => Clean-Syntax-Lift.string-tr ctxt (Symbol-Pos.cartouche-content o Symbol-Pos.explode)))]
⟩
```

1.5 Support for (direct recursive) Clean Function Specifications

Based on the machinery for the State-Management and implicitly cooperating with the cartouches for assignment syntax, the function-specification **function-spec**-package coordinates:

1. the parsing and type-checking of parameters,
2. the parsing and type-checking of pre and post conditions in MOAL notation (using λ-lifting cartouches and implicit reference to parameters, pre and post states),
3. the parsing local variable section with the local-variable space generation,
4. the parsing of the body in this extended variable space,
5. and optionally the support of measures for recursion proofs.

   The reader interested in details is referred to the `../examples/Quicksort_concept.thy`-example, accompanying this distribution.

definition *old* :: *'a* ⇒ *'a* **where** *old x* = *x*

ML‹
›

ML ‹

```
structure Function-Specification-Parser  =
  struct

    type funct-spec-src = {
        binding:  binding,                         (* name *)
        params: (binding*string) list,               (* parameters and their type*)
        ret-type: string,                          (* return type; default unit *)
        locals: (binding*string*mixfix)list,        (* local variables *)
        pre-src: string,                           (* precondition src *)
        post-src: string,                          (* postcondition src *)
        variant-src: string option,                      (* variant src *)
        body-src: string * Position.T                (* body src *)
      }

    type funct-spec-sem = {
        params: (binding*typ) list,                  (* parameters and their type*)
        ret-ty: typ,                              (* return type *)
        pre: term,                                (* precondition  *)
        post: term,                               (* postcondition  *)
        variant: term option                         (* variant  *)
      }


    val parse-arg-decl = Parse.binding -- (Parse.$$$ :: |-- Parse.typ)

    val parse-param-decls = Args.parens (Parse.enum , parse-arg-decl)

    val parse-returns-clause = Scan.optional (keyword‹returns› |--  Parse.typ) unit

    val locals-clause = (Scan.optional ( keyword‹local-vars›
                                  -- (Scan.repeat1 Parse.const-binding)) (, []))

    val parse-proc-spec = (
          Parse.binding
       -- parse-param-decls
       -- parse-returns-clause
       --| keyword‹pre›              -- Parse.term
       --| keyword‹post›             -- Parse.term
       -- (Scan.option  ( keyword‹variant› |-- Parse.term))
       -- (Scan.optional( keyword‹local-vars› |-- (Scan.repeat1 Parse.const-binding))([]))
       --| keyword‹defines›          -- (Parse.position (Parse.term))
     ) >> (fn ((((((((binding,params),ret-ty),pre-src),post-src),variant-src),locals)),body-src) =>

        {
          binding = binding,
```

```
      params=params,
      ret-type=ret-ty,
      pre-src=pre-src,
      post-src=post-src,
      variant-src=variant-src,
      locals=locals,
      body-src=body-src} : funct-spec-src
     )

fun read-params params ctxt =
  let
    val Ts = Syntax.read-typs ctxt (map snd params);
  in (Ts, fold Variable.declare-typ Ts ctxt) end;

fun read-result ret-ty ctxt =
      let val [ty] = Syntax.read-typs ctxt [ret-ty]
          val ctxt' = Variable.declare-typ ty ctxt
      in  (ty, ctxt') end

fun read-function-spec ({ params, ret-type, variant-src, ...} : funct-spec-src) ctxt =
    let val (params-Ts, ctxt') = read-params params ctxt
        val (rty, ctxt'') = read-result ret-type ctxt'
        val variant = Option.map (Syntax.read-term ctxt'')  variant-src
    in ({params = (params, params-Ts), ret-ty = rty,variant = variant},ctxt'') end


fun check-absence-old term =
        let fun test (s,ty) = if s = @{const-name old} andalso fst (dest-Type ty) = fun
                              then error(the old notation is not allowed here!)
                              else false
        in  exists-Const test term end

fun transform-old sty term =
    let fun  transform-old0 (Const(@{const-name old}, Type (fun, [-,-])) $ term )
                          = (case term of
                              (Const(s,ty) $ Bound x) =>  (Const(s,ty) $ Bound (x+1))
                            | - => error(illegal application of the old notation.))
           |transform-old0 (t1 $ t2) = transform-old0 t1 $ transform-old0 t2
           |transform-old0 (Abs(s,ty,term)) = Abs(s,ty,transform-old0 term)
           |transform-old0 term = term
    in  Abs(σ_pre, sty, transform-old0 term) end

fun define-cond binding f-sty transform-old src-suff check-absence-old params src ctxt =
    let val src' = case transform-old (Syntax.read-term ctxt src) of
                         Abs(nn, sty-pre, term) => mk-pat-tupleabs (map (apsnd #2) params)
(Abs(nn,sty-pre(* sty root ! !*),term))
                     | - => error (define abstraction for result ^ Position.here here)
        val bdg = Binding.suffix-name src-suff binding
        val - = check-absence-old src'
```

```
            val eq =  mk-meta-eq(Free(Binding.name-of bdg, HOLogic.mk-tupleT(map (#2 o #2)
params) --> f-sty HOLogic.boolT),src')
            val args = (SOME(bdg,NONE,NoSyn), (Binding.empty-atts,eq),[],[])
        in  StateMgt.cmd args true ctxt end

    fun define-precond binding sty =
      define-cond binding (fn boolT => sty --> boolT) I -pre check-absence-old

    fun define-postcond binding rty sty =
      define-cond binding (fn boolT => sty --> sty --> rty --> boolT) (transform-old sty)
-post I

    fun define-body-core binding args-ty sty params body =
        let val bdg-core = Binding.suffix-name -core binding
            val bdg-core-name = Binding.name-of bdg-core

            val umty = args-ty --> StateMgt.MON-SE-T @{typ unit} sty

          val eq = mk-meta-eq(Free (bdg-core-name, umty),mk-pat-tupleabs(map(apsnd #2)params)
body)
           val args-core =(SOME (bdg-core, SOME umty, NoSyn), (Binding.empty-atts, eq), [], [])

        in StateMgt.cmd args-core true
        end

    fun define-body-main {recursive = x:bool} binding rty sty params variant-src - ctxt =
        let val push-name = StateMgt.mk-push-name (StateMgt.mk-local-state-name binding)
            val pop-name = StateMgt.mk-pop-name (StateMgt.mk-local-state-name binding)
            val bdg-core = Binding.suffix-name -core binding
            val bdg-core-name = Binding.name-of bdg-core
            val bdg-rec-name = Binding.name-of(Binding.suffix-name -rec binding)
            val bdg-ord-name = Binding.name-of(Binding.suffix-name -order binding)

            val args-ty = HOLogic.mk-tupleT (map (#2 o #2) params)
            val params' = map (apsnd #2) params
            val rmty = StateMgt-core.MON-SE-T rty sty

            val umty = StateMgt.MON-SE-T @{typ unit} sty
            val argsProdT = HOLogic.mk-prodT(args-ty,args-ty)
            val argsRelSet = HOLogic.mk-setT argsProdT
            val measure-term = case variant-src of
                                   NONE => Free(bdg-ord-name,args-ty --> HOLogic.natT)
                               | SOME str => (Syntax.read-term ctxt str |> mk-pat-tupleabs params')
                val measure =  Const(@{const-name Wellfounded.measure}, (args-ty --> HO-
Logic.natT)
                                                              --> argsRelSet )
                          $ measure-term
            val lhs-main = if x andalso is-none variant-src
                          then Free(Binding.name-of binding, (args-ty --> HOLogic.natT)
```

```
                                                  --> args-ty --> rmty) $
                                    Free(bdg-ord-name, args-ty --> HOLogic.natT)
                            else Free(Binding.name-of binding, args-ty --> rmty)
          val rhs-main = mk-pat-tupleabs params'
                            (Const(@{const-name Clean.block_C}, umty --> umty  --> rmty -->
rmty)
                            $ Const(read-constname ctxt (Binding.name-of push-name),umty)
                            $ (Const(read-constname ctxt bdg-core-name, args-ty --> umty)
                              $ HOLogic.mk-tuple (map Free params'))
                            $ Const(read-constname ctxt (Binding.name-of pop-name),rmty))
          val rhs-main-rec = wfrecT
                              measure
                              (Abs(bdg-rec-name, (args-ty --> umty) ,
                                  mk-pat-tupleabs params'
                            (Const(@{const-name Clean.block_C}, umty-->umty-->rmty-->rmty)
                                  $ Const(read-constname ctxt (Binding.name-of push-name),umty)
                                  $ (Const(read-constname ctxt bdg-core-name,
                                        (args-ty --> umty) --> args-ty --> umty)
                                    $ (Bound (length params))
                                    $ HOLogic.mk-tuple (map Free params'))
                                  $ Const(read-constname ctxt (Binding.name-of pop-name),rmty))))
          val eq-main = mk-meta-eq(lhs-main, if x then rhs-main-rec else rhs-main )
          val args-main = (SOME(binding,NONE,NoSyn), (Binding.empty-atts,eq-main),[],[])
      in  ctxt |> StateMgt.cmd args-main true
      end


   fun checkNsem-function-spec {recursive = false} ({variant-src=SOME -, ...}) - =
                               error No measure required in non-recursive call
      |checkNsem-function-spec (isrec as {recursive = -:bool})
                             (args as {binding, ret-type, variant-src, locals, body-src, pre-src, post-src,
...} : funct-spec-src)
                               thy =
      let val (theory-map, thy') =
             Named-Target.theory-map-result
               (K (fn f => Named-Target.theory-map o f))
               (read-function-spec args
               #> uncurry (fn {params=(params, Ts),ret-ty,variant = -} =>
                         pair (fn f =>
                               Proof-Context.add-fixes (map2 (fn (b, -) => fn T => (b, SOME T,
NoSyn)) params Ts)
                                 (* this declares the parameters of a function specification
                                    as Free variables (overrides a possible constant declaration)
                                    and assigns the declared type to them *)
                               #> uncurry (fn params' => f (@{map 3} (fn b' => fn (b, -) =>
fn T => (b',(b,T))) params' params Ts) ret-ty))))
                 thy
      in  thy' |> theory-map
                let val sty-old = StateMgt-core.get-state-type-global thy'
```

```
            in fn params => fn ret-ty =>
                define-precond binding sty-old params pre-src
             #> define-postcond binding ret-ty sty-old params post-src end
        |> StateMgt.new-state-record false ((([],binding), SOME ret-type),locals)
        |> theory-map
                (fn params => fn ret-ty => fn ctxt =>
                 let val sty = StateMgt-core.get-state-type ctxt
                     val args-ty = HOLogic.mk-tupleT (map (#2 o #2) params)
                     val mon-se-ty = StateMgt-core.MON-SE-T ret-ty sty
                     val ctxt' =
                       if #recursive isrec then
                         Proof-Context.add-fixes
                           [(binding, SOME (args-ty --> mon-se-ty), NoSyn)] ctxt |> #2
                       else
                         ctxt
                     val body = Syntax.read-term ctxt' (fst body-src)
                 in  ctxt' |> define-body-core binding args-ty sty params body
                 end)
        |> theory-map
                (fn params => fn ret-ty => fn ctxt =>
                 let val sty = StateMgt-core.get-state-type ctxt
                     val body = Syntax.read-term ctxt (fst body-src)
                in  ctxt |> define-body-main isrec binding ret-ty sty params variant-src body
                 end)
  end


val - =
  Outer-Syntax.command
     command-keyword ‹function-spec›
     define Clean function specification
     (parse-proc-spec >> (Toplevel.theory o checkNsem-function-spec {recursive = false}));

val - =
  Outer-Syntax.command
     command-keyword ‹rec-function-spec›
     define recursive Clean function specification
     (parse-proc-spec >> (Toplevel.theory o checkNsem-function-spec {recursive = true}));

end
›
```

1.6 The Rest of Clean: Break/Return aware Version of If, While, etc.

definition *if-C* :: $[('\sigma\text{-}ext)\ control\text{-}state\text{-}ext \Rightarrow bool,$
$('\beta, ('\sigma\text{-}ext)\ control\text{-}state\text{-}ext)MON_{SE},$
$('\beta, ('\sigma\text{-}ext)\ control\text{-}state\text{-}ext)MON_{SE}] \Rightarrow ('\beta, ('\sigma\text{-}ext)\ control\text{-}state\text{-}ext)MON_{SE}$

where $\mathit{if\text{-}C}\ c\ E\ F = (\lambda\sigma.\ \mathit{if}\ \mathit{exec\text{-}stop}\ \sigma$
$\quad \mathit{then}\ \mathit{Some}(\mathit{undefined},\ \sigma)$ — state unchanged, return arbitrary
$\quad \mathit{else}\ \mathit{if}\ c\ \sigma\ \mathit{then}\ E\ \sigma\ \mathit{else}\ F\ \sigma)$

syntax (*xsymbols*)
$\mathit{\text{-}if\text{-}SECLEAN} :: [{}'\sigma \Rightarrow \mathit{bool},({}'o,{}'\sigma)\mathit{MON}_{SE},({}'o',{}'\sigma)\mathit{MON}_{SE}] \Rightarrow ({}'o',{}'\sigma)\mathit{MON}_{SE}$
$((\mathit{if}_C\ \text{-}\ \mathit{then}\ \text{-}\ \mathit{else}\ \text{-}\mathit{fi})\ [5,8,8]8)$
translations
$(\mathit{if}_C\ \mathit{cond}\ \mathit{then}\ \mathit{T1}\ \mathit{else}\ \mathit{T2}\ \mathit{fi}) == \mathit{CONST}\ \mathit{Clean.if\text{-}C}\ \mathit{cond}\ \mathit{T1}\ \mathit{T2}$

definition $\mathit{while\text{-}C} :: (({}'\sigma\text{-}\mathit{ext})\ \mathit{control\text{-}state\text{-}ext} \Rightarrow \mathit{bool})$
$\quad \Rightarrow (\mathit{unit},\ ({}'\sigma\text{-}\mathit{ext})\ \mathit{control\text{-}state\text{-}ext})\mathit{MON}_{SE}$
$\quad \Rightarrow (\mathit{unit},\ ({}'\sigma\text{-}\mathit{ext})\ \mathit{control\text{-}state\text{-}ext})\mathit{MON}_{SE}$
where $\mathit{while\text{-}C}\ c\ B \equiv (\lambda\sigma.\ \mathit{if}\ \mathit{exec\text{-}stop}\ \sigma\ \mathit{then}\ \mathit{Some}((),\ \sigma)$
$\quad \mathit{else}\ ((\mathit{MonadSE.while\text{-}SE}\ (\lambda\ \sigma.\ \neg \mathit{exec\text{-}stop}\ \sigma \wedge c\ \sigma)\ B)\ ;-$
$\quad\quad \mathit{unset\text{-}break\text{-}status})\ \sigma)$

syntax (*xsymbols*)
$\mathit{\text{-}while\text{-}C} :: [{}'\sigma \Rightarrow \mathit{bool},\ (\mathit{unit},\ {}'\sigma)\mathit{MON}_{SE}] \Rightarrow (\mathit{unit},\ {}'\sigma)\mathit{MON}_{SE}$
$((\mathit{while}_C\ \text{-}\ \mathit{do}\ \text{-}\ \mathit{od})\ [8,8]8)$
translations
$\mathit{while}_C\ c\ \mathit{do}\ b\ \mathit{od} == \mathit{CONST}\ \mathit{Clean.while\text{-}C}\ c\ b$

end

2 Clean Semantics : A Coding-Concept Example

The following show-case introduces subsequently a non-trivial example involving local and global variable declarations, declarations of operations with pre-post conditions as well as direct-recursive operations (i.e. C-like functions with side-effects on global and local variables.

theory *Quicksort-concept*
  imports *Clean-Main.Clean*
          Clean-Main.Hoare-MonadSE
begin

2.1 The Quicksort Example

We present the following quicksort algorithm in some conceptual, high-level notation:

```
algorithm (A,i,j) =
    tmp := A[i];
    A[i]:=A[j];
    A[j]:=tmp

algorithm partition(A, lo, hi) is
    pivot := A[hi]
    i := lo
    for j := lo to hi - 1 do
        if A[j] < pivot then
            swap A[i] with A[j]
            i := i + 1
    swap A[i] with A[hi]
    return i

algorithm quicksort(A, lo, hi) is
    if lo < hi then
        p := partition(A, lo, hi)
        quicksort(A, lo, p - 1)
        quicksort(A, p + 1, hi)
```

In the following, we will present the Quicksort program alternatingly in Clean high-level notation and simulate its effect by an alternative formalisation representing the semantic effects of the high-level notation on a step-buy-step basis. Note that Clean does not posses the concept of call-by-reference parameters; consequently, the algorithm must be specialized to a variant where A is just a global variable.

2.2 Clean Encoding of the Global State of Quicksort

We demonstrate the accumulating effect of some key Clean commands by highlighting the changes of Clean's state-management module state. At the beginning, the state-type of the Clean state management is just the type of the *'a control-state-scheme*, while the table of global and local variables is empty.

ML‹ *val Type(s,t) = StateMgt-core.get-state-type-global* @{*theory*};
  StateMgt-core.get-state-field-tab-global @{*theory*}; ›

The *global-vars* command, described and defined in `Clean.thy`, declares the global variable `A`. This has the following effect:

global-vars *state*
  A :: *int list*

... which is reflected in Clean's state-management table:

ML‹ *val Type(Quicksort-concept.global-state-state-scheme,t)*
    = *StateMgt-core.get-state-type-global* @{*theory*};
  StateMgt-core.get-state-field-tab-global @{*theory*}›

Note that the state-management uses long-names for complete disambiguation.

2.3 Encoding swap in Clean

2.3.1 `swap` in High-level Notation

Unfortunately, the name *result* is already used in the logical context; we use local binders instead.

definition $i = ()$ — check that i can exist as a constant with an arbitrary type before treating **function-spec**
definition $j = ()$ — check that j can exist as a constant with an arbitrary type before treating **function-spec**

function-spec *swap* (*i*::*nat*,*j*::*nat*) — TODO: the hovering on parameters produces a number of report equal to the number of `Proof_Context.add_fixes` called in `Function_Specification_Parser.checkN`
pre ‹$i < length\ A \wedge j < length\ A$›
post ‹$\lambda res.\ length\ A = length(old\ A) \wedge res = ()$›
local-vars *tmp* :: *int*
defines ‹ *tmp* := *A* ! *i*› ;−
‹ *A* := *list-update A i* (*A* ! *j*)› ;−
‹ *A* := *list-update A j tmp*›

The body — heavily using the λ-lifting cartouche — corresponds to the low level term:

‹*defines* ((*assign-local tmp-update* (λσ. (*A* σ) ! *i*)) ;−
(*assign-global A-update* (λσ. *list-update* (*A* σ) (*i*) (*A* σ ! *j*))) ;−
(*assign-global A-update* (λσ. *list-update* (*A* σ) (*j*) ((*hd o tmp*) σ))))›

The effect of this statement is generation of the following definitions in the logical context:

term (i, j) — check that i and j are pointing to the constants defined before treating **function-spec**
thm *push-local-swap-state-def*
thm *pop-local-swap-state-def*
thm *swap-pre-def*
thm *swap-post-def*
thm *swap-core-def*
thm *swap-def*

The state-management is in the following configuration:

ML‹ *val Type(s,t) = StateMgt-core.get-state-type-global* @{*theory*};
StateMgt-core.get-state-field-tab-global @{*theory*}›

2.3.2 A Similation of `swap` in elementary specification constructs:

Note that we prime identifiers in order to avoid confusion with the definitions of the previous section. The pre- and postconditions are just definitions of the following form:

definition $swap'\text{-}pre :: \; nat \times nat \Rightarrow {'}a\ global\text{-}state\text{-}state\text{-}scheme \Rightarrow bool$
where $swap'\text{-}pre \equiv \lambda(i, j)\ \sigma.\ i < length\ (A\ \sigma) \wedge j < length\ (A\ \sigma)$
definition $swap'\text{-}post :: {'}a \times {'}b \Rightarrow {'}c\ global\text{-}state\text{-}state\text{-}scheme \Rightarrow {'}d\ global\text{-}state\text{-}state\text{-}scheme \Rightarrow unit \Rightarrow bool$
where $swap'\text{-}post \equiv \lambda(i, j)\ \sigma_{pre}\ \sigma\ res.\ length\ (A\ \sigma) = length\ (A\ \sigma_{pre}) \wedge res = ()$

The somewhat vacuous parameter *res* for the result of the swap-computation is the conseqeuence of the implicit definition of the return-type as *unit*

We simulate the effect of the local variable space declaration by the following command factoring out the functionality into the command *local-vars-test*

2.4 Encoding `partition` in Clean

2.4.1 `partition` in High-level Notation

function-spec *partition* (*lo*::*nat*, *hi*::*nat*) **returns** *nat*
pre ‹$lo < length\ A \wedge hi < length\ A$›
post ‹$\lambda res::nat.\ length\ A = length(old\ A) \wedge res = 3$›
local-vars *pivot* :: *int*
i :: *nat*
j :: *nat*
defines (‹$pivot := A\ !\ hi$› ;− ‹$i := lo$› ;− ‹$j := lo$› ;−
($while_C$ ‹$j \leq hi - 1$›
do (if_C ‹$A\ !\ j < pivot$›
then $call_C$ *swap* ‹(i , j)› ;−
‹$i := i + 1$›
else $skip_{SE}$
fi) ;−
‹$j := j + 1$›

od) ;−
call$_C$ *swap* ‹(*i*, *j*)› ;−
return$_C$ *result-value-update* ‹*i*›
)

The body is a fancy syntax for :

‹*defines* ((*assign-local pivot-update* ($\lambda\sigma$. *A* σ ! *hi*)) ;−
(*assign-local i-update* ($\lambda\sigma$. *lo*)) ;−

(*assign-local j-update* ($\lambda\sigma$. *lo*)) ;−
(*while*$_C$ ($\lambda\sigma$. (*hd o j*) $\sigma \leq$ *hi* − *1*)
do (*if*$_C$ ($\lambda\sigma$. *A* σ ! (*hd o j*) σ < (*hd o pivot*)σ)
then *call*$_C$ (*swap*) ($\lambda\sigma$. ((*hd o i*) σ, (*hd o j*) σ)) ;−
assign-local i-update ($\lambda\sigma$. ((*hd o i*) σ) + *1*)
else skip$_{SE}$
fi) ;−
(*assign-local j-update* ($\lambda\sigma$. ((*hd o j*) σ) + *1*))
od) ;−
call$_C$ (*swap*) ($\lambda\sigma$. ((*hd o i*) σ, (*hd o j*) σ)) ;−
assign-local result-value-update ($\lambda\sigma$. (*hd o i*) σ)
— the meaning of the return stmt
) ›

The effect of this statement is generation of the following definitions in the logical context:

thm *partition-pre-def*
thm *partition-post-def*
thm *push-local-partition-state-def*
thm *pop-local-partition-state-def*
thm *partition-core-def*
thm *partition-def*

The state-management is in the following configuration:

ML‹ *val Type*(*s*,*t*) = *StateMgt-core.get-state-type-global* @{*theory*};
StateMgt-core.get-state-field-tab-global @{*theory*}›

2.4.2 A Similation of `partition` in elementary specification constructs:

definition *partition′-pre* ≡ λ(*lo*, *hi*) σ. *lo* < *length* (*A* σ) ∧ *hi* < *length* (*A* σ)
definition *partition′-post* ≡ λ(*lo*, *hi*) σ_{pre} σ *res*. *length* (*A* σ) = *length* (*A* σ_{pre}) ∧ *res* = *3*

Recall: list-lifting is automatic in *local-vars-test*:

local-vars-test *partition′ nat*
pivot :: *int*
i :: *nat*
j :: *nat*

... which results in the internal definition of the respective push and pop operations for the *partition′* local variable space:

thm *push-local-partition′-state-def*
thm *pop-local-partition′-state-def*

definition *push-local-partition-state′* :: $(unit, 'a\ local\text{-}partition'\text{-}state\text{-}scheme)\ MON_{SE}$
where $push\text{-}local\text{-}partition\text{-}state'\ \sigma = Some((),$
$\sigma(\!|local\text{-}partition\text{-}state.pivot := undefined\ \#\ local\text{-}partition\text{-}state.pivot\ \sigma,$
$local\text{-}partition\text{-}state.i := undefined\ \#\ local\text{-}partition\text{-}state.i\ \sigma,$
$local\text{-}partition\text{-}state.j := undefined\ \#\ local\text{-}partition\text{-}state.j\ \sigma,$
$local\text{-}partition\text{-}state.result\text{-}value$
$:= undefined\ \#\ local\text{-}partition\text{-}state.result\text{-}value\ \sigma\ |\!))$

definition *pop-local-partition-state′* :: $(nat,'a\ local\text{-}partition\text{-}state\text{-}scheme)\ MON_{SE}$
where $pop\text{-}local\text{-}partition\text{-}state'\ \sigma = Some(hd(local\text{-}partition\text{-}state.result\text{-}value\ \sigma),$
$\sigma(\!|local\text{-}partition\text{-}state.pivot := tl(local\text{-}partition\text{-}state.pivot\ \sigma),$
$local\text{-}partition\text{-}state.i := tl(local\text{-}partition\text{-}state.i\ \sigma),$
$local\text{-}partition\text{-}state.j := tl(local\text{-}partition\text{-}state.j\ \sigma),$
$local\text{-}partition\text{-}state.result\text{-}value :=$
$tl(local\text{-}partition\text{-}state.result\text{-}value\ \sigma)\ |\!))$

definition *partition′-core* :: $nat \times nat \Rightarrow (unit,'a\ local\text{-}partition'\text{-}state\text{-}scheme)\ MON_{SE}$
where $partition'\text{-}core \equiv \lambda(lo,hi).$
$((assign\text{-}local\ pivot\text{-}update\ (\lambda\sigma.\ A\ \sigma\ !\ hi\ ))\ ;-$
$(assign\text{-}local\ i\text{-}update\ (\lambda\sigma.\ lo\ ))\ ;-$

$(assign\text{-}local\ j\text{-}update\ (\lambda\sigma.\ lo\ ))\ ;-$
$(while_C\ (\lambda\sigma.\ (hd\ o\ j)\ \sigma \leq hi - 1\ )$
$do\ (if_C\ (\lambda\sigma.\ A\ \sigma\ !\ (hd\ o\ j)\ \sigma < (hd\ o\ pivot)\sigma\ )$
$then\ \ call_C\ (swap)\ (\lambda\sigma.\ ((hd\ o\ i)\ \sigma,\ (hd\ o\ j)\ \sigma))\ ;-$
$assign\text{-}local\ i\text{-}update\ (\lambda\sigma.\ ((hd\ o\ i)\ \sigma) + 1)$
$else\ skip_{SE}$
$fi)$
$od)\ ;-$
$(assign\text{-}local\ j\text{-}update\ (\lambda\sigma.\ ((hd\ o\ j)\ \sigma) + 1))\ ;-$
$call_C\ (swap)\ (\lambda\sigma.\ ((hd\ o\ i)\ \sigma,\ (hd\ o\ j)\ \sigma))\ ;-$
$assign\text{-}local\ result\text{-}value\text{-}update\ (\lambda\sigma.\ (hd\ o\ i)\ \sigma)$
— the meaning of the return stmt
)

thm *partition-core-def*

definition *partition′* :: $nat \times nat \Rightarrow (nat,'a\ local\text{-}partition'\text{-}state\text{-}scheme)\ MON_{SE}$
where $partition' \equiv \lambda(lo,hi).\ block_C\ push\text{-}local\text{-}partition\text{-}state$
$(partition\text{-}core\ (lo,hi))$
$pop\text{-}local\text{-}partition\text{-}state$

2.5 Encoding the toplevel : `quicksort` in Clean

2.5.1 `quicksort` in High-level Notation

rec-function-spec *quicksort* (*lo*::*nat*, *hi*::*nat*) **returns** *unit*
pre ‹$lo \leq hi \wedge hi < length\ A$›
post ‹$\lambda res::unit.\ \forall i \in \{lo\ ..\ hi\}.\ \forall j \in \{lo\ ..\ hi\}.\ i \leq j \longrightarrow A!i \leq A!j$›
variant $hi - lo$
local-vars $p :: nat$
defines if_C ‹$lo < hi$›
then ($p_{tmp} \leftarrow call_C$ *partition* ‹(lo, hi)› ; *assign-local p-update* ($\lambda\sigma.\ p_{tmp}$)) ;−
$call_C$ *quicksort* ‹$(lo, p - 1)$› ;−
$call_C$ *quicksort* ‹$(lo, p + 1)$›
else $skip_{SE}$
fi

thm *quicksort-core-def*
thm *quicksort-def*
thm *quicksort-pre-def*
thm *quicksort-post-def*

2.5.2 A Similation of `quicksort` in elementary specification constructs:

This is the most complex form a Clean function may have: it may be directly recursive. Two subcases are to be distinguished: either a measure is provided or not.

We start again with our simulation: First, we define the local variable *p*.

local-vars-test *quicksort′ unit*
p :: *nat*

ML‹ *val* (*x*,*y*) = *StateMgt-core.get-data-global* @{*theory*}; ›

thm *pop-local-quicksort′-state-def*
thm *push-local-quicksort′-state-def*

definition *push-local-quicksort-state′* :: (*unit*, *′a local-quicksort′-state-scheme*) MON_{SE}
where *push-local-quicksort-state′* σ =
Some((), σ(|*local-quicksort′-state.p* := *undefined* # *local-quicksort′-state.p* σ,
local-quicksort′-state.result-value := *undefined* # *local-quicksort′-state.result-value*
σ |))

definition *pop-local-quicksort-state′* :: (*unit*,*′a local-quicksort′-state-scheme*) MON_{SE}
where *pop-local-quicksort-state′* σ = *Some*(*hd*(*local-quicksort′-state.result-value* σ),
σ(|*local-quicksort′-state.p* := *tl*(*local-quicksort′-state.p* σ),

$local\text{-}quicksort'\text{-}state.result\text{-}value :=$
$tl(local\text{-}quicksort'\text{-}state.result\text{-}value\ \sigma)\ \|))$

We recall the structure of the direct-recursive call in Clean syntax:

funct quicksort$(lo::int,\ hi::int)$ *returns unit*
pre True
post True
local-vars p :: int
$\langle if_{CLEAN}\ \langle lo < hi\rangle$ *then*
$p := partition(lo,\ hi)$;–
$quicksort(lo,\ p - 1)$;–
$quicksort(p + 1,\ hi)$
else Skip$\rangle$

definition $quicksort'\text{-}pre :: nat \times nat \Rightarrow 'a\ local\text{-}quicksort'\text{-}state\text{-}scheme \Rightarrow\ bool$
where $quicksort'\text{-}pre \equiv \lambda(i,j).\ \lambda\sigma.\ \ True$

definition $quicksort'\text{-}post :: nat \times nat \Rightarrow unit \Rightarrow 'a\ local\text{-}quicksort'\text{-}state\text{-}scheme \Rightarrow\ bool$
where $quicksort'\text{-}post \equiv \lambda(i,j).\ \lambda\ res.\ \lambda\sigma.\ \ True$

definition $quicksort'\text{-}core ::\ \ (nat \times nat \Rightarrow (unit,'a\ local\text{-}quicksort'\text{-}state\text{-}scheme)\ MON_{SE})$
$\Rightarrow (nat \times nat \Rightarrow (unit,'a\ local\text{-}quicksort'\text{-}state\text{-}scheme)\ MON_{SE})$
where $quicksort'\text{-}core\ quicksort\text{-}rec \equiv \lambda(lo,\ hi).$
$((if_C\ (\lambda\sigma.\ lo < hi\ )$
$then\ (p_{tmp} \leftarrow call_C\ partition\ (\lambda\sigma.\ (lo,\ hi))\ ;$
$assign\text{-}local\ p\text{-}update\ (\lambda\sigma.\ p_{tmp}))$;–
$call_C\ quicksort\text{-}rec\ (\lambda\sigma.\ (lo,\ (hd\ o\ p)\ \sigma - 1))$;–
$call_C\ quicksort\text{-}rec\ (\lambda\sigma.\ ((hd\ o\ p)\ \sigma + 1,\ hi))$
$else\ skip_{SE}$
$fi))$

term $((quicksort'\text{-}core\ X)\ (lo,hi))$

definition $quicksort' ::\ ((nat \times nat) \times (nat \times nat))\ set \Rightarrow$
$(nat \times nat \Rightarrow (unit,'a\ local\text{-}quicksort'\text{-}state\text{-}scheme)\ MON_{SE})$
where $quicksort'\ order \equiv wfrec\ order\ (\lambda X.\ \lambda(lo,\ hi).\ block_C\ push\text{-}local\text{-}quicksort'\text{-}state$
$(quicksort'\text{-}core\ X\ (lo,hi))$
$pop\text{-}local\text{-}quicksort'\text{-}state)$

2.5.3 Setup for Deductive Verification

The coupling between the pre- and the post-condition state is done by the free variable (serving as a kind of ghost-variable) σ_{pre}. This coupling can also be used to express framing conditions; i.e. parts of the state which are independent and/or not affected by the computations to be verified.

lemma *quicksort-correct* :
  $\{\!\!\{\lambda\sigma.\ \ \neg exec\text{-}stop\ \sigma \wedge quicksort\text{-}pre\ (lo,\ hi)(\sigma) \wedge \sigma = \sigma_{pre}\ \}\!\!\}$
    quicksort (*lo*, *hi*)
   $\{\!\!\{\lambda r\ \sigma.\ \neg exec\text{-}stop\ \sigma \wedge quicksort\text{-}post(lo,\ hi)(\sigma_{pre})(\sigma)(r)\ \}\!\!\}$
   oops

end

2.6 The Squareroot Example for Symbolic Execution

theory *SquareRoot-concept*
  imports *Clean-Main.Test-Clean*
begin

2.6.1 The Conceptual Algorithm in Clean Notation

In high-level notation, the algorithm we are investigating looks like this:

```
‹
function-spec sqrt (a::int) returns int
pre          ‹0 ≤ a›
post         ‹λres::int. (res + 1)² > a ∧ a ≥ (res)²›
defines       (‹tm := 1› ;−
              ‹sqsum := 1› ;−
              ‹i := 0› ;−
              (while_SE ‹sqsum <= a› do
                 ‹i := i+1› ;−
                 ‹tm := tm + 2› ;−
                 ‹sqsum := tm + sqsum›
              od) ;−
              return_C result-value-update ‹i›
              )
›
```

2.6.2 Definition of the Global State

The state is just a record; and the global variables correspond to fields in this record. This corresponds to typed, structured, non-aliasing states. Note that the types in the state can be arbitrary HOL-types - want to have sets of functions in a ghost-field ? No problem !

The state of the square-root program looks like this :

typ *Clean.control-state*

ML‹
val Type(s,t) = StateMgt-core.get-state-type-global @{*theory*}
val Type(u,v) = @{*typ unit*}
›

global-vars *state*
  tm :: *int*
  i :: *int*
  sqsum :: *int*

ML‹
val Type(s,t) = StateMgt-core.get-state-type-global @{*theory*}
val Type(u,v) = @{*typ unit*}
›

lemma *tm-independent* [*simp*]: $\sharp$ *tm-update*
  unfolding *control-independence-def* **by** *auto*

lemma *i-independent* [*simp*]: $\sharp$ *i-update*
  unfolding *control-independence-def* **by** *auto*

lemma *sqsum-independent* [*simp*]: $\sharp$ *sqsum-update*
  unfolding *control-independence-def* **by** *auto*

2.6.3 Setting for Symbolic Execution

Some lemmas to reason about memory

lemma *tm-simp* : $tm\ (\sigma(\!|tm := t|\!)) = t$
  using [[*simp-trace*]] **by** *simp*

lemma *tm-simp1* : $tm\ (\sigma(\!|sqsum := s|\!)) = tm\ \sigma$ **by** *simp*
lemma *tm-simp2* : $tm\ (\sigma(\!|i := s|\!)) = tm\ \sigma$ **by** *simp*
lemma *sqsum-simp* : $sqsum\ (\sigma(\!|sqsum := s|\!)) = s$ **by** *simp*
lemma *sqsum-simp1* : $sqsum\ (\sigma(\!|tm := t|\!)) = sqsum\ \sigma$ **by** *simp*
lemma *sqsum-simp2* : $sqsum\ (\sigma(\!|i := t|\!)) = sqsum\ \sigma$ **by** *simp*
lemma *i-simp* : $i\ (\sigma(\!|i := i'|\!)) = i'$ **by** *simp*
lemma *i-simp1* : $i\ (\sigma(\!|tm := i'|\!)) = i\ \sigma$ **by** *simp*
lemma *i-simp2* : $i\ (\sigma(\!|sqsum := i'|\!)) = i\ \sigma$ **by** *simp*

lemmas *memory-theory* =
  tm-simp tm-simp1 tm-simp2
  sqsum-simp sqsum-simp1 sqsum-simp2
  i-simp i-simp1 i-simp2

declare *memory-theory* [*memory-theory*]

lemma *non-exec-assign-globalD'*:
  assumes ♯ *upd*
  shows $\sigma \models$ *assign-global upd rhs* ;− $M \Longrightarrow \neg$ *exec-stop* $\sigma \Longrightarrow$ *upd* (λ-. *rhs* σ) $\sigma \models M$
  apply(*drule non-exec-assign-global'*[*THEN iffD1*])
  using *assms exec-stop-vs-control-independence* **apply** *blast*
  by *auto*

lemmas *non-exec-assign-globalD'-tm* = *non-exec-assign-globalD'*[*OF tm-independent*]
lemmas *non-exec-assign-globalD'-i* = *non-exec-assign-globalD'*[*OF i-independent*]
lemmas *non-exec-assign-globalD'-sqsum* = *non-exec-assign-globalD'*[*OF sqsum-independent*]

Now we run a symbolic execution. We run match-tactics (rather than the Isabelle simplifier which would do the trick as well) in order to demonstrate a symbolic execution in Isabelle.

2.6.4 A Symbolic Execution Simulation

lemma
  assumes *non-exec-stop*[*simp*]: $\neg$ *exec-stop* σ_0
   and   *pos* : $0 \leq (a{::}int)$
   and   *annotated-program*:
      $\sigma_0 \models$ ⟨*tm* := *1*⟩ ;−
        ⟨*sqsum* := *1*⟩ ;−
        ⟨*i* := *0*⟩ ;−
        (*while*$_{SE}$ ⟨*sqsum* <= *a*⟩ *do*
          ⟨*i* := *i*+*1*⟩ ;−
          ⟨*tm* := *tm* + *2*⟩ ;−
          ⟨*sqsum* := *tm* + *sqsum*⟩
        od) ;−
        assert$_{SE}$($\lambda\sigma.\ \sigma{=}\sigma_R$)

    shows $\sigma_R \models$*assert*$_{SE}$ ⟨$i^2 \leq a \wedge a < (i + 1)^2$⟩

  apply(*insert annotated-program*)

  apply(*tactic dmatch-tac* @{*context*} [@{*thm non-exec-assign-globalD'-tm*}] *1*,*simp*)
  apply(*tactic dmatch-tac* @{*context*} [@{*thm non-exec-assign-globalD'-sqsum*}] *1*,*simp*)
  apply(*tactic dmatch-tac* @{*context*} [@{*thm non-exec-assign-globalD'-i*}] *1*,*simp*)

  apply(*tactic dmatch-tac* @{*context*} [@{*thm exec-whileD*}] *1*)
  apply(*tactic ematch-tac* @{*context*} [@{*thm if-SE-execE''*}] *1*)
   apply(*simp-all only*: *memory-theory MonadSE.bind-assoc'*)

   apply(*tactic dmatch-tac* @{*context*} [@{*thm non-exec-assign-globalD'-i*}] *1*,*simp*)
   apply(*tactic dmatch-tac* @{*context*} [@{*thm non-exec-assign-globalD'-tm*}] *1*,*simp*)

apply(*tactic dmatch-tac* @{*context*} [@{*thm non-exec-assign-globalD′-sqsum*}] *1*,*simp*)

apply(*tactic dmatch-tac* @{*context*} [@{*thm exec-whileD*}] *1*)
apply(*tactic ematch-tac* @{*context*} [@{*thm if-SE-execE′′*}] *1*)
apply(*simp-all only*: *memory-theory MonadSE.bind-assoc′*)

apply(*tactic dmatch-tac* @{*context*} [@{*thm non-exec-assign-globalD′-i*}] *1*,*simp*)
apply(*tactic dmatch-tac* @{*context*} [@{*thm non-exec-assign-globalD′-tm*}] *1*,*simp*)
apply(*tactic dmatch-tac* @{*context*} [@{*thm non-exec-assign-globalD′-sqsum*}] *1*,*simp*)

apply(*tactic dmatch-tac* @{*context*} [@{*thm exec-whileD*}] *1*)
apply(*tactic ematch-tac* @{*context*} [@{*thm if-SE-execE′′*}] *1*)
apply(*simp-all only*: *memory-theory MonadSE.bind-assoc′*)

apply(*tactic dmatch-tac* @{*context*} [@{*thm non-exec-assign-globalD′-i*}] *1*,*simp*)
apply(*tactic dmatch-tac* @{*context*} [@{*thm non-exec-assign-globalD′-tm*}] *1*,*simp*)
apply(*tactic dmatch-tac* @{*context*} [@{*thm non-exec-assign-globalD′-sqsum*}] *1*,*simp*)
apply(*simp-all*)

Here are all abstract test-cases explicit. Each subgoal correstponds to a path taken through the loop.

push away the test-hyp: postcond is true for programs with more than three loop traversals (criterion: all-paths(k). This reveals explicitly the three test-cases for $k < (3::'b)$.

defer *1*

oops

TODO: re-establish automatic test-coverage tactics of [4].

end

3 Appendix : Used Monad Libraries

theory *MonadSE*
  imports *Main*
begin

3.1 Definition : Standard State Exception Monads

State exception monads in our sense are a direct, pure formulation of automata with a partial transition function.

3.1.1 Definition : Core Types and Operators

type-synonym $('o, '\sigma)\ MON_{SE} = '\sigma \rightharpoonup ('o \times '\sigma)$

definition $bind\text{-}SE :: ('o,'\sigma)MON_{SE} \Rightarrow ('o \Rightarrow ('o','\sigma)MON_{SE}) \Rightarrow ('o','\sigma)MON_{SE}$
where $bind\text{-}SE\ f\ g = (\lambda\sigma.\ case\ f\ \sigma\ of\ None \Rightarrow None$
$\qquad\qquad\qquad\qquad | \ Some\ (out, \sigma') \Rightarrow g\ out\ \sigma')$

notation $bind\text{-}SE\ (bind_{SE})$

syntax $(xsymbols)$
$\quad$ $\text{-}bind\text{-}SE :: [pttrn,('o,'\sigma)MON_{SE},('o','\sigma)MON_{SE}] \Rightarrow ('o','\sigma)MON_{SE}$
$\quad$ $((2\ \text{-}\leftarrow\ \text{-};\ \text{-})\ [5,8,8]8)$
translations
$\quad$ $x \leftarrow f;\ g == CONST\ bind\text{-}SE\ f\ (\%\ x\ .\ g)$

definition $unit\text{-}SE :: 'o \Rightarrow ('o, '\sigma)MON_{SE}\quad ((result\ \text{-})\ 8)$
where $unit\text{-}SE\ e = (\lambda\sigma.\ Some(e,\sigma))$
notation $unit\text{-}SE\ (unit_{SE})$

In the following, we prove the required Monad-laws

lemma $bind\text{-}right\text{-}unit[simp]$: $(x \leftarrow m;\ result\ x) = m$
  apply (*simp add: unit-SE-def bind-SE-def*)
  apply (*rule ext*)
  apply (*case-tac m* σ, *simp-all*)
  done

lemma *bind-left-unit* [*simp*]: $(x \leftarrow \textit{result } c;\ P\ x) = P\ c$
by (*simp add*: *unit-SE-def bind-SE-def*)

lemma *bind-assoc*[*simp*]: $(y \leftarrow (x \leftarrow m;\ k\ x);\ h\ y) = (x \leftarrow m;\ (y \leftarrow k\ x;\ h\ y))$
apply (*simp add*: *unit-SE-def bind-SE-def*, *rule ext*)
apply (*case-tac m* σ, *simp-all*)
apply (*case-tac a*, *simp-all*)
done

3.1.2 Definition : More Operators and their Properties

definition *fail-SE* :: $('o,\ '\sigma)MON_{SE}$
where *fail-SE* = ($\lambda\sigma$. *None*)
notation *fail-SE* ($fail_{SE}$)

definition *assert-SE* :: $('\sigma \Rightarrow bool) \Rightarrow (bool,\ '\sigma)MON_{SE}$
where *assert-SE P* = ($\lambda\sigma$. *if P* σ *then Some*(*True*,σ) *else None*)
notation *assert-SE* ($assert_{SE}$)

definition *assume-SE* :: $('\sigma \Rightarrow bool) \Rightarrow (unit,\ '\sigma)MON_{SE}$
where *assume-SE P* = ($\lambda\sigma$. *if* $\exists\sigma$. *P* σ *then Some*((), *SOME* σ . *P* σ) *else None*)
notation *assume-SE* ($assume_{SE}$)

lemma *bind-left-fail-SE*[*simp*] : $(x \leftarrow fail_{SE};\ P\ x) = fail_{SE}$
by (*simp add*: *fail-SE-def bind-SE-def*)

We also provide a "Pipe-free" - variant of the bind operator. Just a "standard" programming sequential operator without output frills.

definition *bind-SE'* :: $('\alpha,\ '\sigma)MON_{SE} \Rightarrow ('\beta,\ '\sigma)MON_{SE} \Rightarrow ('\beta,\ '\sigma)MON_{SE}$ (**infixr** ;– *60*)
where f ;– g = (- $\leftarrow$ f ; g)

lemma *bind-assoc'*[*simp*]: ((*m*;– *k*);– *h*) = (*m*;– (*k*;– *h*))
by(*simp add*:*bind-SE'-def*)

lemma *bind-left-unit'* [*simp*]: ((*result c*);– *P*) = *P*
by (*simp add*: *bind-SE'-def*)

lemma *bind-left-fail-SE'*[*simp*]: ($fail_{SE}$;– *P*) = $fail_{SE}$
by (*simp add*: *bind-SE'-def*)

lemma *bind-right-unit'*[*simp*]: (*m*;– (*result* ())) = *m*
by (*simp add*: *bind-SE'-def*)

The bind-operator in the state-exception monad yields already a semantics for the concept of an input sequence on the meta-level:

lemma *syntax-test*: (*o1* $\leftarrow$ *f1* ; *o2* $\leftarrow$ *f2*; *result* (*post o1 o2*)) = *X*

oops

definition $yield_C$ :: $('a \Rightarrow 'b) \Rightarrow ('b,'a)\ MON_{SE}$
where $yield_C\ f \equiv (\lambda\sigma.\ Some(f\ \sigma, \sigma))$

definition *try-SE* :: $('o,'\sigma)\ MON_{SE} \Rightarrow ('o\ option,'\sigma)\ MON_{SE}\ (try_{SE})$
where $try_{SE}\ ioprog = (\lambda\sigma.\ case\ ioprog\ \sigma\ of$
$None \Rightarrow Some(None, \sigma)$
$|\ Some(outs, \sigma') \Rightarrow Some(Some\ outs, \sigma'))$

In contrast, mbind as a failure safe operator can roughly be seen as a foldr on bind - try: m1 ; try m2 ; try m3; ... Note, that the rough equivalence only holds for certain predicates in the sequence - length equivalence modulo None, for example. However, if a conditional is added, the equivalence can be made precise:

On this basis, a symbolic evaluation scheme can be established that reduces mbind-code to try_SE_code and ite-cascades.

definition *alt-SE* :: $[('o, '\sigma)MON_{SE}, ('o, '\sigma)MON_{SE}] \Rightarrow ('o, '\sigma)MON_{SE}$ (**infixl** $\sqcap_{SE}$ *10*)
where $(f \sqcap_{SE} g) = (\lambda\ \sigma.\ case\ f\ \sigma\ of\ None \Rightarrow g\ \sigma$
$|\ Some\ H \Rightarrow Some\ H)$

definition *malt-SE* :: $('o, '\sigma)MON_{SE}\ list \Rightarrow ('o, '\sigma)MON_{SE}$
where *malt-SE S = foldr alt-SE S* $fail_{SE}$
notation *malt-SE* $(\bigsqcap_{SE})$

lemma *malt-SE-mt* [*simp*]: $\bigsqcap_{SE}\ [] = fail_{SE}$
by(*simp add*: *malt-SE-def*)

lemma *malt-SE-cons* [*simp*]: $\bigsqcap_{SE}\ (a\ \#\ S) = (a \sqcap_{SE} (\bigsqcap_{SE}\ S))$
by(*simp add*: *malt-SE-def*)

3.1.3 Definition : Programming Operators and their Properties

definition $skip_{SE} = unit_{SE}\ ()$

definition *if-SE* :: $['\sigma \Rightarrow bool, ('\alpha, '\sigma)MON_{SE}, ('\alpha, '\sigma)MON_{SE}] \Rightarrow ('\alpha, '\sigma)MON_{SE}$
where *if-SE c E F* $= (\lambda\sigma.\ if\ c\ \sigma\ then\ E\ \sigma\ else\ F\ \sigma)$

syntax (*xsymbols*)
-if-SE :: $['\sigma \Rightarrow bool,('o,'\sigma)MON_{SE},('o','\sigma)MON_{SE}] \Rightarrow ('o','\sigma)MON_{SE}$
$((if_{SE}$ *- then - else -fi*) *[5,8,8]8*)
translations
$(if_{SE}$ *cond then T1 else T2 fi*) == *CONST if-SE cond T1 T2*

3.1.4 Theory of a Monadic While

Prerequisites

fun *replicator* :: $[('a, '\sigma)MON_{SE}, nat] \Rightarrow (unit, '\sigma)MON_{SE}$ (**infixr** ^^^ *60*)

where f ^^^ 0 $= (result\ ())$
$\mid f$ ^^^ $(Suc\ n) = (f\ ;-\ f$ ^^^ $n)$

fun *replicator2* :: $[('a, '\sigma)MON_{SE}, nat, ('b, '\sigma)MON_{SE}] \Rightarrow ('b, '\sigma)MON_{SE}$ (**infixr** ^:^ *60*)
where $(f$ ^:^ $0)\ M = (M\ )$
$\mid (f$ ^:^ $(Suc\ n))\ M = (f\ ;-\ ((f$ ^:^ $n)\ M))$

First Step : Establishing an embedding between partial functions and relations

definition *Mon2Rel* :: $(unit, '\sigma)MON_{SE} \Rightarrow ('\sigma \times '\sigma)\ set$
where *Mon2Rel f* $= \{(x, y).\ (f\ x = Some((), y))\}$

definition *Rel2Mon* :: $('\sigma \times '\sigma)\ set \Rightarrow (unit, '\sigma)MON_{SE}$
where *Rel2Mon S* $= (\lambda\ \sigma.\ if\ \exists\sigma'.\ (\sigma, \sigma') \in S\ then\ Some((), SOME\ \sigma'.\ (\sigma, \sigma') \in S)\ else\ None)$

lemma *Mon2Rel-Rel2Mon-id*: **assumes** *det*:*single-valued R* **shows** $(Mon2Rel \circ Rel2Mon)\ R = R$
apply (*simp add*: *comp-def Mon2Rel-def Rel2Mon-def*,*auto*)
apply (*case-tac* $\exists\sigma'.\ (a, \sigma') \in R$, *auto*)
apply (*subst* (*2*) *some-eq-ex*)
using *det*[*simplified single-valued-def*] **by** *auto*

lemma *Rel2Mon-Id*: $(Rel2Mon \circ Mon2Rel)\ x = x$
apply (*rule ext*)
apply (*auto simp*: *comp-def Mon2Rel-def Rel2Mon-def*)
apply (*erule contrapos-pp*, *drule HOL.not-sym*, *simp*)
done

lemma *single-valued-Mon2Rel*: *single-valued* (*Mon2Rel B*)
by (*auto simp*: *single-valued-def Mon2Rel-def*)

Second Step : Proving an induction principle allowing to establish that lfp remains deterministic

definition *chain* :: $(nat \Rightarrow 'a\ set) \Rightarrow bool$
where $chain\ S = (\forall\ i.\ S\ i \subseteq S(Suc\ i))$

lemma *chain-total*: $chain\ S ==> S\ i \leq S\ j \vee S\ j \leq S\ i$
by (*metis chain-def le-cases lift-Suc-mono-le*)

definition *cont* :: $('a\ set => 'b\ set) => bool$
where $cont\ f = (\forall\ S.\ chain\ S \longrightarrow f(UN\ n.\ S\ n) = (UN\ n.\ f(S\ n)))$

lemma *mono-if-cont*: **fixes** $f :: 'a\ set \Rightarrow 'b\ set$
  assumes *cont f* **shows** *mono f*
proof
  fix $a\ b :: 'a\ set$ **assume** $a \subseteq b$
  let $?S = \lambda n{::}nat.\ if\ n{=}0\ then\ a\ else\ b$
  have *chain ?S* **using** $\langle a \subseteq b\rangle$ **by**(*auto simp*: *chain-def*)

hence $f(UN\ n.\ ?S\ n) = (UN\ n.\ f(?S\ n))$
  using *assms* **by** (*metis cont-def*)
moreover have $(UN\ n.\ ?S\ n) = b$ **using** ‹$a \subseteq b$› **by** (*auto split*: *if-splits*)
moreover have $(UN\ n.\ f(?S\ n)) = f\ a \cup f\ b$ **by** (*auto split*: *if-splits*)
ultimately show $f\ a \subseteq f\ b$ **by** (*metis Un-upper1*)
qed

lemma *chain-iterates*: **fixes** $f :: 'a\ set \Rightarrow 'a\ set$
assumes *mono f* **shows** $chain(\lambda n.\ (f\verb|^^|n)\ \{\})$
proof−
{ fix n **have** $(f\ \verb|^^|\ n)\ \{\} \subseteq (f\ \verb|^^|\ Suc\ n)\ \{\}$ **using** *assms*
  by(*induction n*) (*auto simp*: *mono-def*) **}**
thus *?thesis* **by**(*auto simp*: *chain-def*)
qed

theorem *lfp-if-cont*:
assumes *cont f* **shows** $lfp\ f = (\bigcup n.\ (f\ \verb|^^|\ n)\ \{\})$ (**is** - = *?U*)
proof
show $lfp\ f \subseteq ?U$
proof (*rule lfp-lowerbound*)
  have $f\ ?U = (UN\ n.\ (f\verb|^^|Suc\ n)\{\})$
    using *chain-iterates*[*OF mono-if-cont*[*OF assms*]] *assms*
    by(*simp add*: *cont-def*)
  also have $\ldots = (f\verb|^^|0)\{\} \cup \ldots$ **by** *simp*
  also have $\ldots = ?U$
    apply(*auto simp del*: *funpow.simps*)
    by (*metis empty-iff funpow-0 old.nat.exhaust*)
  finally show $f\ ?U \subseteq ?U$ **by** *simp*
qed
next
{ fix $n\ p$ **assume** $f\ p \subseteq p$
  have $(f\verb|^^|n)\{\} \subseteq p$
  proof(*induction n*)
    case *0* **show** *?case* **by** *simp*
  next
    case *Suc*
    from *monoD*[*OF mono-if-cont*[*OF assms*] *Suc*] ‹$f\ p \subseteq p$›
    show *?case* **by** *simp*
  qed
}
thus $?U \subseteq lfp\ f$ **by**(*auto simp*: *lfp-def*)
qed

lemma *single-valued-UN-chain*:
assumes *chain S* $(!!n.\ single\text{-}valued\ (S\ n))$
shows $single\text{-}valued(UN\ n.\ S\ n)$
proof(*auto simp*: *single-valued-def*)
fix $m\ n\ x\ y\ z$ **assume** $(x,\ y) \in S\ m\ (x,\ z) \in S\ n$

with *chain-total*[*OF assms*(*1*), *of m n*] *assms*(*2*)
show $y = z$ **by** (*auto simp*: *single-valued-def*)
qed

lemma *single-valued-lfp*:
fixes f :: $('a \times 'a)$ *set* $\Rightarrow$ $('a \times 'a)$ *set*
assumes *cont f* $\bigwedge r$. *single-valued r* $\Longrightarrow$ *single-valued* (*f r*)
shows *single-valued*(*lfp f*)
unfolding *lfp-if-cont*[*OF assms*(*1*)]
proof(*rule single-valued-UN-chain*[*OF chain-iterates*[*OF mono-if-cont*[*OF assms*(*1*)]]])
fix *n* **show** *single-valued* ((*f* ^^ *n*) {})
by(*induction n*)(*auto simp*: *assms*(*2*))
qed

Third Step: Definition of the Monadic While

definition Γ :: $['\sigma \Rightarrow bool,('\sigma \times '\sigma)\ set] \Rightarrow (('\sigma \times '\sigma)\ set \Rightarrow ('\sigma \times '\sigma)\ set)$
where $\Gamma\ b\ cd = (\lambda cw.\ \{(s,t).\ if\ b\ s\ then\ (s,\ t) \in cd\ O\ cw\ else\ s = t\})$

definition *while-SE* :: $['\sigma \Rightarrow bool,\ (unit,\ '\sigma)MON_{SE}] \Rightarrow (unit,\ '\sigma)MON_{SE}$
where *while-SE c B* $\equiv$ (*Rel2Mon*(*lfp*(Γ *c* (*Mon2Rel B*))))

syntax (*xsymbols*)
-*while-SE* :: $['\sigma \Rightarrow bool,\ (unit,\ '\sigma)MON_{SE}] \Rightarrow (unit,\ '\sigma)MON_{SE}$
(($while_{SE}$ - *do* - *od*) [*8*,*8*]*8*)
translations
$while_{SE}$ *c do b od* == *CONST while-SE c b*

lemma *cont*-Γ: *cont* (Γ *c b*)
by (*auto simp*: *cont-def* Γ-*def*)

The fixpoint theory now allows us to establish that the lfp constructed over *Mon2Rel* remains deterministic

theorem *single-valued-lfp-Mon2Rel*: *single-valued* (*lfp*(Γ *c* (*Mon2Rel B*)))
apply(*rule single-valued-lfp*, *simp-all add*: *cont*-Γ)
apply(*auto simp*: Γ-*def single-valued-def*)
apply(*metis single-valued-Mon2Rel*[*of B*] *single-valued-def*)
done

lemma *Rel2Mon-if*:
Rel2Mon $\{(s,\ t).\ if\ b\ s\ then\ (s,\ t) \in Mon2Rel\ c\ O\ lfp\ (\Gamma\ b\ (Mon2Rel\ c))\ else\ s = t\}\ \sigma =$
$(if\ b\ \sigma\ then\ Rel2Mon\ (Mon2Rel\ c\ O\ lfp\ (\Gamma\ b\ (Mon2Rel\ c)))\ \sigma\ else\ Some\ ((),\ \sigma))$
by (*simp add*: *Rel2Mon-def*)

lemma *Rel2Mon-homomorphism*:
assumes *determ-X*: *single-valued X* **and** *determ-Y*: *single-valued Y*
shows *Rel2Mon* (*X O Y*) = (*Rel2Mon X*) ;− (*Rel2Mon Y*)
proof −

have *relational-partial-next-in-O*: $\bigwedge x\ E\ F.\ (\exists\, y.\ (x,\ y) \in (E\ O\ F)) \Longrightarrow (\exists\, y.\ (x,\ y) \in E)$
by (*auto*)
have *some-eq-intro*: $\bigwedge\ X\ x\ y\ .\ single\text{-}valued\ X \Longrightarrow (x,\ y) \in X \Longrightarrow (SOME\ y.\ (x,\ y) \in X) = y$
by (*auto simp*: *single-valued-def*)

show *?thesis*
apply (*simp add*: *Rel2Mon-def bind-SE$'$-def bind-SE-def*)
apply (*rule ext*, *rename-tac* σ)
apply (*case-tac* $\exists\ \sigma'.\ (\sigma,\ \sigma') \in X\ O\ Y$)
apply (*simp only*: *HOL.if-True*)
apply (*frule relational-partial-next-in-O*)
apply (*auto simp*: *single-valued-relcomp some-eq-intro determ-X determ-Y relcomp.relcompI*)
by *blast*
qed

Putting everything together, the theory of embedding and the invariance of determinism of the while-body, gives us the usual unfold-theorem:

theorem *while-SE-unfold*:
$(while_{SE}\ b\ do\ c\ od) = (if_{SE}\ b\ then\ (c\ ;-\ (while_{SE}\ b\ do\ c\ od))\ else\ result\ ()\ fi)$
apply (*simp add*: *if-SE-def bind-SE$'$-def while-SE-def unit-SE-def*)
apply (*subst lfp-unfold* [*OF mono-if-cont*, *OF cont-*Γ])
apply (*rule ext*)
apply (*subst* Γ*-def*)
apply (*auto simp*: *Rel2Mon-if Rel2Mon-homomorphism bind-SE$'$-def Rel2Mon-Id* [*simplified comp-def*]
single-valued-Mon2Rel single-valued-lfp-Mon2Rel)
done

lemma *bind-cong* : $f\ \sigma = g\ \sigma \Longrightarrow\ (x \leftarrow f\ ;\ M\ x)\sigma = (x \leftarrow g\ ;\ M\ x)\sigma$
unfolding *bind-SE$'$-def bind-SE-def* **by** *simp*

lemma *bind$'$-cong* : $f\ \sigma = g\ \sigma \Longrightarrow\ (f\ ;-\ M\ )\sigma = (g\ ;-\ M\ )\sigma$
unfolding *bind-SE$'$-def bind-SE-def* **by** *simp*

lemma if_{SE}*-True* [*simp*]: $(if_{SE}\ (\lambda\ x.\ True)\ then\ c\ else\ d\ fi) = c$
apply(*rule ext*) **by** (*simp add*: *MonadSE.if-SE-def*)

lemma if_{SE}*-False* [*simp*]: $(if_{SE}\ (\lambda\ x.\ False)\ then\ c\ else\ d\ fi) = d$
apply(*rule ext*) **by** (*simp add*: *MonadSE.if-SE-def*)

lemma if_{SE}*-cond-cong* : $f\ \sigma = g\ \sigma \Longrightarrow$
$(if_{SE}\ f\ then\ c\ else\ d\ fi)\ \sigma =$
$(if_{SE}\ g\ then\ c\ else\ d\ fi)\ \sigma$
unfolding *if-SE-def* **by** *simp*

lemma *while*$_{SE}$*-skip*[*simp*] : (*while*$_{SE}$ (λ *x. False*) *do c od*) = *skip*$_{SE}$
apply(*rule ext,subst MonadSE.while-SE-unfold*)
by (*simp add*: *MonadSE.if-SE-def skip*$_{SE}$*-def*)

end

theory *Seq-MonadSE*
imports *MonadSE*
begin

3.1.5 Chaining Monadic Computations : Definitions of Multi-bind Operators

In order to express execution sequences inside HOL— rather than arguing over a certain pattern of terms on the meta-level — and in order to make our theory amenable to formal reasoning over execution sequences, we represent them as lists of input and generalize the bind-operator of the state-exception monad accordingly. The approach is straightforward, but comes with a price: we have to encapsulate all input and output data into one type, and restrict ourselves to a uniform step function. Assume that we have a typed interface to a module with the operations $op_1, op_2, \ldots, op_n$ with the inputs $\iota_1, \iota_2, \ldots, \iota_n$ (outputs are treated analogously). Then we can encode for this interface the general input - type:

$$\texttt{datatype in} = op_1 :: \iota_1 \mid \ldots \mid \iota_n$$

Obviously, we loose some type-safety in this approach; we have to express that in traces only *corresponding* input and output belonging to the same operation will occur; this form of side-conditions have to be expressed inside HOL. From the user perspective, this will not make much difference, since junk-data resulting from too weak typing can be ruled out by adopted front-ends.

Note that the subsequent notion of a test-sequence allows the io stepping function (and the special case of a program under test) to stop execution *within* the sequence; such premature terminations are characterized by an output list which is shorter than the input list.

Intuitively, *mbind* corresponds to a sequence of operation calls, separated by ";", in Java. The operation calls may fail (raising an exception), which means that the state is maintained and the exception can still be caught at the end of the execution sequence.

fun *mbind* :: $'\iota$ *list* $\Rightarrow$ ($'\iota \Rightarrow$ ($'o$,$'\sigma$) *MON*$_{SE}$) $\Rightarrow$ ($'o$ *list*,$'\sigma$) *MON*$_{SE}$
where *mbind* [] *iostep* σ = *Some*([], σ)
| *mbind* (*a#S*) *iostep* σ =
(*case iostep a* σ *of*
None $\Rightarrow$ *Some*([], σ)
| *Some* (*out*, σ') $\Rightarrow$ (*case mbind S iostep* σ' *of*
None $\Rightarrow$ *Some*([*out*],σ')

$| \; Some(outs,\sigma'') \Rightarrow Some(out\#outs,\sigma'')))$

notation *mbind* ($mbind_{FailSave}$)

This definition is fail-safe; in case of an exception, the current state is maintained, the computation as a whole is marked as success. Compare to the fail-strict variant *mbind'*:

lemma *mbind-unit* [*simp*]:
  mbind [] f = (*result* [])
   by(*rule ext*, *simp add*: *unit-SE-def*)

The characteristic property of $mbind_{FailSave}$ — which distinguishes it from *mbind* defined in the sequel — is that it never fails; it "swallows" internal errors occuring during the computation.

lemma *mbind-nofailure* [*simp*]:
  mbind S f $\sigma \neq$ *None*
   apply(*rule-tac* $x{=}\sigma$ **in** *spec*)
   apply(*induct S*, *auto simp*:*unit-SE-def*)
   apply(*case-tac f a x*, *auto*)
   apply(*erule-tac* $x{=}b$ **in** *allE*)
   apply(*erule exE*, *erule exE*, *simp*)
   done

In contrast, we define a fail-strict sequential execution operator. He has more the characteristic to fail globally whenever one of its operation steps fails.

Intuitively speaking, *mbind'* corresponds to an execution of operations where a results in a System-Halt. Another interpretation of *mbind'* is to view it as a kind of *foldl* foldl over lists via $bind_{SE}$.

fun $mbind' :: '\iota\ list \Rightarrow ('\iota \Rightarrow ('o,'\sigma)\ MON_{SE}) \Rightarrow ('o\ list,'\sigma)\ MON_{SE}$
where $mbind'\ []\ iostep\ \sigma = Some([], \sigma)\ |$
  $mbind'\ (a\#S)\ iostep\ \sigma =$
    $(case\ iostep\ a\ \sigma\ of$
      $None \Rightarrow None$
     $|\ Some\ (out, \sigma') \Rightarrow (case\ mbind'\ S\ iostep\ \sigma'\ of$
          $None \Rightarrow None$ — fail-strict
         $|\ Some(outs,\sigma'') \Rightarrow Some(out\#outs,\sigma'')))$
notation *mbind'* ($mbind_{FailStop}$)

lemma *mbind'-unit* [*simp*]:
  mbind' [] f = (*result* [])
   by(*rule ext*, *simp add*: *unit-SE-def*)

lemma *mbind'-bind* [*simp*]:
  $(x \leftarrow mbind'\ (a\#S)\ F;\ M\ x) = (a \leftarrow (F\ a);\ (x \leftarrow mbind'\ S\ F;\ M\ (a\ \#\ x)))$
   by(*rule ext*, *rename-tac z*,*simp add*: *bind-SE-def split*: *option.split*)

declare *mbind'.simps*[*simp del*]

The next *mbind* sequential execution operator is called Fail-Purge. He has more the characteristic to never fail, just "stuttering" above operation steps that fail. Another alternative in modeling.

fun $mbind'' :: {}'\iota\ list \Rightarrow ({}'\iota \Rightarrow ({}'o,{}'\sigma)\ MON_{SE}) \Rightarrow ({}'o\ list,{}'\sigma)\ MON_{SE}$
where $mbind''\ []\ iostep\ \sigma = Some([], \sigma)\ |$
$\quad mbind''\ (a\#S)\ iostep\ \sigma =$
$\qquad (case\ iostep\ a\ \sigma\ of$
$\qquad\quad None \qquad\qquad \Rightarrow mbind''\ S\ iostep\ \sigma$
$\qquad\ |\ Some\ (out, \sigma') \Rightarrow (case\ mbind''\ S\ iostep\ \sigma'\ of$
$\qquad\qquad\qquad\qquad None \quad \Rightarrow None$ — does not occur
$\qquad\qquad\qquad\quad |\ Some(outs,\sigma'') \Rightarrow Some(out\#outs,\sigma'')))$

notation $mbind''\ (mbind_{FailPurge})$
declare $mbind''.simps[simp\ del]$

mbind' as failure strict operator can be seen as a foldr on bind - if the types would match . . .

Definition : Miscellaneous Operators and their Properties

lemma *mbind-try*:
$\ (x \leftarrow mbind\ (a\#S)\ F;\ M\ x) =$
$\ \ (a' \leftarrow try_{SE}(F\ a);$
$\quad if\ a' = None$
$\quad then\ (M\ [])$
$\quad else\ (x \leftarrow mbind\ S\ F;\ M\ (the\ a'\ \#\ x)))$
apply(*rule ext*)
apply(*simp add*: *bind-SE-def try-SE-def*)
apply(*case-tac F a x*, *auto*)
apply(*simp add*: *bind-SE-def try-SE-def*)
apply(*case-tac mbind S F b*, *auto*)
done

end

theory *Symbex-MonadSE*
imports *Seq-MonadSE*
begin

3.1.6 Definition and Properties of Valid Execution Sequences

A key-notion in our framework is the *valid* execution sequence, i.e. a sequence that:

1. terminates (not obvious since while),
2. results in a final *True*,

3. does not fail globally (but recall the FailSave and FailPurge variants of $mbind_{FailSave}$-operators, that handle local exceptions in one or another way).

Seen from an automata perspective (where the monad - operations correspond to the step function), valid execution sequences can be used to model "feasible paths" across an automaton.

definition $\mathit{valid\text{-}SE} :: {}'\sigma \Rightarrow (\mathit{bool},{}'\sigma)\ MON_{SE} \Rightarrow \mathit{bool}$ (**infix** $\models$ *15*)
where $(\sigma \models m) = (m\ \sigma \neq \mathit{None} \wedge \mathit{fst}(\mathit{the}\ (m\ \sigma)))$

This notation consideres failures as valid – a definition inspired by I/O conformance.

Valid Execution Sequences and their Symbolic Execution

lemma *exec-unit-SE* [*simp*]: $(\sigma \models (\mathit{result}\ P)) = (P)$
by(*auto simp*: *valid-SE-def unit-SE-def*)

lemma *exec-unit-SE′* [*simp*]: $(\sigma_0 \models (\lambda\sigma.\ \mathit{Some}\ (f\ \sigma,\ \sigma))) = (f\ \sigma_0)$
by(*simp add*: *valid-SE-def*)

lemma *exec-fail-SE* [*simp*]: $(\sigma \models \mathit{fail}_{SE}) = \mathit{False}$
by(*auto simp*: *valid-SE-def fail-SE-def*)

lemma *exec-fail-SE′*[*simp*]: $\neg(\sigma_0 \models (\lambda\sigma.\ \mathit{None}))$
by(*simp add*: *valid-SE-def*)

The following the rules are in a sense the heart of the entire symbolic execution approach

lemma *exec-bind-SE-failure*:
$A\ \sigma = \mathit{None} \Longrightarrow \neg(\sigma \models ((s \leftarrow A\ ;\ M\ s)))$
by(*simp add*: *valid-SE-def unit-SE-def bind-SE-def*)

lemma *exec-bind-SE-failure2*:
$A\ \sigma = \mathit{None} \Longrightarrow \neg(\sigma \models ((A\ ;\!-\ M)))$
by(*simp add*: *valid-SE-def unit-SE-def bind-SE-def bind-SE′-def*)

lemma *exec-bind-SE-success*:
$A\ \sigma = \mathit{Some}(b,\sigma') \Longrightarrow (\sigma \models ((s \leftarrow A\ ;\ M\ s))) = (\sigma' \models (M\ b))$
by(*simp add*: *valid-SE-def unit-SE-def bind-SE-def*)

lemma *exec-bind-SE-success2*:
$A\ \sigma = \mathit{Some}(b,\sigma') \Longrightarrow (\sigma \models ((A\ ;\!-\ M))) = (\sigma' \models M)$
by(*simp add*: *valid-SE-def unit-SE-def bind-SE-def bind-SE′-def*)

lemma *exec-bind-SE-success′*:
$M\ \sigma = \mathit{Some}(f\ \sigma,\sigma) \Longrightarrow (\sigma \models M) = f\ \sigma$
by(*simp add*: *valid-SE-def unit-SE-def bind-SE-def*)

lemma *exec-bind-SE-success''*:
$\sigma \models ((s \leftarrow A \; ; \; M\ s)) \Longrightarrow \; \exists \; v\ \sigma'. \; the(A\ \sigma) = (v,\sigma') \wedge \sigma' \models (M\ v)$
apply(*auto simp*: *valid-SE-def unit-SE-def bind-SE-def*)
apply(*cases A* σ, *simp-all*)
apply(*drule-tac x=A* σ **and** *f=the* **in** *arg-cong*, *simp*)
apply(*rule-tac x=fst aa* **in** *exI*)
apply(*rule-tac x=snd aa* **in** *exI*, *auto*)
done

lemma *exec-bind-SE-success'''*:
$\sigma \models ((s \leftarrow A \; ; \; M\ s)) \Longrightarrow \; \exists \; a. \; (A\ \sigma) = Some\ a \wedge (snd\ a) \models (M\ (fst\ a))$
apply(*auto simp*: *valid-SE-def unit-SE-def bind-SE-def*)
apply(*cases A* σ, *simp-all*)
apply(*drule-tac x=A* σ **and** *f=the* **in** *arg-cong*, *simp*)
apply(*rule-tac x=fst aa* **in** *exI*)
apply(*rule-tac x=snd aa* **in** *exI*, *auto*)
done

lemma *exec-bind-SE-success''''* :
$\sigma \models ((s \leftarrow A \; ; \; M\ s)) \Longrightarrow \; \exists \; v\ \sigma'. \; A\ \sigma = Some(v,\sigma') \wedge \sigma' \models (M\ v)$
apply(*auto simp*: *valid-SE-def unit-SE-def bind-SE-def*)
apply(*cases A* σ, *simp-all*)
apply(*drule-tac x=A* σ **and** *f=the* **in** *arg-cong*, *simp*)
apply(*rule-tac x=fst aa* **in** *exI*)
apply(*rule-tac x=snd aa* **in** *exI*, *auto*)
done

lemma *valid-bind-cong* : $f\ \sigma = g\ \sigma \Longrightarrow (\sigma \models (x \leftarrow f \; ; \; M\ x)) = (\sigma \models (x \leftarrow g \; ; \; M\ x))$
  unfolding *bind-SE'-def bind-SE-def valid-SE-def*
    by *simp*

lemma *valid-bind'-cong* : $f\ \sigma = g\ \sigma \Longrightarrow (\sigma \models f \; ;\!- \; M) = (\sigma \models g \; ;\!- \; M)$
  unfolding *bind-SE'-def bind-SE-def valid-SE-def*
    by *simp*

Recall `mbind_unit` for the base case.

lemma *valid-mbind-mt* : $(\sigma \models (\; s \leftarrow \; mbind_{FailSave}\ []\ f; \; unit_{SE}\ (P\ s))) = P\ []$ **by** *simp*
lemma *valid-mbind-mtE*: $\sigma \models (\; s \leftarrow mbind_{FailSave}\ []\ f; \; unit_{SE}\ (P\ s)) \Longrightarrow (P\ [] \Longrightarrow Q) \Longrightarrow Q$
by(*auto simp*: *valid-mbind-mt*)

lemma *valid-mbind'-mt* : $(\sigma \models (\; s \leftarrow mbind_{FailStop}\ []\ f; \; unit_{SE}\ (P\ s))) = P\ []$ **by** *simp*
lemma *valid-mbind'-mtE*: $\sigma \models (\; s \leftarrow mbind_{FailStop}\ []\ f; \; unit_{SE}\ (P\ s)) \Longrightarrow (P\ [] \Longrightarrow Q) \Longrightarrow Q$

by(*auto simp*: *valid-mbind′-mt*)

lemma *valid-mbind′′-mt* : $(\sigma \models (\ s \leftarrow mbind_{FailPurge}\ []\ f;\ unit_{SE}\ (P\ s))) = P\ []$
by(*simp add*: *mbind′′.simps valid-SE-def bind-SE-def unit-SE-def*)
lemma *valid-mbind′′-mtE*: $\sigma \models (\ s \leftarrow mbind_{FailPurge}\ []\ f;\ unit_{SE}\ (P\ s)) \Longrightarrow (P\ [] \Longrightarrow Q) \Longrightarrow Q$
by(*auto simp*: *valid-mbind′′-mt*)

lemma *exec-mbindFSave-failure*:
$ioprog\ a\ \sigma = None \Longrightarrow$
$(\sigma \models (s \leftarrow mbind_{FailSave}\ (a\#S)\ ioprog\ ;\ M\ s)) = (\sigma \models (M\ []))$
by(*simp add*: *valid-SE-def unit-SE-def bind-SE-def*)

lemma *exec-mbindFStop-failure*:
$ioprog\ a\ \sigma = None \Longrightarrow$
$(\sigma \models (s \leftarrow mbind_{FailStop}\ (a\#S)\ ioprog\ ;\ M\ s)) = (False)$
by(*simp add*: *exec-bind-SE-failure*)

lemma *exec-mbindFPurge-failure*:
$ioprog\ a\ \sigma = None \Longrightarrow$
$(\sigma \models (s \leftarrow mbind_{FailPurge}\ (a\#S)\ ioprog\ ;\ M\ s)) = (\sigma \models (s \leftarrow mbind_{FailPurge}\ (S)\ ioprog\ ;\ M\ s))$
by(*simp add*: *valid-SE-def unit-SE-def bind-SE-def mbind′′.simps*)

lemma *exec-mbindFSave-success* :
$ioprog\ a\ \sigma = Some(b,\sigma') \Longrightarrow$
$(\sigma\ \models (s \leftarrow mbind_{FailSave}\ (a\#S)\ ioprog\ ;\ M\ s)) =$
$(\sigma' \models (s \leftarrow mbind_{FailSave}\ S\ ioprog\ ;\ M\ (b\#s)))$
unfolding *valid-SE-def unit-SE-def bind-SE-def*
by(*cases* $mbind_{FailSave}\ S\ ioprog\ \sigma'$, *auto*)

lemma *exec-mbindFStop-success* :
$ioprog\ a\ \sigma = Some(b,\sigma') \Longrightarrow$
$(\sigma\ \models (s \leftarrow mbind_{FailStop}\ (a\#S)\ ioprog\ ;\ M\ s)) =$
$(\sigma' \models (s \leftarrow mbind_{FailStop}\ S\ ioprog\ ;\ M\ (b\#s)))$
unfolding *valid-SE-def unit-SE-def bind-SE-def*
by(*cases* $mbind_{FailStop}\ S\ ioprog\ \sigma'$, *auto simp*: *mbind′.simps*)

lemma *exec-mbindFPurge-success* :
$ioprog\ a\ \sigma = Some(b,\sigma') \Longrightarrow$
$(\sigma\ \models (s \leftarrow mbind_{FailPurge}\ (a\#S)\ ioprog\ ;\ M\ s)) =$
$(\sigma' \models (s \leftarrow mbind_{FailPurge}\ S\ ioprog\ ;\ M\ (b\#s)))$
unfolding *valid-SE-def unit-SE-def bind-SE-def*
by(*cases* $mbind_{FailPurge}\ S\ ioprog\ \sigma'$, *auto simp*: *mbind′′.simps*)

lemma *exec-mbindFSave*:
$(\sigma \models (s \leftarrow mbind_{FailSave}\ \ (a\#S)\ ioprog\ ;\ return\ (P\ s))) =$

$(case\ ioprog\ a\ \sigma\ of$
$None \Rightarrow (\sigma \models (return\ (P\ [])))$
$\mid Some(b,\sigma') \Rightarrow (\sigma' \models (s \leftarrow mbind_{FailSave}\ S\ ioprog\ ;\ return\ (P\ (b\#s)))))$
apply(*case-tac ioprog a* σ)
apply(*auto simp*: *exec-mbindFSave-failure exec-mbindFSave-success split*: *prod.splits*)
done

lemma *mbind-eq-sexec*:
assumes $* : \bigwedge b\ \sigma'.\ f\ a\ \sigma = Some(b,\sigma') \Longrightarrow$
$(os \leftarrow mbind_{FailStop}\ S\ f;\ P\ (b\#os)) = (os \leftarrow mbind_{FailStop}\ S\ f;\ P'\ (b\#os))$
shows $(\ a \leftarrow f\ a;\ \ x \leftarrow mbind_{FailStop}\ S\ f;\ P\ (a\ \#\ x))\ \sigma =$
$(\ a \leftarrow f\ a;\ \ x \leftarrow mbind_{FailStop}\ S\ f;\ P'(a\ \#\ x))\ \sigma$
apply(*cases f a* σ = *None*)
apply(*subst bind-SE-def*, *simp*)
apply(*subst bind-SE-def*, *simp*)
apply *auto*
apply(*subst bind-SE-def*, *simp*)
apply(*subst bind-SE-def*, *simp*)
apply(*simp add*: $*$)
done

lemma *mbind-eq-sexec'*:
assumes $* : \bigwedge b\ \sigma'.\ f\ a\ \sigma = Some(b,\sigma') \Longrightarrow$
$(P\ (b))\sigma' = (P'\ (b))\sigma'$
shows $(\ a \leftarrow f\ a;\ \ P\ (a))\ \sigma =$
$(\ a \leftarrow f\ a;\ \ P'(a))\ \sigma$
apply(*cases f a* σ = *None*)
apply(*subst bind-SE-def*, *simp*)
apply(*subst bind-SE-def*, *simp*)
apply *auto*
apply(*subst bind-SE-def*, *simp*)
apply(*subst bind-SE-def*, *simp*)
apply(*simp add*: $*$)
done

lemma *mbind'-concat*:
$(os \leftarrow mbind_{FailStop}\ (S@T)\ f;\ P\ os) = (os \leftarrow mbind_{FailStop}\ S\ f;\ os' \leftarrow mbind_{FailStop}\ T\ f;\ P\ (os\ @\ os'))$
proof (*rule ext*, *rename-tac* σ, *induct S arbitrary*: σ *P*)
case *Nil* **show** *?case* **by** *simp*
next
case (*Cons a S*) **show** *?case*
apply(*insert Cons.hyps*, *simp*)
by(*rule mbind-eq-sexec'*,*simp*)
qed

lemma *assert-suffix-inv* :
$\sigma \models (\ \text{-} \leftarrow mbind_{FailStop}\ xs\ istep;\ assert_{SE}\ (P))$

$\Longrightarrow \forall\sigma.\ P\ \sigma \longrightarrow (\sigma \models (\text{-} \leftarrow istep\ x;\ assert_{SE}\ (P)))$
$\Longrightarrow \sigma \models (\ \text{-} \leftarrow mbind_{FailStop}\ (xs\ @\ [x])\ istep;\ assert_{SE}\ (P))$
apply(*subst mbind′-concat*, *simp*)
unfolding *bind-SE-def assert-SE-def valid-SE-def*
apply(*auto split*: *option.split option.split-asm*)
apply(*case-tac aa*,*simp-all*)
apply(*case-tac P bb*,*simp-all*)
apply (*metis option.distinct*(*1*))
apply(*case-tac aa*,*simp-all*)
apply(*case-tac P bb*,*simp-all*)
by (*metis option.distinct*(*1*))

Universal splitting and symbolic execution rule

lemma *exec-mbindFSave-E*:
assumes *seq* : $(\sigma \models (s \leftarrow mbind_{FailSave}\ (a\#S)\ ioprog\ ;\ \ (P\ s)))$
and *none*: $ioprog\ a\ \sigma = None \Longrightarrow (\sigma \models (P\ [])) \Longrightarrow Q$
and *some*: $\bigwedge\ b\ \sigma'.\ ioprog\ a\ \sigma = Some(b,\sigma') \Longrightarrow (\sigma' \models (s \leftarrow mbind_{FailSave}\ S\ ioprog;(P\ (b\#s)))) \Longrightarrow Q$
shows Q
using *seq*
proof(*cases ioprog a* σ)
case *None* **assume** *ass*:$ioprog\ a\ \sigma = None$ **show** Q
apply(*rule none*[*OF ass*])
apply(*insert ass*, *erule-tac ioprog1*=*ioprog* **in** *exec-mbindFSave-failure*[*THEN iffD1*],*rule seq*)
done
next
case (*Some aa*) **assume** *ass*:$ioprog\ a\ \sigma = Some\ aa$ **show** Q
apply(*insert ass*,*cases aa*,*simp*, *rename-tac out* σ')
apply(*erule some*)
apply(*insert ass*,*simp*)
apply(*erule-tac ioprog1*=*ioprog* **in** *exec-mbindFSave-success*[*THEN iffD1*],*rule seq*)
done
qed

The next rule reveals the particular interest in deduction; as an elimination rule, it allows for a linear conversion of a validity judgement $mbind_{FailStop}$ over an input list S into a constraint system; without any branching ... Symbolic execution can even be stopped tactically whenever $ioprog\ a\ \sigma = Some\ (b,\ \sigma')$ comes to a contradiction.

lemma *exec-mbindFStop-E*:
assumes *seq* : $(\sigma \models (s \leftarrow mbind_{FailStop}\ (a\#S)\ ioprog\ ;\ (P\ s)))$
and *some*: $\bigwedge b\ \sigma'.\ ioprog\ a\ \sigma = Some(b,\sigma') \Longrightarrow (\sigma' \models (s \leftarrow mbind_{FailStop}\ S\ ioprog;(P(b\#s)))) \Longrightarrow Q$
shows Q
using *seq*
proof(*cases ioprog a* σ)
case *None* **assume** *ass*:$ioprog\ a\ \sigma = None$ **show** Q
apply(*insert ass seq*)
apply(*drule-tac* σ=σ **and** S=S **and** M=P **in** *exec-mbindFStop-failure*, *simp*)

done
next
case (*Some aa*) **assume** *ass*:*ioprog a* σ = *Some aa* **show** *Q*
apply(*insert ass*,*cases aa*,*simp*, *rename-tac out* σ')
apply(*erule some*)
apply(*insert ass*,*simp*)
apply(*erule-tac ioprog1*=*ioprog* **in** *exec-mbindFStop-success*[*THEN iffD1*],*rule seq*)
done
qed

lemma *exec-mbindFPurge-E*:
assumes *seq* : $(\sigma \models (s \leftarrow mbind_{FailPurge}\ (a\#S)\ ioprog\ ;\ \ (P\ s)))$
and *none*: $ioprog\ a\ \sigma = None \Longrightarrow (\sigma \models (s \leftarrow mbind_{FailPurge}\ S\ ioprog;(P\ (s)))) \Longrightarrow Q$
and *some*: $\bigwedge\ b\ \sigma'.\ ioprog\ a\ \sigma = Some(b,\sigma') \Longrightarrow (\sigma' \models (s \leftarrow mbind_{FailPurge}\ S\ ioprog;(P\ (b\#s)))) \Longrightarrow Q$
shows *Q*
using *seq*
proof(*cases ioprog a* σ)
case *None* **assume** *ass*:*ioprog a* σ = *None* **show** *Q*
apply(*rule none*[*OF ass*])
apply(*insert ass*, *erule-tac ioprog1*=*ioprog* **in** *exec-mbindFPurge-failure*[*THEN iffD1*],*rule seq*)
done
next
case (*Some aa*) **assume** *ass*:*ioprog a* σ = *Some aa* **show** *Q*
apply(*insert ass*,*cases aa*,*simp*, *rename-tac out* σ')
apply(*erule some*)
apply(*insert ass*,*simp*)
apply(*erule-tac ioprog1*=*ioprog* **in** *exec-mbindFPurge-success*[*THEN iffD1*],*rule seq*)
done
qed

lemma *assert-disch1* : $P\ \sigma \Longrightarrow (\sigma \models (x \leftarrow assert_{SE}\ P;\ M\ x)) = (\sigma \models (M\ True))$
by(*auto simp*: *bind-SE-def assert-SE-def valid-SE-def*)

lemma *assert-disch2* : $\neg\ P\ \sigma \Longrightarrow \neg\ (\sigma \models (x \leftarrow assert_{SE}\ P\ ;\ M\ s))$
by(*auto simp*: *bind-SE-def assert-SE-def valid-SE-def*)

lemma *assert-disch3* : $\neg\ P\ \sigma \Longrightarrow \neg\ (\sigma \models (assert_{SE}\ P))$
by(*auto simp*: *bind-SE-def assert-SE-def valid-SE-def*)

lemma *assert-disch4* : $P\ \sigma \Longrightarrow\ (\sigma \models (assert_{SE}\ P))$
by(*auto simp*: *bind-SE-def assert-SE-def valid-SE-def*)

lemma *assert-simp* : $(\sigma \models assert_{SE}\ P) = P\ \sigma$
by (*meson assert-disch3 assert-disch4*)

lemmas *assert-D* = *assert-simp*[*THEN iffD1*]

lemma *assert-bind-simp* : $(\sigma \models (x \leftarrow assert_{SE}\ P;\ M\ x)) = (P\ \sigma \wedge (\sigma \models (M\ True)))$
by(*auto simp*: *bind-SE-def assert-SE-def valid-SE-def split*: *HOL.if-split-asm*)

lemmas *assert-bindD* = *assert-bind-simp*[*THEN iffD1*]

lemma *assume-D* : $(\sigma \models (\text{-} \leftarrow assume_{SE}\ P;\ M)) \Longrightarrow \exists\ \sigma.\ (P\ \sigma \wedge (\sigma \models M)\ )$
apply(*auto simp*: *bind-SE-def assume-SE-def valid-SE-def split*: *HOL.if-split-asm*)
apply(*rule-tac x=Eps P* **in** *exI*, *auto*)
apply(*subst Hilbert-Choice.someI*,*assumption*,*simp*)
done

lemma *assume-E* :
assumes $*\ : \sigma \models (\ \text{-} \leftarrow assume_{SE}\ P;\ M)$
and $** : \bigwedge \sigma.\ P\ \sigma \Longrightarrow \sigma \models M\ \Longrightarrow Q$
shows Q
apply(*insert* ∗)
by(*insert* ∗[*THEN assume-D*], *auto intro*: ∗∗)

lemma *assume-E′* :
assumes $*\ : \sigma \models assume_{SE}\ P\ ;-\ M$
and $** : \bigwedge \sigma.\ P\ \sigma \Longrightarrow \sigma \models M\ \Longrightarrow Q$
shows Q
by(*insert* ∗[*simplified bind-SE′-def*, *THEN assume-D*], *auto intro*: ∗∗)

These two rule prove that the SE Monad in connection with the notion of valid sequence is actually sufficient for a representation of a Boogie-like language. The SBE monad with explicit sets of states — to be shown below — is strictly speaking not necessary (and will therefore be discontinued in the development).

term $if_{SE}\ P\ then\ B_1\ else\ B_2\ fi$

lemma *if-SE-D1* : $P\ \sigma \Longrightarrow (\sigma \models (if_{SE}\ P\ then\ B_1\ else\ B_2\ fi)) = (\sigma \models B_1)$
by(*auto simp*: *if-SE-def valid-SE-def*)

lemma *if-SE-D1′* : $P\ \sigma \Longrightarrow (\sigma \models (if_{SE}\ P\ then\ B_1\ else\ B_2\ fi);-M) = (\sigma \models (B_1;-M))$
by(*auto simp*: *if-SE-def valid-SE-def bind-SE′-def bind-SE-def*)

lemma *if-SE-D2* : $\neg\ P\ \sigma \Longrightarrow (\sigma \models (if_{SE}\ P\ then\ B_1\ else\ B_2\ fi)) = (\sigma \models B_2)$
by(*auto simp*: *if-SE-def valid-SE-def*)

lemma *if-SE-D2′* : $\neg\ P\ \sigma \Longrightarrow (\sigma \models (if_{SE}\ P\ then\ B_1\ else\ B_2\ fi);-M) = (\sigma \models B_2;-M)$
by(*auto simp*: *if-SE-def valid-SE-def bind-SE′-def bind-SE-def*)

lemma *if-SE-split-asm* :

$(\sigma \models (\mathit{if}_{SE}\ P\ \mathit{then}\ B_1\ \mathit{else}\ B_2\ \mathit{fi})) = ((P\ \sigma \wedge (\sigma \models B_1)) \vee (\neg\ P\ \sigma \wedge (\sigma \models B_2)))$
by(*cases P σ*,*auto simp*: *if-SE-D1 if-SE-D2*)

lemma *if-SE-split-asm′*:
$(\sigma \models (\mathit{if}_{SE}\ P\ \mathit{then}\ B_1\ \mathit{else}\ B_2\ \mathit{fi});-M) = ((P\ \sigma \wedge (\sigma \models B_1;-M)) \vee (\neg\ P\ \sigma \wedge (\sigma \models B_2;-M)))$
by(*cases P σ*,*auto simp*: *if-SE-D1′ if-SE-D2′*)

lemma *if-SE-split*:
$(\sigma \models (\mathit{if}_{SE}\ P\ \mathit{then}\ B_1\ \mathit{else}\ B_2\ \mathit{fi})) = ((P\ \sigma \longrightarrow (\sigma \models B_1)) \wedge (\neg\ P\ \sigma \longrightarrow (\sigma \models B_2)))$
by(*cases P σ*, *auto simp*: *if-SE-D1 if-SE-D2*)

lemma *if-SE-split′*:
$(\sigma \models (\mathit{if}_{SE}\ P\ \mathit{then}\ B_1\ \mathit{else}\ B_2\ \mathit{fi});-M) = ((P\ \sigma \longrightarrow (\sigma \models B_1;-M)) \wedge (\neg\ P\ \sigma \longrightarrow (\sigma \models B_2;-M)))$
by(*cases P σ*, *auto simp*: *if-SE-D1′ if-SE-D2′*)

lemma *if-SE-execE*:
  assumes A: $\sigma \models ((\mathit{if}_{SE}\ P\ \mathit{then}\ B_1\ \mathit{else}\ B_2\ \mathit{fi}))$
   and B: $P\ \sigma \implies \sigma \models (B_1) \implies Q$
   and C: $\neg\ P\ \sigma \implies \sigma \models (B_2) \implies Q$
  shows Q
by(*insert A* [*simplified if-SE-split*],*cases P σ*, *simp-all*, *auto elim*: *B C*)

lemma *if-SE-execE′*:
  assumes A: $\sigma \models ((\mathit{if}_{SE}\ P\ \mathit{then}\ B_1\ \mathit{else}\ B_2\ \mathit{fi});-M)$
   and B: $P\ \sigma \implies \sigma \models (B_1;-M) \implies Q$
   and C: $\neg\ P\ \sigma \implies \sigma \models (B_2;-M) \implies Q$
  shows Q
by(*insert A* [*simplified if-SE-split′*],*cases P σ*, *simp-all*, *auto elim*: *B C*)

lemma *exec-while* :
$(\sigma \models ((\mathit{while}_{SE}\ b\ \mathit{do}\ c\ \mathit{od})\ ;-\ M)) =$
$(\sigma \models ((\mathit{if}_{SE}\ b\ \mathit{then}\ c\ ;-\ (\mathit{while}_{SE}\ b\ \mathit{do}\ c\ \mathit{od})\ \mathit{else}\ \mathit{unit}_{SE}\ ()\mathit{fi})\ ;-\ M))$
apply(*subst while-SE-unfold*)
by(*simp add*: *bind-SE′-def*)

lemmas *exec-whileD* = *exec-while*[*THEN iffD1*]

lemma *if-SE-execE′′*:
$\sigma \models (\mathit{if}_{SE}\ P\ \mathit{then}\ B_1\ \mathit{else}\ B_2\ \mathit{fi})\ ;-\ M$
$\implies (P\ \sigma \implies \sigma \models B_1\ ;-\ M \implies Q)$
$\implies (\neg\ P\ \sigma \implies \sigma \models B_2\ ;-\ M \implies Q)$
$\implies Q$
by(*auto elim*: *if-SE-execE′*)

definition *opaque* $(x::bool) = x$
lemma *if-SE-execE''-pos*:
$\sigma \models (\mathit{if}_{SE}\ P\ \mathit{then}\ B_1\ \mathit{else}\ B_2\ \mathit{fi})\ ;-\ M$
$\Longrightarrow (P\ \sigma \Longrightarrow \sigma \models B_1\ ;-\ M \Longrightarrow Q)$
$\Longrightarrow (\mathit{opaque}\ (\sigma \models (\mathit{if}_{SE}\ P\ \mathit{then}\ B_1\ \mathit{else}\ B_2\ \mathit{fi})\ ;-\ M) \Longrightarrow Q)$
$\Longrightarrow Q$
using *opaque-def* **by** *auto*

lemma [*code*]:
$(\sigma \models m) = (\mathit{case}\ (m\ \sigma)\ \mathit{of}\ \mathit{None}\ \Rightarrow \mathit{False}\ |\ (\mathit{Some}\ (x,y))\ \Rightarrow x)$
apply(*simp add*: *valid-SE-def*)
apply(*cases* $m\ \sigma = \mathit{None}$, *simp-all*)
apply(*insert not-None-eq*, *auto*)
done

lemma $P\ \sigma \models (\text{-} \leftarrow \mathit{assume}_{SE}\ P\ ;\ x \leftarrow M;\ \mathit{assert}_{SE}\ (\lambda\sigma.\ (x{=}X) \wedge Q\ x\ \sigma))$
oops

lemma $\forall\sigma.\ \exists\ X.\ \sigma \models (\text{-} \leftarrow \mathit{assume}_{SE}\ P\ ;\ x \leftarrow M;\ \mathit{assert}_{SE}\ (\lambda\sigma.\ x{=}X \wedge Q\ x\ \sigma))$
oops

lemma *monadic-sequence-rule*:
$\bigwedge X\ \sigma_1.\ (\sigma \models (\text{-} \leftarrow \mathit{assume}_{SE}\ (\lambda\sigma'.\ (\sigma{=}\sigma') \wedge\ P\ \sigma)\ ;\ x \leftarrow M;\ \mathit{assert}_{SE}\ (\lambda\sigma.\ (x{=}X) \wedge (\sigma{=}\sigma_1) \wedge Q\ x\ \sigma)))$
$\wedge$
$(\sigma_1 \models (\text{-} \leftarrow \mathit{assume}_{SE}\ (\lambda\sigma.\ (\sigma{=}\sigma_1) \wedge Q\ x\ \sigma)\ ;\ y \leftarrow M';\ \mathit{assert}_{SE}\ (\lambda\sigma.\ R\ x\ y\ \sigma)))$
$\Longrightarrow$
$\sigma \models (\text{-} \leftarrow \mathit{assume}_{SE}\ (\lambda\sigma'.\ (\sigma{=}\sigma') \wedge\ P\ \sigma)\ ;\ x \leftarrow M;\ y \leftarrow M';\ \mathit{assert}_{SE}\ (R\ x\ y))$
apply(*elim exE impE conjE*)
apply(*drule assume-D*)
apply(*elim exE impE conjE*)
unfolding *valid-SE-def assume-SE-def assert-SE-def bind-SE-def*
apply(*auto split*: *if-split HOL.if-split-asm Option.option.split Option.option.split-asm*)
apply (*metis* (*mono-tags*, *lifting*) *option.simps*(*3*) *someI-ex*)
oops

lemma $\exists\ X.\ \sigma \models (\text{-} \leftarrow \mathit{assume}_{SE}\ P\ ;\ x \leftarrow M;\ \mathit{assert}_{SE}\ (\lambda\sigma.\ x{=}X \wedge Q\ x\ \sigma))$
$\Longrightarrow$
$\sigma \models (\text{-} \leftarrow \mathit{assume}_{SE}\ P\ ;\ x \leftarrow M;\ \mathit{assert}_{SE}\ (\lambda\sigma.\ Q\ x\ \sigma))$
unfolding *valid-SE-def assume-SE-def assert-SE-def bind-SE-def*
by(*auto split*: *if-split HOL.if-split-asm Option.option.split Option.option.split-asm*)

lemma *exec-skip*:
$(\sigma \models skip_{SE} \;;\!- \; M) = (\sigma \models M)$
by (*simp add*: $skip_{SE}$*-def*)

lemmas *exec-skipD* = *exec-skip*[*THEN iffD1*]

Test-Refinements will be stated in terms of the failsave $mbind_{FailSave}$, opting more generality. The following lemma allows for an optimization both in test execution as well as in symbolic execution for an important special case of the post-codition: Whenever the latter has the constraint that the length of input and output sequence equal each other (that is to say: no failure occured), failsave mbind can be reduced to failstop mbind ...

lemma *mbindFSave-vs-mbindFStop* :
  $(\sigma \models (os \leftarrow (mbind_{FailSave}\ \iota s\ ioprog);\ result(length\ \iota s = length\ os \wedge P\ \iota s\ os))) =$
   $(\sigma \models (os \leftarrow (mbind_{FailStop}\ \iota s\ ioprog);\ result(P\ \iota s\ os)))$
  apply(*rule-tac* $x{=}P$ **in** *spec*)
  apply(*rule-tac* $x{=}\sigma$ **in** *spec*)
  proof(*induct* ιs)
    case *Nil* **show** *?case* **by**(*simp-all add*: *mbind-try try-SE-def del*: *Seq-MonadSE.mbind.simps*)
     case (*Cons a* ιs) **show** *?case*
          apply(*rule allI*, *rename-tac* σ,*rule allI*, *rename-tac P*)
          apply(*insert Cons.hyps*)
          apply(*case-tac ioprog a* σ)
          apply(*simp only*: *exec-mbindFSave-failure exec-mbindFStop-failure*, *simp*)
          apply(*simp add*: *split-paired-all del*: *Seq-MonadSE.mbind.simps*)
          apply(*rename-tac* σ')
          apply(*subst exec-mbindFSave-success*, *assumption*)
          apply(*subst* (*2*) *exec-bind-SE-success*, *assumption*)
          apply(*erule-tac* $x{=}\sigma'$ **in** *allE*)
          apply(*erule-tac* $x{=}\lambda\iota s\ s.\ P\ (a\ \#\ \iota s)\ (aa\ \#\ s)$ **in** *allE*)
          apply(*simp*)
      done
  qed

lemma $mbind_{FailSave}$*-vs-*$mbind_{FailStop}$:
assumes *A*: $\forall\ \iota\ \sigma.\ ioprog\ \iota\ \sigma \neq None$
shows $(\sigma \models (os \leftarrow (mbind_{FailSave}\ \iota s\ ioprog);\ P\ os)) =$
      $(\sigma \models (os \leftarrow (mbind_{FailStop}\ \iota s\ ioprog);\ P\ os))$
proof(*induct* ιs)
  case *Nil* **show** *?case* **by** *simp*
next
  case (*Cons a* ιs)
      from *Cons.hyps*
      have *B*:$\forall\ S\ f\ \sigma.\ mbind_{FailSave}\ S\ f\ \sigma \neq None$ **by** *simp*
      have *C*:$\forall \sigma.\ mbind_{FailStop}\ \iota s\ ioprog\ \sigma = mbind_{FailSave}\ \iota s\ ioprog\ \sigma$
            apply(*induct* ιs, *simp*)
            apply(*rule allI*,*rename-tac* σ)
            apply(*simp add*: *Seq-MonadSE.mbind*$'$*.simps*(*2*))

```
          apply(insert A, erule-tac x=a in allE)
          apply(erule-tac x=σ and P=λσ . ioprog a σ ≠ None in allE)
          apply(auto split:option.split)
          done
    show ?case
    apply(insert A,erule-tac x=a in allE,erule-tac x=σ in allE)
    apply(simp, elim exE)
    apply(rename-tac  out σ')
     apply(insert B, erule-tac x=ιs in allE, erule-tac x=ioprog in allE, erule-tac x=σ' in
allE)
    apply(subst(asm) not-None-eq, elim exE)
    apply(subst  exec-bind-SE-success)
    apply(simp   split: option.split, auto)
    apply(rule-tac s=(λ a b c. a # (fst c)) out σ' (aa, b) in trans, simp,rule refl)
    apply(rule-tac s=(λ a b c. (snd c)) out σ' (aa, b) in trans, simp,rule refl)
    apply(simp-all)
    apply(subst  exec-bind-SE-success, assumption)
    apply(subst  exec-bind-SE-success)
    apply(rule-tac s=Some (aa, b) in  trans,simp-all add:C)
    apply(subst(asm)  exec-bind-SE-success, assumption)
    apply(subst(asm)  exec-bind-SE-success)
    apply(rule-tac s=Some (aa, b) in  trans,simp-all add:C)
  done
qed
```

3.1.7 Miscellaneous

no-notation *unit-SE* ((*result* -) *8*)

end

theory *Clean-Symbex*
  imports *Clean*
begin

3.2 Clean Symbolic Execution Rules

3.2.1 Basic NOP - Symbolic Execution Rules.

As they are equalities, they can also be used as program optimization rules.

lemma *non-exec-assign* :
assumes ¬ *exec-stop* σ
shows (σ ⊨ (- ← *assign f*; *M*)) = ((*f* σ) ⊨ *M*)
by (*simp add*: *assign-def assms exec-bind-SE-success*)

lemma *non-exec-assign'* :
assumes ¬ *exec-stop* σ

shows $(\sigma \models (assign\ f;- M)) = ((f\ \sigma) \models M)$
by (*simp add*: *assign-def assms exec-bind-SE-success bind-SE′-def*)

lemma *exec-assign* :
assumes *exec-stop* σ
shows $(\sigma \models (\text{-} \leftarrow assign\ f;\ M)) = (\sigma \models M)$
by (*simp add*: *assign-def assms exec-bind-SE-success*)

lemma *exec-assign′* :
assumes *exec-stop* σ
shows $(\sigma \models (assign\ f;- M)) = (\sigma \models M)$
by (*simp add*: *assign-def assms exec-bind-SE-success bind-SE′-def*)

3.2.2 Assign Execution Rules.

lemma *non-exec-assign-global* :
assumes $\neg$ *exec-stop* σ
shows $(\sigma \models (\text{-} \leftarrow assign\text{-}global\ upd\ rhs;\ M)) = ((upd\ (\lambda\text{-}.\ rhs\ \sigma)\ \sigma) \models M)$
by(*simp add*: *assign-global-def non-exec-assign assms*)

lemma *non-exec-assign-global′* :
assumes $\neg$ *exec-stop* σ
shows $(\sigma \models (assign\text{-}global\ upd\ rhs;- M)) = ((upd\ (\lambda\text{-}.\ rhs\ \sigma)\ \sigma) \models M)$
by (*metis* (*full-types*) *assms bind-SE′-def non-exec-assign-global*)

lemma *exec-assign-global* :
assumes *exec-stop* σ
shows $(\sigma \models (\text{-} \leftarrow assign\text{-}global\ upd\ rhs;\ M)) = (\sigma \models M)$
by (*simp add*: *assign-global-def assign-def assms exec-bind-SE-success*)

lemma *exec-assign-global′* :
assumes *exec-stop* σ
shows $(\sigma \models (assign\text{-}global\ upd\ rhs;- M)) = (\sigma \models M)$
by (*simp add*: *assign-global-def assign-def assms exec-bind-SE-success bind-SE′-def*)

lemma *non-exec-assign-local* :
assumes $\neg$ *exec-stop* σ
shows $(\sigma \models (\text{-} \leftarrow assign\text{-}local\ upd\ rhs;\ M)) = ((upd\ (map\text{-}hd\ (\lambda\text{-}.\ rhs\ \sigma))\ \sigma) \models M)$
by(*simp add*: *assign-local-def non-exec-assign assms*)

lemma *non-exec-assign-local′* :
assumes $\neg$ *exec-stop* σ
shows $(\sigma \models (assign\text{-}local\ upd\ rhs;- M)) = ((upd\ (map\text{-}hd\ (\lambda\text{-}.\ rhs\ \sigma))\ \sigma) \models M)$
by (*metis assms bind-SE′-def non-exec-assign-local*)

lemmas *non-exec-assign-localD′*= *non-exec-assign*[*THEN iffD1*]

lemma *exec-assign-local* :

assumes *exec-stop* σ
shows $(\sigma \models (\ \text{-} \leftarrow \textit{assign-local upd rhs};\ M)) = (\ \sigma \models\ M)$
by (*simp add*: *assign-local-def assign-def assms exec-bind-SE-success*)

lemma *exec-assign-local'* :
assumes *exec-stop* σ
shows $(\sigma \models (\ \textit{assign-local upd rhs};-\ M)) = (\ \sigma \models\ M)$
unfolding *assign-local-def assign-def*
by (*simp add*: *assms exec-bind-SE-success2*)

lemmas *exec-assignD* = *exec-assign*[*THEN iffD1*]
thm *exec-assignD*

lemmas *exec-assignD'* = *exec-assign'*[*THEN iffD1*]
thm *exec-assignD'*

lemmas *exec-assign-globalD* = *exec-assign-global*[*THEN iffD1*]

lemmas *exec-assign-globalD'* = *exec-assign-global'*[*THEN iffD1*]

lemmas *exec-assign-localD* = *exec-assign-local*[*THEN iffD1*]
thm *exec-assign-localD*

lemmas *exec-assign-localD'* = *exec-assign-local'*[*THEN iffD1*]

3.2.3 Basic Call Symbolic Execution Rules.

lemma *exec-call-0* :
assumes *exec-stop* σ
shows $(\sigma \models (\ \text{-} \leftarrow \textit{call-0}_C\ M;\ M')) = (\sigma \models\ M')$
by (*simp add*: *assms* $\textit{call-0}_C$*-def exec-bind-SE-success*)

lemma *exec-call-0'* :
assumes *exec-stop* σ
shows $(\sigma \models (\textit{call-0}_C\ M;-\ M')) = (\sigma \models\ M')$
by (*simp add*: *assms bind-SE'-def exec-call-0*)

lemma *exec-call-1* :
assumes *exec-stop* σ
shows $(\sigma \models (\ x \leftarrow \textit{call-1}_C\ M\ A_1;\ M'\ x)) = (\sigma \models\ M'\ \textit{undefined})$
by (*simp add*: *assms* $\textit{call-1}_C$*-def* $\textit{call}_C$*-def exec-bind-SE-success*)

lemma *exec-call-1'* :
assumes *exec-stop* σ
shows $(\sigma \models (\textit{call-1}_C\ M\ A_1;-\ M')) = (\sigma \models\ M')$
by (*simp add*: *assms bind-SE'-def exec-call-1*)

lemma *exec-call* :
assumes *exec-stop* σ
shows $(\sigma \models (\ x \leftarrow call_C\ M\ A_1;\ M'\ x)) = (\sigma \models\ M'\ undefined)$
by (*simp add*: *assms* $call_C$*-def* *call-1*$_C$*-def* *exec-bind-SE-success*)

lemma *exec-call'* :
assumes *exec-stop* σ
shows $(\sigma \models (call_C\ M\ A_1;-\ M')) = (\sigma \models\ M')$
by (*metis assms call-1*$_C$*-def exec-call-1'*)

lemma *exec-call-2* :
assumes *exec-stop* σ
shows $(\sigma \models (\ \text{-} \leftarrow call\text{-}2_C\ M\ A_1\ A_2;\ M')) = (\sigma \models\ M')$
by (*simp add*: *assms call-2*$_C$*-def exec-bind-SE-success*)

lemma *exec-call-2'* :
assumes *exec-stop* σ
shows $(\sigma \models (call\text{-}2_C\ M\ A_1\ A_2;-\ M')) = (\sigma \models M')$
by (*simp add*: *assms bind-SE'-def exec-call-2*)

3.2.4 Basic Call Symbolic Execution Rules.

lemma *non-exec-call-0* :
assumes $\neg$ *exec-stop* σ
shows $(\sigma \models (\ \text{-} \leftarrow call\text{-}0_C\ M;\ M')) = (\sigma \models M;-\ M')$
by (*simp add*: *assms bind-SE'-def bind-SE-def call-0*$_C$*-def valid-SE-def*)

lemma *non-exec-call-0'* :
assumes $\neg$ *exec-stop* σ
shows $(\sigma \models call\text{-}0_C\ M;-\ M') = (\sigma \models M;-\ M')$
by (*simp add*: *assms bind-SE'-def non-exec-call-0*)

lemma *non-exec-call-1* :
assumes $\neg$ *exec-stop* σ
shows $(\sigma \models (\ x \leftarrow (call\text{-}1_C\ M\ A_1);\ M'\ x)) = (\sigma \models (x \leftarrow M\ (A_1\ \sigma);\ M'\ x))$
by (*simp add*: *assms bind-SE'-def* $call_C$*-def bind-SE-def call-1*$_C$*-def valid-SE-def*)

lemma *non-exec-call-1'* :
assumes $\neg$ *exec-stop* σ
shows $(\sigma \models call\text{-}1_C\ M\ A_1;-\ M') = (\sigma \models\ M\ (A_1\ \sigma);-\ M')$
by (*simp add*: *assms bind-SE'-def non-exec-call-1*)

lemma *non-exec-call* :
assumes $\neg$ *exec-stop* σ
shows $(\sigma \models (\ x \leftarrow (call_C\ M\ A_1);\ M'\ x)) = (\sigma \models (x \leftarrow M\ (A_1\ \sigma);\ M'\ x))$
by (*simp add*: *assms* $call_C$*-def bind-SE'-def bind-SE-def call-1*$_C$*-def valid-SE-def*)

lemma *non-exec-call'* :

assumes $\neg$ *exec-stop* σ
shows $(\sigma \models call_C\ M\ A_1;-\ M') = (\sigma \models\ M\ (A_1\ \sigma);-\ M')$
by (*simp add*: *assms bind-SE'-def non-exec-call*)

lemma *non-exec-call-2* :
assumes $\neg$ *exec-stop* σ
shows $(\sigma \models (\ \text{-} \leftarrow (call\text{-}2_C\ M\ A_1\ A_2);\ M')) = (\sigma \models M\ (A_1\ \sigma)\ (A_2\ \sigma);-\ M')$
by (*simp add*: *assms bind-SE'-def bind-SE-def call-2*$_C$*-def valid-SE-def*)

lemma *non-exec-call-2'* :
assumes $\neg$ *exec-stop* σ
shows $(\sigma \models call\text{-}2_C\ M\ A_1\ A_2;-\ M') = (\sigma \models\ M\ (A_1\ \sigma)\ (A_2\ \sigma);-\ M')$
by (*simp add*: *assms bind-SE'-def non-exec-call-2*)

3.2.5 Conditional.

lemma *exec-If*$_C$*-If*$_{SE}$:
assumes $\neg$ *exec-stop* σ
shows $((if_C\ P\ then\ B_1\ else\ B_2\ fi))\sigma = ((if_{SE}\ P\ then\ B_1\ else\ B_2\ fi))\ \sigma$
unfolding *if-SE-def MonadSE.if-SE-def Symbex-MonadSE.valid-SE-def MonadSE.bind-SE'-def*
by (*simp add*: *assms bind-SE-def if-C-def*)

lemma *valid-exec-If*$_C$:
assumes $\neg$ *exec-stop* σ
shows $(\sigma \models (if_C\ P\ then\ B_1\ else\ B_2\ fi);-M) = (\sigma \models (if_{SE}\ P\ then\ B_1\ else\ B_2\ fi);-M)$
by (*meson assms exec-If*$_C$*-If*$_{SE}$ *valid-bind'-cong*)

lemma *exec-If*$_C$*'* :
assumes *exec-stop* σ
shows $(\sigma \models (if_C\ P\ then\ B_1\ else\ B_2\ fi);-M) = (\sigma \models M)$
unfolding *if-SE-def MonadSE.if-SE-def Symbex-MonadSE.valid-SE-def MonadSE.bind-SE'-def bind-SE-def*
by (*simp add*: *assms if-C-def*)

lemma *exec-While*$_C$*'* :
assumes *exec-stop* σ
shows $(\sigma \models (while_C\ P\ do\ B_1\ od);-M) = (\sigma \models M)$
unfolding *while-C-def MonadSE.if-SE-def Symbex-MonadSE.valid-SE-def MonadSE.bind-SE'-def bind-SE-def*
apply *simp* **using** *assms* **by** *blast*

lemma *if*$_C$*-cond-cong* : $f\ \sigma = g\ \sigma \Longrightarrow$

$(\mathit{if}_C\ f\ \mathit{then}\ c\ \mathit{else}\ d\ \mathit{fi})\ \sigma =$
$(\mathit{if}_C\ g\ \mathit{then}\ c\ \mathit{else}\ d\ \mathit{fi})\ \sigma$
unfolding *if-C-def*
by *simp*

3.2.6 Break - Rules.

lemma *break-assign-skip* [*simp*]: *break* ;− *assign f* = *break*
apply(*rule ext*)
unfolding *break-def assign-def exec-stop-def bind-SE′-def bind-SE-def*
by *auto*

lemma *break-if-skip* [*simp*]: *break* ;− (if_C *b then c else d fi*) = *break*
apply(*rule ext*)
unfolding *break-def assign-def exec-stop-def if-C-def bind-SE′-def bind-SE-def*
by *auto*

lemma *break-while-skip* [*simp*]: *break* ;− (while_C *b do c od*) = *break*
apply(*rule ext*)
unfolding *while-C-def* skip_{SE}*-def unit-SE-def bind-SE′-def bind-SE-def break-def exec-stop-def*
by *simp*

lemma *unset-break-idem* [*simp*] :
(*unset-break-status* ;− *unset-break-status* ;− *M*) = (*unset-break-status* ;− *M*)
apply(*rule ext*) **unfolding** *unset-break-status-def bind-SE′-def bind-SE-def* **by** *auto*

lemma *return-cancel1-idem* [*simp*] :
(return_C *X E* ;− *assign-global X E′* ;− *M*) = (return_C *X E* ;− *M*)
apply(*rule ext*, *rename-tac* σ)
unfolding *unset-break-status-def bind-SE′-def bind-SE-def*
assign-def return_C*-def assign-global-def assign-local-def*
apply(*case-tac exec-stop* σ)
apply *auto*
by (*simp add*: *exec-stop-def set-return-status-def*)

lemma *return-cancel2-idem* [*simp*] :
(return_C *X E* ;− *assign-local X E′* ;− *M*) = (return_C *X E* ;− *M*)
apply(*rule ext*, *rename-tac* σ)
unfolding *unset-break-status-def bind-SE′-def bind-SE-def*
assign-def return_C*-def assign-global-def assign-local-def*
apply(*case-tac exec-stop* σ)
apply *auto*
by (*simp add*: *exec-stop-def set-return-status-def*)

3.2.7 While.

lemma $while_C$*-skip* [*simp*]: ($while_C$ (λ *x*. *False*) *do c od*) = $skip_{SE}$
apply(*rule ext*)
unfolding *while-C-def* $skip_{SE}$*-def unit-SE-def*
apply *auto*
unfolding *exec-stop-def* $skip_{SE}$*-def unset-break-status-def bind-SE′-def unit-SE-def bind-SE-def*
by *simp*

Various tactics for various coverage criteria

definition *while-k* :: *nat* ⇒ ((′σ*-ext*) *control-state-ext* ⇒ *bool*)
⇒ (*unit*, (′σ*-ext*) *control-state-ext*)MON_{SE}
⇒ (*unit*, (′σ*-ext*) *control-state-ext*)MON_{SE}
where *while-k* - ≡ *while-C*

Somewhat amazingly, this unfolding lemma crucial for symbolic execution still holds ... Even in the presence of break or return...

lemma *exec*-$while_C$:
(σ ⊨ (($while_C$ *b do c od*) ;− *M*)) =
(σ ⊨ ((if_C *b then c* ;− (($while_C$ *b do c od*) ;− *unset-break-status*) *else* $skip_{SE}$ *fi*) ;− *M*))
proof (*cases exec-stop* σ)
case *True*
then show *?thesis*
by (*simp add*: *True exec*-If_C′ *exec*-$While_C$′)
next
case *False*
then show *?thesis*
proof (*cases* ¬ *b* σ)
case *True*
then show *?thesis*
apply(*subst valid-bind′-cong*)
using ‹¬ *exec-stop* σ› **apply** *simp-all*
apply (*auto simp*: $skip_{SE}$*-def unit-SE-def*)
apply(*subst while-C-def*, *simp*)
apply(*subst bind′-cong*)
apply(*subst MonadSE.while-SE-unfold*)
apply(*subst* if_{SE}*-cond-cong* [*of* - - λ-. *False*])
apply *simp-all*
apply(*subst* if_C*-cond-cong* [*of* - - λ-. *False*], *simp add*:)
apply(*subst exec*-If_C-If_{SE},*simp-all*)
by (*simp add*: *exec-stop-def unset-break-status-def*)
next
case *False*
have ∗ : *b* σ **using** *False* **by** *auto*
then show *?thesis*
unfolding *while-k-def*
apply(*subst while-C-def*)
apply(*subst if-C-def*)
apply(*subst valid-bind′-cong*)
apply (*simp add*: ‹¬ *exec-stop* σ›)

apply(*subst* (2) *valid-bind′-cong*)
apply (*simp add*: ‹¬ *exec-stop* σ›)
apply(*subst MonadSE.while-SE-unfold*)
apply(*subst valid-bind′-cong*)
apply(*subst bind′-cong*)
apply(*subst* if_{SE}*-cond-cong* [*of* - - λ-. *True*])
apply(*simp-all add*: ‹¬ *exec-stop* σ›)
apply(*subst bind-assoc′*, *subst bind-assoc′*)
proof(*cases c* σ)
case *None*
then show $(\sigma \models c;-((while_{SE}\ (\lambda\sigma.\ \neg\ exec\text{-}stop\ \sigma \wedge b\ \sigma)\ do\ c\ od);-unset\text{-}break\text{-}status);-M)$
=
$(\sigma \models c;-(while_C\ \ b\ \ do\ \ c\ \ od)\ ;-\ unset\text{-}break\text{-}status\ ;-\ M)$
by (*simp add*: *bind-SE′-def exec-bind-SE-failure*)
next
case (*Some a*)
then show $(\sigma \models c\ ;-\ ((while_{SE}\ (\lambda\sigma.\ \neg\ exec\text{-}stop\ \sigma \wedge b\ \sigma)\ do\ c\ od);-unset\text{-}break\text{-}status);-M)$
=
$(\sigma \models c\ ;-\ (while_C\ \ b\ \ do\ \ c\ \ od)\ ;-\ unset\text{-}break\text{-}status\ ;-\ M)$
apply(*insert* ‹*c* σ = *Some a*›, *subst* (*asm*) *surjective-pairing*[*of a*])
apply(*subst exec-bind-SE-success2*, *assumption*)
apply(*subst exec-bind-SE-success2*, *assumption*)
proof(*cases exec-stop* (*snd a*))
case *True*
then show $(snd\ a \models ((while_{SE}\ (\lambda\sigma.\ \neg\ exec\text{-}stop\ \sigma \wedge b\ \sigma)\ do\ c\ od);-unset\text{-}break\text{-}status);-M) =$
$(snd\ a \models (while_C\ \ b\ \ do\ \ c\ \ od)\ ;-\ unset\text{-}break\text{-}status\ ;-\ M)$
by (*metis* (*no-types*, *lifting*) *bind-assoc′ exec-*$While_C$*′ exec-skip if-SE-D2′*
$skip_{SE}$*-def while-SE-unfold*)
next
case *False*
then show $(snd\ a \models ((while_{SE}(\lambda\sigma.\ \neg exec\text{-}stop\ \sigma \wedge b\ \sigma)\ do\ c\ od);-unset\text{-}break\text{-}status);-M) =$
$(snd\ a \models (while_C\ \ b\ \ do\ \ c\ \ od)\ ;-\ unset\text{-}break\text{-}status\ ;-\ M)$
unfolding *while-C-def*
by(*subst* (2) *valid-bind′-cong*,*simp*)(*simp*)
qed
qed
qed
qed

lemma *while-k-SE* : *while-C* = *while-k k*
by (*simp only*: *while-k-def*)

corollary *exec-while-k* :
$(\sigma \models ((while\text{-}k\ (Suc\ n)\ b\ c)\ ;-\ M)) =$
$(\sigma \models ((if_C\ b\ then\ c\ ;-\ (while\text{-}k\ n\ b\ c)\ ;-\ unset\text{-}break\text{-}status\ else\ skip_{SE}\ fi)\ \ ;-\ M))$
by (*metis exec-*$while_C$ *while-k-def*)

Necessary prerequisite: turning ematch and dmatch into a proper Isar Method.

ML‹
```
local
fun method-setup b tac =
  Method.setup b
    (Attrib.thms >> (fn rules => fn ctxt => METHOD (HEADGOAL o K (tac ctxt rules))))
in
val - =
  Theory.setup (    method-setup @{binding ematch} ematch-tac fast elimination matching
                #> method-setup @{binding dmatch} dmatch-tac fast destruction matching
                #> method-setup @{binding match} match-tac resolution based on fast matching)
end
```
›

lemmas *exec-while-kD* = *exec-while-k*[*THEN iffD1*]

end

theory *Test-Clean*
  imports *Clean-Symbex*
          ~~/*src*/*HOL*/*Eisbach*/*Eisbach*

begin

named-theorems *memory-theory*

method *memory-theory* = (*simp only*: *memory-theory MonadSE.bind-assoc′*)
method *norm* = (*auto dest*!: *assert-D*)

end

theory *Hoare-MonadSE*
  imports *Symbex-MonadSE*
begin

3.3 Hoare

definition $hoare_3$ :: $('\sigma \Rightarrow bool) \Rightarrow ('\alpha, '\sigma)MON_{SE} \Rightarrow ('\alpha \Rightarrow '\sigma \Rightarrow bool) \Rightarrow bool$ ((⦃(1-)⦄/ (-)/ ⦃(1-)⦄) 50)
where $⦃P⦄\ M\ ⦃Q⦄ \equiv (\forall \sigma.\ P\ \sigma \longrightarrow (case\ M\ \sigma\ of\ None => False \mid Some(x, \sigma') => Q\ x\ \sigma'))$

definition $hoare_3'$ :: $('\sigma \Rightarrow bool) \Rightarrow ('\alpha, '\sigma)MON_{SE} \Rightarrow bool$ ((⦃(1-)⦄/ (-)/†) 50)

where $\{\!|P|\!\}\ M\ \dagger \equiv (\forall\,\sigma.\ P\ \sigma \longrightarrow (\mathit{case}\ M\ \sigma\ \mathit{of}\ \mathit{None} \Rightarrow \mathit{True}\ |\ \text{-} \Rightarrow \mathit{False}))$

3.3.1 Basic rules

lemma *skip*: $\{\!|P|\!\}\ \mathit{skip}_{SE}\ \{\!|\lambda\text{-}.\ P|\!\}$
  unfolding *hoare$_3$-def skip$_{SE}$-def unit-SE-def*
  by *auto*

lemma *fail*: $\{\!|P|\!\}\ \mathit{fail}_{SE}\ \dagger$
  unfolding *hoare$_3$'-def fail-SE-def unit-SE-def* **by** *auto*

lemma *assert*: $\{\!|P|\!\}\ \mathit{assert}_{SE}\ P\ \{\!|\lambda\ \text{-}\ \text{-}.\ \mathit{True}|\!\}$
  unfolding *hoare$_3$-def assert-SE-def unit-SE-def*
  by *auto*

lemma *assert-conseq*: $\mathit{Collect}\ P \subseteq \mathit{Collect}\ Q \Longrightarrow \{\!|P|\!\}\ \mathit{assert}_{SE}\ Q\ \{\!|\lambda\ \text{-}\ \text{-}.\ \mathit{True}|\!\}$
  unfolding *hoare$_3$-def assert-SE-def unit-SE-def*
  by *auto*

lemma *assume-conseq*:
  assumes $\exists\ \sigma.\ Q\ \sigma$
  shows $\{\!|P|\!\}\ \mathit{assume}_{SE}\ Q\ \{\!|\lambda\ \text{-}\ .\ Q|\!\}$
  unfolding *hoare$_3$-def assume-SE-def unit-SE-def*
  apply (*auto simp* : *someI2*)
  using *assms* **by** *auto*

assignment missing in the calculus because this is viewed as a state specific operation, definable for concrete instances of $'\sigma$.

3.3.2 Generalized and special sequence rules

The decisive idea is to factor out the post-condition on the results of M :

lemma *sequence* :
  $\{\!|P|\!\}\ M\ \{\!|\lambda x\ \sigma.\ x{\in}A \wedge Q\ x\ \sigma|\!\}$
  $\Longrightarrow \forall\,x{\in}A.\ \{\!|Q\ x|\!\}\ M'\ x\ \{\!|R|\!\}$
  $\Longrightarrow \{\!|P|\!\}\ x \leftarrow M;\ M'\ x\ \{\!|R|\!\}$
  unfolding *hoare$_3$-def bind-SE-def*
  by(*auto*,*erule-tac* $x{=}\sigma$ **in** *allE*, *auto split*: *Option.option.split-asm Option.option.split*)

lemma *sequence-irpt-l* : $\{\!|P|\!\}\ M\ \dagger\ \Longrightarrow \{\!|P|\!\}\ x \leftarrow M;\ M'\ x\ \dagger$
  unfolding *hoare$_3$'-def bind-SE-def*
  by(*auto*,*erule-tac* $x{=}\sigma$ **in** *allE*, *auto split*: *Option.option.split-asm Option.option.split*)

lemma *sequence-irpt-r* : $\{\!|P|\!\}\ M\ \{\!|\lambda x\ \sigma.\ x{\in}A \wedge Q\ x\ \sigma|\!\} \Longrightarrow \forall\,x{\in}A.\ \{\!|Q\ x|\!\}\ M'\ x\ \dagger\ \Longrightarrow \{\!|P|\!\}\ x \leftarrow M;\ M'\ x\ \dagger$
  unfolding *hoare$_3$'-def hoare$_3$-def bind-SE-def*
  by(*auto*,*erule-tac* $x{=}\sigma$ **in** *allE*, *auto split*: *Option.option.split-asm Option.option.split*)

lemma *sequence'* : $\{\!|P|\!\}\ M\ \{\!|\lambda\text{-}.\ Q\ |\!\} \Longrightarrow \{\!|Q|\!\}\ M'\ \{\!|R|\!\} \Longrightarrow \{\!|P|\!\}\ M;\!-\ M'\ \{\!|R|\!\}$

unfolding *hoare*$_3$*-def hoare*$_3$*-def bind-SE-def bind-SE'-def*
by(*auto*,*erule-tac* $x=\sigma$ **in** *allE*, *auto split*: *Option.option.split-asm Option.option.split*)

lemma *sequence-irpt-l'* : $\{\!|P|\!\}$ M † $\Longrightarrow$ $\{\!|P|\!\}$ $M;-$ M' †
unfolding *hoare*$_3$*'-def bind-SE-def bind-SE'-def*
by(*auto*,*erule-tac* $x=\sigma$ **in** *allE*, *auto split*: *Option.option.split-asm Option.option.split*)

lemma *sequence-irpt-r'* : $\{\!|P|\!\}$ M $\{\!|\lambda\text{-}.\ Q\ |\!\}$ $\Longrightarrow$ $\{\!|Q|\!\}$ M' † $\Longrightarrow$ $\{\!|P|\!\}$ $M;-$ M' †
unfolding *hoare*$_3$*'-def hoare*$_3$*-def bind-SE-def bind-SE'-def*
by(*auto*,*erule-tac* $x=\sigma$ **in** *allE*, *auto split*: *Option.option.split-asm Option.option.split*)

3.3.3 Generalized and special consequence rules

lemma *consequence* :
$Collect\ P \subseteq Collect\ P'$
$\Longrightarrow$ $\{\!|P'|\!\}$ M $\{\!|\lambda x\ \sigma.\ x{\in}A \wedge Q'\ x\ \sigma|\!\}$
$\Longrightarrow$ $\forall\ x{\in}A.\ Collect(Q'\ x) \subseteq Collect\ (Q\ x)$
$\Longrightarrow$ $\{\!|P|\!\}$ M $\{\!|\lambda x\ \sigma.\ x{\in}A \wedge Q\ x\ \sigma|\!\}$
unfolding *hoare*$_3$*-def bind-SE-def*
by(*auto*,*erule-tac* $x=\sigma$ **in** *allE*,*auto split*: *Option.option.split-asm Option.option.split*)

lemma *consequence-unit* :
assumes $(\bigwedge\ \sigma.\ P\ \sigma \longrightarrow P'\ \sigma)$
and $\{\!|P'|\!\}$ M $\{\!|\lambda x{::}unit.\ \lambda\ \sigma.\ \ Q'\ \sigma|\!\}$
and $(\bigwedge\ \sigma.\ Q'\ \ \sigma \longrightarrow Q\ \ \sigma)$
shows $\{\!|P|\!\}$ M $\{\!|\lambda x\ \sigma.\ Q\ \sigma|\!\}$
proof −
have $*$: $(\lambda x\ \sigma.\ Q\ \ \sigma) = (\lambda x{::}unit.\ \lambda\ \sigma.\ x{\in}UNIV \wedge Q\ \ \sigma)$ **by** *auto*
show *?thesis*
apply(*subst* $*$)
apply(*rule-tac* $P' = P'$ **and** $Q' = \%\text{-}.\ Q'$ **in** *consequence*)
apply (*simp add*: *Collect-mono assms*(*1*))
using *assms*(*2*) **apply** *auto*[*1*]
by (*simp add*: *Collect-mono assms*(*3*))
qed

lemma *consequence-irpt* :
$Collect\ P \subseteq Collect\ P'$
$\Longrightarrow$ $\{\!|P'|\!\}$ M †
$\Longrightarrow$ $\{\!|P|\!\}$ M †
unfolding *hoare*$_3$*-def hoare*$_3$*'-def bind-SE-def*
by(*auto*)

lemma *consequence-mt-swap* :
$(\{\!|\lambda\text{-}.\ False|\!\}\ M\ †) = (\{\!|\lambda\text{-}.\ False|\!\}\ M\ \{\!|P|\!\})$
unfolding *hoare*$_3$*-def hoare*$_3$*'-def bind-SE-def*
by *auto*

3.3.4 Condition rules

lemma *cond* :
$\{\!|\lambda\sigma.\ P\ \sigma \wedge cond\ \sigma|\!\}\ M\ \{\!|Q|\!\}$
$\Longrightarrow \{\!|\lambda\sigma.\ P\ \sigma \wedge \neg\ cond\ \sigma|\!\}\ M'\ \{\!|Q|\!\}$
$\Longrightarrow \{\!|P|\!\} if_{SE}\ cond\ then\ M\ else\ M'\ fi \{\!|Q|\!\}$
unfolding *hoare*$_3$*-def hoare*$_3$*'-def bind-SE-def if-SE-def*
by *auto*

lemma *cond-irpt* :
$\{\!|\lambda\sigma.\ P\ \sigma \wedge cond\ \sigma|\!\}\ M\ \dagger$
$\Longrightarrow \{\!|\lambda\sigma.\ P\ \sigma \wedge \neg\ cond\ \sigma|\!\}\ M'\ \dagger$
$\Longrightarrow \{\!|P|\!\} if_{SE}\ cond\ then\ M\ else\ M'\ fi\ \dagger$
unfolding *hoare*$_3$*-def hoare*$_3$*'-def bind-SE-def if-SE-def*
by *auto*

Note that the other four combinations can be directly derived via the ($\{\!|\lambda$-. *False*$|\!\}$ *?M*$\dagger$) = ($\{\!|\lambda$-. *False*$|\!\}$ *?M* $\{\!|$*?P*$|\!\}$) rule.

3.3.5 While rules

The only non-trivial proof is, of course, the while loop rule. Note that non-terminating loops were mapped to *None* following the principle that our monadic state-transformers represent partial functions in the mathematical sense.

lemma *while* :
assumes $*$: $\{\!|\lambda\sigma.\ cond\ \sigma \wedge P\ \sigma|\!\}\ \ M\ \{\!|\lambda\text{-}.\ P|\!\}$
and *measure*: $\forall\sigma.\ cond\ \sigma \wedge P\ \sigma \longrightarrow M\ \sigma \neq None \wedge f(snd(the(M\ \sigma))) < ((f\ \sigma)::nat)$
shows $\{\!|P|\!\} while_{SE}\ cond\ do\ M\ od\ \{\!|\lambda\text{-}\ \sigma.\ \neg cond\ \sigma \wedge P\ \sigma|\!\}$

unfolding *hoare*$_3$*-def hoare*$_3$*'-def bind-SE-def if-SE-def*
proof *auto*
have $*$: $\forall n.\ \forall\ \sigma.\ P\ \sigma \wedge f\ \sigma \leq n \longrightarrow$
$(case\ (while_{SE}\ cond\ do\ M\ od)\ \sigma\ of$
$None \Rightarrow False$
$|\ Some\ (x, \sigma') \Rightarrow \neg\ cond\ \sigma' \wedge P\ \sigma')$ (**is** $\forall n.$ *?P n*)
proof (*rule allI*, *rename-tac n*, *induct-tac n*)
fix *n* **show** *?P 0*
apply(*auto*)
apply(*subst while-SE-unfold*)
by (*metis* (*no-types*, *lifting*) *gr-implies-not0 if-SE-def measure option.case-eq-if option.sel option.simps*(*3*) *prod.sel*(*2*) *split-def unit-SE-def*)
next
fix *n* **show** *?P n* $\Longrightarrow$ *?P* (*Suc n*)
apply(*auto*,*subst while-SE-unfold*)
apply(*case-tac* $\neg$*cond* σ)
apply (*simp add*: *if-SE-def unit-SE-def*)
apply(*simp add*: *if-SE-def*)
apply(*case-tac* $M\ \sigma = None$)
using *measure* **apply** *blast*

proof (*auto simp*: *bind-SE′-def bind-SE-def*)
fix σ σ'
assume *1* : *cond* σ
and *2* : $M\ \sigma = Some\ ((), \sigma')$
and *3* : $P\ \sigma$
and *4* : $f\ \sigma \leq Suc\ n$
and *hyp* : *?P n*
have *5* : $P\ \sigma'$
by (*metis* (*no-types*, *lifting*) ∗ *1 2 3 case-prodD* $hoare_3$*-def option.simps*(*5*))
have *6* : $snd(the(M\ \sigma)) = \sigma'$
by (*simp add*: *2*)
have *7* : $cond\ \sigma' \Longrightarrow f\ \sigma' \leq n$
using *1 3 4 6 leD measure* **by** *auto*
show $case\ (while_{SE}\ cond\ do\ M\ od)\ \sigma'\ of\ None \Rightarrow False$
$\mid Some\ (xa, \sigma') \Rightarrow \neg\ cond\ \sigma' \wedge P\ \sigma'$
using *1 3 4 5 6 hyp measure* **by** *auto*
qed
qed
show $\bigwedge\sigma.\ P\ \sigma \Longrightarrow$
$case\ (while_{SE}\ cond\ do\ M\ od)\ \sigma\ of\ None \Rightarrow False$
$\mid Some\ (x, \sigma') \Rightarrow \neg\ cond\ \sigma' \wedge P\ \sigma'$
using ∗ **by** *blast*
qed

lemma *while-irpt* :
assumes ∗ : $\{\!|\lambda\sigma.\ cond\ \sigma \wedge P\ \sigma|\!\}\ M\ \{\!|\lambda\text{-}.\ P|\!\} \vee \{\!|\lambda\sigma.\ cond\ \sigma \wedge P\ \sigma|\!\}\ M\ \dagger$
and *measure*: $\forall\sigma.\ cond\ \sigma \wedge P\ \sigma \longrightarrow M\ \sigma = None \vee f(snd(the(M\ \sigma))) < ((f\ \sigma)::nat)$
and *enabled*: $\forall\sigma.\ P\ \sigma \longrightarrow cond\ \sigma$
shows $\{\!|P|\!\}while_{SE}\ cond\ do\ M\ od\ \dagger$
unfolding $hoare_3$*-def* $hoare_3$*′-def bind-SE-def if-SE-def*
proof *auto*
have ∗ : $\forall n.\ \forall\ \sigma.\ P\ \sigma \wedge f\ \sigma \leq n \longrightarrow$
$(case\ (while_{SE}\ cond\ do\ M\ od)\ \sigma\ of\ None \Rightarrow True \mid Some\ a \Rightarrow False)$
(**is** $\forall n.$ *?P n*)
proof (*rule allI*, *rename-tac n*, *induct-tac n*)
fix *n*
have *1* : $\bigwedge\sigma.\ P\ \sigma \Longrightarrow cond\ \sigma$
by (*simp add*: *enabled* ∗)
show *?P 0*
apply(*auto*,*frule 1*)
by (*metis assms*(*2*) *bind-SE′-def bind-SE-def gr-implies-not0 if-SE-def option.case*(*1*)
option.case-eq-if while-SE-unfold)
next
fix *k n*
assume *hyp* : *?P n*
have *1* : $\bigwedge\sigma.\ P\ \sigma \Longrightarrow cond\ \sigma$
by (*simp add*: *enabled* ∗)
show *?P* (*Suc n*)

apply(*auto*, *frule 1*)
apply(*subst while-SE-unfold*, *auto simp*: *if-SE-def*)
proof(*insert* $*$,*simp-all add*: *hoare*$_3$*-def hoare*$_3$′*-def*, *erule disjE*)
fix σ
assume $P\ \sigma$
and $f\ \sigma \leq Suc\ n$
and $cond\ \sigma$
and $** : \forall\sigma.\ cond\ \sigma \wedge P\ \sigma \longrightarrow (case\ M\ \sigma\ of\ None \Rightarrow False \mid Some\ (x, \sigma') \Rightarrow P\ \sigma')$
obtain $(case\ M\ \sigma\ of\ None \Rightarrow False \mid Some\ (x, \sigma') \Rightarrow P\ \sigma')$
by (*simp add*: $**$ ‹$P\ \sigma$› ‹$cond\ \sigma$›)
then
show $case\ (M\ ;\!-\ (while_{SE}\ cond\ do\ M\ od))\ \sigma\ of\ None \Rightarrow True \mid Some\ a \Rightarrow False$
apply(*case-tac* $M\ \sigma$, *auto*, *rename-tac* σ', *simp add*: *bind-SE*′*-def bind-SE-def*)
proof $-$
fix σ'
assume $P\ \sigma'$
and $M\ \sigma = Some\ ((), \sigma')$
have $cond\ \sigma'$ **by** (*simp add*: ‹$P\ \sigma'$› *enabled*)
have $f\ \sigma' \leq n$
using ‹$M\ \sigma = Some\ ((), \sigma')$› ‹$P\ \sigma$› ‹$cond\ \sigma$› ‹$f\ \sigma \leq Suc\ n$› *measure* **by** *fastforce*
show $case\ (while_{SE}\ cond\ do\ M\ od)\ \sigma'\ of\ None \Rightarrow True \mid Some\ a \Rightarrow False$
using *hyp* **by** (*simp add*: ‹$P\ \sigma'$› ‹$f\ \sigma' \leq n$›)
qed
next
fix σ
assume $P\ \sigma$
and $f\ \sigma \leq Suc\ n$
and $cond\ \sigma$
and $* : \forall\sigma.\ cond\ \sigma \wedge P\ \sigma \longrightarrow (case\ M\ \sigma\ of\ None \Rightarrow True \mid Some\ a \Rightarrow False)$
obtain $** : (case\ M\ \sigma\ of\ None \Rightarrow True \mid Some\ a \Rightarrow False)$
by (*simp add*: $*$ ‹$P\ \sigma$› ‹$cond\ \sigma$›)
have $M\ \sigma = None$
by (*simp add*: $**$ *option.disc-eq-case*(*1*))
show $case\ (M\ ;\!-\ (while_{SE}\ cond\ do\ M\ od))\ \sigma\ of\ None \Rightarrow True \mid Some\ a \Rightarrow False$
by (*simp add*: ‹$M\ \sigma = None$› *bind-SE*′*-def bind-SE-def*)
qed
qed
show $\bigwedge\sigma.\ P\ \sigma \Longrightarrow case\ (while_{SE}\ cond\ do\ M\ od)\ \sigma\ of\ None \Rightarrow True \mid Some\ a \Rightarrow False$ **using** $*$ **by** *blast*
qed

3.3.6 Experimental Alternative Definitions (Transformer-Style Rely-Guarantee)

definition $hoare_1 :: ('\sigma \Rightarrow bool) \Rightarrow ('\alpha, '\sigma)MON_{SE} \Rightarrow ('\alpha \Rightarrow '\sigma \Rightarrow bool) \Rightarrow bool$ ($\vdash_1$ ({(*1*-)}/ (-)/ {(*1*-)}) *50*)
where $(\vdash_1\{P\}\ M\ \{Q\}\ ) = (\forall\sigma.\ \sigma \models (\text{-} \leftarrow assume_{SE}\ P\ ;\ x \leftarrow M;\ assert_{SE}\ (Q\ x)))$

definition *hoare*$_2$ :: $('\sigma \Rightarrow bool) \Rightarrow ('\alpha, '\sigma)MON_{SE} \Rightarrow ('\alpha \Rightarrow '\sigma \Rightarrow bool) \Rightarrow bool$ ($\vdash_2$ ({(*1*-)}/ (-)/ {(*1*-)}) *50*)
where $(\vdash_2\{P\}\ M\ \{Q\}\ ) = (\forall \sigma.\ P\ \sigma \longrightarrow (\sigma \models\ (x \leftarrow M;\ assert_{SE}\ (Q\ x))))$

end

theory *Hoare-Clean*
imports *Hoare-MonadSE*
Clean
begin

3.3.7 Clean Control Rules

lemma *break1*:
$\{\!|\lambda\sigma.\ P\ (\sigma\ (\!|\ break\text{-}status := True\ |\!))\ |\!\}\ break\ \{\!|\lambda r\ \sigma.\ P\ \sigma \wedge break\text{-}status\ \sigma\ |\!\}$
unfolding *hoare*$_3$*-def break-def unit-SE-def* **by** *auto*

lemma *unset-break1*:
$\{\!|\lambda\sigma.\ P\ (\sigma\ (\!|\ break\text{-}status := False\ |\!))\ |\!\}\ unset\text{-}break\text{-}status\ \{\!|\lambda r\ \sigma.\ P\ \sigma \wedge \neg\ break\text{-}status\ \sigma\ |\!\}$
unfolding *hoare*$_3$*-def unset-break-status-def unit-SE-def* **by** *auto*

lemma *set-return1*:
$\{\!|\lambda\sigma.\ P\ (\sigma\ (\!|\ return\text{-}status := True\ |\!))\ |\!\}\ set\text{-}return\text{-}status\ \{\!|\lambda r\ \sigma.\ P\ \sigma \wedge return\text{-}status\ \sigma\ |\!\}$
unfolding *hoare*$_3$*-def set-return-status-def unit-SE-def* **by** *auto*

lemma *unset-return1*:
$\{\!|\lambda\sigma.\ P\ (\sigma\ (\!|\ return\text{-}status := False\ |\!))\ |\!\}\ unset\text{-}return\text{-}status\ \{\!|\lambda r\ \sigma.\ P\ \sigma \wedge \neg return\text{-}status\ \sigma\ |\!\}$
unfolding *hoare*$_3$*-def unset-return-status-def unit-SE-def* **by** *auto*

3.3.8 Clean Skip Rules

lemma *assign-global-skip*:
$\{\!|\lambda\sigma.\ exec\text{-}stop\ \sigma \wedge P\ \sigma\ |\!\}\ assign\text{-}global\ upd\ rhs\ \{\!|\lambda r\ \sigma.\ exec\text{-}stop\ \sigma \wedge P\ \sigma\ |\!\}$
unfolding *hoare*$_3$*-def skip*$_{SE}$*-def unit-SE-def*
by (*simp add*: *assign-def assign-global-def*)

lemma *assign-local-skip*:
$\{\!|\lambda\sigma.\ exec\text{-}stop\ \sigma \wedge P\ \sigma\ |\!\}\ assign\text{-}local\ upd\ rhs\ \{\!|\lambda r\ \sigma.\ exec\text{-}stop\ \sigma \wedge P\ \sigma\ |\!\}$
unfolding *hoare*$_3$*-def skip*$_{SE}$*-def unit-SE-def*
by (*simp add*: *assign-def assign-local-def*)

lemma *return-skip*:
$\{\!|\lambda\sigma.\ exec\text{-}stop\ \sigma \wedge P\ \sigma\ |\!\}\ return_C\ upd\ rhs\ \{\!|\lambda r\ \sigma.\ exec\text{-}stop\ \sigma \wedge P\ \sigma\ |\!\}$
unfolding *hoare*$_3$*-def return*$_C$*-def unit-SE-def assign-local-def assign-def bind-SE'-def bind-SE-def*
by *auto*

lemma *assign-clean-skip*:
$\{\!|\lambda\sigma.\ \ exec\text{-}stop\ \sigma \wedge P\ \sigma\ |\!\}$ $assign\ tr$ $\{\!|\lambda r\ \sigma.\ exec\text{-}stop\ \sigma \wedge P\ \sigma\ |\!\}$
  unfolding *hoare*$_3$*-def skip*$_{SE}$*-def unit-SE-def*
  by (*simp add*: *assign-def assign-def*)

lemma *if-clean-skip*:
$\{\!|\lambda\sigma.\ \ exec\text{-}stop\ \sigma \wedge P\ \sigma\ |\!\}$ $if_C\ C\ then\ E\ else\ F\ fi$ $\{\!|\lambda r\ \sigma.\ exec\text{-}stop\ \sigma \wedge P\ \sigma\ |\!\}$
  unfolding *hoare*$_3$*-def skip*$_{SE}$*-def unit-SE-def if-SE-def*
  by (*simp add*: *if-C-def*)

lemma *while-clean-skip*:
$\{\!|\lambda\sigma.\ \ exec\text{-}stop\ \sigma \wedge P\ \sigma\ |\!\}$ $while_C\ cond\ do\ body\ od$ $\{\!|\lambda r\ \sigma.\ exec\text{-}stop\ \sigma \wedge P\ \sigma\ |\!\}$
  unfolding *hoare*$_3$*-def skip*$_{SE}$*-def unit-SE-def while-C-def*
  by *auto*

lemma *if-opcall-skip*:
$\{\!|\lambda\sigma.\ \ exec\text{-}stop\ \sigma \wedge P\ \sigma|\!\}$ $(call_C\ M\ A_1)$ $\{\!|\lambda r\ \sigma.\ exec\text{-}stop\ \sigma \wedge P\ \sigma|\!\}$
  unfolding *hoare*$_3$*-def skip*$_{SE}$*-def unit-SE-def call*$_C$*-def*
  by *simp*

lemma *if-funcall-skip*:
$\{\!|\lambda\sigma.\ exec\text{-}stop\ \sigma \wedge P\ \sigma|\!\}(p_{tmp} \leftarrow call_C\ fun\ E\ ;\ assign\text{-}local\ upd\ (\lambda\sigma.\ p_{tmp}))$ $\{\!|\lambda r\ \sigma.\ exec\text{-}stop\ \sigma \wedge P\ \sigma|\!\}$
  unfolding *hoare*$_3$*-def skip*$_{SE}$*-def unit-SE-def call*$_C$*-def assign-local-def assign-def*
  by (*simp add*: *bind-SE-def*)

lemma *if-funcall-skip'*:
$\{\!|\lambda\sigma.\ exec\text{-}stop\ \sigma \wedge P\ \sigma\ |\!\}(p_{tmp} \leftarrow call_C\ fun\ E\ ;\ assign\text{-}global\ upd\ (\lambda\sigma.\ p_{tmp}))$ $\{\!|\lambda r\ \sigma.\ exec\text{-}stop\ \sigma \wedge P\ \sigma\ |\!\}$
  unfolding *hoare*$_3$*-def skip*$_{SE}$*-def unit-SE-def call*$_C$*-def assign-global-def assign-def*
  by (*simp add*: *bind-SE-def*)

3.3.9 Clean Assign Rules

lemma *assign-global*:
  assumes $* : \sharp\ upd$
  shows $\{\!|\lambda\sigma.\ \neg exec\text{-}stop\ \sigma \wedge P\ (upd\ (\lambda\text{-}.\ rhs\ \sigma)\ \sigma)\ |\!\}$
    assign-global upd rhs
    $\{\!|\lambda r\ \sigma.\ \neg exec\text{-}stop\ \sigma \wedge P\ \sigma\ |\!\}$
  unfolding *hoare*$_3$*-def skip*$_{SE}$*-def unit-SE-def assign-global-def assign-def*
  by(*auto simp*: *assms*)

lemma *assign-local*:
  assumes $* : \sharp\ (upd \circ map\text{-}hd)$
  shows $\{\!|\lambda\sigma.\ \neg\ exec\text{-}stop\ \sigma \wedge P\ ((upd \circ map\text{-}hd)\ (\lambda\text{-}.\ rhs\ \sigma)\ \sigma)\ |\!\}$
    assign-local upd rhs
    $\{\!|\lambda r\ \sigma.\ \neg\ exec\text{-}stop\ \sigma \wedge P\ \sigma\ |\!\}$
  unfolding *hoare*$_3$*-def skip*$_{SE}$*-def unit-SE-def assign-local-def assign-def*

using *assms exec-stop-vs-control-independence* **by** *fastforce*

lemma *return-assign*:
assumes $*$: $\sharp$ (*upd* $\circ$ *map-hd*)
shows $\{\!|\lambda\ \sigma.\ \neg\ \textit{exec-stop}\ \sigma \wedge P\ ((\textit{upd} \circ \textit{map-hd})\ (\lambda\text{-}.\ \textit{rhs}\ \sigma)\ (\sigma\ (\!|\ \textit{return-status} := \textit{True}\ |\!)))|\!\}$
$\textit{return}_C\ \textit{upd rhs}$
$\{\!|\lambda r\ \sigma.\ P\ \sigma \wedge \textit{return-status}\ \sigma\ |\!\}$
unfolding $\textit{return}_C$*-def* $\textit{hoare}_3$*-def* $\textit{skip}_{SE}$*-def unit-SE-def assign-local-def assign-def set-return-status-def bind-SE'-def bind-SE-def*
proof (*auto*)
fix σ :: *'b control-state-scheme*
assume *a1*: $P\ (\textit{upd}\ (\textit{map-hd}\ (\lambda\text{-}.\ \textit{rhs}\ \sigma))\ (\sigma(\!|\textit{return-status} := \textit{True}|\!)))$
assume $\neg\ \textit{exec-stop}\ \sigma$
show $P\ (\textit{upd}\ (\textit{map-hd}\ (\lambda\text{-}.\ \textit{rhs}\ \sigma))\ \sigma(\!|\textit{return-status} := \textit{True}|\!))$
using *a1 assms exec-stop-vs-control-independence'* **by** *fastforce*
qed

3.3.10 Clean Construct Rules

lemma *cond-clean* :
$\{\!|\lambda\sigma.\ \neg\ \textit{exec-stop}\ \sigma \wedge P\ \sigma \wedge \textit{cond}\ \sigma|\!\}\ M\ \{\!|Q|\!\}$
$\Longrightarrow \{\!|\lambda\sigma.\ \neg\ \textit{exec-stop}\ \sigma \wedge P\ \sigma \wedge \neg\ \textit{cond}\ \sigma|\!\}\ M'\ \{\!|Q|\!\}$
$\Longrightarrow \{\!|\lambda\sigma.\ \neg\ \textit{exec-stop}\ \sigma \wedge P\ \sigma|\!\}\ \textit{if}_C\ \textit{cond then}\ M\ \textit{else}\ M'\ \textit{fi}\{\!|Q|\!\}$
unfolding $\textit{hoare}_3$*-def* $\textit{hoare}_3{}'$*-def bind-SE-def if-SE-def*
by (*simp add*: *if-C-def*)

There is a particular difficulty with a verification of (terminating) while rules in a Hoare-logic for a language involving break. The first is, that break is not used in the toplevel of a body of a loop (there might be breaks inside an inner loop, though). This scheme is covered by the rule below, which is a generalisation of the classical while loop (as presented by $[\![\{\!|\lambda\sigma.\ ?cond\ \sigma \wedge ?P\ \sigma|\!\}\ ?M\ \{\!|\lambda\text{-}.\ ?P|\!\};\ \forall\sigma.\ ?cond\ \sigma \wedge ?P\ \sigma \longrightarrow ?M\ \sigma \neq \textit{None} \wedge ?f\ (\textit{snd}\ (\textit{the}\ (?M\ \sigma))) < ?f\ \sigma]\!] \Longrightarrow \{\!|?P|\!\}\ \textit{-while-SE}\ ?cond\ ?M\ \{\!|\lambda\text{-}\ \sigma.\ \neg\ ?cond\ \sigma \wedge ?P\ \sigma|\!\}$.

lemma *while-clean-no-break* :
assumes $*$: $\{\!|\lambda\sigma.\ \neg\ \textit{break-status}\ \sigma \wedge \textit{cond}\ \sigma \wedge P\ \sigma|\!\}\ \ M\ \{\!|\lambda\text{-}.\ \lambda\sigma.\ \neg\ \textit{break-status}\ \sigma \wedge P\ \sigma\ |\!\}$
and *measure*: $\forall\sigma.\ \neg\ \textit{exec-stop}\ \sigma \wedge \textit{cond}\ \sigma \wedge P\ \sigma$
$\longrightarrow M\ \sigma \neq \textit{None} \wedge f(\textit{snd}(\textit{the}(M\ \sigma))) < ((f\ \sigma)::\textit{nat})$
(**is** $\forall\sigma.$ - $\wedge\ \textit{cond}\ \sigma \wedge P\ \sigma \longrightarrow ?\textit{decrease}\ \sigma$)
shows $\{\!|\lambda\sigma.\ \neg\ \textit{exec-stop}\ \sigma \wedge P\ \sigma|\!\}$
$\textit{while}_C\ \textit{cond do}\ M\ \textit{od}$
$\{\!|\lambda\text{-}\ \sigma.\ (\textit{return-status}\ \sigma \vee \neg\ \textit{cond}\ \sigma) \wedge \neg\ \textit{break-status}\ \sigma \wedge P\ \sigma|\!\}$
(**is** $\{\!|?pre|\!\}\ \textit{while}_C\ \textit{cond do}\ M\ \textit{od}\ \{\!|\lambda\text{-}\ \sigma.\ ?post1\ \sigma \wedge ?post2\ \sigma|\!\}$)
unfolding *while-C-def* $\textit{hoare}_3$*-def* $\textit{hoare}_3{}'$*-def*
proof (*simp add*: $\textit{hoare}_3$*-def*[*symmetric*],*rule sequence'*)
show $\{\!|?pre|\!\}$
$\textit{while}_{SE}\ (\lambda\sigma.\ \neg\ \textit{exec-stop}\ \sigma \wedge \textit{cond}\ \sigma)\ \textit{do}\ M\ \textit{od}$
$\{\!|\lambda\text{-}\ \sigma.\ \neg\ (\neg\ \textit{exec-stop}\ \sigma \wedge \textit{cond}\ \sigma) \wedge \neg\ \textit{break-status}\ \sigma \wedge P\ \sigma|\!\}$
(**is** $\{\!|?pre|\!\}\ \textit{while}_{SE}\ ?cond'\ \textit{do}\ M\ \textit{od}\ \{\!|\lambda\text{-}\ \sigma.\ \neg\ (\ ?cond'\ \sigma) \wedge ?post2\ \sigma|\!\}$)
proof (*rule consequence-unit*)

fix σ **show** *?pre* $\sigma \longrightarrow$ *?post2* σ **using** *exec-stop1* **by** *blast*
next
show $\{\!|$ *?post2* $|\!\}$ *while*$_{SE}$ *?cond'* *do M od* $\{\!|\lambda x\ \sigma.\ \neg($*?cond'* $\sigma) \wedge$ *?post2* $\sigma|\!\}$
proof (*rule-tac f = f* **in** *while*, *rule consequence-unit*)
fix σ **show** *?cond'* $\sigma \wedge$ *?post2* $\sigma \longrightarrow \neg$*break-status* $\sigma \wedge$ *cond* $\sigma \wedge P\ \sigma$ **by** *simp*
next
show $\{\!|\lambda\sigma.\ \neg$ *break-status* $\sigma \wedge$ *cond* $\sigma \wedge P\ \sigma|\!\}$ *M* $\{\!|\lambda x\ \sigma.$ *?post2* $\sigma|\!\}$ **using** $*$ **by** *blast*
next
fix σ **show** *?post2* $\sigma \longrightarrow$ *?post2* σ **by** *blast*
next
show $\forall\,\sigma.$*?cond'* $\sigma \wedge$ *?post2* $\sigma \longrightarrow$ *?decrease* σ **using** *measure* **by** *blast*
qed
next
fix σ **show** $\neg$*?cond'* $\sigma \wedge$ *?post2* $\sigma \longrightarrow \neg$*?cond'* $\sigma \wedge$ *?post2* σ **by** *blast*
qed
next
show $\{\!|\lambda\sigma.\ \neg\ (\neg$ *exec-stop* $\sigma \wedge$ *cond* $\sigma) \wedge$ *?post2* $\sigma|\!\}$ *unset-break-status*
$\{\!|\lambda\text{-}\ \sigma'.$ (*return-status* $\sigma' \vee \neg$ *cond* σ') $\wedge$ *?post2* $\sigma'|\!\}$
(**is** $\{\!|\lambda\sigma.\ \neg$ (*?cond''* σ) $\wedge$ *?post2* $\sigma|\!\}$ *unset-break-status* $\{\!|\lambda\text{-}\ \sigma'.$ *?post3* $\sigma' \wedge$ *?post2* $\sigma'\ |\!\}$)
proof (*rule consequence-unit*)
fix σ
show $\neg$ *?cond''* $\sigma \wedge$ *?post2* $\sigma \longrightarrow (\lambda\sigma.\ P\ \sigma \wedge$ *?post3* $\sigma)\ (\sigma(\!|$*break-status* := *False*$|\!))$
by (*metis* (*full-types*) *exec-stop-def surjective update-convs*(*1*))
next
show $\{\!|\lambda\sigma.\ (\lambda\sigma.\ P\ \sigma \wedge$ *?post3* $\sigma)\ (\sigma(\!|$*break-status* := *False*$|\!))|\!\}$
unset-break-status
$\{\!|\lambda x\ \sigma.$ *?post3* $\sigma \wedge \neg$ *break-status* $\sigma \wedge P\ \sigma|\!\}$
apply(*subst* (*2*) *conj-commute*,*subst conj-assoc*,*subst* (*2*) *conj-commute*)
by(*rule unset-break1*)
next
fix σ **show** *?post3* $\sigma \wedge$ *?post2* $\sigma \longrightarrow$ *?post3* $\sigma \wedge$ *?post2* σ **by** *simp*
qed
qed

In the following we present a version allowing a break inside the body, which implies that the invariant has been established at the break-point and the condition is irrelevant. A return may occur, but the *break-status* is guaranteed to be true after leaving the loop.

lemma *while-clean'*:
assumes *M-inv* : $\{\!|\lambda\sigma.\ \neg$ *exec-stop* $\sigma \wedge$ *cond* $\sigma \wedge P\ \sigma|\!\}$ *M* $\{\!|\lambda\text{-}.\ P|\!\}$
and *cond-idpc* : $\forall\,x\ \sigma.$ (*cond* $(\sigma(\!|$*break-status* := $x|\!)))$ = *cond* σ
and *inv-idpc* : $\forall\,x\ \sigma.$ $(P\ (\sigma(\!|$*break-status* := $x|\!))) = P\ \sigma$
and *f-is-measure* : $\forall\,\sigma.\ \neg$ *exec-stop* $\sigma \wedge$ *cond* $\sigma \wedge P\ \sigma \longrightarrow$
$M\ \sigma \neq None \wedge f(snd(the(M\ \sigma))) < ((f\ \sigma)::nat)$
shows $\{\!|\lambda\sigma.\ \neg$ *exec-stop* $\sigma \wedge P\ \sigma|\!\}$
while$_C$ *cond do M od*
$\{\!|\lambda\text{-}\ \sigma.\ \neg$ *break-status* $\sigma \wedge P\ \sigma|\!\}$
unfolding *while-C-def hoare*$_3$*-def hoare*$_3$*'-def*
proof (*simp add*: *hoare*$_3$*-def*[*symmetric*], *rule sequence'*)
show $\{\!|\lambda\sigma.\ \neg$ *exec-stop* $\sigma \wedge P\ \sigma|\!\}$

$while_{SE}$ *(λσ. ¬ exec-stop σ ∧ cond σ) do M od*
⦃λ- σ. P (σ⦇break-status := False⦈)⦄
apply(*rule consequence-unit, rule impI, erule conjunct2*)
apply(*rule-tac f = f* **in** *while*)
using *M-inv f-is-measure inv-idpc* **by** *auto*
next
show *⦃λσ. P (σ⦇break-status := False⦈)⦄ unset-break-status*
⦃λx σ. ¬ break-status σ ∧ P σ⦄
apply(*subst conj-commute*)
by(*rule Hoare-Clean.unset-break1*)
qed

Consequence and Sequence rules where inherited from the underlying Hoare-Monad theory.

end

Bibliography

[1] J. N. Foster. *Bidirectional programming languages*. PhD thesis, University of Pennsylvania, 2009. URL https://repository.upenn.edu/edissertations/56/.

[2] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3):17, 2007. doi: 10.1145/1232420.1232424. URL https://doi.org/10.1145/1232420.1232424.

[3] S. Foster, F. Zeyda, and J. Woodcock. Unifying heterogeneous state-spaces with lenses. In A. Sampaio and F. Wang, editors, *Theoretical Aspects of Computing - ICTAC 2016 - 13th International Colloquium, Taipei, Taiwan, ROC, October 24-31, 2016, Proceedings*, volume 9965 of *Lecture Notes in Computer Science*, pages 295–314, 2016. ISBN 978-3-319-46749-8. doi: 10.1007/978-3-319-46750-4_17. URL https://doi.org/10.1007/978-3-319-46750-4_17.

[4] C. Keller. Tactic program-based testing and bounded verification in isabelle/hol. In *Tests and Proofs - 12th International Conference, TAP 2018, Held as Part of STAF 2018, Toulouse, France, June 27-29, 2018, Proceedings*, pages 103–119, 2018. doi: 10.1007/978-3-319-92994-1_6. URL https://doi.org/10.1007/978-3-319-92994-1_6.

[5] F. Tuong and B. Wolff. Deeply integrating C11 code support into isabelle/pide. In *Formal IDE - 5th International Workshop, F-IDE 2019, Porto, Portugal, October 7, 2019. Proceedings*, 2019. URL https://gitlri.lri.fr/ftuong/isabelle_c.