Clean - An Abstract Imperative Programming Language and its Theory

Frédéric Tuong Burkhart Wolff
(with Contributions by Chantal Keller)

December 7, 2022

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 [**?**], or for the test-generation techniques described in [4].

Contents

1 The Clean Language

theory *Clean*
imports *Optics 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** $('o, 'σ)\ MON_{SE} = ‹'σ ⇀ ('o × 'σ)›$ 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 $'σ$ 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 $'σ$ extensible record type $'σ$ *control-state-ext* where the $'σ$ 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*

⟨*ML*⟩

definition *break* :: (*unit*, ('σ-*ext*) *control-state-ext*) MON_{SE}
where *break* ≡ (λ σ. *Some*((), σ (| *break-status* := *True* |)))

definition *unset-break-status* :: (*unit*, ('σ-*ext*) *control-state-ext*) MON_{SE}
where *unset-break-status* ≡ (λ σ. *Some*((), σ (| *break-status* := *False* |)))

definition *set-return-status* :: (*unit*, ('σ-*ext*) *control-state-ext*) MON_{SE}
where *set-return-status* = (λ σ. *Some*((), σ (| *return-status* := *True* |)))

definition *unset-return-status* :: (*unit*, ('σ-*ext*) *control-state-ext*) MON_{SE}
where *unset-return-status* = (λ σ. *Some*((), σ (| *return-status* := *False* |)))

definition *exec-stop* :: ('σ-*ext*) *control-state-ext* ⇒ *bool*
where *exec-stop* = (λ σ. *break-status* σ ∨ *return-status* σ)

abbreviation *normal-execution* :: ('σ-*ext*) *control-state-ext* ⇒ *bool*
where (*normal-execution s*) ≡ (¬ *exec-stop s*)
notation *normal-execution* (▷)

lemma *exec-stop1*[*simp*] : *break-status* σ ⟹ *exec-stop* σ
⟨*proof*⟩

lemma *exec-stop2*[*simp*] : *return-status* σ ⟹ *exec-stop* σ
⟨*proof*⟩

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 $break\text{-}status_L$
where $break\text{-}status_L$ = $create_L$ *control-state.break-status control-state.break-status-update*
lemma *vwb-lens* $break\text{-}status_L$
⟨*proof*⟩

definition *return-status*$_L$
where *return-status*$_L$ = *create*$_L$ *control-state.return-status control-state.return-status-update*
lemma *vwb-lens return-status*$_L$
⟨*proof*⟩

lemma *break-return-indep* : *break-status*$_L$ ⋈ *return-status*$_L$
⟨*proof*⟩

definition *strong-control-independence* (♯!)
where ♯! L = (*break-status*$_L$ ⋈ L ∧ *return-status*$_L$ ⋈ L)

lemma *vwb-lens break-status*$_L$
⟨*proof*⟩

definition *control-independence* ::
(('*b*⇒'*b*)⇒'*a control-state-scheme* ⇒ '*a control-state-scheme*) ⇒ *bool* (♯)
where ♯ *upd* ≡ (∀ σ *T b*. *break-status* (*upd T* σ) = *break-status* σ
∧ *return-status* (*upd T* σ) = *return-status* σ
∧ *upd T* (σ(| *return-status* := *b* |)) = (*upd T* σ)(| *return-status* := *b* |)
∧ *upd T* (σ(| *break-status* := *b* |)) = (*upd T* σ)(| *break-status* := *b* |))

lemma *strong-vs-weak-ci* : ♯! L ⟹ ♯ (λ*f*. λσ. *lens-put L* σ (*f* (*lens-get L* σ)))
⟨*proof*⟩

lemma *expimnt* :♯! (*create*$_L$ *getv updv*) ⟹ (λ*f* σ. *updv* (λ-. *f* (*getv* σ)) σ) = *updv*
⟨*proof*⟩

lemma *expimnt* :
vwb-lens (*create*$_L$ *getv updv*) ⟹ (λ*f* σ. *updv* (λ-. *f* (*getv* σ)) σ) = *updv*
⟨*proof*⟩

lemma *strong-vs-weak-upd* :
assumes ∗ : ♯! (*create*$_L$ *getv updv*)
and ∗∗ : (λ*f* σ. *updv* (λ-. *f* (*getv* σ)) σ) = *updv*
shows ♯ (*updv*)
⟨*proof*⟩

This quite tricky proof establishes the fact that the special case $hd(getv\ \sigma) = []$ for $getv\ \sigma = []$ is finally irrelevant in our setting. This implies that we don't need the list-lense-construction (so far).

lemma *strong-vs-weak-upd-list* :
assumes ∗ : ♯! (*create*$_L$ (*getv*:: '*b control-state-scheme* ⇒ '*c list*)
(*updv*:: ('*c list* ⇒ '*c list*) ⇒ '*b control-state-scheme* ⇒ '*b control-state-scheme*))

and ∗∗ : (λ*f* σ. *updv* (λ-. *f* (*getv* σ)) σ) = *updv*

shows $\sharp\ (updv \circ upd\text{-}hd)$
⟨*proof*⟩

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

lemma *exec-stop-vs-control-independence′* [*simp*]:
$\sharp\ upd \Longrightarrow (upd\ f\ (\sigma\ (\!|\ return\text{-}status := b\ |\!))) = (upd\ f\ \sigma)(\!|\ return\text{-}status := b\ |\!)$
⟨*proof*⟩

lemma *exec-stop-vs-control-independence″* [*simp*]:
$\sharp\ upd \Longrightarrow (upd\ f\ (\sigma\ (\!|\ break\text{-}status := b\ |\!))) = (upd\ f\ \sigma)\ (\!|\ break\text{-}status := b\ |\!)$
⟨*proof*⟩

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*)

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

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

definition *assign-global* :: (('*a* ⇒ '*a*) ⇒ 'σ-*ext control-state-scheme* ⇒ 'σ-*ext control-state-scheme*)
⇒ ('σ-*ext control-state-scheme* ⇒ '*a*)
⇒ (*unit*,'σ-*ext control-state-scheme*) MON_{SE} (**infix** :==$_G$ *100*)
where *assign-global upd rhs* = *assign*(λσ. ((*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* (λσ. *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).

definition *assign-local* :: (('*a list* ⇒ '*a list*)
⇒ 'σ-*ext control-state-scheme* ⇒ 'σ-*ext control-state-scheme*)
⇒ ('σ-*ext control-state-scheme* ⇒ '*a*)
⇒ (*unit*,'σ-*ext control-state-scheme*) MON_{SE} (**infix** :==$_L$ *100*)
where *assign-local upd rhs* = *assign*(λσ. ((*upd o upd-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 (*upd* :==$_L$ *rhs*) = ((*upd* ∘ *upd-hd*) :==$_G$ *rhs*)
⟨*proof*⟩

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*$_C$ *0*
where *return*$_C$ *0 A* = (λσ. *if exec-stop* σ *then Some*((), σ)
else (*A* ;− *set-return-status*) σ)

definition *return*$_C$ :: (('*a list* ⇒ '*a list*) ⇒ 'σ-*ext control-state-scheme* ⇒ 'σ-*ext control-state-scheme*)
⇒ ('σ-*ext control-state-scheme* ⇒ '*a*)
⇒ (*unit*,'σ-*ext control-state-scheme*) MON_{SE} (*return*$_1$)
where *return*$_C$ *upd rhs* = *return*$_C$ *0* (*assign-local upd rhs*)

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* (λσ. (*A* σ) ! *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*⟩

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
```

$\Rightarrow$ ('α, ('σ-*ext*) *control-state-ext*)MON_{SE}
$\Rightarrow$ ('α, ('σ-*ext*) *control-state-ext*)MON_{SE}
where *block*$_C$ *push core pop* $\equiv$ (— 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
($x \leftarrow$ *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* $\times$ *nat* $\Rightarrow$ (*unit*,'*a local-swap-state-scheme*) MON_{SE}
where swap-core $\equiv$ ($\lambda(i,j)$. ((*assign-local tmp-update* ($\lambda\sigma$. *A* σ ! *i*)) ;–
(*assign-global A-update* ($\lambda\sigma$. *list-update* (*A* σ) (*i*) (*A* σ ! *j*))) ;–
(*assign-global A-update* ($\lambda\sigma$. *list-update* (*A* σ) (*j*) ((*hd o tmp*) σ)))))

definition swap :: *nat* $\times$ *nat* $\Rightarrow$ (*unit*,'*a local-swap-state-scheme*) MON_{SE}
where swap $\equiv \lambda(i,j)$. *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*$_{upd}$-aware.

definition *call*$_C$ :: ('α $\Rightarrow$ ('ϱ, ('σ-*ext*) *control-state-ext*)MON_{SE}) $\Rightarrow$
((('σ-*ext*) *control-state-ext*) $\Rightarrow$ 'α) $\Rightarrow$
('ϱ, ('σ-*ext*) *control-state-ext*)MON_{SE}
where *call*$_C$ *M A*$_1$ = ($\lambda\sigma$. *if exec-stop* σ *then Some*(*undefined*, σ) *else M* (*A*$_1$ σ) σ)

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

The generic version using tuples is identical with *call-1*$_C$.

definition *call-1*$_C$ :: ('α $\Rightarrow$ ('ϱ, ('σ-*ext*) *control-state-ext*)MON_{SE}) $\Rightarrow$
((('σ-*ext*) *control-state-ext*) $\Rightarrow$ 'α) $\Rightarrow$
('ϱ, ('σ-*ext*) *control-state-ext*)MON_{SE}
where *call-1*$_C$ = *call*$_C$

definition *call-2*$_C$ :: ('α $\Rightarrow$ 'β $\Rightarrow$ ('ϱ, ('σ-*ext*) *control-state-ext*)MON_{SE}) $\Rightarrow$
((('σ-*ext*) *control-state-ext*) $\Rightarrow$ 'α) $\Rightarrow$
((('σ-*ext*) *control-state-ext*) $\Rightarrow$ 'β) $\Rightarrow$

$('\varrho, ('\sigma\text{-}ext)\ control\text{-}state\text{-}ext)MON_{SE}$
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 $call\text{-}3_C :: (\ 'a \Rightarrow 'b \Rightarrow\ 'c \Rightarrow ('\varrho, ('\sigma\text{-}ext)\ control\text{-}state\text{-}ext)MON_{SE}) \Rightarrow$
$((('\sigma\text{-}ext)\ control\text{-}state\text{-}ext) \Rightarrow 'a) \Rightarrow$
$((('\sigma\text{-}ext)\ control\text{-}state\text{-}ext) \Rightarrow 'b) \Rightarrow$
$((('\sigma\text{-}ext)\ control\text{-}state\text{-}ext) \Rightarrow 'c) \Rightarrow$
$('\varrho, ('\sigma\text{-}ext)\ control\text{-}state\text{-}ext)MON_{SE}$
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*⟩

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*⟩

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

⟨*ML*⟩

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

⟨*ML*⟩

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 co-ordinates:

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.

In order to support the `old`-notation known from JML and similar annotation languages, we introduce the following definition:

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

The core module of the parser and operation specification construct is implemented in the following module:

⟨*ML*⟩

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

definition *if-C* :: [('σ-ext) *control-state-ext* ⇒ *bool*,
('β, ('σ-ext) *control-state-ext*)MON_{SE},
('β, ('σ-ext) *control-state-ext*)MON_{SE}] ⇒ ('β, ('σ-ext) *control-state-ext*)MON_{SE}
where *if-C c E F* = (λσ. *if exec-stop* σ
then Some(*undefined*, σ) — state unchanged, return arbitrary
else if c σ *then E* σ *else F* σ)

syntax (*xsymbols*)
-if-SECLEAN :: ['σ ⇒ *bool*,('o,'σ)MON_{SE},('o','σ)MON_{SE}] ⇒ ('o','σ)MON_{SE}
((*if*$_C$ - *then* - *else* -*fi*) [5,8,8]20)
translations
(*if*$_C$ *cond then T1 else T2 fi*) == *CONST Clean.if-C cond T1 T2*

definition *while-C* :: (('σ-ext) *control-state-ext* ⇒ *bool*)
⇒ (*unit*, ('σ-ext) *control-state-ext*)MON_{SE}
⇒ (*unit*, ('σ-ext) *control-state-ext*)MON_{SE}
where *while-C c B* ≡ (λσ. *if exec-stop* σ *then Some*((), σ)
else ((*MonadSE.while-SE* (λ σ. ¬*exec-stop* σ ∧ *c* σ) *B*) ;–
unset-break-status) σ)

syntax (*xsymbols*)
-while-C :: ['σ ⇒ *bool*, (*unit*, 'σ)MON_{SE}] ⇒ (*unit*, 'σ)MON_{SE}
((*while*$_C$ - *do* - *od*) [8,8]20)

translations
while$_C$ c do b od == CONST Clean.while-C c b

1.7 Miscellaneous

Since `int` were mapped to Isabelle/HOL *int* and `unsigned int` to *nat*, there is the need for a common interface for accesses in arrays, which were represented by Isabelle/HOL lists:

consts *nth$_C$ :: 'a list ⇒ 'b ⇒ 'a*
overloading *nth$_C$ ≡ nth$_C$ :: 'a list ⇒ nat ⇒ 'a*
begin
definition
nth$_C$-nat : nth$_C$ (S::'a list) (a) ≡ nth S a
end

overloading *nth$_C$ ≡ nth$_C$ :: 'a list ⇒ int ⇒ 'a*
begin
definition
nth$_C$-int : nth$_C$ (S::'a list) (a) ≡ nth S (nat a)
end

definition *while-C-A :: (('σ-ext) control-state-scheme ⇒ bool)*
⇒ (('σ-ext) control-state-scheme ⇒ 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-C-A Inv f c B ≡ while-C c B*

⟨*ML*⟩

1.8 Function-calls in Expressions

The precise semantics of function-calls appearing inside expressions is underspecified in C, which is a notorious problem for compilers and analysis tools. In Clean, it is impossible by construction — and the type displine — to have function-calls inside expressions. However, there is a somewhat *recommended coding-scheme* for this feature, which leaves this issue to decisions in the front-end:

```
a = f() + g();
```

can be represented in Clean by: $x \leftarrow f(); y \leftarrow g(); \langle a := x + y \rangle$ or $x \leftarrow g(); y \leftarrow f(); \langle a := y + x \rangle$ which makes the evaluation order explicit without introducing local variables or any form of explicit trace on the state-space of the Clean program. We assume, however, even in this coding scheme, that `f()` and `g()` are atomic actions; note that

this assumption is not necessarily justified in modern compilers, where actually neither of these two (atomic) serializations of `f()` and `g()` may exists.

Note, furthermore, that expressions may not only be right-hand-sides of (local or global) assignments or conceptually similar return-statements, but also passed as argument of other function calls, where the same problem arises.

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.Clean*
Clean.Hoare-Clean
Clean.Clean-Symbex
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*⟩

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*

find-theorems $create_L$ *name*:*Quick*

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

⟨*ML*⟩

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

A Simulation of Synthesis of Typed Assignment-Rules

definition $A_L{}'$ **where** $A_L{}' \equiv create_L$ *global-state-state.A global-state-state.A-update*

lemma $A_L{}'$*-control-indep* : (*break-status*$_L \bowtie A_L{}' \wedge$ *return-status*$_L \bowtie A_L{}'$)
  ⟨*proof*⟩

lemma $A_L{}'$*-strong-indep* : $\sharp!\ A_L{}'$
  ⟨*proof*⟩

Specialized Assignment Rule for Global Variable *A*. Note that this specialized rule of $\sharp$ *?upd* $\Longrightarrow \{\!|\lambda\sigma.\ \triangleright\ \sigma \wedge$ *?P* (*?upd* (λ-. *?rhs* σ) σ)$|\!\}$ *?upd* $:==_G$ *?rhs* $\{\!|\lambda r\ \sigma.\ \triangleright\ \sigma \wedge$ *?P* σ$|\!\}$ does not need any further side-conditions referring to independence from the control. Consequently, backward inference in an *wp*-calculus will just maintain the invariant $\triangleright\ \sigma$.

lemma *assign-global-A*:
  $\{\!|\lambda\sigma.\ \triangleright\ \sigma \wedge\ P\ (\sigma(\!|A := rhs\ \sigma|\!))|\!\}$ *A-update* $:==_G$ *rhs* $\{\!|\lambda r\ \sigma.\ \triangleright\ \sigma \wedge P\ \sigma\ |\!\}$
  ⟨*proof*⟩

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.checkNsem_functior`
pre ‹$i < length\ A \wedge j < length\ A$›
post ‹$\lambda res.\ length\ A = length(old\ A) \wedge res = ()$›
local-vars *tmp* :: *int*
defines ‹ $tmp := A\ !\ i$› ;–
‹ $A := list\text{-}update\ A\ i\ (A\ !\ j)$› ;–
‹ $A := list\text{-}update\ A\ j\ tmp$›

value (|*break-status* = *False*, *return-status* = *False*, $A = [1,2,3]$,
tmp = [], *result-value* = [], ... = *X*|)

value *swap* (*0*,*1*) (|*break-status* = *False*, *return-status* = *False*, $A = [1,2,3]$,
tmp = [],
result-value = [],... = *X*|)

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

‹*defines* ((*assign-local tmp-update* ($\lambda\sigma.$ ($A\ \sigma$) ! i)) ;–
(*assign-global A-update* ($\lambda\sigma.$ *list-update* ($A\ \sigma$) (i) ($A\ \sigma$! j))) ;–
(*assign-global A-update* ($\lambda\sigma.$ *list-update* ($A\ \sigma$) (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*⟩

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′-pre* :: $nat \times nat \Rightarrow$ *'a global-state-state-scheme* $\Rightarrow$ *bool*
where *swap′-pre* $\equiv \lambda(i,\ j)\ \sigma.\ i < length\ (A\ \sigma) \wedge j < length\ (A\ \sigma)$
definition *swap′-post* :: *'a* $\times$ *'b* $\Rightarrow$ *'c global-state-state-scheme* $\Rightarrow$ *'d global-state-state-scheme* $\Rightarrow$ *unit* $\Rightarrow$ *bool*
where *swap′-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*

local-vars-test *swap′ unit*
  tmp :: *int*

The immediate effect of this command on the internal Clean State Management can be made explicit as follows:

⟨*ML*⟩
This has already the effect of the definition:

thm *push-local-swap-state-def*
thm *pop-local-swap-state-def*

Again, we simulate the effect of this command by more elementary HOLspecification constructs:

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* |))

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*),
      — recall : returns op value
      — which happens to be unit
      σ(|*local-swap-state.tmp*:= *tl*(*local-swap-state.tmp σ*) |))

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

a block manages the "dynamically" created fresh instances for the local variables of swap

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′*

NOTE: If local variables were only used in single-assignment style, it is possible to drastically simplify the encoding. These variables were not stored in the state, just kept as part of the monadic calculation. The simplifications refer both to calculation as well as well as symbolic execution and deduction.

The could be represented by the following alternative, optimized version :

definition *swap-opt* :: *nat* × *nat* ⇒ (*unit*,′*a global-state-state-scheme*) MON_{SE}

where *swap-opt* $\equiv \lambda(i,j).\ (tmp \leftarrow\ yield_C\ (\lambda\sigma.\ A\ \sigma\ !\ i)\ ;$
$((assign\text{-}global\ A\text{-}update\ (\lambda\sigma.\ list\text{-}update\ (A\ \sigma)\ (i)\ (A\ \sigma\ !\ j)))\ ;-$
$(assign\text{-}global\ A\text{-}update\ (\lambda\sigma.\ list\text{-}update\ (A\ \sigma)\ (j)\ (tmp)))))$

In case that all local variables are single-assigned in swap, the entire local var definition could be ommitted.

A more pretty-printed term representation is:

term‹ *swap-opt* $= (\lambda(i,\ j).$
$tmp \leftarrow (yield_C\ (\lambda\sigma.\ A\ \sigma\ !\ i));$
$(A\text{-}update :==_G\ (\lambda\sigma.\ (A\ \sigma)[i := A\ \sigma\ !\ j])\ ;-$
$A\text{-}update :==_G\ (\lambda\sigma.\ (A\ \sigma)[j := tmp]))))$›

A Simulation of Synthesis of Typed Assignment-Rules

definition tmp_L
where $tmp_L \equiv create_L\ local\text{-}swap'\text{-}state.tmp\ local\text{-}swap'\text{-}state.tmp\text{-}update$

lemma $tmp_L\text{-}control\text{-}indep : (break\text{-}status_L \bowtie tmp_L \wedge return\text{-}status_L \bowtie tmp_L)$
⟨*proof*⟩

lemma $tmp_L\text{-}strong\text{-}indep : \sharp!\ tmp_L$
⟨*proof*⟩

Specialized Assignment Rule for Local Variable *tmp*. Note that this specialized rule of $\sharp\ (?upd \circ upd\text{-}hd) \Longrightarrow \{\!|\lambda\sigma.\ \triangleright\ \sigma \wedge\ ?P\ ((?upd \circ upd\text{-}hd)\ (\lambda\text{-}.\ ?rhs\ \sigma)\ \sigma)|\!\}\ ?upd :==_L\ ?rhs\ \{\!|\lambda r\ \sigma.\ \triangleright\ \sigma \wedge\ ?P\ \sigma|\!\}$ does not need any further side-conditions referring to independence from the control. Consequently, backward inference in an *wp*-calculus will just maintain the invariant $\triangleright\ \sigma$.

lemma *assign-local-tmp*:
$\{\!|\lambda\sigma.\ \triangleright\ \sigma \wedge\ P\ ((tmp\text{-}update \circ upd\text{-}hd)\ (\lambda\text{-}.\ rhs\ \sigma)\ \sigma)|\!\}$
$local\text{-}swap'\text{-}state.tmp\text{-}update :==_L\ rhs$
$\{\!|\lambda r\ \sigma.\ \triangleright\ \sigma \wedge\ P\ \sigma\ |\!\}$
⟨*proof*⟩

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 (λσ. A σ ! hi ))   ;–
            (assign-local i-update (λσ. lo )) ;–

            (assign-local j-update (λσ. lo )) ;–
            (while_C (λσ. (hd o j) σ ≤ hi − 1 )
             do (if_C (λσ. A σ ! (hd o j) σ < (hd o pivot)σ )
                 then  call_C (swap) (λσ. ((hd o i) σ,  (hd o j) σ))  ;–
                       assign-local i-update (λσ. ((hd o i) σ) + 1)
                 else skip_SE
                 fi) ;–
                 (assign-local j-update (λσ. ((hd o j) σ) + 1))
             od) ;–
             call_C (swap) (λσ. ((hd o i) σ,  (hd o j) σ))  ;–
             assign-local result-value-update (λσ. (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*⟩

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

Contract-Elements

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*

Memory-Model

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-partition′-state-scheme*) MON_{SE}
where *push-local-partition-state′ σ* = *Some*((),
σ(|*local-partition-state.pivot* := *undefined* # *local-partition-state.pivot σ*,
local-partition-state.i := *undefined* # *local-partition-state.i σ*,
local-partition-state.j := *undefined* # *local-partition-state.j σ*,
local-partition-state.result-value
:= *undefined* # *local-partition-state.result-value σ* |))

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

Memory-Model

Independence of Control-Block:

Monadic Representation of the Body

definition *partition′-core* :: *nat* × *nat* ⇒ (*unit*,*′a local-partition′-state-scheme*) MON_{SE}
where *partition′-core* ≡ *λ*(*lo*,*hi*).
((*assign-local pivot-update* (*λσ*. *A σ* ! *hi*)) ;–
(*assign-local i-update* (*λσ*. *lo*)) ;–

(*assign-local j-update* (*λσ*. *lo*)) ;–
(*while*$_C$ (*λσ*. (*hd o j*) *σ* ≤ *hi* − *1*)
do (*if*$_C$ (*λσ*. *A σ* ! (*hd o j*) *σ* < (*hd o pivot*)*σ*)
then *call*$_C$ (*swap*) (*λσ*. ((*hd o i*) *σ*, (*hd o j*) *σ*)) ;–
assign-local i-update (*λσ*. ((*hd o i*) *σ*) + *1*)
else *skip*$_{SE}$
fi)
od) ;–
(*assign-local j-update* (*λσ*. ((*hd o j*) *σ*) + *1*)) ;–
call$_C$ (*swap*) (*λσ*. ((*hd o i*) *σ*, (*hd o j*) *σ*)) ;–
assign-local result-value-update (*λσ*. (*hd o i*) *σ*)

— the meaning of the return stmt
)

thm *partition-core-def*

definition *partition′* :: *nat* × *nat* ⇒ (*nat*,′*a local-partition′-state-scheme*) MON_{SE}
where *partition′* ≡ λ(*lo*,*hi*). $block_C$ *push-local-partition-state*
(*partition-core* (*lo*,*hi*))
pop-local-partition-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* ≤ *hi* ∧ *hi* < *length A*›
post ‹λ*res*::*unit*. ∀ *i*∈{*lo* .. *hi*}. ∀ *j*∈{*lo* .. *hi*}. *i* ≤ *j* ⟶ *A*!*i* ≤ *A*!*j*›
variant *hi* − *lo*
local-vars *p* :: *nat*
defines *if*$_C$ ‹*lo* < *hi*›
then (*p*$_{tmp}$ ← *call*$_C$ *partition* ‹(*lo*, *hi*)› ; *assign-local p-update* (λσ. *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*⟩

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-quicksort′-state.result-value :=
tl(local-quicksort′-state.result-value σ) |))

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
    ‹if_CLEAN ‹lo < hi› then
      p := partition(lo, hi) ;–
      quicksort(lo, p − 1) ;–
      quicksort(p + 1, hi)
     else Skip›
```

definition *quicksort′-pre* :: *nat × nat ⇒ ′a local-quicksort′-state-scheme ⇒ bool*
where *quicksort′-pre ≡ λ(i,j). λσ. True*

definition *quicksort′-post* :: *nat × nat ⇒ unit ⇒ ′a local-quicksort′-state-scheme ⇒ bool*
where *quicksort′-post ≡ λ(i,j). λ res. λσ. True*

definition *quicksort′-core* :: *(nat × nat ⇒ (unit,′a local-quicksort′-state-scheme) MON$_{SE}$)*
⇒ (nat × nat ⇒ (unit,′a local-quicksort′-state-scheme) MON$_{SE}$)
where *quicksort′-core quicksort-rec ≡ λ(lo, hi).*
((if$_C$ (λσ. lo < hi)
then (p$_{tmp}$ ← call$_C$ partition (λσ. (lo, hi)) ;
assign-local p-update (λσ. p$_{tmp}$)) ;–
call$_C$ quicksort-rec (λσ. (lo, (hd o p) σ − 1)) ;–
call$_C$ quicksort-rec (λσ. ((hd o p) σ + 1, hi))
else skip$_{SE}$
fi))

term *((quicksort′-core X) (lo,hi))*

definition *quicksort′* :: *((nat × nat) × (nat × nat)) set ⇒*
(nat × nat ⇒ (unit,′a local-quicksort′-state-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* :
$\{\!|\lambda\sigma.\quad \triangleright\ \sigma\ \wedge\ quicksort\text{-}pre\ (lo,\ hi)(\sigma)\ \wedge\ \sigma = \sigma_{pre}\ |\!\}$
$quicksort\ (lo,\ hi)$
$\{\!|\lambda r\ \sigma.\ \triangleright\ \sigma\ \wedge\ quicksort\text{-}post(lo,\ hi)(\sigma_{pre})(\sigma)(r)\ |\!\}$
$\langle proof \rangle$

end

3 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*
imports *Clean.Clean*
Clean.Hoare-Clean
Clean.Clean-Symbex
begin

3.1 The Quicksort Example - At a Glance

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)
```

3.2 Clean Encoding of the Global State of Quicksort

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

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 ‹ $tmp := A\ !\ i$› ;–
‹ $A := list\text{-}update\ A\ i\ (A\ !\ j)$› ;–
‹ $A := list\text{-}update\ A\ j\ tmp$›

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* ‹$(i,\ j)$› ;–
$return_{local\text{-}partition\text{-}state.result\text{-}value\text{-}update}$ ‹i›

thm *partition-core-def*

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*

3.3 Possible Application Sketch

lemma *quicksort-correct* :
$\{\!|\lambda\sigma. \quad \triangleright\ \sigma \wedge$ *quicksort-pre* $(lo, hi)(\sigma) \wedge \sigma = \sigma_{pre}\ |\!\}$
quicksort (lo, hi)
$\{\!|\lambda r\ \sigma. \triangleright\ \sigma \wedge$ *quicksort-post*$(lo, hi)(\sigma_{pre})(\sigma)(r)\ |\!\}$
⟨*proof*⟩

end

3.4 The Squareroot Example for Symbolic Execution

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

3.4.1 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 ‹$0 \leq a$›
post ‹λres::*int*. $(res + 1)^2 > a \wedge a \geq (res)^2$›
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*›
)
›

3.4.2 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*⟩

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

⟨*ML*⟩

lemma *tm-independent* [*simp*]: ♯ *tm-update*
  ⟨*proof*⟩

lemma *i-independent* [*simp*]: ♯ *i-update*
  ⟨*proof*⟩

lemma *sqsum-independent* [*simp*]: ♯ *sqsum-update*
  ⟨*proof*⟩

3.4.3 Setting for Symbolic Execution

Some lemmas to reason about memory

lemma *tm-simp* : *tm* (*σ*(|*tm* := *t*|)) = *t*
  ⟨*proof*⟩

lemma *tm-simp1* : *tm* (*σ*(|*sqsum* := *s*|)) = *tm σ* ⟨*proof*⟩
lemma *tm-simp2* : *tm* (*σ*(|*i* := *s*|)) = *tm σ* ⟨*proof*⟩
lemma *sqsum-simp* : *sqsum* (*σ*(|*sqsum* := *s*|)) = *s* ⟨*proof*⟩
lemma *sqsum-simp1* : *sqsum* (*σ*(|*tm* := *t*|)) = *sqsum σ* ⟨*proof*⟩
lemma *sqsum-simp2* : *sqsum* (*σ*(|*i* := *t*|)) = *sqsum σ* ⟨*proof*⟩
lemma *i-simp* : *i* (*σ*(|*i* := *i′*|)) = *i′* ⟨*proof*⟩
lemma *i-simp1* : *i* (*σ*(|*tm* := *i′*|)) = *i σ* ⟨*proof*⟩
lemma *i-simp2* : *i* (*σ*(|*sqsum* := *i′*|)) = *i σ* ⟨*proof*⟩

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 $\sigma \models upd :==_G rhs \;- M \Longrightarrow \triangleright\, \sigma \Longrightarrow upd\ (\lambda\text{-}.\ rhs\ \sigma)\ \sigma \models M$
⟨*proof*⟩

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.

3.4.4 A Symbolic Execution Simulation

lemma
assumes *non-exec-stop*[*simp*]: ¬ *exec-stop* σ_0
and *pos* : *0* ≤ (*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$_{SE}$(λσ. σ=σ_R)

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

⟨*proof*⟩

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

end

4 Clean Semantics : Another Clean Example

theory *IsPrime*
imports *Clean.Clean*
Clean.Hoare-Clean
Clean.Clean-Symbex
HOL−Computational-Algebra.Primes
begin

4.1 The Primality-Test Example at a Glance

definition *SQRT-UINT-MAX* = (*65536*::*nat*)
definition *UINT-MAX* = (*2^32*::*nat*) − *1*

function-spec *isPrime*(*n* :: *nat*) **returns** *bool*
pre ‹*n* ≤ *SQRT-UINT-MAX*›
post ‹λ*res*. *res* ⟷ *prime n* ›
local-vars *i* :: *nat*
defines *if*$_C$ ‹*n* < *2*›
then return$_{local\text{-}isPrime\text{-}state.result\text{-}value\text{-}update}$ ‹*False*›
else skip$_{SE}$
fi ;−
‹*i* := *2*› ;−
while$_C$ ‹*i* < *SQRT-UINT-MAX* ∧ *i*∗*i* ≤ *n* ›
do if$_C$ ‹*n mod i* = *0*›
then return$_{local\text{-}isPrime\text{-}state.result\text{-}value\text{-}update}$ ‹*False*›
else skip$_{SE}$
fi ;−
‹*i* := *i* + *1* ›
od ;−
return$_{local\text{-}isPrime\text{-}state.result\text{-}value\text{-}update}$ ‹*True*›

find-theorems *name*:*isPrime name*:*core*

lemma *XXX* :
isPrime-core n ≡
if$_C$ (λσ. *n* < *2*) *then* (*return*$_{result\text{-}value\text{-}update}$ (λσ. *False*))
else skip$_{SE}$ *fi*;−
i-update :==$_L$ (λσ. *2*) ;−
while$_C$ (λσ. (*hd*∘*i*)σ < *SQRT-UINT-MAX* ∧ (*hd*∘*i*)σ ∗ (*hd*∘*i*)σ ≤ *n*)
do

$(if_C\ (\lambda\sigma.\ n\ mod\ (hd \circ i)\ \sigma = 0)$
$then\ (return_{result\text{-}value\text{-}update}\ (\lambda\sigma.\ False))$
$else\ skip_{SE}\ fi\ ;-$
$i\text{-}update :==_L\ (\lambda\sigma.\ (hd \circ i)\ \sigma + 1))$
$od\ ;-$
$return_{result\text{-}value\text{-}update}\ (\lambda\sigma.\ True)$

⟨*proof*⟩

lemma *YYY*:
isPrime n ≡ $block_C$ *push-local-isPrime-state*
(*isPrime-core n*)
pop-local-isPrime-state
⟨*proof*⟩

lemma *isPrime-correct* :
$\{\!|\lambda\sigma.\quad \triangleright\ \sigma \wedge isPrime\text{-}pre\ (n)(\sigma) \wedge \sigma = \sigma_{pre}\ |\!\}$
isPrime n
$\{\!|\lambda r\ \sigma.\ \triangleright\ \sigma \wedge isPrime\text{-}post(n)\ (\sigma_{pre})(\sigma)(r)\ |\!\}$
⟨*proof*⟩

end

5 A Clean Semantics Example : Linear Search

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 *LinearSearch*
imports *Clean.Clean*
Clean.Hoare-MonadSE
begin

5.1 The LinearSearch Example

definition *bool2int* **where** *bool2int x* = (*if x then 1*::*int else 0*)

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

function-spec *linearsearch* (*x*::*int*, *n*::*int*) **returns** *int*
pre ‹ $0 \leq n \wedge n < int(length\ t) \wedge sorted\ t$ ›
post ‹$\lambda res{::}int.\ res = bool2int\ (\exists\ i \in \{0\ ..{<}\ length\ t\}.\ t!i = x)$ ›
local-vars *i* :: *int*
defines ‹*i* := *0* › ;–
while$_C$ ‹*i* < *n* ›
do if$_C$ ‹*t* ! (*nat i*) < *x*›
then ‹*i* := *i* + *1* ›
else return$_C$ *result-value-update* ‹*bool2int*(*t*!(*nat i*) = *x*)›
fi
od

end

6 Appendix : Used Monad Libraries

theory *MonadSE*
imports *Main*
begin

6.1 Definition : Standard State Exception Monads

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

6.1.1 Definition : Core Types and Operators

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

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

notation *bind-SE* $(bind_{SE})$

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

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

In the following, we prove the required Monad-laws

lemma *bind-right-unit*[*simp*]: $(x \leftarrow m;\ result\ x) = m$
⟨*proof*⟩

lemma *bind-left-unit* [*simp*]: $(x \leftarrow result\ c;\ P\ x) = P\ c$
⟨*proof*⟩

lemma *bind-assoc*[*simp*]: $(y \leftarrow (x \leftarrow m;\ k\ x);\ h\ y) = (x \leftarrow m;\ (y \leftarrow k\ x;\ h\ y))$
⟨*proof*⟩

6.1.2 Definition : More Operators and their Properties

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

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

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

lemma *bind-left-fail-SE*[*simp*] : $(x \leftarrow fail_{SE};\ P\ x) = fail_{SE}$
⟨*proof*⟩

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

definition *bind-SE′* :: $('α,\ 'σ)MON_{SE} \Rightarrow ('β,\ 'σ)MON_{SE} \Rightarrow ('β,\ 'σ)MON_{SE}$ (**infixr** ;− *10*)
where $(f\ ;-\ g) = (\text{-} \leftarrow f\ ;\ g)$

lemma *bind-assoc′*[*simp*]: $((m\ ;-\ k);-\ h\ ) = (m;-\ (k;-\ h))$
⟨*proof*⟩

lemma *bind-left-unit′* [*simp*]: $((result\ c);-\ P) = P$
⟨*proof*⟩

lemma *bind-left-fail-SE′*[*simp*]: $(fail_{SE};-\ P) = fail_{SE}$
⟨*proof*⟩

lemma *bind-right-unit′*[*simp*]: $(m;-\ (result\ ())) = m$
⟨*proof*⟩

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*: $(o1 \leftarrow f1\ ;\ o2 \leftarrow f2;\ result\ (post\ o1\ o2)) = X$
⟨*proof*⟩

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

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

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 $(f \sqcap_{SE} g) = (\lambda\ \sigma.\ case\ f\ \sigma\ of\ None \Rightarrow g\ \sigma$
$\mid Some\ H \Rightarrow Some\ H)$

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

lemma *malt-SE-mt* [*simp*]: $\bigsqcap_{SE}\ [] = fail_{SE}$
⟨*proof*⟩

lemma *malt-SE-cons* [*simp*]: $\bigsqcap_{SE}\ (a\ \#\ S) = (a \sqcap_{SE} (\bigsqcap_{SE}\ S))$
⟨*proof*⟩

6.1.3 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 $if\text{-}SE\ c\ E\ F = (\lambda\sigma.\ if\ c\ \sigma\ then\ E\ \sigma\ else\ F\ \sigma)$

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
$(if_{SE}\ cond\ then\ T1\ else\ T2\ fi) == CONST\ if\text{-}SE\ cond\ T1\ T2$

6.1.4 Theory of a Monadic While

Prerequisites

fun *replicator* :: $[('a, '\sigma)MON_{SE}, nat] \Rightarrow (unit, '\sigma)MON_{SE}$ (**infixr** $\frown\frown$ *60*)
where $f \frown\frown 0 = (result\ ())$
$\mid f \frown\frown (Suc\ n) = (f\ ;-\ f \frown\frown\ n)$

fun *replicator2* :: $[('a, '\sigma)MON_{SE}, nat, ('b, '\sigma)MON_{SE}] \Rightarrow ('b, '\sigma)MON_{SE}$ (**infixr** $\widehat{}:\widehat{}$ *60*)

where $(f \mathbin{\widehat{:}\widehat{}} 0)\ M = (M\ )$
$\mid (f \mathbin{\widehat{:}\widehat{}} (Suc\ n))\ M = (f\ ;\!-\ ((f \mathbin{\widehat{:}\widehat{}}\ n)\ M))$

First Step : Establishing an embedding between partial functions and relations

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

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

lemma *Mon2Rel-Rel2Mon-id*: **assumes** *det:single-valued R* **shows** $(Mon2Rel \circ Rel2Mon)\ R = R$
⟨*proof*⟩

lemma *Rel2Mon-Id*: $(Rel2Mon \circ Mon2Rel)\ x = x$
⟨*proof*⟩

lemma *single-valued-Mon2Rel*: *single-valued* $(Mon2Rel\ B)$
⟨*proof*⟩

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*: $chain\ S \Longrightarrow S\ i \leq S\ j \vee S\ j \leq S\ i$
⟨*proof*⟩

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*⟩

lemma *chain-iterates*: **fixes** $f :: 'a\ set \Rightarrow 'a\ set$
assumes *mono f* **shows** $chain(\lambda n.\ (f\widehat{\ }\widehat{\ }n)\ \{\})$
⟨*proof*⟩

theorem *lfp-if-cont*:
assumes *cont f* **shows** $lfp\ f = (\bigcup n.\ (f \mathbin{\widehat{\ }\widehat{\ }} n)\ \{\})$ (**is** - = *?U*)
⟨*proof*⟩

lemma *single-valued-UN-chain*:
assumes $chain\ S\ (!!n.\ single\text{-}valued\ (S\ n))$
shows $single\text{-}valued(UN\ n.\ S\ n)$
⟨*proof*⟩

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)
⟨*proof*⟩

Third Step: Definition of the Monadic While

definition $\Gamma :: ['\sigma \Rightarrow bool, ('\sigma \times '\sigma)\ set] \Rightarrow (('\sigma \times '\sigma)\ set \Rightarrow ('\sigma \times '\sigma)\ set)$
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*)
⟨*proof*⟩

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*)))
⟨*proof*⟩

lemma *Rel2Mon-if*:
Rel2Mon {(*s*, *t*). *if b s then* (*s*, *t*) ∈ *Mon2Rel c O lfp* (Γ *b* (*Mon2Rel c*)) *else s* = *t*} *σ* =
(*if b σ then Rel2Mon* (*Mon2Rel c O lfp* (Γ *b* (*Mon2Rel c*))) *σ else Some* ((), *σ*))
⟨*proof*⟩

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*⟩

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*:
$(while_{SE}\ b\ do\ c\ od) = (if_{SE}\ b\ then\ (c\ ;–\ (while_{SE}\ b\ do\ c\ od))\ else\ result\ ()\ fi)$
⟨*proof*⟩

lemma *bind-cong* : $f\ \sigma = g\ \sigma \Longrightarrow\ (x \leftarrow f\ ;\ M\ x)\sigma = (x \leftarrow g\ ;\ M\ x)\sigma$

⟨*proof*⟩

lemma *bind′-cong* : $f\ \sigma = g\ \sigma \Longrightarrow (f\ ;-\ M\ )\sigma = (g\ ;-\ M\ )\sigma$
⟨*proof*⟩

lemma if_{SE}*-True* [*simp*]: $(if_{SE}\ (\lambda\ x.\ True)\ then\ c\ else\ d\ fi) = c$
⟨*proof*⟩

lemma if_{SE}*-False* [*simp*]: $(if_{SE}\ (\lambda\ x.\ False)\ then\ c\ else\ d\ fi) = d$
⟨*proof*⟩

lemma if_{SE}*-cond-cong* : $f\ \sigma = g\ \sigma \Longrightarrow$
$(if_{SE}\ f\ then\ c\ else\ d\ fi)\ \sigma =$
$(if_{SE}\ g\ then\ c\ else\ d\ fi)\ \sigma$
⟨*proof*⟩

lemma $while_{SE}$*-skip*[*simp*] : $(while_{SE}\ (\lambda\ x.\ False)\ do\ c\ od) = skip_{SE}$
⟨*proof*⟩

end

theory *Seq-MonadSE*
imports *MonadSE*
begin

6.1.5 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, ..., op_n with the inputs ι_1, ι_2, ..., ι_n (outputs are treated analogously). Then we can encode for this interface the general input - type:

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

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 :: 'ι list ⇒ ('ι ⇒ ('o,'σ) MON_SE) ⇒ ('o list,'σ) MON_SE
where mbind [] iostep σ = Some([], σ)
  | mbind (a#S) iostep σ =
          (case iostep a σ of
              None   ⇒ Some([], σ)
            | Some (out, σ') ⇒ (case mbind S iostep σ' of
                                  None    ⇒ Some([out],σ')
                                | Some(outs,σ'') ⇒ Some(out#outs,σ'')))
```

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* [])
⟨*proof*⟩

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 σ ≠ *None*
⟨*proof*⟩

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' :: 'ι list ⇒ ('ι ⇒ ('o,'σ) MON_SE) ⇒ ('o list,'σ) MON_SE
where mbind' [] iostep σ = Some([], σ) |
     mbind' (a#S) iostep σ =
          (case iostep a σ of
              None   ⇒ None
            | Some (out, σ') ⇒ (case mbind' S iostep σ' of
                                  None    ⇒ None   — fail-strict
                                | Some(outs,σ'') ⇒ Some(out#outs,σ'')))
```

notation $mbind'$ ($mbind_{FailStop}$)

lemma *mbind'-unit* [*simp*]:
$mbind'\ []\ f = (result\ [])$
⟨*proof*⟩

lemma *mbind'-bind* [*simp*]:
$(x \leftarrow mbind'\ (a\#S)\ F;\ M\ x) = (a \leftarrow (F\ a);\ (x \leftarrow mbind'\ S\ F;\ M\ (a\ \#\ x)))$
⟨*proof*⟩

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

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*:
$(x \leftarrow mbind\ (a\#S)\ F;\ M\ x) =$
$(a' \leftarrow try_{SE}(F\ a);$
$if\ a' = None$
$then\ (M\ [])$
$else\ (x \leftarrow mbind\ S\ F;\ M\ (the\ a'\ \#\ x)))$
⟨*proof*⟩

end

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

6.1.6 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 *valid-SE* :: $'\sigma \Rightarrow (bool,'\sigma)\ MON_{SE} \Rightarrow bool$ (**infix** $\models$ *9*)
where $(\sigma \models m) = (m\ \sigma \neq None \wedge fst(the\ (m\ \sigma)))$

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*]: $(\sigma \models (result\ P)) = (P)$
⟨*proof*⟩

lemma *exec-unit-SE'* [*simp*]: $(\sigma_0 \models (\lambda\sigma.\ Some\ (f\ \sigma,\ \sigma))) = (f\ \sigma_0)$
⟨*proof*⟩

lemma *exec-fail-SE* [*simp*]: $(\sigma \models fail_{SE}) = False$
⟨*proof*⟩

lemma *exec-fail-SE'*[*simp*]: $\neg(\sigma_0 \models (\lambda\sigma.\ None))$
⟨*proof*⟩

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

lemma *exec-bind-SE-failure*:
$A\ \sigma = None \Longrightarrow \neg(\sigma \models ((s \leftarrow A\ ;\ M\ s)))$
⟨*proof*⟩

lemma *exec-bind-SE-failure2*:
$A\ \sigma = None \Longrightarrow \neg(\sigma \models ((A\ ;\!-\ M)))$
⟨*proof*⟩

lemma *exec-bind-SE-success*:
$A\ \sigma = Some(b,\sigma') \Longrightarrow (\sigma \models ((s \leftarrow A\ ;\ M\ s))) = (\sigma' \models (M\ b))$
⟨*proof*⟩

lemma *exec-bind-SE-success2*:
$A\ \sigma = Some(b,\sigma') \Longrightarrow (\sigma \models ((A\ ;\!-\ M))) = (\sigma' \models M)$
⟨*proof*⟩

lemma *exec-bind-SE-success'*:
$M\ \sigma = Some(f\ \sigma,\sigma) \Longrightarrow\ (\sigma \models M) = f\ \sigma$
⟨*proof*⟩

lemma *exec-bind-SE-success''*:
$\sigma \models ((s \leftarrow A\ ;\ M\ s)) \Longrightarrow\ \exists\ v\ \sigma'.\ the(A\ \sigma) = (v,\sigma') \wedge (\sigma' \models M\ v)$
⟨*proof*⟩

lemma *exec-bind-SE-success'''*:
$\sigma \models ((s \leftarrow A\ ;\ M\ s)) \Longrightarrow\ \exists\ a.\ (A\ \sigma) = Some\ a \wedge (snd\ a \models M\ (fst\ a))$
⟨*proof*⟩

lemma *exec-bind-SE-success''''* :
$\sigma \models ((s \leftarrow A\ ;\ M\ s)) \Longrightarrow\ \exists\ v\ \sigma'.\ A\ \sigma = Some(v,\sigma') \wedge (\sigma' \models M\ v)$
⟨*proof*⟩

lemma *valid-bind-cong* : $f\ \sigma = g\ \sigma \Longrightarrow (\sigma \models (x \leftarrow f\ ;\ M\ x)) = (\sigma \models (x \leftarrow g\ ;\ M\ x))$
⟨*proof*⟩

lemma *valid-bind'-cong* : $f\ \sigma = g\ \sigma \Longrightarrow (\sigma \models f\ ;-\ M) = (\sigma \models g\ ;-\ M)$
⟨*proof*⟩

Recall `mbind_unit` for the base case.

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

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

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

lemma *exec-mbindFSave-failure*:
$ioprog\ a\ \sigma = None \Longrightarrow$

$(\sigma \models (s \leftarrow mbind_{FailSave}\ (a\#S)\ ioprog\ ;\ M\ s)) =\ (\sigma \models (M\ []))$
⟨*proof*⟩

lemma *exec-mbindFStop-failure*:
ioprog a σ = *None* $\Longrightarrow$
$(\sigma \models (s \leftarrow mbind_{FailStop}\ (a\#S)\ ioprog\ ;\ M\ s)) =\ (False)$
⟨*proof*⟩

lemma *exec-mbindFPurge-failure*:
ioprog a σ = *None* $\Longrightarrow$
$(\sigma \models (s \leftarrow mbind_{FailPurge}\ (a\#S)\ ioprog\ ;\ M\ s)) =$
$(\sigma \models (s \leftarrow mbind_{FailPurge}\ (S)\ ioprog\ ;\ M\ s))$
⟨*proof*⟩

lemma *exec-mbindFSave-success* :
ioprog a σ = *Some*(b,σ') $\Longrightarrow$
$(\sigma\ \models (s \leftarrow mbind_{FailSave}\ (a\#S)\ ioprog\ ;\ M\ s)) =$
$(\sigma' \models (s \leftarrow mbind_{FailSave}\ S\ ioprog\ ;\ M\ (b\#s)))$
⟨*proof*⟩

lemma *exec-mbindFStop-success* :
ioprog a σ = *Some*(b,σ') $\Longrightarrow$
$(\sigma\ \models (s \leftarrow mbind_{FailStop}\ (a\#S)\ ioprog\ ;\ M\ s)) =$
$(\sigma' \models (s \leftarrow mbind_{FailStop}\ S\ ioprog\ ;\ M\ (b\#s)))$
⟨*proof*⟩

lemma *exec-mbindFPurge-success* :
ioprog a σ = *Some*(b,σ') $\Longrightarrow$
$(\sigma\ \models (s \leftarrow mbind_{FailPurge}\ (a\#S)\ ioprog\ ;\ M\ s)) =$
$(\sigma' \models (s \leftarrow mbind_{FailPurge}\ S\ ioprog\ ;\ M\ (b\#s)))$
⟨*proof*⟩

lemma *exec-mbindFSave*:
$(\sigma \models (s \leftarrow mbind_{FailSave}\ \ (a\#S)\ ioprog\ ;\ return\ (P\ s))) =$
(case ioprog a σ *of*
None $\Rightarrow$ $(\sigma\ \models (return\ (P\ [])))$
| *Some*(b,σ') $\Rightarrow$ $(\sigma'\ \models (s \leftarrow mbind_{FailSave}\ \ S\ ioprog\ ;\ return\ (P\ (b\#s)))))$
⟨*proof*⟩

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 $(\ a \leftarrow f\ a;\ \ x \leftarrow mbind_{FailStop}\ S\ f;\ P\ (a\ \#\ x))\ \sigma =$
$(\ a \leftarrow f\ a;\ \ x \leftarrow mbind_{FailStop}\ S\ f;\ P'(a\ \#\ x))\ \sigma$
⟨*proof*⟩

lemma *mbind-eq-sexec'*:

assumes $* : \bigwedge b\ \sigma'.\ f\ a\ \sigma = Some(b,\sigma') \Longrightarrow$
$(P\ (b))\sigma' = (P'\ (b))\sigma'$
shows $(\ a \leftarrow f\ a;\ \ P\ (a))\ \sigma =$
$(\ a \leftarrow f\ a;\ \ P'(a))\ \sigma$
⟨*proof*⟩

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*⟩

lemma *assert-suffix-inv* :
$\sigma \models (\ \text{-} \leftarrow mbind_{FailStop}\ xs\ istep;\ assert_{SE}\ (P))$
$\Longrightarrow \forall \sigma.\ P\ \sigma \longrightarrow (\sigma \models (\text{-} \leftarrow istep\ x;\ assert_{SE}\ (P)))$
$\Longrightarrow \sigma \models (\ \text{-} \leftarrow mbind_{FailStop}\ (xs\ @\ [x])\ istep;\ assert_{SE}\ (P))$
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

lemma *assert-disch1* : $P\ \sigma \Longrightarrow (\sigma \models (x \leftarrow assert_{SE}\ P;\ M\ x)) = (\sigma \models (M\ True))$
⟨*proof*⟩

lemma *assert-disch2* : $\neg\ P\ \sigma \Longrightarrow \neg\ (\sigma \models (x \leftarrow assert_{SE}\ P\ ;\ M\ s))$
⟨*proof*⟩

lemma *assert-disch3* : $\neg\ P\ \sigma \Longrightarrow \neg\ (\sigma \models (assert_{SE}\ P))$
⟨*proof*⟩

lemma *assert-disch4* : $P\ \sigma \Longrightarrow\ (\sigma \models (assert_{SE}\ P))$
⟨*proof*⟩

lemma *assert-simp* : $(\sigma \models assert_{SE}\ P) = P\ \sigma$
⟨*proof*⟩

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

lemma *assert-bind-simp* : $(\sigma \models (x \leftarrow assert_{SE}\ P;\ M\ x)) = (P\ \sigma \wedge (\sigma \models (M\ True)))$
⟨*proof*⟩

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

lemma *assume-D* : $(\sigma \models (\text{-} \leftarrow assume_{SE}\ P;\ M)) \Longrightarrow \exists\ \sigma.\ (P\ \sigma \wedge (\sigma \models M)\ )$
⟨*proof*⟩

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
⟨*proof*⟩

lemma *assume-E′* :
assumes $*\ :\ \sigma \models assume_{SE}\ P\ ;\!-\ M$
and $**\ :\ \bigwedge \sigma.\ P\ \sigma \Longrightarrow \sigma \models M\ \Longrightarrow Q$
shows Q
⟨*proof*⟩

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* : $P\ \sigma \Longrightarrow (\sigma \models (if_{SE}\ P\ then\ B_1\ else\ B_2\ fi)) = (\sigma \models B_1)$
⟨*proof*⟩

lemma *if-SE-D1′* : $P\ \sigma \Longrightarrow (\sigma \models (if_{SE}\ P\ then\ B_1\ else\ B_2\ fi);\!-M) = (\sigma \models (B_1;\!-M))$
⟨*proof*⟩

lemma *if-SE-D2* : $\neg\ P\ \sigma \Longrightarrow (\sigma \models (if_{SE}\ P\ then\ B_1\ else\ B_2\ fi)) = (\sigma \models B_2)$
⟨*proof*⟩

lemma *if-SE-D2′* : $\neg\ P\ \sigma \Longrightarrow (\sigma \models (if_{SE}\ P\ then\ B_1\ else\ B_2\ fi);-M) = (\sigma \models B_2;-M)$
⟨*proof*⟩

lemma *if-SE-split-asm* :
$(\sigma \models (if_{SE}\ P\ then\ B_1\ else\ B_2\ fi)) = ((P\ \sigma \wedge (\sigma \models B_1)) \vee (\neg\ P\ \sigma \wedge (\sigma \models B_2)))$
⟨*proof*⟩

lemma *if-SE-split-asm′*:
$(\sigma \models (if_{SE}\ P\ then\ B_1\ else\ B_2\ fi);-M) = ((P\ \sigma \wedge (\sigma \models B_1;-M)) \vee (\neg\ P\ \sigma \wedge (\sigma \models B_2;-M)))$
⟨*proof*⟩

lemma *if-SE-split*:
$(\sigma \models (if_{SE}\ P\ then\ B_1\ else\ B_2\ fi)) = ((P\ \sigma \longrightarrow (\sigma \models B_1)) \wedge (\neg\ P\ \sigma \longrightarrow (\sigma \models B_2)))$
⟨*proof*⟩

lemma *if-SE-split′*:
$(\sigma \models (if_{SE}\ P\ then\ B_1\ else\ B_2\ fi);-M) = ((P\ \sigma \longrightarrow (\sigma \models B_1;-M)) \wedge (\neg\ P\ \sigma \longrightarrow (\sigma \models B_2;-M)))$
⟨*proof*⟩

lemma *if-SE-execE*:
assumes *A*: $\sigma \models ((if_{SE}\ P\ then\ B_1\ else\ B_2\ fi))$
and *B*: $P\ \sigma \Longrightarrow \sigma \models (B_1) \Longrightarrow Q$
and *C*: $\neg\ P\ \sigma \Longrightarrow \sigma \models (B_2) \Longrightarrow Q$
shows Q
⟨*proof*⟩

lemma *if-SE-execE′*:
assumes *A*: $\sigma \models ((if_{SE}\ P\ then\ B_1\ else\ B_2\ fi);-M)$
and *B*: $P\ \sigma \Longrightarrow \sigma \models (B_1;-M) \Longrightarrow Q$
and *C*: $\neg\ P\ \sigma \Longrightarrow \sigma \models (B_2;-M) \Longrightarrow Q$
shows Q
⟨*proof*⟩

lemma *exec-while* :
$(\sigma \models ((while_{SE}\ b\ do\ c\ od)\ ;-\ M)) =$
$(\sigma \models ((if_{SE}\ b\ then\ c\ ;-\ (while_{SE}\ b\ do\ c\ od)\ else\ unit_{SE}\ ()fi)\ ;-\ M))$
⟨*proof*⟩

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

lemma *if-SE-execE''*:
$\sigma \models (if_{SE}\ P\ then\ B_1\ else\ B_2\ fi)\ ;-\ M$
$\Longrightarrow (P\ \sigma \Longrightarrow \sigma \models B_1\ ;-\ M \Longrightarrow Q)$
$\Longrightarrow (\neg\ P\ \sigma \Longrightarrow \sigma \models B_2\ ;-\ M \Longrightarrow Q)$
$\Longrightarrow Q$
⟨*proof*⟩

definition *opaque* $(x{::}bool) = x$
lemma *if-SE-execE''-pos*:
$\sigma \models (if_{SE}\ P\ then\ B_1\ else\ B_2\ fi)\ ;-\ M$
$\Longrightarrow (P\ \sigma \Longrightarrow \sigma \models B_1\ ;-\ M \Longrightarrow Q)$
$\Longrightarrow (opaque\ (\sigma \models (if_{SE}\ P\ then\ B_1\ else\ B_2\ fi)\ ;-\ M) \Longrightarrow Q)$
$\Longrightarrow Q$
⟨*proof*⟩

lemma [*code*]:
$(\sigma \models m) = (case\ (m\ \sigma)\ of\ None\ \Rightarrow False \mid (Some\ (x,y))\ \Rightarrow x)$
⟨*proof*⟩

lemma $P\ \ \sigma \models (\text{-}\ \leftarrow assume_{SE}\ P\ ;\ x\ \leftarrow M;\ assert_{SE}\ (\lambda\sigma.\ \ (x{=}X) \wedge Q\ x\ \sigma))$
⟨*proof*⟩

lemma $\forall\sigma.\ \exists\ X.\ \sigma \models (\text{-}\ \leftarrow assume_{SE}\ P\ ;\ x\ \leftarrow M;\ assert_{SE}\ (\lambda\sigma.\ \ x{=}X \wedge Q\ x\ \sigma))$
⟨*proof*⟩

lemma *monadic-sequence-rule*:
$\bigwedge X\ \sigma_1.\ (\sigma \models (\text{-}\ \leftarrow assume_{SE}\ (\lambda\sigma'.\ (\sigma{=}\sigma') \wedge\ \ P\ \sigma)\ ;\ x\ \leftarrow M;\ assert_{SE}\ (\lambda\sigma.\ \ (x{=}X) \wedge (\sigma{=}\sigma_1) \wedge Q\ x\ \sigma)))$
$\wedge$
$(\sigma_1 \models (\text{-}\ \leftarrow assume_{SE}\ (\lambda\sigma.\ \ (\sigma{=}\sigma_1) \wedge Q\ x\ \sigma)\ ;\ y\ \leftarrow M';\ assert_{SE}\ (\lambda\sigma.\ \ R\ x\ y\ \sigma)))$
$\Longrightarrow$
$\sigma \models (\text{-}\ \leftarrow assume_{SE}\ (\lambda\sigma'.\ (\sigma{=}\sigma') \wedge\ \ P\ \sigma)\ ;\ x\ \leftarrow M;\ y\ \leftarrow M';\ assert_{SE}\ (R\ x\ y))$
⟨*proof*⟩

lemma $\exists\ X.\ \sigma \models (\text{-}\ \leftarrow assume_{SE}\ P\ ;\ x\ \leftarrow M;\ assert_{SE}\ (\lambda\sigma.\ \ x{=}X \wedge Q\ x\ \sigma))$
$\Longrightarrow$
$\sigma \models (\text{-}\ \leftarrow assume_{SE}\ P\ ;\ x\ \leftarrow M;\ assert_{SE}\ (\lambda\sigma.\ Q\ x\ \sigma))$
⟨*proof*⟩

lemma *exec-skip*:
$(\sigma \models skip_{SE}\ ;-\ M) = (\sigma \models M)$

⟨*proof*⟩

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* :
$(\sigma \models (os \leftarrow (mbind_{FailSave}\ \iota s\ ioprog);\ result(length\ \iota s = length\ os \wedge P\ \iota s\ os))) =$
$(\sigma \models (os \leftarrow (mbind_{FailStop}\ \iota s\ ioprog);\ result(P\ \iota s\ os)))$
⟨*proof*⟩

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*⟩

6.1.7 Miscellaneous

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

end

theory *Clean-Symbex*
imports *Clean*
begin

6.2 Clean Symbolic Execution Rules

6.2.1 Basic NOP - Symbolic Execution Rules.

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

lemma *non-exec-assign* :
assumes $\triangleright\ \sigma$
shows $(\sigma \models (\ \text{-} \leftarrow assign\ f;\ M)) = ((f\ \sigma) \models\ M)$
⟨*proof*⟩

lemma *non-exec-assign′* :
assumes $\triangleright\ \sigma$
shows $(\sigma \models (assign\ f;\!-\ M)) = ((f\ \sigma) \models\ M)$
⟨*proof*⟩

lemma *exec-assign* :
assumes *exec-stop* σ
shows $(\sigma \models (\ \text{-} \leftarrow \mathit{assign}\ f;\ M)) = (\sigma \models M)$
⟨*proof*⟩

lemma *exec-assign′* :
assumes *exec-stop* σ
shows $(\sigma \models (\mathit{assign}\ f;\!-\ M)) = (\sigma \models M)$
⟨*proof*⟩

6.2.2 Assign Execution Rules.

lemma *non-exec-assign-global* :
assumes $\triangleright\ \sigma$
shows $(\sigma \models (\ \text{-} \leftarrow \mathit{assign\text{-}global}\ \mathit{upd}\ \mathit{rhs};\ M)) = ((\mathit{upd}\ (\lambda\text{-}.\ \mathit{rhs}\ \sigma)\ \sigma) \models\ M)$
⟨*proof*⟩

lemma *non-exec-assign-global′* :
assumes $\triangleright\ \sigma$
shows $(\sigma \models (\mathit{assign\text{-}global}\ \mathit{upd}\ \mathit{rhs};\!-\ M)) = ((\mathit{upd}\ (\lambda\text{-}.\ \mathit{rhs}\ \sigma)\ \sigma) \models\ M)$
⟨*proof*⟩

lemma *exec-assign-global* :
assumes *exec-stop* σ
shows $(\sigma \models (\ \text{-} \leftarrow \mathit{assign\text{-}global}\ \mathit{upd}\ \mathit{rhs};\ M)) = (\ \sigma \models\ M)$
⟨*proof*⟩

lemma *exec-assign-global′* :
assumes *exec-stop* σ
shows $(\sigma \models (\mathit{assign\text{-}global}\ \mathit{upd}\ \mathit{rhs};\!-\ M)) = (\ \sigma \models\ M)$
⟨*proof*⟩

lemma *non-exec-assign-local* :
assumes $\triangleright\ \sigma$
shows $(\sigma \models (\ \text{-} \leftarrow \mathit{assign\text{-}local}\ \mathit{upd}\ \mathit{rhs};\ M)) = ((\mathit{upd}\ (\mathit{upd\text{-}hd}\ (\lambda\text{-}.\ \mathit{rhs}\ \sigma))\ \sigma) \models\ M)$
⟨*proof*⟩

lemma *non-exec-assign-local′* :
assumes $\triangleright\ \sigma$
shows $(\sigma \models (\mathit{assign\text{-}local}\ \mathit{upd}\ \mathit{rhs};\!-\ M)) = ((\mathit{upd}\ (\mathit{upd\text{-}hd}\ (\lambda\text{-}.\ \mathit{rhs}\ \sigma))\ \sigma) \models\ M)$
⟨*proof*⟩

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

lemma *exec-assign-local* :
assumes *exec-stop* σ
shows $(\sigma \models (\ \text{-} \leftarrow \mathit{assign\text{-}local}\ \mathit{upd}\ \mathit{rhs};\ M)) = (\ \sigma \models\ M)$
⟨*proof*⟩

lemma *exec-assign-local′* :
assumes *exec-stop* σ
shows $(\sigma \models (\ \textit{assign-local upd rhs};\!-\ M)) = (\ \sigma \models\ M)$
⟨*proof*⟩

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*]

6.2.3 Basic Call Symbolic Execution Rules.

lemma *exec-call-0* :
assumes *exec-stop* σ
shows $(\sigma \models (\ \text{-} \leftarrow \textit{call-0}_C\ M;\ M')) = (\sigma \models\ M')$
⟨*proof*⟩

lemma *exec-call-0′* :
assumes *exec-stop* σ
shows $(\sigma \models (\textit{call-0}_C\ M;\!-\ M')) = (\sigma \models\ M')$
⟨*proof*⟩

lemma *exec-call-1* :
assumes *exec-stop* σ
shows $(\sigma \models (\ x \leftarrow \textit{call-1}_C\ M\ A_1;\ M'\ x)) = (\sigma \models\ M'\ \textit{undefined})$
⟨*proof*⟩

lemma *exec-call-1′* :
assumes *exec-stop* σ
shows $(\sigma \models (\textit{call-1}_C\ M\ A_1;\!-\ M')) = (\sigma \models\ M')$
⟨*proof*⟩

lemma *exec-call* :
assumes *exec-stop* σ
shows $(\sigma \models (\ x \leftarrow \textit{call}_C\ M\ A_1;\ M'\ x)) = (\sigma \models\ M'\ \textit{undefined})$
⟨*proof*⟩

lemma *exec-call′* :
assumes *exec-stop* σ
shows $(\sigma \models (call_C\ M\ A_1;-\ M')) = (\sigma \models\ M')$
⟨*proof*⟩

lemma *exec-call-2* :
assumes *exec-stop* σ
shows $(\sigma \models (\ \text{-} \leftarrow call\text{-}2_C\ M\ A_1\ A_2;\ M')) = (\sigma \models\ M')$
⟨*proof*⟩

lemma *exec-call-2′* :
assumes *exec-stop* σ
shows $(\sigma \models (call\text{-}2_C\ M\ A_1\ A_2;-\ M')) = (\sigma \models M')$
⟨*proof*⟩

6.2.4 Basic Call Symbolic Execution Rules.

lemma *non-exec-call-0* :
assumes $\triangleright\ \sigma$
shows $(\sigma \models (\ \text{-} \leftarrow call\text{-}0_C\ M;\ M')) = (\sigma \models M;-\ M')$
⟨*proof*⟩

lemma *non-exec-call-0′* :
assumes $\triangleright\ \sigma$
shows $(\sigma \models call\text{-}0_C\ M;-\ M') = (\sigma \models M;-\ M')$
⟨*proof*⟩

lemma *non-exec-call-1* :
assumes $\triangleright\ \sigma$
shows $(\sigma \models (\ x \leftarrow (call\text{-}1_C\ M\ (A_1));\ M'\ x)) = (\sigma \models (x \leftarrow M\ (A_1\ \sigma);\ M'\ x))$
⟨*proof*⟩

lemma *non-exec-call-1′* :
assumes $\triangleright\ \sigma$
shows $(\sigma \models call\text{-}1_C\ M\ (A_1);-\ M') = (\sigma \models\ M\ (A_1\ \sigma);-\ M')$
⟨*proof*⟩

lemma *non-exec-call* :
assumes $\triangleright\ \sigma$
shows $(\sigma \models (\ x \leftarrow (call_C\ M\ (A_1));\ M'\ x)) = (\sigma \models (x \leftarrow M\ (A_1\ \sigma);\ M'\ x))$
⟨*proof*⟩

lemma *non-exec-call′* :
assumes $\triangleright\ \sigma$
shows $(\sigma \models call_C\ M\ (A_1);-\ M') = (\sigma \models\ M\ (A_1\ \sigma);-\ M')$
⟨*proof*⟩

lemma *non-exec-call-2* :
assumes $\triangleright\ \sigma$
shows $(\sigma \models (\ \text{-} \leftarrow (\textit{call-2}_C\ M\ (A_1)\ (A_2));\ M')) = (\sigma \models M\ (A_1\ \sigma)\ (A_2\ \sigma);-\ M')$
⟨*proof*⟩

lemma *non-exec-call-2'* :
assumes $\triangleright\ \sigma$
shows $(\sigma \models \textit{call-2}_C\ M\ (A_1)\ (A_2);-\ M') = (\sigma \models\ M\ (A_1\ \sigma)\ (A_2\ \sigma);-\ M')$
⟨*proof*⟩

6.2.5 Conditional.

lemma *exec-If$_C$-If$_{SE}$* :
assumes $\triangleright\ \sigma$
shows $((\textit{if}_C\ P\ \textit{then}\ B_1\ \textit{else}\ B_2\ \textit{fi}))\sigma = ((\textit{if}_{SE}\ P\ \textit{then}\ B_1\ \textit{else}\ B_2\ \textit{fi}))\ \sigma$
⟨*proof*⟩

lemma *valid-exec-If$_C$* :
assumes $\triangleright\ \sigma$
shows $(\sigma \models (\textit{if}_C\ P\ \textit{then}\ B_1\ \textit{else}\ B_2\ \textit{fi});-M) = (\sigma \models (\textit{if}_{SE}\ P\ \textit{then}\ B_1\ \textit{else}\ B_2\ \textit{fi});-M)$
⟨*proof*⟩

lemma *exec-If$_C$'* :
assumes *exec-stop* σ
shows $(\sigma \models (\textit{if}_C\ P\ \textit{then}\ B_1\ \textit{else}\ B_2\ \textit{fi});-M) = (\sigma \models M)$
⟨*proof*⟩

lemma *exec-While$_C$'* :
assumes *exec-stop* σ
shows $(\sigma \models (\textit{while}_C\ P\ \textit{do}\ B_1\ \textit{od});-M) = (\sigma \models M)$
⟨*proof*⟩

lemma *if$_C$-cond-cong* : $f\ \sigma = g\ \sigma \Longrightarrow (\textit{if}_C\ f\ \textit{then}\ c\ \textit{else}\ d\ \textit{fi})\ \sigma = (\textit{if}_C\ g\ \textit{then}\ c\ \textit{else}\ d\ \textit{fi})\ \sigma$
⟨*proof*⟩

6.2.6 Break - Rules.

lemma *break-assign-skip* [*simp*]: $(\textit{break}\ ;-\ \textit{assign}\ f) = \textit{break}$
⟨*proof*⟩

lemma *break-if-skip* [*simp*]: $(break\ ;-\ if_C\ b\ then\ c\ else\ d\ fi) = break$
⟨*proof*⟩

lemma *break-while-skip* [*simp*]: $(break\ ;-\ while_C\ b\ do\ c\ od) = break$
⟨*proof*⟩

lemma *unset-break-idem* [*simp*] :
$(unset\text{-}break\text{-}status\ ;-\ unset\text{-}break\text{-}status\ ;-\ M) = (unset\text{-}break\text{-}status\ ;-\ M)$
⟨*proof*⟩

lemma *return-cancel1-idem* [*simp*] :
$(return_X(E)\ ;-\ X :==_G E'\ ;-\ M) = (\ return_C\ X\ E\ ;-\ M)$
⟨*proof*⟩

lemma *return-cancel2-idem* [*simp*] :
$(\ return_X(E)\ ;-\ X :==_L E'\ ;-\ M) = (\ return_C\ X\ E\ ;-\ M)$
⟨*proof*⟩

6.2.7 While.

lemma $while_C$*-skip* [*simp*]: $(while_C\ (\lambda\ x.\ False)\ do\ c\ od) = skip_{SE}$
⟨*proof*⟩

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 $exec\text{-}while_C$:
$(\sigma \models ((while_C\ b\ do\ c\ od)\ ;-\ M)) =$
$(\sigma \models ((if_C\ b\ then\ c\ ;-\ ((while_C\ b\ do\ c\ od)\ ;-\ unset\text{-}break\text{-}status)\ else\ skip_{SE}\ fi)\ \ ;-\ M))$
⟨*proof*⟩

lemma *while-k-SE* : *while-C* = *while-k k*
⟨*proof*⟩

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

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

⟨*ML*⟩

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

end

theory *Test-Clean*
imports *Clean-Symbex*
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

6.3 Hoare

definition $hoare_3$ :: (′σ ⇒ *bool*) ⇒ (′α, ′σ)MON_{SE} ⇒ (′α ⇒ ′σ ⇒ *bool*) ⇒ *bool* (({⦇(*1-*)⦈}/ (-)/ {⦇(*1-*)⦈}) *50*)
where {⦇*P*⦈} *M* {⦇*Q*⦈} ≡ (∀ σ. *P* σ ⟶ (*case M* σ *of None* => *False* | *Some*(*x*, σ′) => *Q x* σ′))

definition $hoare_3'$ :: (′σ ⇒ *bool*) ⇒ (′α, ′σ)MON_{SE} ⇒ *bool* (({⦇(*1-*)⦈}/ (-)/†) *50*)
where {⦇*P*⦈} *M* † ≡ (∀ σ. *P* σ ⟶ (*case M* σ *of None* => *True* | - => *False*))

6.3.1 Basic rules

lemma *skip*: {⦇*P*⦈} $skip_{SE}$ {⦇λ-. *P*⦈}
⟨*proof*⟩

lemma *fail*: {⦇*P*⦈} $fail_{SE}$ †
⟨*proof*⟩

lemma *assert*: {⦇*P*⦈} $assert_{SE}$ *P* {⦇λ - -. *True*⦈}
⟨*proof*⟩

lemma *assert-conseq*: $Collect\ P \subseteq Collect\ Q \Longrightarrow \{\!|P|\!\}\ assert_{SE}\ Q\ \{\!|\lambda\ \text{-}\ \text{-}.\ True|\!\}$
⟨*proof*⟩

lemma *assume-conseq*:
assumes $\exists\ \sigma.\ Q\ \sigma$
shows $\{\!|P|\!\}\ assume_{SE}\ Q\ \{\!|\lambda\ \text{-}\ .\ Q|\!\}$
⟨*proof*⟩

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

6.3.2 Generalized and special sequence rules

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

lemma *sequence* :
$\{\!|P|\!\}\ M\ \{\!|\lambda x\ \sigma.\ x{\in}A \wedge Q\ x\ \sigma|\!\}$
$\Longrightarrow \forall\, x{\in}A.\ \{\!|Q\ x|\!\}\ M'\ x\ \{\!|R|\!\}$
$\Longrightarrow \{\!|P|\!\}\ x \leftarrow M;\ M'\ x\ \{\!|R|\!\}$
⟨*proof*⟩

lemma *sequence-irpt-l* : $\{\!|P|\!\}\ M\ \dagger\ \Longrightarrow \{\!|P|\!\}\ x \leftarrow M;\ M'\ x\ \dagger$
⟨*proof*⟩

lemma *sequence-irpt-r* : $\{\!|P|\!\}\ M\ \{\!|\lambda x\ \sigma.\ x{\in}A \wedge Q\ x\ \sigma|\!\} \Longrightarrow \forall\, x{\in}A.\ \{\!|Q\ x|\!\}\ M'\ x\ \dagger\ \Longrightarrow \{\!|P|\!\}\ x \leftarrow M;\ M'\ x\ \dagger$
⟨*proof*⟩

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

lemma *sequence-irpt-l'* : $\{\!|P|\!\}\ M\ \dagger \Longrightarrow \{\!|P|\!\}\ M;\!-\ M'\ \dagger$
⟨*proof*⟩

lemma *sequence-irpt-r'* : $\{\!|P|\!\}\ M\ \{\!|\lambda\text{-}.\ Q\ |\!\} \Longrightarrow \{\!|Q|\!\}\ M'\ \dagger \Longrightarrow \{\!|P|\!\}\ M;\!-\ M'\ \dagger$
⟨*proof*⟩

6.3.3 Generalized and special consequence rules

lemma *consequence* :
$Collect\ P \subseteq Collect\ P'$
$\Longrightarrow \{\!|P'|\!\}\ M\ \{\!|\lambda x\ \sigma.\ x{\in}A \wedge Q'\ x\ \sigma|\!\}$
$\Longrightarrow \forall\ x{\in}A.\ Collect(Q'\ x) \subseteq Collect\ (Q\ x)$
$\Longrightarrow \{\!|P|\!\}\ M\ \{\!|\lambda x\ \sigma.\ x{\in}A \wedge Q\ x\ \sigma|\!\}$
⟨*proof*⟩

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*⟩

lemma *consequence-irpt* :
$\quad Collect\ P \subseteq Collect\ P'$
$\Longrightarrow \{\!|P'|\!\}\ M\ \dagger$
$\Longrightarrow \{\!|P|\!\}\ \ M\ \dagger$
⟨*proof*⟩

lemma *consequence-mt-swap* :
$(\{\!|\lambda\text{-}.\ False|\!\}\ M\ \dagger) = (\{\!|\lambda\text{-}.\ False|\!\}\ M\ \{\!|P|\!\})$
⟨*proof*⟩

6.3.4 Condition rules

lemma *cond* :
$\quad \{\!|\lambda\sigma.\ P\ \sigma \wedge cond\ \sigma|\!\}\ M\ \{\!|Q|\!\}$
$\Longrightarrow \{\!|\lambda\sigma.\ P\ \sigma \wedge \neg\ cond\ \sigma|\!\}\ M'\ \{\!|Q|\!\}$
$\Longrightarrow \{\!|P|\!\} if_{SE}\ cond\ then\ M\ else\ M'\ fi \{\!|Q|\!\}$
⟨*proof*⟩

lemma *cond-irpt* :
$\quad \{\!|\lambda\sigma.\ P\ \sigma \wedge cond\ \sigma|\!\}\ M\ \dagger$
$\Longrightarrow \{\!|\lambda\sigma.\ P\ \sigma \wedge \neg\ cond\ \sigma|\!\}\ M'\ \dagger$
$\Longrightarrow \{\!|P|\!\} if_{SE}\ cond\ then\ M\ else\ M'\ fi\ \dagger$
⟨*proof*⟩

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

6.3.5 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 $\{\!|P|\!\} while_{SE}\ cond\ do\ M\ od\ \{\!|\lambda\text{-}\ \sigma.\ \neg cond\ \sigma \wedge P\ \sigma|\!\}$

⟨*proof*⟩

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 $\{\!|P|\!\}\, while_{SE}\ cond\ do\ M\ od\ \dagger$
⟨*proof*⟩

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

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

definition $hoare_2 :: ('\sigma \Rightarrow bool) \Rightarrow ('\alpha, '\sigma)MON_{SE} \Rightarrow ('\alpha \Rightarrow '\sigma \Rightarrow bool) \Rightarrow bool\ (\vdash_2 (\{(1\text{-})\}/ (\text{-})/ \{(1\text{-})\})\ 50)$
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

6.3.7 Clean Control Rules

lemma *break1*:
$\{\!|\lambda\sigma.\ \ P\ (\sigma\ (\!|\ break\text{-}status := True\ |\!))\ |\!\}\ break\ \{\!|\lambda r\ \sigma.\ \ P\ \sigma \wedge break\text{-}status\ \sigma\ |\!\}$
⟨*proof*⟩

lemma *unset-break1*:
$\{\!|\lambda\sigma.\ \ P\ (\sigma\ (\!|\ break\text{-}status := False\ |\!))\ |\!\}\ unset\text{-}break\text{-}status\ \{\!|\lambda r\ \sigma.\ P\ \sigma \wedge \neg\ break\text{-}status\ \sigma\ |\!\}$
⟨*proof*⟩

lemma *set-return1*:
$\{\!|\lambda\sigma.\ \ P\ (\sigma\ (\!|\ return\text{-}status := True\ |\!))\ |\!\}\ set\text{-}return\text{-}status\ \{\!|\lambda r\ \sigma.\ P\ \sigma \wedge return\text{-}status\ \sigma\ |\!\}$
⟨*proof*⟩

lemma *unset-return1*:
$\{\!|\lambda\sigma.\ \ P\ (\sigma\ (\!|\ return\text{-}status := False\ |\!))\ |\!\}\ unset\text{-}return\text{-}status\ \{\!|\lambda r\ \sigma.\ P\ \sigma \wedge \neg return\text{-}status\ \sigma\ |\!\}$
⟨*proof*⟩

6.3.8 Clean Skip Rules

lemma *assign-global-skip*:
$\{\!|\lambda\sigma.\quad exec\text{-}stop\ \sigma \wedge P\ \sigma\ |\!\}\ \ upd :==_G rhs\ \ \{\!|\lambda r\ \sigma.\ exec\text{-}stop\ \sigma \wedge P\ \sigma\ |\!\}$
⟨*proof*⟩

lemma *assign-local-skip*:
$\{\!|\lambda\sigma.\quad exec\text{-}stop\ \sigma \wedge P\ \sigma\ |\!\}\ upd :==_L\ rhs\ \ \{\!|\lambda r\ \sigma.\ exec\text{-}stop\ \sigma \wedge P\ \sigma\ |\!\}$
⟨*proof*⟩

lemma *return-skip*:
$\{\!|\lambda\sigma.\ \ exec\text{-}stop\ \sigma \wedge P\ \sigma\ |\!\}\ return_C\ upd\ rhs\ \{\!|\lambda r\ \sigma.\ exec\text{-}stop\ \sigma \wedge P\ \sigma\ |\!\}$
⟨*proof*⟩

lemma *assign-clean-skip*:
$\{\!|\lambda\sigma.\quad exec\text{-}stop\ \sigma \wedge P\ \sigma\ |\!\}\ \ assign\ tr\ \ \{\!|\lambda r\ \sigma.\ exec\text{-}stop\ \sigma \wedge P\ \sigma\ |\!\}$
⟨*proof*⟩

lemma *if-clean-skip*:
$\{\!|\lambda\sigma.\quad exec\text{-}stop\ \sigma \wedge P\ \sigma\ |\!\}\ \ if_C\ C\ then\ E\ else\ F\ fi\ \{\!|\lambda r\ \sigma.\ exec\text{-}stop\ \sigma \wedge P\ \sigma\ |\!\}$
⟨*proof*⟩

lemma *while-clean-skip*:
$\{\!|\lambda\sigma.\quad exec\text{-}stop\ \sigma \wedge P\ \sigma\ |\!\}\ \ while_C\ cond\ do\ body\ od\ \ \{\!|\lambda r\ \sigma.\ exec\text{-}stop\ \sigma \wedge P\ \sigma\ |\!\}$
⟨*proof*⟩

lemma *if-opcall-skip*:
$\{\!|\lambda\sigma.\quad exec\text{-}stop\ \sigma \wedge P\ \sigma|\!\}\ (call_C\ M\ A_1)\ \{\!|\lambda r\ \sigma.\ exec\text{-}stop\ \sigma \wedge P\ \sigma|\!\}$
⟨*proof*⟩

lemma *if-funcall-skip*:
$\{\!|\lambda\sigma.\ exec\text{-}stop\ \sigma \wedge P\ \sigma|\!\}(p_{tmp} \leftarrow call_C\ fun\ E\ ;\ assign\text{-}local\ upd\ (\lambda\sigma.\ p_{tmp}))\ \{\!|\lambda r\ \sigma.\ exec\text{-}stop\ \sigma \wedge P\ \sigma|\!\}$
⟨*proof*⟩

lemma *if-funcall-skip′*:
$\{\!|\lambda\sigma.\ exec\text{-}stop\ \sigma \wedge P\ \sigma\ |\!\}(p_{tmp} \leftarrow call_C\ fun\ E\ ;\ assign\text{-}global\ upd\ (\lambda\sigma.\ p_{tmp}))\ \{\!|\lambda r\ \sigma.\ exec\text{-}stop\ \sigma \wedge P\ \sigma\ |\!\}$
⟨*proof*⟩

6.3.9 Clean Assign Rules

lemma *assign-global*:
assumes $* : \sharp\ upd$
shows $\{\!|\lambda\sigma.\ \triangleright\ \sigma \wedge P\ (upd\ (\lambda\text{-}.\ rhs\ \sigma)\ \sigma)\ |\!\}\ upd :==_G\ rhs\ \{\!|\lambda r\ \sigma.\ \triangleright\ \sigma \wedge P\ \sigma\ |\!\}$
⟨*proof*⟩

find-theorems $\sharp$ -

lemma *assign-local*:
assumes $* : \sharp\ (upd \circ upd\text{-}hd)$
shows $\{\!|\lambda\sigma.\ \triangleright\ \sigma \wedge P\ ((upd \circ upd\text{-}hd)\ (\lambda\text{-}.\ rhs\ \sigma)\ \sigma)\ |\!\}\ upd :==_L\ rhs\ \ \{\!|\lambda r\ \sigma.\ \triangleright\ \sigma \wedge P\ \sigma\ |\!\}$
⟨*proof*⟩

lemma *return-assign*:
assumes $* : \sharp\ (upd \circ upd\text{-}hd)$
shows $\{\!|\lambda\ \sigma.\ \triangleright\ \sigma \wedge P\ ((upd \circ upd\text{-}hd)\ (\lambda\text{-}.\ rhs\ \sigma)\ (\sigma\ (\!|\ return\text{-}status := True\ |\!)))|\!\}$
$return_{upd}(rhs)$
$\{\!|\lambda r\ \sigma.\ P\ \sigma \wedge return\text{-}status\ \sigma\ |\!\}$
⟨*proof*⟩

6.3.10 Clean Construct Rules

lemma *cond-clean* :
$\{\!|\lambda\sigma.\ \triangleright\ \sigma \wedge P\ \sigma \wedge cond\ \sigma|\!\}\ M\ \{\!|Q|\!\}$
$\Longrightarrow \{\!|\lambda\sigma.\ \triangleright\ \sigma \wedge P\ \sigma \wedge \neg\ cond\ \sigma|\!\}\ M'\ \{\!|Q|\!\}$
$\Longrightarrow \{\!|\lambda\sigma.\ \triangleright\ \sigma \wedge P\ \sigma|\!\}\ if_C\ cond\ then\ M\ else\ M'\ fi\{\!|Q|\!\}$
⟨*proof*⟩

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 $[\![\{\!|\lambda\sigma.\ ?cond\ \sigma \wedge ?P\ \sigma|\!\}\ ?M\ \{\!|\lambda\text{-}.\ ?P|\!\};\ \forall\sigma.\ ?cond\ \sigma \wedge ?P\ \sigma \longrightarrow ?M\ \sigma \neq None \wedge ?f\ (snd\ (the\ (?M\ \sigma))) < ?f\ \sigma]\!] \Longrightarrow \{\!|?P|\!\}$ *-while-SE ?cond ?M* $\{\!|\lambda\text{-}\ \sigma.\ \neg\ ?cond\ \sigma \wedge ?P\ \sigma|\!\}$.

lemma *while-clean-no-break* :
assumes $* : \{\!|\lambda\sigma.\ \neg\ break\text{-}status\ \sigma \wedge cond\ \sigma \wedge P\ \sigma|\!\}\ \ M\ \{\!|\lambda\text{-}.\ \lambda\sigma.\ \ \neg\ break\text{-}status\ \sigma \wedge P\ \sigma\ |\!\}$
and *measure*: $\forall\sigma.\ \neg\ exec\text{-}stop\ \sigma \wedge cond\ \sigma \wedge P\ \sigma$
$\longrightarrow M\ \sigma \neq None \wedge f(snd(the(M\ \sigma))) < ((f\ \sigma)::nat)$
(**is** $\forall\sigma.\ \text{-} \wedge cond\ \sigma \wedge P\ \sigma \longrightarrow ?decrease\ \sigma$)
shows $\{\!|\lambda\sigma.\ \triangleright\ \sigma \wedge P\ \sigma|\!\}$
$while_C\ cond\ do\ M\ od$
$\{\!|\lambda\text{-}\ \sigma.\ (return\text{-}status\ \sigma \vee \neg\ cond\ \sigma) \wedge \neg\ break\text{-}status\ \sigma \wedge P\ \sigma|\!\}$
(**is** $\{\!|?pre|\!\}\ while_C\ cond\ do\ M\ od\ \{\!|\lambda\text{-}\ \sigma.\ ?post1\ \sigma \wedge ?post2\ \sigma|\!\}$)
⟨*proof*⟩

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* : $\{\!|\lambda\sigma.\ \triangleright\ \sigma \wedge cond\ \sigma \wedge P\ \sigma|\!\}\ \ M\ \{\!|\lambda\text{-}.\ P|\!\}$
and *cond-idpc* : $\forall x\ \sigma.\ \ (cond\ (\sigma(\!|break\text{-}status := x|\!))) = cond\ \sigma$
and *inv-idpc* : $\forall x\ \sigma.\ \ (P\ (\sigma(\!|break\text{-}status := x|\!))) = P\ \sigma$
and *f-is-measure* : $\forall\sigma.\ \triangleright\ \sigma \wedge cond\ \sigma \wedge P\ \sigma \longrightarrow\ \ M\ \sigma \neq None \wedge f(snd(the(M\ \sigma))) < ((f\ \sigma)::nat)$
shows $\{\!|\lambda\sigma.\ \triangleright\ \sigma \wedge P\ \sigma|\!\}\ \ while_C\ cond\ do\ M\ od\ \ \{\!|\lambda\text{-}\ \sigma.\ \ \neg\ break\text{-}status\ \sigma \wedge P\ \sigma|\!\}$
⟨*proof*⟩

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.