

Clean - An Abstract Imperative Programming Language and its Theory

Frédéric Tuong Burkhart Wolff
(with Contributions by Chantal Keller)

May 26, 2024

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	7
1.1	A High-level Description of the Clean Memory Model	8
1.1.1	A Simple Typed Memory Model of Clean: An Introduction	8
1.1.2	Formally Modeling Control-States	8
1.1.3	An Example for Global Variable Declarations.	12
1.1.4	The Assignment Operations (embedded in State-Exception Monad)	12
1.1.5	Example for a Local Variable Space	13
1.2	Global and Local State Management via Extensible Records	14
1.2.1	Block-Structures	16
1.2.2	Call Semantics	16
1.3	Some Term-Coding Functions	17
1.4	Syntactic Sugar supporting λ -lifting for Global and Local Variables	25
1.5	Support for (direct recursive) Clean Function Specifications	27
1.6	The Rest of Clean: Break/Return aware Version of If, While, etc.	33
1.7	Miscellaneous	34
1.8	Function-calls in Expressions	37
2	Clean Semantics : A Coding-Concept Example	39
2.1	The Quicksort Example	39
2.2	Clean Encoding of the Global State of Quicksort	40
2.3	Encoding swap in Clean	41
2.3.1	swap in High-level Notation	41
2.3.2	A Simulation of swap in elementary specification constructs:	42
2.4	Encoding partition in Clean	44
2.4.1	partition in High-level Notation	44
2.4.2	A Simulation of partition in elementary specification constructs:	45
2.5	Encoding the toplevel : quicksort in Clean	46
2.5.1	quicksort in High-level Notation	46
2.5.2	A Simulation of quicksort in elementary specification constructs:	47
2.5.3	Setup for Deductive Verification	48
3	Clean Semantics : A Coding-Concept Example	51
3.1	The Quicksort Example - At a Glance	51
3.2	Clean Encoding of the Global State of Quicksort	51
3.3	Possible Application Sketch	53
3.4	The Squareroot Example for Symbolic Execution	53
3.4.1	The Conceptual Algorithm in Clean Notation	53

3.4.2	Definition of the Global State	53
3.4.3	Setting for Symbolic Execution	54
3.4.4	A Symbolic Execution Simulation	55
4	Clean Semantics : Another Clean Example	57
4.1	The Primality-Test Example at a Glance	57
5	A Clean Semantics Example : Linear Search	59
5.1	The LinearSearch Example	59
6	Appendix : Used Monad Libraries	61
6.1	Definition : Standard State Exception Monads	61
6.1.1	Definition : Core Types and Operators	61
6.1.2	Definition : More Operators and their Properties	62
6.1.3	Definition : Programming Operators and their Properties	63
6.1.4	Theory of a Monadic While	63
6.1.5	Chaining Monadic Computations : Definitions of Multi-bind Op- erators	68
6.1.6	Definition and Properties of Valid Execution Sequences	70
6.1.7	Miscellaneous	81
6.2	Clean Symbolic Execution Rules	81
6.2.1	Basic NOP - Symbolic Execution Rules.	81
6.2.2	Assign Execution Rules.	82
6.2.3	Basic Call Symbolic Execution Rules.	83
6.2.4	Basic Call Symbolic Execution Rules.	84
6.2.5	Conditional.	85
6.2.6	Break - Rules.	85
6.2.7	While.	86
6.3	Hoare	89
6.3.1	Basic rules	89
6.3.2	Generalized and special sequence rules	90
6.3.3	Generalized and special consequence rules	90
6.3.4	Condition rules	91
6.3.5	While rules	92
6.3.6	Experimental Alternative Definitions (Transformer-Style Rely-Guarantee)	94
6.3.7	Clean Control Rules	95
6.3.8	Clean Skip Rules	95
6.3.9	Clean Assign Rules	96
6.3.10	Clean Construct Rules	97

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** $(\prime o, \prime \sigma) MON_{SE} = \langle \prime \sigma \rightarrow (\prime o \times \prime \sigma) \rangle$ with the usual definitions of *bind* and *unit*. In this language, sequence operators, conditionals and loops can be integrated.

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

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

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

control-state represents the σ_0 state.

2. Any global variable definition block with definitions $a_1 : \tau_1 \dots a_n : \tau_n$ is translated into a record extension:

```
record  $\sigma_{n+1} = \sigma_n + a_1 :: \tau_1; \dots; a_n :: \tau_n$ 
```

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

```
record  $\sigma_{n+1} = \sigma_n + a_1 :: \tau_1 \text{ list}; \dots; a_n :: \tau_n \text{ list}; \text{result} :: \tau_{\text{result-type}} \text{ list};$ 
```

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

1.1.2 Formally Modeling Control-States

The control state is the “root” of all extensions for local and global variable spaces in Clean. It contains just the information of the current control-flow: a *break* occurred

(meaning all commands till the end of the control block will be skipped) or a *return* occurred (meaning all commands till the end of the current function body will be skipped).

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

ML⟨ *val* *t* = @{*term* σ (| *break-status* := *False* |)}⟩

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

definition *unset-break-status* :: (*unit*, (' σ -*ext*) *control-state-ext*) *MON_{SE}*
where *unset-break-status* \equiv (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* \Rightarrow *bool*
where *exec-stop* = (λ σ . *break-status* σ \vee *return-status* σ)

abbreviation *normal-execution* :: (' σ -*ext*) *control-state-ext* \Rightarrow *bool*
where (*normal-execution* *s*) \equiv (\neg *exec-stop* *s*)

notation *normal-execution* (\triangleright)

lemma *exec-stop1*[*simp*] : *break-status* σ \Longrightarrow *exec-stop* σ
unfolding *exec-stop-def* **by** *simp*

lemma *exec-stop2*[*simp*] : *return-status* σ \Longrightarrow *exec-stop* σ
unfolding *exec-stop-def* **by** *simp*

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

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

definition *break-status_L*
where *break-status_L* = *create_L control-state.break-status control-state.break-status-update*

lemma *vwb-lens break-status_L*
unfolding *break-status_L-def*
by (*simp add: vwb-lens-def create_L-def wb-lens-def mwb-lens-def*)

mwb-lens-axioms-def upd2put-def wb-lens-axioms-def weak-lens-def)

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*

unfolding *return-status_L-def*

by (*simp add: vwb-lens-def create_L-def wb-lens-def mwb-lens-def*
mwb-lens-axioms-def upd2put-def wb-lens-axioms-def weak-lens-def)

lemma *break-return-indep : break-status_L ⋈ return-status_L*

by (*simp add: break-status_L-def lens-indepI return-status_L-def upd2put-def create_L-def*)

definition *strong-control-independence* (#!)

where #! *L* = (*break-status_L ⋈ L* ∧ *return-status_L ⋈ L*)

lemma *vwb-lens break-status_L*

unfolding *vwb-lens-def break-status_L-def create_L-def wb-lens-def mwb-lens-def*

by (*simp add: mwb-lens-axioms-def upd2put-def wb-lens-axioms-def weak-lens-def*)

definition *control-independence* ::

$(('b \Rightarrow 'a) \Rightarrow 'a \text{ control-state-scheme} \Rightarrow 'a \text{ control-state-scheme}) \Rightarrow \text{bool}$ (#)

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 ⇒ # (λf. λσ. lens-put L σ (f (lens-get L σ)))*

unfolding *strong-control-independence-def control-independence-def*

by (*simp add: break-status_L-def lens-indep-def return-status_L-def upd2put-def create_L-def*)

lemma *expimnt : #! (create_L getv updv) ⇒ (λf σ. updv (λ-. f (getv σ)) σ) = updv*

unfolding *create_L-def strong-control-independence-def*

break-status_L-def return-status_L-def lens-indep-def

apply(*rule ext, rule ext*)

apply *auto*

unfolding *upd2put-def*

oops

lemma *expimnt :*

vwb-lens (create_L getv updv) ⇒ (λf σ. updv (λ-. f (getv σ)) σ) = updv

unfolding *create_L-def strong-control-independence-def lens-indep-def*

break-status_L-def return-status_L-def vwb-lens-def

apply(*rule ext, rule ext*)

apply *auto*

unfolding *upd2put-def wb-lens-def weak-lens-def wb-lens-axioms-def mwb-lens-def*

mwb-lens-axioms-def

apply *auto*

oops

lemma *strong-vs-weak-upd* :
assumes * : $\#!$ (*create_L getv upd*)
and ** : $(\lambda f \sigma. \text{upd } (\lambda \cdot. f (\text{getv } \sigma)) \sigma) = \text{upd}$
shows $\#!$ (*upd*)
apply (*insert * ***)
unfolding *create_L-def upd2put-def*
by (*drule strong-vs-weak-ci, auto*)

This quite tricky proof establishes the fact that the special case $hd(\text{getv } \sigma) = []$ for $\text{getv } \sigma = []$ 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 \Rightarrow 'c list)*)
(upd:: ('c list \Rightarrow 'c list) \Rightarrow 'b control-state-scheme \Rightarrow 'b control-state-scheme))

and ** : $(\lambda f \sigma. \text{upd } (\lambda \cdot. f (\text{getv } \sigma)) \sigma) = \text{upd}$
shows $\#!$ (*upd \circ upd-hd*)

proof –

have *** : $\#!$ (*create_L (hd \circ getv) (upd \circ upd-hd)*)
using * ** **by** (*simp add: indep-list-lift strong-control-independence-def*)
show $\#!$ (*upd \circ upd-hd*)
apply (*rule strong-vs-weak-upd*)
apply (*rule ****)
apply (*rule ext, rule ext, simp*)
apply (*subst (2) ** [symmetric]*)

proof –

fix *f*:: 'c \Rightarrow 'c **fix** σ :: 'b control-state-scheme
show *upd \circ upd-hd* ($\lambda \cdot. f (\text{hd } (\text{getv } \sigma))$) $\sigma = \text{upd } (\lambda \cdot. \text{upd-hd } f (\text{getv } \sigma)) \sigma$
proof (*cases getv σ*)

case *Nil*

then show *?thesis*

by (*simp,metis (no-types) ** upd-hd.simps(1)*)

next

case (*Cons a list*)

then show *?thesis*

proof –

have $(\lambda c. f (\text{hd } (\text{getv } \sigma))) = ((\lambda c. f a):: 'c \Rightarrow 'c)$

using *local.Cons by auto*

then show *?thesis*

by (*metis (no-types) ** local.Cons upd-hd.simps(2)*)

qed

qed

qed

qed

lemma *exec-stop-vs-control-independence* [simp]:
upd \implies *exec-stop* (upd f σ) = *exec-stop* σ
unfolding *control-independence-def exec-stop-def* **by** *simp*

lemma *exec-stop-vs-control-independence'* [simp]:
upd \implies (upd f (σ ($\text{return-status} := b$))) = (upd f σ) ($\text{return-status} := b$)
unfolding *control-independence-def exec-stop-def* **by** *simp*

lemma *exec-stop-vs-control-independence''* [simp]:
upd \implies (upd f (σ ($\text{break-status} := b$))) = (upd f σ) ($\text{break-status} := b$)
unfolding *control-independence-def exec-stop-def* **by** *simp*

1.1.3 An Example for Global Variable Declarations.

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

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

record *state0* = *control-state* + $A :: \text{int list}$

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

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

A further extension would be needed if a (potentially recursive) function f with some local variable tmp is defined: **record** *state2* = *state1* + $\text{tmp} :: \text{nat stack result-value} :: \text{nat stack}$, 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* :: ($\alpha \Rightarrow \text{int}$) \Rightarrow $\text{int} \Rightarrow \text{term}$ (**infix** := 60)

definition *assign* :: ((σ -ext) *control-state-scheme* \Rightarrow

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

$(\sigma\text{-ext control-state-scheme}) \Rightarrow$
 $(\text{unit}, (\sigma\text{-ext control-state-scheme}) \text{MON}_{SE})$

where $\text{assign } f = (\lambda\sigma. \text{if exec-stop } \sigma \text{ then Some}(\cdot), \sigma) \text{ else Some}(\cdot), f \sigma)$

definition $\text{assign-global} :: ((\sigma \Rightarrow \sigma) \Rightarrow \sigma\text{-ext control-state-scheme} \Rightarrow \sigma\text{-ext control-state-scheme})$
 $\Rightarrow (\sigma\text{-ext control-state-scheme} \Rightarrow \sigma)$
 $\Rightarrow (\text{unit}, \sigma\text{-ext control-state-scheme}) \text{MON}_{SE} (\mathbf{infix} ::=_G 100)$

where $\text{assign-global upd rhs} = \text{assign}(\lambda\sigma. ((\text{upd}) (\lambda\cdot. \text{rhs } \sigma)) \sigma)$

An update of the variable A based on the state of the previous example is done by $\text{assign-global } A\text{-upd } (\lambda\sigma. \text{list-update } (A \sigma) (i) (A \sigma ! j))$ 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 $\text{assign-local} :: ((\sigma \Rightarrow \sigma) \Rightarrow \sigma\text{-ext control-state-scheme} \Rightarrow \sigma\text{-ext control-state-scheme})$
 $\Rightarrow (\sigma\text{-ext control-state-scheme} \Rightarrow \sigma)$
 $\Rightarrow (\text{unit}, \sigma\text{-ext control-state-scheme}) \text{MON}_{SE} (\mathbf{infix} ::=_L 100)$

where $\text{assign-local upd rhs} = \text{assign}(\lambda\sigma. ((\text{upd } \circ \text{upd-hd}) (\sigma\text{-ext control-state-scheme} \Rightarrow \sigma)) \sigma)$

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 $(\text{upd} ::=_L \text{rhs}) = ((\text{upd } \circ \text{upd-hd}) ::=_G \text{rhs})$
unfolding $\text{assign-local-def assign-global-def}$ by *simp*

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

definition $\text{return}_C 0$

where $\text{return}_C 0 A = (\lambda\sigma. \text{if exec-stop } \sigma \text{ then Some}(\cdot), \sigma)$
 $\text{else } (A ; - \text{set-return-status}) \sigma)$

definition $\text{return}_C :: ((\sigma \Rightarrow \sigma) \Rightarrow \sigma\text{-ext control-state-scheme} \Rightarrow \sigma\text{-ext control-state-scheme})$
 $\Rightarrow (\sigma\text{-ext control-state-scheme} \Rightarrow \sigma)$
 $\Rightarrow (\text{unit}, \sigma\text{-ext control-state-scheme}) \text{MON}_{SE} (\text{return})$

where $\text{return}_C \text{upd rhs} = \text{return}_C 0 (\text{assign-local upd rhs})$

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

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

```

definition push-local-swap-state :: (unit,'a local-swap-state-scheme) MONSE
where push-local-swap-state  $\sigma =$ 
  Some((), $\sigma$ (local-swap-state.tmp := undefined # local-swap-state.tmp  $\sigma$ ,
    local-swap-state.result-value := undefined #
    local-swap-state.result-value  $\sigma$  ))

```

```

definition pop-local-swap-state :: (unit,'a local-swap-state-scheme) MONSE
where pop-local-swap-state  $\sigma =$ 
  Some(hd(local-swap-state.result-value  $\sigma$ ),
     $\sigma$ (local-swap-state.tmp:= tl( local-swap-state.tmp  $\sigma$  )))

```

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

1.2 Global and Local State Management via Extensible Records

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

ML⟨

```

structure StateMgt-core =
struct

```

```

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

```

```

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

```

```

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

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

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

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

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

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

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

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

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

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

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

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

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

```

| - => false

```

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

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

end>

```

1.2.1 Block-Structures

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

definition $block_C :: (unit, ('\sigma\text{-ext}$ control-state-ext) MON_{SE}

$$\Rightarrow (unit, ('\sigma\text{-ext}$$
 control-state-ext) MON_{SE}

$$\Rightarrow ('\alpha, ('\sigma\text{-ext}$$
 control-state-ext) MON_{SE}

$$\Rightarrow ('\alpha, ('\sigma\text{-ext}$$
 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 $\equiv (\lambda(i,j). ((assign\text{-local}$ tmp-update ($\lambda\sigma. A \sigma ! i$)) ;—

(assign-global A-update ($\lambda\sigma. list\text{-update}$ (A σ) (i) (A $\sigma ! j$))) ;—

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

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

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

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

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

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

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

1.3 Some Term-Coding Functions

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

ML
 $(* HOLogic \text{ extended} *)$

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

```

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

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

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

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

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

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

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

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

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

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

fun mk-lens-type from-ty to-ty = Type(@{type-name lens.lens-ext},
  [from-ty, to-ty, HOLogic.unitT]);

```

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

```
signature STATEMGT = sig
  structure Data: GENERIC-DATA
  datatype var-kind = global-var of typ | local-var of typ
  type state-field-tab = var-kind Symtab.table
  val MON-SE-T: typ -> typ -> typ
  val add-record-cmd:
    {overloaded: bool} ->
    bool ->
    (string * string option) list ->
    binding -> string option -> (binding * string * mixfix) list -> theory -> theory
  val add-record-cmd':
    {overloaded: bool} ->
    bool ->
    (string * string option) list ->
    binding -> string option -> (binding * typ * mixfix) list -> theory -> theory
  val add-record-cmd0:
    ('a -> Proof.context -> (binding * typ * mixfix) list * Proof.context) ->
    {overloaded: bool} ->
    bool -> (string * string option) list -> binding -> string option -> 'a -> theory
-> theory
  val cmd:
    (binding * typ option * mixfix) option * (Attrib.binding * term) * term list *
    (binding * typ option * mixfix) list
    -> local-theory -> local-theory
  val construct-update: bool -> binding -> typ -> theory -> term
  val control-stateS: typ
  val control-stateT: typ
  val declare-state-variable-global: (typ -> var-kind) -> string -> theory -> theory
  val declare-state-variable-local: (typ -> var-kind) -> string -> Context.generic -> Con-
text.generic
  val define-lense: binding -> typ -> binding * typ * 'a -> Proof.context -> local-theory
  val fetch-state-field: string * 'a -> (string * string) * 'a
  val filter-attr-of: string -> theory -> ((string * string) * var-kind) list
  val filter-name: string -> string * 'a -> ((string * string) * 'a) option
  val get-data: Proof.context -> Data.T
  val get-data-global: theory -> Data.T
  val get-result-value-conf: string -> theory -> (string * string) * var-kind
  val get-state-field-tab: Proof.context -> state-field-tab
  val get-state-field-tab-global: theory -> state-field-tab
  val get-state-type: Proof.context -> typ
  val get-state-type-global: theory -> typ
  val is-global-program-variable: Symtab.key -> theory -> bool
  val is-local-program-variable: Symtab.key -> theory -> bool
  val is-program-variable: Symtab.key -> theory -> bool
  val map-data: (Data.T -> Data.T) -> Context.generic -> Context.generic
  val map-data-global: (Data.T -> Data.T) -> theory -> theory
```

```

    val map-to-update: typ -> bool -> theory -> (string * string) * var-kind -> term ->
term
    val merge-control-stateS: typ * typ -> typ
    val mk-global-state-name: binding -> binding
    val mk-lense-name: binding -> binding
    val mk-local-state-name: binding -> binding
    val mk-lookup-result-value-term: string -> typ -> theory -> term
    val mk-pop-def: binding -> typ -> typ -> Proof.context -> local-theory
    val mk-pop-name: binding -> binding
    val mk-push-def: binding -> typ -> Proof.context -> local-theory
    val mk-push-name: binding -> binding
    val new-state-record:
        bool ->
            (((string * string option) list * binding) * string option) option *
            (binding * string * mixfix) list
            -> theory -> theory
    val new-state-record':
        bool ->
            (((string * string option) list * binding) * typ option) option * (binding * typ * mixfix)
list ->
        theory -> theory
    val new-state-record0:
        ({overloaded: bool} ->
            bool -> 'a list -> binding -> string option -> (binding * 'b * mixfix) list -> theory
-> theory)
        -> bool -> (('a list * binding) * 'b option) option * (binding * 'b * mixfix) list ->
theory -> theory
    val optionT: typ -> typ
    val parse-tyt-'a: Proof.context -> binding -> typ
    val pop-eq: binding -> string -> typ -> typ -> Proof.context -> term
    val push-eq: binding -> string -> typ -> typ -> Proof.context -> term
    val read-fields: ('a * string * 'b) list -> Proof.context -> ('a * typ * 'b) list * Proof.context
    val read-parent: string option -> Proof.context -> (typ list * string) option * Proof.context
    val result-name: string
    val typ-2-string-raw: typ -> string
    val type-of: var-kind -> typ
    val upd-state-type: (typ -> typ) -> Context.generic -> Context.generic
    val upd-state-type-global: (typ -> typ) -> theory -> theory

```

end

```

structure StateMgt : STATEMGT =
struct

```

```

open StateMgt-core

```

```

val result-name = result-value

```

```

fun get-result-value-conf name thy =

```

```

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

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

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

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

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

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

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

fun mk-lense-name binding = Binding.suffix-name L binding

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

end;

fun mk-push-def binding sty lthy =
  let val name-pushop = mk-push-name binding
      val rty = typ ⟨unit⟩
  in
  end

```

```

    val eq = push-eq binding (Binding.name-of name-pushop) rty sty lthy
    val mty = StateMgt-core.MON-SE-T rty sty
    val args = (SOME(name-pushop, SOME mty, NoSyn), (Binding.empty-atts,eq),[],[])
  in cmd args lthy end;

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

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

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

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

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

fun parse-ty-'a ctxt binding =
  let val ty-bind = Binding.prefix-name 'a (Binding.suffix-name -scheme binding)
  in case Syntax.parse-ty ctxt (Binding.name-of ty-bind) of
    Type (s, -) => Type (s, [@\{typ 'a::type\}])
  | - => error (Unexpected type ^ Position.here here)
  end

fun define-lense binding sty (attr-name,rty,-) lthy =
  let val prefix = Binding.name-of binding ^

```

```

val name-L = attr-name |> Binding.prefix-name prefix
                |> mk-lense-name
val name-upd = Binding.suffix-name -update attr-name
val acc-ty = sty --> rty
val upd-ty = (rty --> rty) --> sty --> sty
val cr = Const(@{const-name Optics.createL},
                acc-ty --> upd-ty --> mk-lens-type rty sty)
val thy = Proof-Context.theory-of lthy
val acc-name = Sign.intern-const thy (Binding.name-of attr-name)
val upd-name = Sign.intern-const thy (Binding.name-of name-upd)
val acc = Const(acc-name, acc-ty)
val upd = Const(upd-name, upd-ty)
val lens-ty = mk-lens-type rty sty
val eq = mk-meta-eq (Free(Binding.name-of name-L, lens-ty), cr $ acc $ upd)
val args = (SOME(name-L, SOME lens-ty, NoSyn), (Binding.empty-atts,eq),[],[])
in cmd args lthy end

```

```

fun add-record-cmd0 read-fields overloaded is-global-kind raw-params binding raw-parent raw-fields
thy =

```

```

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

```

```

    in thy |> Named-Target.theory-map (fold (define-lense binding sty) fields') end
in thy |> Record.add-record overloaded (params', binding) parent fields'
    |> (fn thy => List.foldr insert-var (thy) (fields'))
    |> upd-state-typ
    |> define-push-pop
    |> define-lenses
end;

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

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

fun new-state-record0 add-record-cmd is-global-kind (aS, raw-fields) thy =
    let val state-index = (Int.toString o length o Syntab.dest)
        (StateMgt-core.get-state-field-tab-global thy)
        val state-pos = (Binding.pos-of o #1 o hd) raw-fields
        val ((raw-params, binding), res-ty) = case aS of
            SOME d => d
            | NONE => ([], Binding.make(state-index,state-pos)),
NONE)
        val binding = if is-global-kind
            then mk-global-state-name binding
            else mk-local-state-name binding
        val raw-parent = SOME(typ-2-string-raw (StateMgt-core.get-state-type-global thy))
        val - = writeln(XXXXX ^@{make-string} raw-params ^CCC ^@{make-string} binding
            ^@{make-string} raw-fields)
        val pos = Binding.pos-of binding
        fun upd-state-typ thy = StateMgt-core.upd-state-type-global
            (K (parse-typ-'a (Proof-Context.init-global thy) binding)) thy
        val result-binding = Binding.make(result-name,pos)
        val raw-fields' = case res-ty of
            NONE => raw-fields
            | SOME res-ty => raw-fields @ [(result-binding,res-ty, NoSyn)]
    in thy |> add-record-cmd {overloaded = false} is-global-kind
        raw-params binding raw-parent raw-fields'
        |> upd-state-typ
    end

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

```



```

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

```

```

fun clean-ctxt-parser b = Parse.$$$ (
  |-- (Parse.type-args-constrained -- Parse.binding)
  -- (if b then Scan.succeed NONE else Parse.typ >> SOME)
  --| Parse.$$$ )
  : (((string * string option) list * binding) * string option) parser

```

```

val - =
  Outer-Syntax.command
  command-keyword <global-vars>
  define global state record
  (Scan.option (clean-ctxt-parser true) -- Scan.repeat1 Parse.const-binding
   >> (Toplevel.theory o new-state-record true));

```

```

val - =
  Outer-Syntax.command
  command-keyword <local-vars-test>
  define local state record
  (Scan.option (clean-ctxt-parser false) -- Scan.repeat1 Parse.const-binding
   >> (Toplevel.theory o new-state-record false));

```

```

end
>

```

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

```

ML <
structure Clean-Syntax-Lift =
struct
  type T = { is-local : string -> bool
            , is-global : string -> bool }

  val init =
    Proof-Context.theory-of
    #> (fn thy =>
        { is-local = fn name => StateMgt-core.is-local-program-variable name thy
        , is-global = fn name => StateMgt-core.is-global-program-variable name thy })

  local
    fun mk-local-access X = Const (@{const-name Fun.comp}, dummyT)

```

```

    $ Const (@{const-name List.list.hd}, dummyT) $ X
in
  fun app-sigma0 (st : T) db tm = case tm of
    Const(name, -) => if #is-global st name
      then tm $ (Bound db) (* lambda lifting *)
      else if #is-local st name
        then (mk-local-access tm) $ (Bound db) (* lambda lifting local *)
        else tm (* no lifting *)
  | Free - => tm
  | Var - => tm
  | Bound n => if n > db then Bound(n + 1) else Bound n
  | Abs (x, ty, tm') => Abs(x, ty, app-sigma0 st (db+1) tm')
  | t1 $ t2 => (app-sigma0 st db t1) $ (app-sigma0 st db t2)

  fun app-sigma db tm = init #> (fn st => app-sigma0 st db tm)

  fun scope-var st name =
    if #is-global st name then SOME true
    else if #is-local st name then SOME false
    else NONE

  fun assign-update var = var ^ Record.updateN

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

  fun transform-term st sty =
    transform-term0
    (fn tm => Abs (σ, sty, app-sigma0 st 0 tm))
    (scope-var st)

  fun transform-term' st = transform-term st dummyT

  fun string-tr ctxt content args =
    let fun err () = raise TERM (string-tr, args)
    in
      (case args of
        [(Const (@{syntax-const -constrain}, -)) $ (Free (s, -)) $ p] =>

```

```

      (case Term-Position.decode-position p of
        SOME (pos, -) => Symbol-Pos.implode (content (s, pos))
          |> Syntax.parse-term ctxt
          |> transform-term (init ctxt) (StateMgt-core.get-state-type ctxt)
          |> Syntax.check-term ctxt
        | NONE => err ())
    | - => err ()
  end
end
end
>

syntax -cartouche-string :: cartouche-position => string (-)

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

```

1.5 Support for (direct recursive) Clean Function Specifications

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

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

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

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 <
  structure Function-Specification-Parser =
    struct

      type funct-spec-src = {
        binding: binding,

```

(* name *)

```

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

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

type funct-spec-sem = {
  binding: binding,                      (* name *)
  params: (binding*string) list,         (* parameters and their type*)
  ret-type: string,                     (* return type; default unit *)
  locals: (binding*string*mixfix)list,   (* local variables *)
  read-pre: Proof.context -> term,       (* precondition src *)
  read-post: Proof.context -> term,     (* postcondition src *)
  read-variant-opt: (Proof.context->term) option, (* variant src *)
  read-body: Proof.context -> typ -> term (* body src *)
}

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

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

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

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

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

```

```

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

```

```

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

```

```

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

```

```

fun read-function-spec ( params, ret-type, read-variant-opt) ctxt =
  let val (params-Ts, ctxt') = read-params params ctxt
      val (rty, ctxt'') = read-result ret-type ctxt'
      val variant = case read-variant-opt of
        NONE => NONE
      | SOME f => SOME(f ctxt'')
      val paramT-l = (map2 (fn (b, -) => fn T => (b, T)) params params-Ts)
  in ((paramT-l, rty, variant), ctxt'') end

```

```

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

```

```

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

```

```

fun define-cond binding f-sty transform-old check-absence-old cond-suffix params read-cond
(ctxt:local-theory) =
  let val params' = map (fn(b, ty) => (Binding.name-of b,ty)) params

```

```

val src' = case transform-old (read-cond ctxt) of
  Abs(nn, sty-pre, term) => mk-pat-tupleabs params' (Abs(nn,sty-pre,term))
  | - => error (define abstraction for result ^ Position.here here)
val bdg = Binding.suffix-name cond-suffix binding
val - = check-absence-old src'
val bdg-ty = HOLogic.mk-tupleT(map (#2) params) --> f-sty HOLogic.boolT
val eq = mk-meta-eq(Free(Binding.name-of bdg, bdg-ty),src')
val args = (SOME(bdg,NONE,NoSyn), (Binding.empty-atts,eq),[],[])
in StateMgt.cmd args ctxt end

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

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

fun define-body-core binding args-ty sty params body =
  let val params' = map (fn(b,ty) => (Binding.name-of b, ty)) params
      val bdg-core = Binding.suffix-name -core binding
      val bdg-core-name = Binding.name-of bdg-core

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

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

  in StateMgt.cmd args-core
  end

fun define-body-main {recursive = x:bool} binding rty sty params read-variant-opt - ctxt =
  let val push-name = StateMgt.mk-push-name (StateMgt.mk-local-state-name binding)
      val pop-name = StateMgt.mk-pop-name (StateMgt.mk-local-state-name binding)
      val bdg-core = Binding.suffix-name -core binding
      val bdg-core-name = Binding.name-of bdg-core
      val bdg-rec-name = Binding.name-of(Binding.suffix-name -rec binding)
      val bdg-ord-name = Binding.name-of(Binding.suffix-name -order binding)
      val args-ty = HOLogic.mk-tupleT (map snd params)
      val rmtty = StateMgt-core.MON-SE-T rty sty
      val umty = StateMgt.MON-SE-T @{typ unit} sty
      val argsProdT = HOLogic.mk-prodT(args-ty,args-ty)
      val argsRelSet = HOLogic.mk-setT argsProdT
      val params' = map (fn(b, ty) => (Binding.name-of b,ty)) params
      val measure-term = case read-variant-opt of
        NONE => Free(bdg-ord-name,args-ty --> HOLogic.natT)
        | SOME f => ((f ctxt) |> mk-pat-tupleabs params')
      val measure = Const(@{const-name Wellfounded.measure}, (args-ty --> HO-
Logic.natT)
--> argsRelSet )

```

```

    $ measure-term
    val lhs-main = if x andalso is-none (read-variant-opt )
                    then Free(Binding.name-of binding, (args-ty --> HOLogic.natT)
                               --> args-ty --> rmtty) $
                               Free(bdg-ord-name, args-ty --> HOLogic.natT)
                    else Free(Binding.name-of binding, args-ty --> rmtty)
    val rhs-main = mk-pat-tupleabs params'
                    (Const(@{const-name Clean.blockC}, umty --> umty --> rmtty -->
rmtty)

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

```

```

val - = Local-Theory.exit-result-global;
val - = Named-Target.theory-map-result;
val - = Named-Target.theory-map;

```

(* This code is in large parts so messy because the extensible record package (used inside `StateMgt.new-state-record`) is only available as transformation on global contexts, which cuts the local context calculations into two halves. The second halves is cut again into two halves because the definition of the core apparently does not take effect before defining the block – structure when not separated (this problem can perhaps be overcome somehow))

Precondition: the terms of the read–functions are full typed in the respective local contexts.

*)

```

fun checkNsem-function-spec-gen {recursive = false} ({read-variant-opt=SOME -, ...}) - =
    error No measure required in non–recursive call

```

```

|checkNsem-function-spec-gen (isrec as {recursive = -:bool})
  ({binding, ret-type, read-variant-opt, locals,
   read-body, read-pre, read-post, params} : funct-spec-sem)
  thy =
let fun addfixes ((params-Ts,ret-ty,t-opt), ctxt) =
  (fn fg => fn ctxt =>
    ctxt
    |> Proof-Context.add-fixes (map (fn (s,ty)=>(s,SOME ty,NoSyn))
      (* this declares the parameters of a function specification
       as Free variables (overrides a possible constant declaration)
       and assigns the declared type to them *)
      |> (fn (X, ctxt) => fg params-Ts ret-ty ctxt)
    , ctxt)
  val (theory-map, thy') = Named-Target.theory-map-result
    (K (fn f => Named-Target.theory-map o f))
    ( read-function-spec (params, ret-type, read-variant-opt)
      #> addfixes
    )
    (thy)
in thy' |> theory-map
  let val sty-old = StateMgt-core.get-state-type-global thy'
      fun parse-contract params ret-ty =
        ( define-precond binding sty-old params read-pre
          #> define-postcond binding ret-ty sty-old params read-post)
      in parse-contract
      end
  |> StateMgt.new-state-record false (SOME ([],binding), SOME ret-type),locals)
  |> theory-map
    (fn params => fn ret-ty => fn ctxt =>
      let val sty = StateMgt-core.get-state-type ctxt
          val args-ty = HOLogic.mk-tupleT (map snd params)
          val mon-se-ty = StateMgt-core.MON-SE-T ret-ty sty
          val body = read-body ctxt mon-se-ty
          val ctxt' =
            if #recursive isrec then
              Proof-Context.add-fixes
                [(binding, SOME (args-ty --> mon-se-ty), NoSyn)] ctxt |> #2
            else
              ctxt
          val body = read-body ctxt' mon-se-ty
          in ctxt' |> define-body-core binding args-ty sty params body
          end) (* separation nasty, but nec. in order to make the body definition
              take effect. No other reason. *)
    |> theory-map
      (fn params => fn ret-ty => fn ctxt =>
        let val sty = StateMgt-core.get-state-type ctxt
            val mon-se-ty = StateMgt-core.MON-SE-T ret-ty sty

```



```

      val body = read-body ctxt mon-se-ty
    in ctxt |> define-body-main isrec binding ret-ty sty
      params read-variant-opt body
    end)
  end

fun checkNsem-function-spec (isrec as {recursive = -:bool})
  ( {binding, ret-type, variant-src, locals,
    body-src, pre-src, post-src, params} : funct-spec-src)
  thy =
  checkNsem-function-spec-gen (isrec)
  ( {binding = binding,
    params = params,
    ret-type = ret-type,
    read-variant-opt = (case variant-src of
      NONE => NONE
    | SOME t=> SOME(fn ctxt
      => Syntax.read-term ctxt t)),
    locals = locals,
    read-body = fn ctxt => fn expected-type
      => Syntax.read-term ctxt (fst body-src),
    read-pre = fn ctxt => Syntax.read-term ctxt pre-src,
    read-post = fn ctxt => Syntax.read-term ctxt post-src} : funct-spec-sem)
  thy

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

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

end

```

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

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

where $if-C \ c \ E \ F = (\lambda\sigma. \text{if exec-stop } \sigma$

then Some(undefined, σ) — state unchanged, return arbitrary
else if $c \sigma$ then $E \sigma$ else $F \sigma$

syntax (*xsymbols*)

-if-SECLEAN :: [$'\sigma \Rightarrow \text{bool}, ('o, ' \sigma) \text{MON}_{SE}, ('o', ' \sigma) \text{MON}_{SE}$] $\Rightarrow ('o', ' \sigma) \text{MON}_{SE}$
 ((*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* :: (($'\sigma$ -*ext*) *control-state-ext* \Rightarrow *bool*)

\Rightarrow (*unit*, ($'\sigma$ -*ext*) *control-state-ext*) *MON_{SE}*

\Rightarrow (*unit*, ($'\sigma$ -*ext*) *control-state-ext*) *MON_{SE}*

where *while-C c B* \equiv ($\lambda \sigma$. *if exec-stop* σ *then* *Some*((), σ)

else ((*MonadSE*.*while-SE* ($\lambda \sigma$. \neg *exec-stop* $\sigma \wedge c \sigma$) *B*) ;-
unset-break-status) σ)

syntax (*xsymbols*)

-while-C :: [$'\sigma \Rightarrow \text{bool}, (\text{unit}, ' \sigma) \text{MON}_{SE}$] $\Rightarrow (\text{unit}, ' \sigma) \text{MON}_{SE}$

((*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* :: $'a \text{ list} \Rightarrow 'b \Rightarrow 'a$

overloading *nth_C* \equiv *nth_C* :: $'a \text{ list} \Rightarrow \text{nat} \Rightarrow 'a$

begin

definition

nth_C-nat : *nth_C* (*S*:: $'a \text{ list}$) (*a*) \equiv *nth S a*

end

overloading *nth_C* \equiv *nth_C* :: $'a \text{ list} \Rightarrow \text{int} \Rightarrow 'a$

begin

definition

nth_C-int : *nth_C* (*S*:: $'a \text{ list}$) (*a*) \equiv *nth S (nat a)*

end

definition *while-C-A* :: (($'\sigma$ -*ext*) *control-state-scheme* \Rightarrow *bool*)

\Rightarrow (($'\sigma$ -*ext*) *control-state-scheme* \Rightarrow *nat*)

\Rightarrow (($'\sigma$ -*ext*) *control-state-ext* \Rightarrow *bool*)

\Rightarrow (*unit*, ($'\sigma$ -*ext*) *control-state-ext*) *MON_{SE}*

\Rightarrow (*unit*, ($'\sigma$ -*ext*) *control-state-ext*) *MON_{SE}*

where *while-C-A Inv f c B* \equiv *while-C c B*

ML

```
structure Clean-Term-interface =  
struct
```

```
fun mk-seq-C C C' = let val t = fastype-of C  
    val t' = fastype-of C'  
    in Const(const-name <bind-SE'>, t --> t' --> t') $ C $ C' end;
```

```
fun mk-skip-C sty = Const(const-name <skip-SE>, StateMgt-core.MON-SE-T HOLogic.unitT  
sty)
```

```
fun mk-break sty =  
    Const(const-name <break>, StateMgt-core.MON-SE-T HOLogic.unitT sty )
```

```
fun mk-return-C upd rhs =  
    let val ty = fastype-of rhs  
        val (sty,rty) = case ty of  
            Type(fun, [sty,rty]) => (sty,rty)  
            | - => error mk-return-C: illegal type for body  
        val upd-ty = (HOLogic.listT rty --> HOLogic.listT rty) --> sty --> sty  
        val rhs-ty = sty --> rty  
        val mty = StateMgt-core.MON-SE-T HOLogic.unitT sty  
    in Const(const-name <return-C>, upd-ty --> rhs-ty --> mty) $ upd $ rhs end
```

```
fun mk-assign-global-C upd rhs =  
    let val ty = fastype-of rhs  
        val (sty,rty) = case ty of  
            Type(fun, [sty,rty]) => (sty,rty)  
            | - => error mk-assign-global-C: illegal type for body  
        val upd-ty = (rty --> rty) --> sty --> sty  
        val rhs-ty = sty --> rty  
        val mty = StateMgt-core.MON-SE-T HOLogic.unitT sty  
    in Const(const-name <assign-global>, upd-ty --> rhs-ty --> mty) $ upd $ rhs end
```

```
fun mk-assign-local-C upd rhs =  
    let val ty = fastype-of rhs  
        val (sty,rty) = case ty of  
            Type(fun, [sty,rty]) => (sty,rty)  
            | - => error mk-assign-local-C: illegal type for body  
        val upd-ty = (HOLogic.listT rty --> HOLogic.listT rty) --> sty --> sty  
        val rhs-ty = sty --> rty  
        val mty = StateMgt-core.MON-SE-T HOLogic.unitT sty  
    in Const(const-name <assign-local>, upd-ty --> rhs-ty --> mty) $ upd $ rhs end
```

```
fun mk-call-C opn args =  
    let val ty = fastype-of opn
```

```

val (argty, mty) = case ty of
  Type(fun, [argty, mty]) => (argty, mty)
  | - => error mk-call-C: illegal type for body
val sty = case mty of
  Type(fun, [sty, -]) => sty
  | - => error mk-call-C: illegal type for body 2
val args-ty = sty --> argty
in Const(const-name ⟨callC⟩, ty --> args-ty --> mty) $ opn $ args end

```

(* missing : a call-assign-local and a call-assign-global. Or define at HOL level ? *)

```

fun mk-if-C c B B' =
  let val ty = fastype-of B
      val ty-cond = case ty of
        Type(fun, [argty, -]) => argty --> HOLogic.boolT
        | - => error mk-if-C: illegal type for body
      in Const(const-name ⟨if-C⟩, ty-cond --> ty --> ty --> ty) $ c $ B $ B'
  end;

```

```

fun mk-while-C c B =
  let val ty = fastype-of B
      val ty-cond = case ty of
        Type(fun, [argty, -]) => argty --> HOLogic.boolT
        | - => error mk-while-C: illegal type for body
      in Const(const-name ⟨while-C⟩, ty-cond --> ty --> ty) $ c $ B
  end;

```

```

fun mk-while-anno-C inv f c B =
  (* no type-check on inv and measure f *)
  let val ty = fastype-of B
      val (ty-cond, ty-m) = case ty of
        Type(fun, [argty, -]) => ( argty --> HOLogic.boolT,
                                   argty --> HOLogic.natT)
        | - => error mk-while-anno-C: illegal type for body
      in Const(const-name ⟨while-C-A⟩, ty-cond --> ty-m --> ty-cond --> ty --> ty)
        $ inv $ f $ c $ B
  end;

```

```

fun mk-block-C push body pop =
  let val body-ty = fastype-of body
      val pop-ty = fastype-of pop
      val bty = body-ty --> body-ty --> pop-ty --> pop-ty
  in Const(const-name ⟨blockC⟩, bty) $ push $ body $ pop end

```

end;>

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 discipline — 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 exist.

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 alternately 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-by-step basis. Note that Clean does not possess the concept of call-by-reference parameters; consequently, the algorithm must be specialized to a variant where A is just a global variable.

2.2 Clean Encoding of the Global State of Quicksort

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

```
ML< val Type(s,t) = StateMgt-core.get-state-type-global @{\theory};
      StateMgt-core.get-state-field-tab-global @{\theory}; >
```

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

```
global-vars (S)
  A :: int list
ML<
(fst o StateMgt-core.get-data-global) @{\theory}
>
global-vars (S2)
  B :: int list
```

```
ML<
(Int.toString o length o Symtab.dest o fst o StateMgt-core.get-data-global) @{\theory}
>
```

```
find-theorems (60) name:global-state2-state
```

```
find-theorems create_L name:Quick
```

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

```
ML< val Type(Quicksort-concept.global-S2-state-scheme,t)
      = StateMgt-core.get-state-type-global @{\theory};
      (Int.toString o length o Symtab.dest)(StateMgt-core.get-state-field-tab-global @{\theory})>
```

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

A Simulation of Synthesis of Typed Assignment-Rules

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

lemma A_L' -control-indep : (break-status_L \bowtie $A_L' \wedge$ return-status_L \bowtie A_L')
unfolding A_L' -def break-status_L-def return-status_L-def create_L-def upd2put-def
by (simp add: lens-indep-def)

lemma A_L' -strong-indep : $\#!$ A_L'
unfolding strong-control-independence-def
using A_L' -control-indep **by** blast

Specialized Assignment Rule for Global Variable `A`. Note that this specialized rule of $\#$
 $?upd \implies \{\lambda\sigma. \triangleright \sigma \wedge ?P (?upd (\lambda-. ?rhs \sigma) \sigma)\} ?upd :=_G ?rhs \{\lambda r \sigma. \triangleright \sigma \wedge ?P \sigma\}$

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-global-A*:

```

  {λσ. ▷ σ ∧ P (σ(A := rhs σ))} A-update ::=G rhs {λr σ. ▷ σ ∧ P σ }
  apply(rule assign-global)
  apply(rule strong-vs-weak-upd [of global-S-state.A global-S-state.A-update])
  apply (metis AL'-def AL'-strong-indep)
  by(rule ext, rule ext, auto)

```

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_function`

pre $\langle i < length\ A \wedge j < length\ A \rangle$

post $\langle \lambda res. length\ A = length(old\ A) \wedge res = () \rangle$

local-vars $tmp :: int$

defines $\langle tmp := A ! i \rangle ;-$

$\langle A := list-update\ A\ i\ (A ! j) \rangle ;-$

$\langle A := list-update\ A\ j\ tmp \rangle$

value $\langle break-status = False, return-status = False, A = [1,2,3],$

$tmp = [], result-value = [], \dots = X \rangle$

term *swap*

find-theorems (70) *name:local-swap-state*

value *swap* (0,1) $\langle break-status = False, return-status = False, A = [1,2,3], B=[],$

$tmp = [],$

$result-value = [], \dots = X \rangle$

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

```

⟨defines ((assign-local tmp-update (λσ. (A σ) ! i)) ;-
  (assign-global A-update (λσ. list-update (A σ) (i) (A σ ! j))) ;-
  (assign-global A-update (λσ. list-update (A σ) (j) ((hd o tmp) σ))))⟩

```

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

term (i, j) — check that i and j are pointing to the constants defined before treating **function-spec**

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

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

thm *swap-pre-def*

thm *swap-post-def*

thm *swap-core-def*

thm *swap-def*

The state-management is in the following configuration:

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

2.3.2 A Simulation 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 $swap'-pre :: nat \times nat \Rightarrow 'a\ global-S-state-scheme \Rightarrow bool$
where $swap'-pre \equiv \lambda(i, j) \sigma. i < length (A \sigma) \wedge j < length (A \sigma)$

definition $swap'-post :: 'a \times 'b \Rightarrow 'c\ global-S-state-scheme \Rightarrow 'd\ global-S-state-scheme \Rightarrow unit \Rightarrow bool$
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 consequence 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 \langle
 $val\ Type(s,t) = StateMgt-core.get-state-type-global @\{theory\};$
 $val\ tab = StateMgt-core.get-state-field-tab-global @\{theory\};$
 $@\{term\ A :: ('a\ local-swap-state-scheme \Rightarrow int\ list)\}$ \rangle

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 $push-local-swap-state' \sigma =$
 $Some((), \sigma (local-swap'-state.tmp := undefined \# local-swap'-state.tmp \sigma))$

definition $pop\text{-}local\text{-}swap\text{-}state' :: (unit, 'a local\text{-}swap'\text{-}state\text{-}scheme) MON_{SE}$

where $pop\text{-}local\text{-}swap\text{-}state' \sigma =$
 $Some(hd(local\text{-}swap\text{-}state.result\text{-}value \sigma),$
 $\quad \text{--- recall : returns op value}$
 $\quad \text{--- which happens to be unit}$
 $\sigma(local\text{-}swap\text{-}state.tmp := tl(local\text{-}swap\text{-}state.tmp \sigma) \))$

definition $swap'\text{-}core :: nat \times nat \Rightarrow (unit, 'a local\text{-}swap'\text{-}state\text{-}scheme) MON_{SE}$

where $swap'\text{-}core \equiv (\lambda(i,j).$
 $\quad ((assign\text{-}local\ tmp\text{-}update (\lambda\sigma. A \sigma ! i)) \ ;\text{---}$
 $\quad (assign\text{-}global\ A\text{-}update (\lambda\sigma. list\text{-}update (A \sigma) (i) (A \sigma ! j))) \ ;\text{---}$
 $\quad (assign\text{-}global\ A\text{-}update (\lambda\sigma. list\text{-}update (A \sigma) (j) ((hd\ o\ tmp) \sigma))))$

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

definition $swap' :: nat \times nat \Rightarrow (unit, 'a local\text{-}swap'\text{-}state\text{-}scheme) MON_{SE}$

where $swap' \equiv \lambda(i,j). block_C\ push\text{-}local\text{-}swap\text{-}state' (swap'\text{-}core (i,j))\ pop\text{-}local\text{-}swap\text{-}state'$

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 $swap\text{-}opt :: nat \times nat \Rightarrow (unit, 'a global\text{-}S\text{-}state\text{-}scheme) MON_{SE}$

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

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

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 $tmp_L\text{-}control\text{-}indep : (break\text{-}status_L \bowtie tmp_L \wedge return\text{-}status_L \bowtie tmp_L)$

unfolding $tmp_L\text{-}def\ break\text{-}status_L\text{-}def\ return\text{-}status_L\text{-}def\ create_L\text{-}def\ upd2put\text{-}def$
by $(simp\ add: lens\text{-}indep\text{-}def)$

lemma $tmp_L\text{-}strong\text{-}indep : \#! tmp_L$

unfolding *strong-control-independence-def*
using *tmp_L-control-indep* **by** *blast*

Specialized Assignment Rule for Local Variable *tmp*. Note that this specialized rule of \sharp ($?upd \circ upd\text{-}hd$) $\implies \{\!\{ \lambda\sigma. \triangleright \sigma \wedge ?P ((?upd \circ upd\text{-}hd) (\lambda\text{-}. ?rhs \sigma) \sigma) \}\!\} ?upd ::=_L ?rhs \{\!\{ \lambda r \sigma. \triangleright \sigma \wedge ?P \sigma \}\!\}$ 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-swap'-state.tmp-update ::=L rhs
 $\{\!\{ \lambda r \sigma. \triangleright \sigma \wedge P \sigma \}\!\}$ 
apply(rule assign-local)
apply(rule strong-vs-weak-upd-list)
apply(rule tmpL-strong-indep[simplified tmpL-def])
by(rule ext, rule ext, auto)

```

2.4 Encoding partition in Clean

2.4.1 partition in High-level Notation

```

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

```

The body is a fancy syntax for :

```

 $\langle defines$ 
   $((assign\text{-}local\ pivot\text{-}update\ (\lambda\sigma. A\ \sigma ! hi)) ; -$ 
   $(assign\text{-}local\ i\text{-}update\ (\lambda\sigma. lo)) ; -$ 
   $(assign\text{-}local\ j\text{-}update\ (\lambda\sigma. lo)) ; -$ 
   $(while_C (\lambda\sigma. (hd\ o\ j)\ \sigma \leq hi - 1)$ 
     $do (if_C (\lambda\sigma. A\ \sigma ! (hd\ o\ j)\ \sigma < (hd\ o\ pivot)\ \sigma)$ 
       $then\ call_C (swap) (\lambda\sigma. ((hd\ o\ i)\ \sigma, (hd\ o\ j)\ \sigma)) ; -$ 
       $assign\text{-}local\ i\text{-}update (\lambda\sigma. ((hd\ o\ i)\ \sigma) + 1)$ 
     $)$ 

```

```

    else skipSE
    fi) ;−
    (assign-local j-update (λσ. ((hd o j) σ) + 1))
od) ;−
callC (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⟨ val Type(s,t) = StateMgt-core.get-state-type-global @{theory};
      StateMgt-core.get-state-field-tab-global @{theory}⟩

```

2.4.2 A Simulation of partition in elementary specification constructs:

Contract-Elements

definition *partition'-pre* $\equiv \lambda(lo, hi) \sigma. lo < length(A \sigma) \wedge hi < length(A \sigma)$

definition *partition'-post* $\equiv \lambda(lo, hi) \sigma_{pre} \sigma \text{ res. } length(A \sigma) = length(A \sigma_{pre}) \wedge res = 3$

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'* $\sigma = \text{Some}()$,
 $\sigma(\text{local-partition-state.pivot} := \text{undefined} \# \text{local-partition-state.pivot } \sigma,$
 $\text{local-partition-state.i} := \text{undefined} \# \text{local-partition-state.i } \sigma,$
 $\text{local-partition-state.j} := \text{undefined} \# \text{local-partition-state.j } \sigma,$

local-partition-state.result-value
 $:= \text{undefined} \# \text{local-partition-state.result-value } \sigma \text{))}$

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

Memory-Model

Independence of Control-Block:

Monadic Representation of the Body

definition *partition'-core* :: nat \times nat \Rightarrow (unit,'a local-partition'-state-scheme) MON_{SE}
where *partition'-core* $\equiv \lambda(lo,hi).$
 $((\text{assign-local pivot-update } (\lambda\sigma. A \sigma ! hi)) \text{ ;-}$
 $(\text{assign-local i-update } (\lambda\sigma. lo)) \text{ ;-}$
 $(\text{assign-local j-update } (\lambda\sigma. lo)) \text{ ;-}$
 $(\text{while}_C (\lambda\sigma. (hd \ o \ j) \ \sigma \leq hi - 1)$
 $\text{do } (\text{if}_C (\lambda\sigma. A \sigma ! (hd \ o \ j) \ \sigma < (hd \ o \ pivot)\sigma)$
 $\text{then } \text{call}_C (\text{swap}) (\lambda\sigma. ((hd \ o \ i) \ \sigma, (hd \ o \ j) \ \sigma)) \text{ ;-}$
 $\text{assign-local i-update } (\lambda\sigma. ((hd \ o \ i) \ \sigma) + 1)$
 else skip_{SE}
 $\text{fi})$
 $\text{od}) \text{ ;-}$
 $(\text{assign-local j-update } (\lambda\sigma. ((hd \ o \ j) \ \sigma) + 1)) \text{ ;-}$
 $\text{call}_C (\text{swap}) (\lambda\sigma. ((hd \ o \ i) \ \sigma, (hd \ o \ j) \ \sigma)) \text{ ;-}$
 $\text{assign-local result-value-update } (\lambda\sigma. (hd \ o \ i) \ \sigma)$
 $\text{— the meaning of the return stmt}$
 $)$

thm *partition-core-def*

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

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    ⟨λres::unit. ∀ i∈{lo .. hi}. ∀ j∈{lo .. hi}. i ≤ j ⟶ A!i ≤ A!j⟩
variant hi - lo
local-vars p :: nat
defines  ifC ⟨lo < hi⟩
           then (ptmp ← callC partition ⟨(lo, hi)⟩ ; assign-local p-update (λσ. ptmp)) ;-
                callC quicksort ⟨(lo, p - 1)⟩ ;-
                callC quicksort ⟨(lo, p + 1)⟩
           else skipSE
           fi

```

```

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

```

2.5.2 A Simulation of quicksort in elementary specification constructs:

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

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

```

local-vars-test (quicksort' unit)
  p :: nat

```

```

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

```

```

thm pop-local-quicksort'-state-def
thm push-local-quicksort'-state-def

```

```

definition push-local-quicksort-state' :: (unit, 'a local-quicksort'-state-scheme) MONSE
  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) MONSE
  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
  ⟨ifCLEAN ⟨lo < hi⟩ then
    p := partition(lo, hi) ;-
    quicksort(lo, p - 1) ;-
    quicksort(p + 1, hi)
  else Skip⟩

```

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

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

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

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

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

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* :

$$\{\lambda\sigma. \triangleright \sigma \wedge \text{quicksort-pre } (lo, hi)(\sigma) \wedge \sigma = \sigma_{pre} \}$$

$$\text{quicksort } (lo, hi)$$

$$\{\lambda r \sigma. \triangleright \sigma \wedge \text{quicksort-post}(lo, hi)(\sigma_{pre})(\sigma)(r) \}$$

oops

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      ⟨ $i < \text{length } A \wedge j < \text{length } A$ ⟩
post     ⟨ $\lambda res. \text{length } A = \text{length}(\text{old } A) \wedge res = ()$ ⟩
local-vars  $tmp :: int$ 
defines  ⟨ $tmp := A ! i$ ⟩ ;−
          ⟨ $A := \text{list-update } A \ i \ (A ! j)$ ⟩ ;−
          ⟨ $A := \text{list-update } A \ j \ tmp$ ⟩

```

function-spec *partition* ($lo::nat, hi::nat$) **returns** *nat*

```

pre      ⟨ $lo < \text{length } A \wedge hi < \text{length } A$ ⟩
post     ⟨ $\lambda res::nat. \text{length } A = \text{length}(\text{old } A) \wedge res = 3$ ⟩
local-vars  $pivot :: int$ 
           $i :: nat$ 
           $j :: nat$ 
defines  ⟨ $pivot := A ! hi$ ⟩ ;− ⟨ $i := lo$ ⟩ ;− ⟨ $j := lo$ ⟩ ;−
           $\text{while}_C \langle j \leq hi - 1 \rangle$ 
           $\text{do if}_C \langle A ! j < pivot \rangle$ 
             $\text{then call}_C \text{ swap } \langle (i, j) \rangle$  ;−
            ⟨ $i := i + 1$ ⟩
           $\text{else skip}_{SE}$ 
           $\text{fi}$  ;−
          ⟨ $j := j + 1$ ⟩
           $\text{od}$ ;−
           $\text{call}_C \text{ swap } \langle (i, j) \rangle$  ;−
           $\text{return}_{\text{local-partition-state.result-value-update}} \langle i \rangle$ 

```

thm *partition-core-def*

rec-function-spec *quicksort* ($lo::nat, hi::nat$) **returns** *unit*

```

pre      ⟨ $lo \leq hi \wedge hi < \text{length } A$ ⟩
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   $\text{if}_C \langle lo < hi \rangle$ 
           $\text{then } (p_{tmp} \leftarrow \text{call}_C \text{ partition } \langle (lo, hi) \rangle ; \text{assign-local } p\text{-update } (\lambda \sigma. p_{tmp}))$  ;−
           $\text{call}_C \text{ quicksort } \langle (lo, p - 1) \rangle$  ;−
           $\text{call}_C \text{ quicksort } \langle (lo, p + 1) \rangle$ 
           $\text{else skip}_{SE}$ 
           $\text{fi}$ 

```

thm *quicksort-core-def*

thm *quicksort-def*

thm *quicksort-pre-def*

thm *quicksort-post-def*

3.3 Possible Application Sketch

```

lemma quicksort-correct :
  {λσ. ▷ σ ∧ quicksort-pre (lo, hi)(σ) ∧ σ = σpre }
    quicksort (lo, hi)
  {λr σ. ▷ σ ∧ quicksort-post(lo, hi)(σpre)(σ)(r) }
  oops

```

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 ≤ a⟩
post        ⟨λres::int. (res + 1)2 > a ∧ a ≥ (res)2⟩
defines     (⟨tm := 1⟩ ;−
            ⟨sqsum := 1⟩ ;−
            ⟨i := 0⟩ ;−
            (whileSE ⟨sqsum ≤ a⟩ do
              ⟨i := i+1⟩ ;−
              ⟨tm := tm + 2⟩ ;−
              ⟨sqsum := tm + sqsum⟩
            od) ;−
            returnC 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⟨
  val Type(s,t) = StateMgt-core.get-state-type-global @{theory}
  val Type(u,v) = @{typ unit}
  ⟩

```

```

global-vars (state)
  tm    :: int
  i     :: int
  sqsum :: int

```

```

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

```

```

lemma tm-independent [simp]: ‡ tm-update
  unfolding control-independence-def by auto

```

```

lemma i-independent [simp]: ‡ i-update
  unfolding control-independence-def by auto

```

```

lemma sqsum-independent [simp]: ‡ sqsum-update
  unfolding control-independence-def by auto

```

3.4.3 Setting for Symbolic Execution

Some lemmas to reason about memory

```

lemma tm-simp : tm (σ(tm := t)) = t
  using [[simp-trace]] by simp

```

```

lemma tm-simp1 : tm (σ(sqsum := s)) = tm σ by simp

```

```

lemma tm-simp2 : tm (σ(i := s)) = tm σ by simp

```

```

lemma sqsum-simp : sqsum (σ(sqsum := s)) = s by simp

```

```

lemma sqsum-simp1 : sqsum (σ(tm := t)) = sqsum σ by simp

```

```

lemma sqsum-simp2 : sqsum (σ(i := t)) = sqsum σ by simp

```

```

lemma i-simp : i (σ(i := i')) = i' by simp

```

```

lemma i-simp1 : i (σ(tm := i')) = i σ by simp

```

```

lemma i-simp2 : i (σ(sqsum := i')) = i σ by simp

```

```

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

```

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

lemma *non-exec-assign-globalD'*:

assumes $\# \text{ upd}$

shows $\sigma \models \text{upd} ::=_G \text{rhs} ; - M \implies \triangleright \sigma \implies \text{upd} (\lambda-. \text{rhs } \sigma) \sigma \models M$

apply(*drule non-exec-assign-global'[THEN iffD1]*)

using *assms exec-stop-vs-control-independence* **apply** *blast*

by *auto*

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

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

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

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

3.4.4 A Symbolic Execution Simulation

lemma

assumes *non-exec-stop[simp]*: $\neg \text{exec-stop } \sigma_0$

and *pos* : $0 \leq (a::\text{int})$

and *annotated-program*:

$\sigma_0 \models \langle \text{tm} := 1 \rangle ; -$
 $\langle \text{sqsum} := 1 \rangle ; -$
 $\langle i := 0 \rangle ; -$
 $(\text{while}_{SE} \langle \text{sqsum} \leq a \rangle \text{ do}$
 $\quad \langle i := i+1 \rangle ; -$
 $\quad \langle \text{tm} := \text{tm} + 2 \rangle ; -$
 $\quad \langle \text{sqsum} := \text{tm} + \text{sqsum} \rangle$
 $\text{od}) ; -$
 $\text{assert}_{SE}(\lambda\sigma. \sigma = \sigma_R)$

shows $\sigma_R \models \text{assert}_{SE} \langle i^2 \leq a \wedge a < (i+1)^2 \rangle$

apply(*insert annotated-program*)

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

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

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

apply(*tactic dmatch-tac @ {context} [@ {thm exec-whileD}] 1*)

apply(*tactic ematch-tac @ {context} [@ {thm if-SE-execE''}] 1*)

apply(*simp-all only: memory-theory MonadSE.bind-assoc'*)

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

```

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

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

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

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

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

```

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

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

defer 1

oops

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

end

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 = (232::nat) - 1
```

```
function-spec isPrime(n :: nat) returns bool
pre      ⟨n ≤ Sqrt-UINT-MAX⟩
post     ⟨λres. res ↔ prime n⟩
local-vars i :: nat
defines  ifC ⟨n < 2⟩
         then returnlocal-isPrime-state.result-value-update ⟨False⟩
         else skipSE
fi ;−
⟨i := 2⟩ ;−
whileC ⟨i < Sqrt-UINT-MAX ∧ i*i ≤ n⟩
do ifC ⟨n mod i = 0⟩
   then returnlocal-isPrime-state.result-value-update ⟨False⟩
   else skipSE
   fi ;−
   ⟨i := i + 1⟩
od ;−
returnlocal-isPrime-state.result-value-update ⟨True⟩
```

```
find-theorems name:isPrime name:core
```

```
term⟨isPrime-core⟩
```

```
lemma XXX :
```

```
isPrime-core n ≡
  ifC (λσ. n < 2) then (returnresult-value-update (λσ. False))
  else skipSE fi ;−
  i-update ::=L (λσ. 2) ;−
  whileC (λσ. (hd◦i)σ < Sqrt-UINT-MAX ∧ (hd◦i)σ * (hd◦i)σ ≤ n)
```

```

do
  (ifC (λσ. n mod (hd ∘ i) σ = 0)
    then (returnresult-value-update (λσ. False))
    else skipSE fi ;−
  i-update ::=L (λσ. (hd ∘ i) σ + 1))
od ;−
returnresult-value-update (λσ. True)

```

by(simp add: isPrime-core-def)

lemma YYY:

```

isPrime n ≡ blockC push-local-isPrime-state
  (isPrime-core n)
  pop-local-isPrime-state

```

by(simp add: isPrime-def)

lemma isPrime-correct :

```

{λσ. ▷ σ ∧ isPrime-pre (n)(σ) ∧ σ = σpre }
  isPrime n
{λr σ. ▷ σ ∧ isPrime-post(n) (σpre)(σ)(r) }
oops

```

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
```

```
global-vars (2)
  tt :: int list
```

```
find-theorems (160) name:2 name:Linear
```

```
function-spec linearsearch (x::int, n::int) returns int
pre      ⟨ 0 ≤ n ∧ n < int(length t) ∧ sorted t ⟩
post     ⟨ λres::int. res = bool2int (∃ i ∈ {0 ..< length t}. t!i = x) ⟩
local-vars i :: int
defines  ⟨ i := 0 ⟩ ; - ⟨ tt := [] ⟩ ; -
        whileC ⟨ i < n ⟩
          do ifC ⟨ t!(nat i) < x ⟩
              then ⟨ i := i + 1 ⟩
              else returnC 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) MONSE = 'σ → ('o × 'σ)
```

```
definition bind-SE :: ('o, 'σ) MONSE ⇒ ('o ⇒ ('o', 'σ) MONSE) ⇒ ('o', 'σ) MONSE
where   bind-SE f g = (λσ. case f σ of None ⇒ None
                          | Some (out, σ') ⇒ g out σ')
```

```
notation bind-SE (bindSE)
```

```
syntax   (xsymbols)
  -bind-SE :: [pttrn, ('o, 'σ) MONSE, ('o', 'σ) MONSE] ⇒ ('o', 'σ) MONSE
  ((λ - ← -; -) [5, 8, 8] 8)
```

translations

```
x ← f; g == CONST bind-SE f (% x . g)
```

```
definition unit-SE :: 'o ⇒ ('o, 'σ) MONSE ((result -) 8)
```

```
where   unit-SE e = (λσ. Some(e, σ))
```

```
notation unit-SE (unitSE)
```

In the following, we prove the required Monad-laws

```
lemma bind-right-unit[simp]: (x ← m; result x) = m
```

```
  apply (simp add: unit-SE-def bind-SE-def)
```

```
  apply (rule ext)
```

```
  apply (case-tac m σ, simp-all)
```

```
done
```

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

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

6.1.2 Definition : More Operators and their Properties

definition *fail-SE* :: $(\prime o, \prime \sigma) \text{MON}_{SE}$

where *fail-SE* = $(\lambda \sigma. \text{None})$

notation *fail-SE* (*fail*_{SE})

definition *assert-SE* :: $(\prime \sigma \Rightarrow \text{bool}) \Rightarrow (\text{bool}, \prime \sigma) \text{MON}_{SE}$

where *assert-SE* *P* = $(\lambda \sigma. \text{if } P \ \sigma \text{ then } \text{Some}(\text{True}, \sigma) \text{ else } \text{None})$

notation *assert-SE* (*assert*_{SE})

definition *assume-SE* :: $(\prime \sigma \Rightarrow \text{bool}) \Rightarrow (\text{unit}, \prime \sigma) \text{MON}_{SE}$

where *assume-SE* *P* = $(\lambda \sigma. \text{if } \exists \sigma . P \ \sigma \text{ then } \text{Some}(()), \text{SOME } \sigma . P \ \sigma) \text{ else } \text{None})$

notation *assume-SE* (*assume*_{SE})

lemma *bind-left-fail-SE* [*simp*]: $(x \leftarrow \text{fail}_{SE}; P \ x) = \text{fail}_{SE}$

by (*simp add: fail-SE-def bind-SE-def*)

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

definition *bind-SE'* :: $(\prime \alpha, \prime \sigma) \text{MON}_{SE} \Rightarrow (\prime \beta, \prime \sigma) \text{MON}_{SE} \Rightarrow (\prime \beta, \prime \sigma) \text{MON}_{SE}$ (**infixr** ;- 10)

where $(f ;- g) = (- \leftarrow f ; g)$

lemma *bind-assoc'* [*simp*]: $((m ;- k);- h) = (m;- (k;- h))$

by(*simp add:bind-SE'-def*)

lemma *bind-left-unit'* [*simp*]: $((\text{result } c);- P) = P$

by (*simp add: bind-SE'-def*)

lemma *bind-left-fail-SE'* [*simp*]: $(\text{fail}_{SE};- P) = \text{fail}_{SE}$

by (*simp add: bind-SE'-def*)

lemma *bind-right-unit'* [*simp*]: $(m;- (\text{result } ())) = m$

by (*simp add: bind-SE'-def*)

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

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

oops

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

definition $try_SE :: ('o, 'σ) MON_{SE} \Rightarrow ('o \text{ option}, 'σ) MON_{SE} (try_{SE})$
where $try_{SE} \text{ ioprogram} = (\lambda \sigma. \text{case } \text{ioprogram } \sigma \text{ of}$
 $\quad None \Rightarrow Some(None, \sigma)$
 $\quad | Some(outs, \sigma') \Rightarrow Some(Some \text{ outs}, \sigma'))$

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 $alt_SE :: [('o, 'σ) MON_{SE}, ('o, 'σ) MON_{SE}] \Rightarrow ('o, 'σ) MON_{SE} \quad (\text{infixl } \sqcap_{SE} 10)$
where $(f \sqcap_{SE} g) = (\lambda \sigma. \text{case } f \sigma \text{ of } None \Rightarrow g \sigma$
 $\quad | Some H \Rightarrow Some H)$

definition $malt_SE :: ('o, 'σ) MON_{SE} \text{ list} \Rightarrow ('o, 'σ) MON_{SE}$
where $malt_SE S = \text{foldr } alt_SE S fail_{SE}$
notation $malt_SE (\sqcap_{SE})$

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

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

6.1.3 Definition : Programming Operators and their Properties

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

definition $if_SE :: ['σ \Rightarrow bool, ('α, 'σ) MON_{SE}, ('α, 'σ) MON_{SE}] \Rightarrow ('α, 'σ) MON_{SE}$
where $if_SE c E F = (\lambda \sigma. \text{if } c \sigma \text{ then } E \sigma \text{ else } F \sigma)$

syntax (*xsymbols*)
 $-if_SE :: ['σ \Rightarrow bool, ('o, 'σ) MON_{SE}, ('o', 'σ) MON_{SE}] \Rightarrow ('o', 'σ) MON_{SE}$
 $((if_{SE} - \text{then} - \text{else} - fi) [5, 8, 8] 8)$

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

6.1.4 Theory of a Monadic While

Prerequisites

fun $replicator :: [('a, 'σ) MON_{SE}, nat] \Rightarrow (unit, 'σ) MON_{SE} \quad (\text{infixr } \sim\sim\sim 60)$

where $f \overset{\sim}{\sim} 0 = (\text{result } ())$
 $| f \overset{\sim}{\sim} (\text{Suc } n) = (f ; - f \overset{\sim}{\sim} n)$

fun *replicator2* :: $[('a, ' \sigma) \text{MON}_{SE}, \text{nat}, ('b, ' \sigma) \text{MON}_{SE}] \Rightarrow ('b, ' \sigma) \text{MON}_{SE}$ (**infixr** $\overset{\sim}{\sim} 60$)
where $(f \overset{\sim}{\sim} 0) M = (M)$
 $| (f \overset{\sim}{\sim} (\text{Suc } n)) M = (f ; - ((f \overset{\sim}{\sim} n) M))$

First Step : Establishing an embedding between partial functions and relations

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

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

lemma *Mon2Rel-Rel2Mon-id*: **assumes** *det:single-valued R* **shows** $(\text{Mon2Rel} \circ \text{Rel2Mon}) R = R$

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

lemma *Rel2Mon-Id*: $(\text{Rel2Mon} \circ \text{Mon2Rel}) x = x$

apply (*rule ext*)
apply (*auto simp: comp-def Mon2Rel-def Rel2Mon-def*)
apply (*erule contrapos-pp, drule HOL.not-sym, simp*)
done

lemma *single-valued-Mon2Rel*: *single-valued* $(\text{Mon2Rel } B)$
by (*auto simp: single-valued-def Mon2Rel-def*)

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

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

lemma *chain-total*: $\text{chain } S \implies S i \leq S j \vee S j \leq S i$
by (*metis chain-def le-cases lift-Suc-mono-le*)

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

lemma *mono-if-cont*: **fixes** $f :: 'a \text{set} \Rightarrow 'b \text{set}$
assumes *cont f* **shows** *mono f*

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

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

lemma *chain-iterates*: fixes $f :: 'a \text{ set} \Rightarrow 'a \text{ set}$
 assumes *mono* f shows $\text{chain}(\lambda n. (f \sim^n) \{\})$

proof–
 { **fix** n have $(f \sim^n) \{\} \subseteq (f \sim^{Suc\ n}) \{\}$ using *assms*
 by (*induction n*) (*auto simp: mono-def*) }
 thus *?thesis* by (*auto simp: chain-def*)
 qed

theorem *lfp-if-cont*:

assumes *cont* f shows $\text{lfp } f = (\bigcup n. (f \sim^n) \{\})$ (*is - = ?U*)

proof

show $\text{lfp } f \subseteq ?U$

proof (*rule lfp-lowerbound*)

have $f ?U = (\text{UN } n. (f \sim^{Suc\ n}) \{\})$

using *chain-iterates*[*OF mono-if-cont*][*OF assms*] *assms*

by (*simp add: cont-def*)

also have $\dots = (f \sim^0) \{\} \cup \dots$ by *simp*

also have $\dots = ?U$

apply (*auto simp del: funpow.simps*)

by (*metis empty-iff funpow-0 old.nat.exhaust*)

finally show $f ?U \subseteq ?U$ by *simp*

qed

next

{ **fix** $n\ p$ assume $f p \subseteq p$

have $(f \sim^n) \{\} \subseteq p$

proof (*induction n*)

case 0 show *?case* by *simp*

next

case *Suc*

from *monoD*[*OF mono-if-cont*][*OF assms*] *Suc* $\langle f p \subseteq p \rangle$

show *?case* by *simp*

qed

}

thus $?U \subseteq \text{lfp } f$ by (*auto simp: lfp-def*)

qed

lemma *single-valued-UN-chain*:

assumes *chain* S (! n . *single-valued* ($S\ n$))

shows *single-valued* ($\text{UN } n. S\ n$)

proof (*auto simp: single-valued-def*)

fix $m\ n\ x\ y\ z$ assume $(x, y) \in S\ m$ $(x, z) \in S\ n$

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

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

Third Step: Definition of the Monadic While

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

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

syntax (*xsymbols*)
 $\text{-while-SE} :: ['\sigma \Rightarrow \text{bool}, (\text{unit}, '\sigma) \text{MON}_{SE}] \Rightarrow (\text{unit}, '\sigma) \text{MON}_{SE}$
 $((\text{while}_{SE} \text{ - do - od}) [8,8]8)$

translations
 $\text{while}_{SE} c \text{ do } b \text{ od} == \text{CONST while-SE } c \text{ b}$

lemma *cont- Γ* : $\text{cont } (\Gamma c b)$
by (*auto simp: cont-def Γ -def*)

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

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

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

lemma *Rel2Mon-homomorphism*:
assumes *determ-X*: $\text{single-valued } X$ **and** *determ-Y*: $\text{single-valued } Y$
shows $\text{Rel2Mon } (X \text{ O } Y) = ((\text{Rel2Mon } X) ; - (\text{Rel2Mon } Y))$
proof –

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

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

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

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

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

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

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

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

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

```

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

```

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, \dots, op_n with the inputs $\iota_1, \iota_2, \dots, \iota_n$ (outputs are treated analogously). Then we can encode for this interface the general input - type:

$$\text{datatype in} = op_1 :: \iota_1 \mid \dots \mid \iota_n$$

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

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

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

```

fun mbind :: ' $\iota$  list  $\Rightarrow$  (' $\iota \Rightarrow$  (' $o, \sigma$ ) MONSE)  $\Rightarrow$  (' $o$  list, ' $\sigma$ ) MONSE
where mbind [] iostep  $\sigma$  = Some([],  $\sigma$ )
  | mbind ( $a\#S$ ) iostep  $\sigma$  =
    (case iostep a  $\sigma$  of
      None  $\Rightarrow$  Some([],  $\sigma$ )
    | Some (out,  $\sigma'$ )  $\Rightarrow$  (case mbind S iostep  $\sigma'$  of
      None  $\Rightarrow$  Some([out],  $\sigma'$ )

```

$$| \text{Some}(\text{outs}, \sigma') \Rightarrow \text{Some}(\text{out} \# \text{outs}, \sigma'))$$

notation mbind ($\text{mbind}_{\text{FailSave}}$)

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*]:
 $\text{mbind } [] f = (\text{result } [])$
by(*rule ext, simp add: unit-SE-def*)

The characteristic property of $\text{mbind}_{\text{FailSave}}$ — which distinguishes it from mbind defined in the sequel — is that it never fails; it “swallows” internal errors occurring during the computation.

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

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

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

fun $\text{mbind}' :: 'l \text{ list} \Rightarrow ('l \Rightarrow ('o, 'σ) \text{MON}_{SE}) \Rightarrow ('o \text{ list}, 'σ) \text{MON}_{SE}$
where $\text{mbind}' [] \text{iostep } \sigma = \text{Some}([], \sigma) |$
 $\text{mbind}' (a \# S) \text{iostep } \sigma =$
 (*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' ($\text{mbind}_{\text{FailStop}}$)

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

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

declare $\text{mbind}'.$ *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'' :: 'l list ⇒ ('l ⇒ ('o,'σ) MONSE) ⇒ ('o list,'σ) MONSE
where mbind'' [] iostep σ = Some([], σ) |
      mbind'' (a#S) iostep σ =
        (case iostep a σ of
         None           ⇒ mbind'' S iostep σ
        | Some (out, σ') ⇒ (case mbind'' S iostep σ' of
                             None ⇒ None — does not occur
                            | Some(outs,σ'') ⇒ 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 ← mbind (a#S) F; M x) =
  (a' ← trySE(F a);
   if a' = None
   then (M [])
   else (x ← mbind S F; M (the a' # x)))
apply(rule ext)
apply(simp add: bind-SE-def try-SE-def)
apply(case-tac F a x, auto)
apply(simp add: bind-SE-def try-SE-def)
apply(case-tac mbind S F b, auto)
done

```

end

```

theory Symbex-MonadSE
imports Seq-MonadSE
begin

```

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 $m\text{bind}_{FailSave}$ -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 $\text{valid-SE} :: 'a \Rightarrow (\text{bool}, 'a) \text{MON}_{SE} \Rightarrow \text{bool}$ (**infix** \models 9)
where $(\sigma \models m) = (m \sigma \neq \text{None} \wedge \text{fst}(\text{the}(m \sigma)))$

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

Valid Execution Sequences and their Symbolic Execution

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Recall `mbind_unit` for the base case.

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

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

by(*auto simp: valid-mbind'-mt*)

lemma *valid-mbind''-mt* : $(\sigma \models (s \leftarrow \text{mbind}_{\text{FailPurge}} [] f; \text{unit}_{SE} (P s))) = P []$

by(*simp add: mbind''.simps valid-SE-def bind-SE-def unit-SE-def*)

lemma *valid-mbind''-mtE*: $\sigma \models (s \leftarrow \text{mbind}_{\text{FailPurge}} [] f; \text{unit}_{SE} (P s)) \implies (P [] \implies Q)$
 $\implies Q$

by(*auto simp: valid-mbind''-mt*)

lemma *exec-mbindFSave-failure*:

ioprogram a $\sigma = \text{None} \implies$

$(\sigma \models (s \leftarrow \text{mbind}_{\text{FailSave}} (a\#S) \text{ioprogram}; M s)) = (\sigma \models (M []))$

by(*simp add: valid-SE-def unit-SE-def bind-SE-def*)

lemma *exec-mbindFStop-failure*:

ioprogram a $\sigma = \text{None} \implies$

$(\sigma \models (s \leftarrow \text{mbind}_{\text{FailStop}} (a\#S) \text{ioprogram}; M s)) = (\text{False})$

by(*simp add: exec-bind-SE-failure*)

lemma *exec-mbindFPurge-failure*:

ioprogram a $\sigma = \text{None} \implies$

$(\sigma \models (s \leftarrow \text{mbind}_{\text{FailPurge}} (a\#S) \text{ioprogram}; M s)) =$

$(\sigma \models (s \leftarrow \text{mbind}_{\text{FailPurge}} S \text{ioprogram}; M s))$

by(*simp add: valid-SE-def unit-SE-def bind-SE-def mbind''.simps*)

lemma *exec-mbindFSave-success* :

ioprogram a $\sigma = \text{Some}(b, \sigma') \implies$

$(\sigma \models (s \leftarrow \text{mbind}_{\text{FailSave}} (a\#S) \text{ioprogram}; M s)) =$

$(\sigma' \models (s \leftarrow \text{mbind}_{\text{FailSave}} S \text{ioprogram}; M (b\#s)))$

unfolding *valid-SE-def unit-SE-def bind-SE-def*

by(*cases mbind_{FailSave} S ioprogram \sigma', auto*)

lemma *exec-mbindFStop-success* :

ioprogram a $\sigma = \text{Some}(b, \sigma') \implies$

$(\sigma \models (s \leftarrow \text{mbind}_{\text{FailStop}} (a\#S) \text{ioprogram}; M s)) =$

$(\sigma' \models (s \leftarrow \text{mbind}_{\text{FailStop}} S \text{ioprogram}; M (b\#s)))$

unfolding *valid-SE-def unit-SE-def bind-SE-def*

by(*cases mbind_{FailStop} S ioprogram \sigma', auto simp: mbind''.simps*)

lemma *exec-mbindFPurge-success* :

ioprogram a $\sigma = \text{Some}(b, \sigma') \implies$

$(\sigma \models (s \leftarrow \text{mbind}_{\text{FailPurge}} (a\#S) \text{ioprogram}; M s)) =$

$(\sigma' \models (s \leftarrow \text{mbind}_{\text{FailPurge}} S \text{ioprogram}; M (b\#s)))$

unfolding *valid-SE-def unit-SE-def bind-SE-def*

by(*cases mbind_{FailPurge} S ioprogram \sigma', auto simp: mbind''.simps*)

lemma *exec-mbindFSave*:

$(\sigma \models (s \leftarrow \text{mbind}_{\text{FailSave}} (a\#S) \text{ioprogram}; \text{return} (P s))) =$

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

lemma mbind-eq-sexec:

assumes * : $\bigwedge b \sigma'. f a \sigma = \text{Some}(b, \sigma') \implies$
 ($os \leftarrow$ 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)) σ
apply(cases f a $\sigma = \text{None}$)
apply(subst bind-SE-def, simp)
apply(subst bind-SE-def, simp)
apply auto
apply(subst bind-SE-def, simp)
apply(subst bind-SE-def, simp)
apply(simp add: *)
done

lemma mbind-eq-sexec':

assumes * : $\bigwedge b \sigma'. f a \sigma = \text{Some}(b, \sigma') \implies$
 (P (b)) $\sigma' =$ (P' (b)) σ'
shows ($a \leftarrow$ f a; P (a)) $\sigma =$
 ($a \leftarrow$ f a; P'(a)) σ
apply(cases f a $\sigma = \text{None}$)
apply(subst bind-SE-def, simp)
apply(subst bind-SE-def, simp)
apply auto
apply(subst bind-SE-def, simp)
apply(subst bind-SE-def, simp)
apply(simp add: *)
done

lemma mbind'-concat:

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

lemma assert-suffix-inv :

$\sigma \models$ (- \leftarrow mbind_{FailStop} xs istep; assert_{SE} (P))

$$\begin{aligned} &\implies \forall \sigma. P \sigma \longrightarrow (\sigma \models (- \leftarrow \text{istep } x; \text{assert}_{SE} (P))) \\ &\implies \sigma \models (- \leftarrow \text{mbind}_{FailStop} (xs @ [x]) \text{istep}; \text{assert}_{SE} (P)) \end{aligned}$$

apply(subst mbind'-concat, simp)
unfolding bind-SE-def assert-SE-def valid-SE-def
apply(auto split: option.split option.split-asm)
apply(case-tac aa, simp-all)
apply(case-tac P bb, simp-all)
apply (metis option.distinct(1))
apply(case-tac aa, simp-all)
apply(case-tac P bb, simp-all)
by (metis option.distinct(1))

Universal splitting and symbolic execution rule

lemma exec-mbindFSave-E:

assumes seq : $(\sigma \models (s \leftarrow \text{mbind}_{FailSave} (a\#S) \text{ioprogram}; (P s)))$
and none: $\text{ioprogram } a \sigma = \text{None} \implies (\sigma \models (P [])) \implies Q$
and some: $\bigwedge b \sigma'. \text{ioprogram } a \sigma = \text{Some}(b, \sigma') \implies (\sigma' \models (s \leftarrow \text{mbind}_{FailSave} S \text{ioprogram}; (P (b\#s)))) \implies Q$
shows Q
using seq
proof(cases ioprogram a σ)
case None **assume** ass:ioprogram a $\sigma = \text{None}$ **show** Q
apply(rule none[OF ass])
apply(insert ass, erule-tac ioprogram1=ioprogram **in** exec-mbindFSave-failure[THEN iffD1], rule seq)
done
next
case (Some aa) **assume** ass:ioprogram a $\sigma = \text{Some } aa$ **show** Q
apply(insert ass, cases aa, simp, rename-tac out σ')
apply(erule some)
apply(insert ass, simp)
apply(erule-tac ioprogram1=ioprogram **in** exec-mbindFSave-success[THEN iffD1], rule seq)
done
qed

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

lemma exec-mbindFStop-E:

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

```

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

```

lemma *exec-mbindFPurge-E*:

```

assumes seq : (σ ⊨ (s ← mbindFailPurge (a#S) ioprog ; (P s)))
  and none: ioprog a σ = None ⇒ (σ ⊨ (s ← mbindFailPurge S ioprog;(P (s)))) ⇒ Q
  and some: ∧ b σ'. ioprog a σ = Some(b,σ') ⇒ (σ' ⊨ (s ← mbindFailPurge S ioprog;(P
(b#s)))) ⇒ Q
shows Q
using seq
proof(cases ioprog a σ)
  case None assume ass:ioprog a σ = None show Q
    apply(rule none[OF ass])
    apply(insert ass, erule-tac ioprog1=ioprog in exec-mbindFPurge-failure[THEN iffD1],rule
seq)
  done
  next
    case (Some aa) assume ass:ioprog a σ = Some aa show Q
      apply(insert ass,cases aa,simp, rename-tac out σ')
      apply(erule some)
      apply(insert ass,simp)
      apply(erule-tac ioprog1=ioprog in exec-mbindFPurge-success[THEN iffD1],rule seq)
    done
  qed

```

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

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

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

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

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

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

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

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

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

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

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

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

term *if_{SE} P then B₁ else B₂ fi*

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

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

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

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

lemma *if-SE-split-asm* :

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

lemma if-SE-split-asm':

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

lemma if-SE-split:

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

lemma if-SE-split':

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

lemma if-SE-execE:

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

lemma if-SE-execE':

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

lemma exec-while :

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

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

lemma if-SE-execE'':

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

definition *opaque* ($x::\text{bool}$) = x

lemma *if-SE-execE''-pos*:

$\sigma \models (\text{if}_{SE} P \text{ then } B_1 \text{ else } B_2 \text{ fi}) ; - M$

$\implies (P \sigma \implies \sigma \models B_1 ; - M \implies Q)$

$\implies (\text{opaque } (\sigma \models (\text{if}_{SE} P \text{ then } B_1 \text{ else } B_2 \text{ fi}) ; - M) \implies Q)$

$\implies Q$

using *opaque-def* **by** *auto*

lemma [*code*]:

$(\sigma \models m) = (\text{case } (m \sigma) \text{ of } \text{None} \Rightarrow \text{False} \mid (\text{Some } (x,y)) \Rightarrow x)$

apply(*simp add: valid-SE-def*)

apply(*cases m \sigma = None, simp-all*)

apply(*insert not-None-eq, auto*)

done

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

oops

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

oops

lemma *monadic-sequence-rule*:

$\bigwedge X \sigma_1. (\sigma \models (- \leftarrow \text{assume}_{SE} (\lambda\sigma'. (\sigma=\sigma') \wedge P \sigma) ; x \leftarrow M ; \text{assert}_{SE} (\lambda\sigma. (x=X) \wedge (\sigma=\sigma_1) \wedge Q x \sigma)))$

\wedge
 $(\sigma_1 \models (- \leftarrow \text{assume}_{SE} (\lambda\sigma. (\sigma=\sigma_1) \wedge Q x \sigma) ; y \leftarrow M'; \text{assert}_{SE} (\lambda\sigma. R x y \sigma)))$

\implies

$\sigma \models (- \leftarrow \text{assume}_{SE} (\lambda\sigma'. (\sigma=\sigma') \wedge P \sigma) ; x \leftarrow M ; y \leftarrow M'; \text{assert}_{SE} (R x y))$

apply(*elim exE impE conjE*)

apply(*drule assume-D*)

apply(*elim exE impE conjE*)

unfolding *valid-SE-def assume-SE-def assert-SE-def bind-SE-def*

apply(*auto split: if-split HOL.if-split-asm Option.option.split Option.option.split-asm*)

apply (*metis (mono-tags, lifting) option.simps(3) someI-ex*)

oops

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

\implies

$\sigma \models (- \leftarrow \text{assume}_{SE} P ; x \leftarrow M ; \text{assert}_{SE} (\lambda\sigma. Q x \sigma))$

unfolding *valid-SE-def assume-SE-def assert-SE-def bind-SE-def*

by(*auto split: if-split HOL.if-split-asm Option.option.split Option.option.split-asm*)

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

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

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

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

lemma *mbindFailSave-vs-mbindFailStop*:
assumes *A*: $\forall \iota \in \text{set} \ \iota s. \forall \sigma. \text{ioprogram} \ \iota \ \sigma \neq \text{None}$
shows $(\sigma \models (os \leftarrow (\text{mbind}_{FailSave} \ \iota s \ \text{ioprogram}); P \ os)) =$
 $(\sigma \models (os \leftarrow (\text{mbind}_{FailStop} \ \iota s \ \text{ioprogram}); P \ os))$
proof(*insert A, erule rev-mp, induct* ιs)
case Nil show ?*case* **by** *simp*
next
case (*Cons a* ιs)
from *Cons.hyps*
have *B*: $\forall S f \sigma. \text{mbind}_{FailSave} \ S f \ \sigma \neq \text{None}$ **by** *simp*

have *C*: $(\forall \iota \in \text{set} \ \iota s. \forall \sigma. \text{ioprogram} \ \iota \ \sigma \neq \text{None})$
 $\longrightarrow (\forall \sigma. \text{mbind}_{FailStop} \ \iota s \ \text{ioprogram} \ \sigma = \text{mbind}_{FailSave} \ \iota s \ \text{ioprogram} \ \sigma)$
apply(*induct* $\iota s, \text{simp}$)


```

    apply(intro impI allI, rename-tac  $\sigma$ )
    apply(simp add: Seq-MonadSE.mbind'.simps(2))
    apply(insert A, erule-tac  $x=a$  in ballE)
    apply(erule-tac  $x=\sigma$  and  $P=\lambda\sigma . ioprogram a \sigma \neq None$  in allE)
    apply(auto split:option.split)
  done
show ?case
  apply(intro impI)
  by (smt (verit, best) C exec-mbindFSave-E exec-mbindFSave-success exec-mbindFStop-E
      exec-mbindFStop-success list.set-intros(1) list.set-intros(2) valid-bind-cong)
qed

```

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

```

```

lemma non-exec-assign' :
  assumes  $\triangleright \sigma$ 
  shows  $(\sigma \models (assign\ f; - M)) = ((f\ \sigma) \models M)$ 
  by (simp add: assign-def assms exec-bind-SE-success bind-SE'-def)

```

```

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

```

```

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

```

6.2.2 Assign Execution Rules.

lemma *non-exec-assign-global* :

assumes $\triangleright \sigma$

shows $(\sigma \models (- \leftarrow \text{assign-global upd rhs}; M)) = ((\text{upd } (\lambda-. \text{rhs } \sigma) \sigma) \models M)$

by (*simp add: assign-global-def non-exec-assign assms*)

lemma *non-exec-assign-global'* :

assumes $\triangleright \sigma$

shows $(\sigma \models (\text{assign-global upd rhs}; - M)) = ((\text{upd } (\lambda-. \text{rhs } \sigma) \sigma) \models M)$

by (*metis (full-types) assms bind-SE'-def non-exec-assign-global*)

lemma *exec-assign-global* :

assumes *exec-stop* σ

shows $(\sigma \models (- \leftarrow \text{assign-global upd rhs}; M)) = (\sigma \models M)$

by (*simp add: assign-global-def assign-def assms exec-bind-SE-success*)

lemma *exec-assign-global'* :

assumes *exec-stop* σ

shows $(\sigma \models (\text{assign-global upd rhs}; - M)) = (\sigma \models M)$

by (*simp add: assign-global-def assign-def assms exec-bind-SE-success bind-SE'-def*)

lemma *non-exec-assign-local* :

assumes $\triangleright \sigma$

shows $(\sigma \models (- \leftarrow \text{assign-local upd rhs}; M)) = ((\text{upd } (\text{upd-hd } (\lambda-. \text{rhs } \sigma)) \sigma) \models M)$

by (*simp add: assign-local-def non-exec-assign assms*)

lemma *non-exec-assign-local'* :

assumes $\triangleright \sigma$

shows $(\sigma \models (\text{assign-local upd rhs}; - M)) = ((\text{upd } (\text{upd-hd } (\lambda-. \text{rhs } \sigma)) \sigma) \models M)$

by (*metis assms bind-SE'-def non-exec-assign-local*)

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

lemma *exec-assign-local* :

assumes *exec-stop* σ

shows $(\sigma \models (- \leftarrow \text{assign-local upd rhs}; M)) = (\sigma \models M)$

by (*simp add: assign-local-def assign-def assms exec-bind-SE-success*)

lemma *exec-assign-local'* :

assumes *exec-stop* σ

shows $(\sigma \models (\text{assign-local upd rhs}; - M)) = (\sigma \models M)$

unfolding *assign-local-def assign-def*

by (*simp add: assms exec-bind-SE-success2*)

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

thm *exec-assignD*

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

thm *exec-assignD'*

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

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

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

thm *exec-assign-localD*

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

6.2.3 Basic Call Symbolic Execution Rules.

lemma *exec-call-0* :

assumes *exec-stop* σ

shows $(\sigma \models (- \leftarrow \text{call-0}_C M; M')) = (\sigma \models M')$

by (*simp add: assms call-0_C-def exec-bind-SE-success*)

lemma *exec-call-0'* :

assumes *exec-stop* σ

shows $(\sigma \models (\text{call-0}_C M; - M')) = (\sigma \models M')$

by (*simp add: assms bind-SE'-def exec-call-0*)

lemma *exec-call-1* :

assumes *exec-stop* σ

shows $(\sigma \models (x \leftarrow \text{call-1}_C M A_1; M' x)) = (\sigma \models M' \text{ undefined})$

by (*simp add: assms call-1_C-def call_C-def exec-bind-SE-success*)

lemma *exec-call-1'* :

assumes *exec-stop* σ

shows $(\sigma \models (\text{call-1}_C M A_1; - M')) = (\sigma \models M')$

by (*simp add: assms bind-SE'-def exec-call-1*)

lemma *exec-call* :

assumes *exec-stop* σ

shows $(\sigma \models (x \leftarrow \text{call}_C M A_1; M' x)) = (\sigma \models M' \text{ undefined})$

by (*simp add: assms call_C-def call-1_C-def exec-bind-SE-success*)

lemma *exec-call'* :

assumes *exec-stop* σ

shows $(\sigma \models (\text{call}_C M A_1; - M')) = (\sigma \models M')$

by (*metis assms call-1_C-def exec-call-1'*)

lemma *exec-call-2* :

assumes *exec-stop* σ

shows $(\sigma \models (- \leftarrow \text{call-2}_C M A_1 A_2; M')) = (\sigma \models M')$

by (*simp add: assms call-2_C-def exec-bind-SE-success*)

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

6.2.4 Basic Call Symbolic Execution Rules.

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

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

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

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

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

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

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

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

6.2.5 Conditional.

lemma *exec-If_C-If_{SE}* :

assumes $\triangleright \sigma$

shows $((\text{if}_C P \text{ then } B_1 \text{ else } B_2 \text{ fi}))\sigma = ((\text{if}_{SE} P \text{ then } B_1 \text{ else } B_2 \text{ fi}))\sigma$

unfolding *if-SE-def MonadSE.if-SE-def Symbex-MonadSE.valid-SE-def MonadSE.bind-SE'-def*
by (*simp add: assms bind-SE-def if-C-def*)

lemma *valid-exec-If_C* :

assumes $\triangleright \sigma$

shows $(\sigma \models (\text{if}_C P \text{ then } B_1 \text{ else } B_2 \text{ fi}); -M) = (\sigma \models (\text{if}_{SE} P \text{ then } B_1 \text{ else } B_2 \text{ fi}); -M)$

by (*meson assms exec-If_C-If_{SE} valid-bind'-cong*)

lemma *exec-If_C'* :

assumes *exec-stop* σ

shows $(\sigma \models (\text{if}_C P \text{ then } B_1 \text{ else } B_2 \text{ fi}); -M) = (\sigma \models M)$

unfolding *if-SE-def MonadSE.if-SE-def Symbex-MonadSE.valid-SE-def MonadSE.bind-SE'-def*
bind-SE-def

by (*simp add: assms if-C-def*)

lemma *exec-While_C'* :

assumes *exec-stop* σ

shows $(\sigma \models (\text{while}_C P \text{ do } B_1 \text{ od}); -M) = (\sigma \models M)$

unfolding *while-C-def MonadSE.if-SE-def Symbex-MonadSE.valid-SE-def MonadSE.bind-SE'-def*
bind-SE-def

apply *simp using assms by blast*

lemma *if_C-cond-cong* : $f \sigma = g \sigma \implies (\text{if}_C f \text{ then } c \text{ else } d \text{ fi}) \sigma =$
 $(\text{if}_C g \text{ then } c \text{ else } d \text{ fi}) \sigma$

unfolding *if-C-def*

by *simp*

6.2.6 Break - Rules.

lemma *break-assign-skip* [*simp*]: $(\text{break} ; - \text{assign } f) = \text{break}$

apply(*rule ext*)

unfolding *break-def assign-def exec-stop-def bind-SE'-def bind-SE-def*

by *auto*

lemma *break-if-skip* [*simp*]: $(\text{break} ; - \text{if}_C b \text{ then } c \text{ else } d \text{ fi}) = \text{break}$

apply(*rule ext*)

unfolding *break-def assign-def exec-stop-def if-C-def bind-SE'-def bind-SE-def*

by auto

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

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

lemma *return-cancel1-idem* [simp] :
 (return_X(E) ;− X ::=_G E' ;− M) = (return_C X E ;− M)
 apply(rule ext, rename-tac σ)
 unfolding unset-break-status-def bind-SE'-def bind-SE-def
 assign-def return_C-def return_C0-def assign-global-def assign-local-def
 apply(case-tac exec-stop σ)
 apply auto
 by (simp add: exec-stop-def set-return-status-def)

lemma *return-cancel2-idem* [simp] :
 (return_X(E) ;− X ::=_L E' ;− M) = (return_C X E ;− M)
 apply(rule ext, rename-tac σ)
 unfolding unset-break-status-def bind-SE'-def bind-SE-def
 assign-def return_C-def return_C0-def assign-global-def assign-local-def
 apply(case-tac exec-stop σ)
 apply auto
 by (simp add: exec-stop-def set-return-status-def)

6.2.7 While.

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

Various tactics for various coverage criteria

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

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

lemma *exec-while_C* :

```

( $\sigma \models ((\text{while}_C b \text{ do } c \text{ od}) ; - M)$ ) =
( $\sigma \models ((\text{if}_C b \text{ then } c ; - ((\text{while}_C b \text{ do } c \text{ od}) ; - \text{unset-break-status}) \text{ else skip}_{SE} fi) ; - M)$ )
proof (cases exec-stop  $\sigma$ )
  case True
  then show ?thesis
    by (simp add: True exec-IfC' exec-WhileC')
next
case False
then show ?thesis
  proof (cases  $\neg b \sigma$ )
    case True
    then show ?thesis
      apply(subst valid-bind'-cong)
      using  $\langle \neg \text{exec-stop } \sigma \rangle$  apply simp-all
      apply (auto simp: skipSE-def unit-SE-def)
      apply(subst while-C-def, simp)
      apply(subst bind'-cong)
      apply(subst MonadSE.while-SE-unfold)
      apply(subst ifSE-cond-cong [of - -  $\lambda$ -. False])
      apply simp-all
      apply(subst ifC-cond-cong [of - -  $\lambda$ -. False], simp add: )
      apply(subst exec-IfC-IfSE,simp-all)
      by (simp add: exec-stop-def unset-break-status-def)
    next
    case False
    have * :  $b \sigma$  using False by auto
    then show ?thesis
      unfolding while-k-def
      apply(subst while-C-def)
      apply(subst if-C-def)
      apply(subst valid-bind'-cong)
      apply (simp add:  $\langle \neg \text{exec-stop } \sigma \rangle$ )
      apply(subst (2) valid-bind'-cong)
      apply (simp add:  $\langle \neg \text{exec-stop } \sigma \rangle$ )
      apply(subst MonadSE.while-SE-unfold)
      apply(subst valid-bind'-cong)
      apply(subst bind'-cong)
      apply(subst ifSE-cond-cong [of - -  $\lambda$ -. True])
      apply(simp-all add:  $\langle \neg \text{exec-stop } \sigma \rangle$  )
      apply(subst bind-assoc', subst bind-assoc')
      proof(cases  $c \sigma$ )
        case None
      then show ( $\sigma \models c ; - ((\text{while}_{SE} (\lambda\sigma. \neg \text{exec-stop } \sigma \wedge b \sigma) \text{ do } c \text{ od}) ; - \text{unset-break-status}) ; - M$ )
    =
      ( $\sigma \models c ; - (\text{while}_C b \text{ do } c \text{ od}) ; - \text{unset-break-status} ; - M$ )
      by (simp add: bind-SE'-def exec-bind-SE-failure)
    next
    case (Some a)
    then show ( $\sigma \models c ; - ((\text{while}_{SE} (\lambda\sigma. \neg \text{exec-stop } \sigma \wedge b \sigma) \text{ do } c \text{ od}) ; - \text{unset-break-status}) ; - M$ )

```

```

=
      ( $\sigma \models c ; - (\text{while}_C b \text{ do } c \text{ od}) ; - \text{unset-break-status} ; - M$ )
apply(insert  $\langle c \ \sigma = \text{Some } a \rangle$ , subst (asm) surjective-pairing[of a])
apply(subst exec-bind-SE-success2, assumption)
apply(subst exec-bind-SE-success2, assumption)
proof(cases exec-stop (snd a))
  case True
then show (snd a  $\models ((\text{while}_{SE} (\lambda\sigma. \neg \text{exec-stop } \sigma \wedge b \ \sigma) \text{ do } c \text{ od}) ; - \text{unset-break-status}) ; - M$ ) =
  (snd a  $\models (\text{while}_C b \text{ do } c \text{ od}) ; - \text{unset-break-status} ; - M$ )
    by (metis (no-types, lifting) bind-assoc' exec-While_C' exec-skip if-SE-D2'
      skip_{SE}-def while-SE-unfold)
  next
    case False
then show (snd a  $\models ((\text{while}_{SE} (\lambda\sigma. \neg \text{exec-stop } \sigma \wedge b \ \sigma) \text{ do } c \text{ od}) ; - \text{unset-break-status}) ; - M$ ) =
  (snd a  $\models (\text{while}_C b \text{ do } c \text{ od}) ; - \text{unset-break-status} ; - M$ )
    unfolding while-C-def
    by(subst (?) valid-bind'-cong,simp)(simp)
  qed
qed
qed

```

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

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

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

```

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

```

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 hoare3 :: ('σ ⇒ bool) ⇒ ('α, 'σ)MONSE ⇒ ('α ⇒ 'σ ⇒ bool) ⇒ bool (({1-})/ (-)/
{(-)}) 50)
where {P} M {Q} ≡ (∀σ. P σ → (case M σ of None => False | Some(x, σ') => Q x σ'))
```

```
definition hoare3' :: ('σ ⇒ bool) ⇒ ('α, 'σ)MONSE ⇒ bool (({1-})/ (-)/†) 50)
where {P} M † ≡ (∀σ. P σ → (case M σ of None => True | - => False))
```

6.3.1 Basic rules

```
lemma skip: {P} skipSE {λ-. P}
  unfolding hoare3-def skip-SE-def unit-SE-def
  by auto
```

```
lemma fail: {P} failSE †
  unfolding hoare3'-def fail-SE-def unit-SE-def by auto
```

```
lemma assert: {P} assertSE P {λ - . True}
  unfolding hoare3-def assert-SE-def unit-SE-def
  by auto
```

```
lemma assert-conseq: Collect P ⊆ Collect Q ⇒ {P} assertSE Q {λ - . True}
  unfolding hoare3-def assert-SE-def unit-SE-def
```

by *auto*

lemma *assume-conseq*:

assumes $\exists \sigma. Q \sigma$
shows $\{P\} \text{assume}_{SE} Q \{\lambda -. Q\}$
unfolding *hoare3-def assume-SE-def unit-SE-def*
apply (*auto simp : someI2*)
using *assms* by *auto*

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

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* :

$\{P\} M \{\lambda x \sigma. x \in A \wedge Q x \sigma\}$
 $\implies \forall x \in A. \{Q x\} M' x \{R\}$
 $\implies \{P\} x \leftarrow M; M' x \{R\}$
unfolding *hoare3-def bind-SE-def*
by(*auto,erule-tac x=σ in allE, auto split: Option.option.split-asm Option.option.split*)

lemma *sequence-irpt-l* : $\{P\} M \dagger \implies \{P\} x \leftarrow M; M' x \dagger$

unfolding *hoare3'-def bind-SE-def*
by(*auto,erule-tac x=σ in allE, auto split: Option.option.split-asm Option.option.split*)

lemma *sequence-irpt-r* : $\{P\} M \{\lambda x \sigma. x \in A \wedge Q x \sigma\} \implies \forall x \in A. \{Q x\} M' x \dagger \implies \{P\} x \leftarrow M; M' x \dagger$

unfolding *hoare3'-def hoare3-def bind-SE-def*
by(*auto,erule-tac x=σ in allE, auto split: Option.option.split-asm Option.option.split*)

lemma *sequence'* : $\{P\} M \{\lambda -. Q\} \implies \{Q\} M' \{R\} \implies \{P\} M; - M' \{R\}$

unfolding *hoare3-def hoare3-def bind-SE-def bind-SE'-def*
by(*auto,erule-tac x=σ in allE, auto split: Option.option.split-asm Option.option.split*)

lemma *sequence-irpt-l'* : $\{P\} M \dagger \implies \{P\} M; - M' \dagger$

unfolding *hoare3'-def bind-SE-def bind-SE'-def*
by(*auto,erule-tac x=σ in allE, auto split: Option.option.split-asm Option.option.split*)

lemma *sequence-irpt-r'* : $\{P\} M \{\lambda -. Q\} \implies \{Q\} M' \dagger \implies \{P\} M; - M' \dagger$

unfolding *hoare3'-def hoare3-def bind-SE-def bind-SE'-def*
by(*auto,erule-tac x=σ in allE, auto split: Option.option.split-asm Option.option.split*)

6.3.3 Generalized and special consequence rules

lemma *consequence* :

Collect P \subseteq *Collect P'*
 $\implies \{P'\} M \{\lambda x \sigma. x \in A \wedge Q' x \sigma\}$
 $\implies \forall x \in A. \text{Collect}(Q' x) \subseteq \text{Collect}(Q x)$

$\implies \{P\} M \{\lambda x \sigma. x \in A \wedge Q x \sigma\}$
unfolding *hoare₃-def bind-SE-def*
by (*auto,erule-tac x=σ in allE,auto split: Option.option.split-asm Option.option.split*)

lemma *consequence-unit* :
assumes $(\bigwedge \sigma. P \sigma \longrightarrow P' \sigma)$
and $\{P'\} M \{\lambda x::unit. \lambda \sigma. Q' \sigma\}$
and $(\bigwedge \sigma. Q' \sigma \longrightarrow Q \sigma)$
shows $\{P\} M \{\lambda x \sigma. Q \sigma\}$

proof –

have $*$: $(\lambda x \sigma. Q \sigma) = (\lambda x::unit. \lambda \sigma. x \in UNIV \wedge Q \sigma)$ **by** *auto*
show *?thesis*
apply (*subst **)
apply (*rule-tac P' = P' and Q' = %- Q' in consequence*)
apply (*simp add: Collect-mono assms(1)*)
using *assms(2)* **apply** *auto[1]*
by (*simp add: Collect-mono assms(3)*)

qed

lemma *consequence-irpt* :
 $Collect P \subseteq Collect P'$
 $\implies \{P'\} M \dagger$
 $\implies \{P\} M \dagger$
unfolding *hoare₃-def hoare₃'-def bind-SE-def*
by (*auto*)

lemma *consequence-mt-swap* :
 $(\{\lambda-. False\} M \dagger) = (\{\lambda-. False\} M \{P\})$
unfolding *hoare₃-def hoare₃'-def bind-SE-def*
by *auto*

6.3.4 Condition rules

lemma *cond* :
 $\{\lambda \sigma. P \sigma \wedge cond \sigma\} M \{Q\}$
 $\implies \{\lambda \sigma. P \sigma \wedge \neg cond \sigma\} M' \{Q\}$
 $\implies \{P\} if_{SE} cond then M else M' fi \{Q\}$
unfolding *hoare₃-def hoare₃'-def bind-SE-def if-SE-def*
by *auto*

lemma *cond-irpt* :
 $\{\lambda \sigma. P \sigma \wedge cond \sigma\} M \dagger$
 $\implies \{\lambda \sigma. P \sigma \wedge \neg cond \sigma\} M' \dagger$
 $\implies \{P\} if_{SE} cond then M else M' fi \dagger$
unfolding *hoare₃-def hoare₃'-def bind-SE-def if-SE-def*
by *auto*

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

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

unfolding *hoare₃-def hoare₃'-def bind-SE-def if-SE-def*

proof *auto*

have $*$: $\forall n. \forall \sigma. P \ \sigma \wedge f \ \sigma \leq n \longrightarrow$
 $(\text{case } (\text{while}_{SE} \ \text{cond} \ \text{do} \ M \ \text{od}) \ \sigma \ \text{of}$
 $\quad \text{None} \Rightarrow \text{False}$
 $\quad | \ \text{Some } (x, \sigma') \Rightarrow \neg \text{cond } \sigma' \wedge P \ \sigma')$ (**is** $\forall n. ?P \ n$)

proof (*rule allI, rename-tac n, induct-tac n*)

fix n **show** $?P \ 0$

apply(*auto*)

apply(*subst while-SE-unfold*)

by (*metis (no-types, lifting) gr-implies-not0 if-SE-def measure option.case-eq-if option.sel option.simps(3) prod.sel(2) split-def unit-SE-def*)

next

fix n **show** $?P \ n \Longrightarrow ?P \ (\text{Suc } n)$

apply(*auto, subst while-SE-unfold*)

apply(*case-tac $\neg \text{cond } \sigma$*)

apply (*simp add: if-SE-def unit-SE-def*)

apply(*simp add: if-SE-def*)

apply(*case-tac $M \ \sigma = \text{None}$*)

using *measure* **apply** *blast*

proof (*auto simp: bind-SE'-def bind-SE-def*)

fix $\sigma \ \sigma'$

assume 1 : *cond* σ

and 2 : $M \ \sigma = \text{Some } ((), \sigma')$

and 3 : $P \ \sigma$

and 4 : $f \ \sigma \leq \text{Suc } n$

and *hyp* : $?P \ n$

have 5 : $P \ \sigma'$

by (*metis (no-types, lifting) * 1 2 3 case-prodD hoare₃-def option.simps(5)*)

have 6 : $\text{snd}(\text{the}(M \ \sigma)) = \sigma'$

by (*simp add: 2*)

have 7 : $\text{cond } \sigma' \Longrightarrow f \ \sigma' \leq n$

using $1 \ 3 \ 4 \ 6$ *leD measure* **by** *auto*

show $\text{case } (\text{while}_{SE} \ \text{cond} \ \text{do} \ M \ \text{od}) \ \sigma' \ \text{of} \ \text{None} \Rightarrow \text{False}$

$| \ \text{Some } (x, \sigma') \Rightarrow \neg \text{cond } \sigma' \wedge P \ \sigma'$

using $1 \ 3 \ 4 \ 5 \ 6$ *hyp measure* **by** *auto*

qed

qed

show $\bigwedge\sigma. P \ \sigma \Longrightarrow$

```

      case (whileSE cond do M od) σ of None ⇒ False
      | Some (x, σ') ⇒ ¬ cond σ' ∧ P σ'
using * by blast
qed

lemma while-irpt :
  assumes * : {λσ. cond σ ∧ P σ} M {λ-. P} ∨ {λσ. cond σ ∧ P σ} M †
  and measure: ∀σ. cond σ ∧ P σ → M σ = None ∨ f(snd(the(M σ))) < ((f σ)::nat)
  and enabled: ∀σ. P σ → cond σ
  shows {P} whileSE cond do M od †
unfolding hoare3-def hoare3'-def bind-SE-def if-SE-def
proof auto
  have * : ∀n. ∀σ. P σ ∧ f σ ≤ n →
    (case (whileSE cond do M od) σ of None ⇒ True | Some a ⇒ False)
    (is ∀n. ?P n)
  proof (rule allI, rename-tac n, induct-tac n)
    fix n
    have 1 : ∧σ. P σ ⇒ cond σ
    by (simp add: enabled *)
  show ?P 0
  apply(auto, frule 1)
  by (metis assms(2) bind-SE'-def bind-SE-def gr-implies-not0 if-SE-def option.case(1)
    option.case-eq-if while-SE-unfold)
next
  fix k n
  assume hyp : ?P n
  have 1 : ∧σ. P σ ⇒ cond σ
  by (simp add: enabled *)
  show ?P (Suc n)
  apply(auto, frule 1)
  apply(subst while-SE-unfold, auto simp: if-SE-def)
  proof(insert *, simp-all add: hoare3-def hoare3'-def, erule disjE)
    fix σ
    assume P σ
    and f σ ≤ Suc n
    and cond σ
    and ** : ∀σ. cond σ ∧ P σ → (case M σ of None ⇒ False | Some (x, σ') ⇒ P σ')
    obtain (case M σ of None ⇒ False | Some (x, σ') ⇒ P σ')
    by (simp add: ** ⟨P σ⟩ ⟨cond σ⟩)
  then
  show case (M ;− (whileSE cond do M od)) σ of None ⇒ True | Some a ⇒ False
  apply(case-tac M σ, auto, rename-tac σ', simp add: bind-SE'-def bind-SE-def)
  proof −
    fix σ'
    assume P σ'
    and M σ = Some ((), σ')
    have cond σ' by (simp add: ⟨P σ'⟩ enabled)
    have f σ' ≤ n

```

```

using ⟨ $M \sigma = \text{Some } ((), \sigma')$ ⟩ ⟨ $P \sigma$ ⟩ ⟨ $\text{cond } \sigma$ ⟩ ⟨ $f \sigma \leq \text{Suc } n$ ⟩ measure by fastforce

show case (whileSE cond do M od)  $\sigma'$  of None  $\Rightarrow$  True | Some a  $\Rightarrow$  False
  using hyp by (simp add: ⟨ $P \sigma'$ ⟩ ⟨ $f \sigma' \leq n$ ⟩)
qed
next
fix  $\sigma$ 
assume  $P \sigma$ 
and  $f \sigma \leq \text{Suc } n$ 
and  $\text{cond } \sigma$ 
and  $*$  :  $\forall \sigma. \text{cond } \sigma \wedge P \sigma \longrightarrow (\text{case } M \sigma \text{ of } \text{None} \Rightarrow \text{True} \mid \text{Some } a \Rightarrow \text{False})$ 
obtain  $**$  : (case  $M \sigma$  of None  $\Rightarrow$  True | Some a  $\Rightarrow$  False)
  by (simp add:  $*$  ⟨ $P \sigma$ ⟩ ⟨ $\text{cond } \sigma$ ⟩)
have  $M \sigma = \text{None}$ 
  by (simp add:  $**$  option.disc-eq-case(1))
show case ( $M ; -$  (whileSE cond do M od))  $\sigma$  of None  $\Rightarrow$  True | Some a  $\Rightarrow$  False
  by (simp add: ⟨ $M \sigma = \text{None}$ ⟩ bind-SE'-def bind-SE-def)
qed
qed
show  $\bigwedge \sigma. P \sigma \Longrightarrow \text{case } (\text{while}_{SE} \text{ cond do } M \text{ od}) \sigma \text{ of } \text{None} \Rightarrow \text{True} \mid \text{Some } a \Rightarrow \text{False}$  using
 $*$  by blast
qed

```

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

definition $\text{hoare}_1 :: ('\sigma \Rightarrow \text{bool}) \Rightarrow ('\alpha, '\sigma) \text{MON}_{SE} \Rightarrow ('\alpha \Rightarrow '\sigma \Rightarrow \text{bool}) \Rightarrow \text{bool}$ (\vdash_1 ($\{(1-)\}/(-)/\{(1-)\}$) 50)

where ($\vdash_1\{P\} M \{Q\}$) = ($\forall \sigma. (\sigma \models (- \leftarrow \text{assume}_{SE} P ; x \leftarrow M ; \text{assert}_{SE} (Q x)))$)

definition $\text{hoare}_2 :: ('\sigma \Rightarrow \text{bool}) \Rightarrow ('\alpha, '\sigma) \text{MON}_{SE} \Rightarrow ('\alpha \Rightarrow '\sigma \Rightarrow \text{bool}) \Rightarrow \text{bool}$ (\vdash_2 ($\{(1-)\}/(-)/\{(1-)\}$) 50)

where ($\vdash_2\{P\} M \{Q\}$) = ($\forall \sigma. P \sigma \longrightarrow (\sigma \models (x \leftarrow M ; \text{assert}_{SE} (Q x)))$)

end

```

theory Hoare-Clean
  imports Hoare-MonadSE
    Clean
begin

```

6.3.7 Clean Control Rules

lemma *break1*:

$\{\lambda\sigma. P (\sigma \langle \text{break-status} := \text{True} \rangle)\} \text{break} \{\lambda r \sigma. P \sigma \wedge \text{break-status } \sigma\}$
unfolding *hoare3-def break-def unit-SE-def by auto*

lemma *unset-break1*:

$\{\lambda\sigma. P (\sigma \langle \text{break-status} := \text{False} \rangle)\} \text{unset-break-status} \{\lambda r \sigma. P \sigma \wedge \neg \text{break-status } \sigma\}$
unfolding *hoare3-def unset-break-status-def unit-SE-def by auto*

lemma *set-return1*:

$\{\lambda\sigma. P (\sigma \langle \text{return-status} := \text{True} \rangle)\} \text{set-return-status} \{\lambda r \sigma. P \sigma \wedge \text{return-status } \sigma\}$
unfolding *hoare3-def set-return-status-def unit-SE-def by auto*

lemma *unset-return1*:

$\{\lambda\sigma. P (\sigma \langle \text{return-status} := \text{False} \rangle)\} \text{unset-return-status} \{\lambda r \sigma. P \sigma \wedge \neg \text{return-status } \sigma\}$
unfolding *hoare3-def unset-return-status-def unit-SE-def by auto*

6.3.8 Clean Skip Rules

lemma *assign-global-skip*:

$\{\lambda\sigma. \text{exec-stop } \sigma \wedge P \sigma\} \text{upd} ::=_G \text{rhs} \{\lambda r \sigma. \text{exec-stop } \sigma \wedge P \sigma\}$
unfolding *hoare3-def skip_{SE}-def unit-SE-def*
by (*simp add: assign-def assign-global-def*)

lemma *assign-local-skip*:

$\{\lambda\sigma. \text{exec-stop } \sigma \wedge P \sigma\} \text{upd} ::=_L \text{rhs} \{\lambda r \sigma. \text{exec-stop } \sigma \wedge P \sigma\}$
unfolding *hoare3-def skip_{SE}-def unit-SE-def*
by (*simp add: assign-def assign-local-def*)

lemma *return-skip*:

$\{\lambda\sigma. \text{exec-stop } \sigma \wedge P \sigma\} \text{return}_C \text{ upd rhs} \{\lambda r \sigma. \text{exec-stop } \sigma \wedge P \sigma\}$
unfolding *hoare3-def return_C-def return_C0-def unit-SE-def assign-local-def assign-def*
bind-SE'-def bind-SE-def
by auto

lemma *assign-clean-skip*:

$\{\lambda\sigma. \text{exec-stop } \sigma \wedge P \sigma\} \text{assign tr} \{\lambda r \sigma. \text{exec-stop } \sigma \wedge P \sigma\}$
unfolding *hoare3-def skip_{SE}-def unit-SE-def*
by (*simp add: assign-def assign-def*)

lemma *if-clean-skip*:

$\{\lambda\sigma. \text{exec-stop } \sigma \wedge P \sigma\} \text{if}_C \text{ C then E else F fi} \{\lambda r \sigma. \text{exec-stop } \sigma \wedge P \sigma\}$
unfolding *hoare3-def skip_{SE}-def unit-SE-def if-SE-def*
by (*simp add: if-C-def*)

lemma *while-clean-skip*:

$\{\lambda\sigma. \text{exec-stop } \sigma \wedge P \sigma\} \text{while}_C \text{ cond do body od} \{\lambda r \sigma. \text{exec-stop } \sigma \wedge P \sigma\}$
unfolding *hoare3-def skip_{SE}-def unit-SE-def while-C-def*
by auto

lemma *if-opcall-skip*:

$\{\lambda\sigma. \text{exec-stop } \sigma \wedge P \sigma\} (\text{call}_C M A_1) \{\lambda r \sigma. \text{exec-stop } \sigma \wedge P \sigma\}$
unfolding *hoare3-def skip_{SE}-def unit-SE-def call_C-def*
by *simp*

lemma *if-funcall-skip*:

$\{\lambda\sigma. \text{exec-stop } \sigma \wedge P \sigma\} (p_{tmp} \leftarrow \text{call}_C \text{fun } E ; \text{assign-local upd } (\lambda\sigma. p_{tmp})) \{\lambda r \sigma. \text{exec-stop } \sigma \wedge P \sigma\}$
unfolding *hoare3-def skip_{SE}-def unit-SE-def call_C-def assign-local-def assign-def*
by (*simp add: bind-SE-def*)

lemma *if-funcall-skip'*:

$\{\lambda\sigma. \text{exec-stop } \sigma \wedge P \sigma\} (p_{tmp} \leftarrow \text{call}_C \text{fun } E ; \text{assign-global upd } (\lambda\sigma. p_{tmp})) \{\lambda r \sigma. \text{exec-stop } \sigma \wedge P \sigma\}$
unfolding *hoare3-def skip_{SE}-def unit-SE-def call_C-def assign-global-def assign-def*
by (*simp add: bind-SE-def*)

6.3.9 Clean Assign Rules

lemma *assign-global*:

assumes $*$: $\# \text{ upd}$
shows $\{\lambda\sigma. \triangleright \sigma \wedge P (\text{upd } (\lambda-. \text{rhs } \sigma) \sigma)\} \text{upd} ::=_G \text{rhs} \{\lambda r \sigma. \triangleright \sigma \wedge P \sigma\}$
unfolding *hoare3-def skip_{SE}-def unit-SE-def assign-global-def assign-def*
by (*auto simp: assms*)

find-theorems $\# -$

lemma *assign-local*:

assumes $*$: $\# (\text{upd} \circ \text{upd-hd})$
shows $\{\lambda\sigma. \triangleright \sigma \wedge P ((\text{upd} \circ \text{upd-hd}) (\lambda-. \text{rhs } \sigma) \sigma)\} \text{upd} ::=_L \text{rhs} \{\lambda r \sigma. \triangleright \sigma \wedge P \sigma\}$
unfolding *hoare3-def skip_{SE}-def unit-SE-def assign-local-def assign-def*
using *assms exec-stop-vs-control-independence* **by** *fastforce*

lemma *return-assign*:

assumes $*$: $\# (\text{upd} \circ \text{upd-hd})$
shows $\{\lambda\sigma. \triangleright \sigma \wedge P ((\text{upd} \circ \text{upd-hd}) (\lambda-. \text{rhs } \sigma) (\sigma \langle \text{return-status} := \text{True} \rangle))\}$
 $\text{return}_{\text{upd}}(\text{rhs})$
 $\{\lambda r \sigma. P \sigma \wedge \text{return-status } \sigma\}$
unfolding *return_C-def return_{C0}-def hoare3-def skip_{SE}-def unit-SE-def assign-local-def assign-def*
set-return-status-def bind-SE'-def bind-SE-def

proof (*auto*)

fix $\sigma :: 'b \text{ control-state-scheme}$

assume *a1*: $P (\text{upd} (\text{upd-hd } (\lambda-. \text{rhs } \sigma)) (\sigma \langle \text{return-status} := \text{True} \rangle))$

assume $\triangleright \sigma$

show $P (\text{upd} (\text{upd-hd } (\lambda-. \text{rhs } \sigma)) (\sigma \langle \text{return-status} := \text{True} \rangle))$

using *a1 assms exec-stop-vs-control-independence'* **by** *fastforce*

qed

6.3.10 Clean Construct Rules

lemma *cond-clean* :

$$\begin{aligned} & \{\lambda\sigma. \triangleright \sigma \wedge P \ \sigma \wedge \text{cond } \sigma\} M \ \{Q\} \\ \implies & \{\lambda\sigma. \triangleright \sigma \wedge P \ \sigma \wedge \neg \text{cond } \sigma\} M' \ \{Q\} \\ \implies & \{\lambda\sigma. \triangleright \sigma \wedge P \ \sigma\} \text{if}_C \text{ cond then } M \text{ else } M' \text{ fi} \ \{Q\} \\ \text{unfolding } & \text{hoare}_3\text{-def hoare}_3'\text{-def bind-SE-def if-SE-def} \\ \text{by (simp add: } & \text{if-C-def)} \end{aligned}$$

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 $\llbracket \{\lambda\sigma. ?\text{cond } \sigma \wedge ?P \ \sigma\} ?M \ \{\lambda-. ?P\}; \forall \sigma. ?\text{cond } \sigma \wedge ?P \ \sigma \longrightarrow ?M \ \sigma \neq \text{None} \wedge ?f(\text{snd}(\text{the}(\ ?M \ \sigma))) < ?f \ \sigma \rrbracket \implies \{\ ?P\} \text{-while-SE } ?\text{cond } ?M \ \{\lambda-. \sigma. \neg ?\text{cond } \sigma \wedge ?P \ \sigma\}$.

lemma *while-clean-no-break* :

$$\begin{aligned} \text{assumes } & * : \{\lambda\sigma. \neg \text{break-status } \sigma \wedge \text{cond } \sigma \wedge P \ \sigma\} M \ \{\lambda-. \lambda\sigma. \neg \text{break-status } \sigma \wedge P \ \sigma\} \\ \text{and measure: } & \forall \sigma. \neg \text{exec-stop } \sigma \wedge \text{cond } \sigma \wedge P \ \sigma \\ & \longrightarrow M \ \sigma \neq \text{None} \wedge f(\text{snd}(\text{the}(M \ \sigma))) < ((f \ \sigma)::\text{nat}) \\ \text{(is } & \forall \sigma. - \wedge \text{cond } \sigma \wedge P \ \sigma \longrightarrow ?\text{decrease } \sigma) \\ \text{shows } & \{\lambda\sigma. \triangleright \sigma \wedge P \ \sigma\} \\ & \text{while}_C \text{ cond do } M \text{ od} \\ & \{\lambda-. \sigma. (\text{return-status } \sigma \vee \neg \text{cond } \sigma) \wedge \neg \text{break-status } \sigma \wedge P \ \sigma\} \\ \text{(is } & \{\ ?pre\} \text{while}_C \text{ cond do } M \text{ od } \{\lambda-. \sigma. ?post1 \ \sigma \wedge ?post2 \ \sigma\}) \\ \text{unfolding } & \text{while-C-def hoare}_3\text{-def hoare}_3'\text{-def} \\ \text{proof (simp add: } & \text{hoare}_3\text{-def[symmetric],rule sequence')} \\ \text{show } & \{\ ?pre\} \\ & \text{while}_{SE} (\lambda\sigma. \triangleright \sigma \wedge \text{cond } \sigma) \text{ do } M \text{ od} \\ & \{\lambda-. \sigma. \neg (\triangleright \sigma \wedge \text{cond } \sigma) \wedge \neg \text{break-status } \sigma \wedge P \ \sigma\} \\ \text{(is } & \{\ ?pre\} \text{while}_{SE} ?\text{cond}' \text{ do } M \text{ od } \{\lambda-. \sigma. \neg (?\text{cond}' \ \sigma) \wedge ?post2 \ \sigma\}) \\ \text{proof (rule } & \text{consequence-unit)} \\ \text{fix } \sigma \text{ show } & ?pre \ \sigma \longrightarrow ?post2 \ \sigma \text{ using exec-stop1 by blast} \\ \text{next} & \\ \text{show } & \{\ ?post2\} \text{while}_{SE} ?\text{cond}' \text{ do } M \text{ od} \{\lambda x \ \sigma. \neg (?\text{cond}' \ \sigma) \wedge ?post2 \ \sigma\} \\ \text{proof (rule-tac } & f = f \text{ in while, rule consequence-unit)} \\ \text{fix } \sigma \text{ show } & ?\text{cond}' \ \sigma \wedge ?post2 \ \sigma \longrightarrow \neg \text{break-status } \sigma \wedge \text{cond } \sigma \wedge P \ \sigma \text{ by simp} \\ \text{next} & \\ \text{show } & \{\lambda\sigma. \neg \text{break-status } \sigma \wedge \text{cond } \sigma \wedge P \ \sigma\} M \ \{\lambda x \ \sigma. ?post2 \ \sigma\} \text{ using } * \text{ by blast} \\ \text{next} & \\ \text{fix } \sigma \text{ show } & ?post2 \ \sigma \longrightarrow ?post2 \ \sigma \text{ by blast} \\ \text{next} & \\ \text{show } & \forall \sigma. ?\text{cond}' \ \sigma \wedge ?post2 \ \sigma \longrightarrow ?\text{decrease } \sigma \text{ using measure by blast} \\ \text{qed} & \\ \text{next} & \\ \text{fix } \sigma \text{ show } & \neg ?\text{cond}' \ \sigma \wedge ?post2 \ \sigma \longrightarrow \neg ?\text{cond}' \ \sigma \wedge ?post2 \ \sigma \text{ by blast} \\ \text{qed} & \\ \text{next} & \\ \text{show } & \{\lambda\sigma. \neg (\triangleright \sigma \wedge \text{cond } \sigma) \wedge ?post2 \ \sigma\} \text{unset-break-status} \end{aligned}$$

```

    {λ- σ'. (return-status σ' ∨ ¬ cond σ') ∧ ?post2 σ'}
    (is {λσ. ¬ (?cond'' σ) ∧ ?post2 σ} unset-break-status {λ- σ'. ?post3 σ' ∧ ?post2 σ' })
proof (rule consequence-unit)
  fix σ
  show ¬ ?cond'' σ ∧ ?post2 σ → (λσ. P σ ∧ ?post3 σ) (σ(|break-status := False|))
    by (metis (full-types) exec-stop-def surjective update-convs(1))
next
  show {λσ. (λσ. P σ ∧ ?post3 σ) (σ(|break-status := False|))}
    unset-break-status
    {λx σ. ?post3 σ ∧ ¬ break-status σ ∧ P σ}
  apply(subst (2) conj-commute,subst conj-assoc,subst (2) conj-commute)
  by(rule unset-break1)
next
  fix σ show ?post3 σ ∧ ?post2 σ → ?post3 σ ∧ ?post2 σ by simp
qed
qed

```

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

lemma *while-clean'*:

```

assumes M-inv : {λσ. ▷ σ ∧ cond σ ∧ P σ} M {λ-. P}
and cond-idpc : ∀x σ. (cond (σ(|break-status := x|))) = cond σ
and inv-idpc : ∀x σ. (P (σ(|break-status := x|))) = P σ
and f-is-measure : ∀σ. ▷ σ ∧ cond σ ∧ P σ → M σ ≠ None ∧ f(snd(the(M σ))) < ((f
σ)::nat)
shows {λσ. ▷ σ ∧ P σ} whileC cond do M od {λ- σ. ¬ break-status σ ∧ P σ}
unfolding while-C-def hoare3-def hoare3'-def
proof (simp add: hoare3-def[symmetric], rule sequence')
  show {λσ. ▷ σ ∧ P σ}
    whileSE (λσ. ▷ σ ∧ cond σ) do M od
    {λ- σ. P (σ(|break-status := False|))}
  apply(rule consequence-unit, rule impI, erule conjunct2)
  apply(rule-tac f = f in while)
  using M-inv f-is-measure inv-idpc by auto
next
  show {λσ. P (σ(|break-status := False|))} unset-break-status
    {λx σ. ¬ break-status σ ∧ P σ}
  apply(subst conj-commute)
  by(rule Hoare-Clean.unset-break1)
qed

```

Consequence and Sequence rules were 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.