

# **Clean - An Abstract Imperative Programming Language and its Theory**

Frédéric Tuong      Burkhart Wolff  
(with Contributions by Chantal Keller)

February 23, 2021

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



## Abstract

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

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

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



# Contents

<b>1</b>	<b>The Clean Language</b>	<b>7</b>
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. . . . .	10
1.1.4	The Assignment Operations (embedded in State-Exception Monad) . . . . .	10
1.1.5	Example for a Local Variable Space . . . . .	11
1.2	Global and Local State Management via Extensible Records . . . . .	12
1.2.1	Block-Structures . . . . .	12
1.2.2	Call Semantics . . . . .	13
1.3	Some Term-Coding Functions . . . . .	14
1.4	Syntactic Sugar supporting $\lambda$ -lifting for Global and Local Variables . . . . .	14
1.5	Support for (direct recursive) Clean Function Specifications . . . . .	14
1.6	The Rest of Clean: Break/Return aware Version of If, While, etc. . . . .	15
<b>2</b>	<b>Clean Semantics : A Coding-Concept Example</b>	<b>17</b>
2.1	The Quicksort Example . . . . .	17
2.2	Clean Encoding of the Global State of Quicksort . . . . .	18
2.3	Encoding swap in Clean . . . . .	18
2.3.1	swap in High-level Notation . . . . .	18
2.3.2	A Simulation of swap in elementary specification constructs: . . . . .	19
2.4	Encoding partition in Clean . . . . .	19
2.4.1	partition in High-level Notation . . . . .	19
2.4.2	A Simulation of partition in elementary specification constructs: . . . . .	20
2.5	Encoding the toplevel : quicksort in Clean . . . . .	22
2.5.1	quicksort in High-level Notation . . . . .	22
2.5.2	A Simulation of quicksort in elementary specification constructs: . . . . .	22
2.5.3	Setup for Deductive Verification . . . . .	23
2.6	The Squareroot Example for Symbolic Execution . . . . .	24
2.6.1	The Conceptual Algorithm in Clean Notation . . . . .	24
2.6.2	Definition of the Global State . . . . .	24
2.6.3	Setting for Symbolic Execution . . . . .	25
2.6.4	A Symbolic Execution Simulation . . . . .	26
<b>3</b>	<b>Appendix : Used Monad Libraries</b>	<b>27</b>
3.1	Definition : Standard State Exception Monads . . . . .	27
3.1.1	Definition : Core Types and Operators . . . . .	27

3.1.2	Definition : More Operators and their Properties . . . . .	28
3.1.3	Definition : Programming Operators and their Properties . . . . .	29
3.1.4	Theory of a Monadic While . . . . .	29
3.1.5	Chaining Monadic Computations : Definitions of Multi-bind Op- erators . . . . .	32
3.1.6	Definition and Properties of Valid Execution Sequences . . . . .	35
3.1.7	Miscellaneous . . . . .	42
3.2	Clean Symbolic Execution Rules . . . . .	42
3.2.1	Basic NOP - Symbolic Execution Rules. . . . .	42
3.2.2	Assign Execution Rules. . . . .	43
3.2.3	Basic Call Symbolic Execution Rules. . . . .	44
3.2.4	Basic Call Symbolic Execution Rules. . . . .	45
3.2.5	Conditional. . . . .	46
3.2.6	Break - Rules. . . . .	46
3.2.7	While. . . . .	47
3.3	Hoare . . . . .	48
3.3.1	Basic rules . . . . .	48
3.3.2	Generalized and special sequence rules . . . . .	49
3.3.3	Generalized and special consequence rules . . . . .	49
3.3.4	Condition rules . . . . .	50
3.3.5	While rules . . . . .	50
3.3.6	Experimental Alternative Definitions (Transformer-Style Rely-Guarantee)	51
3.3.7	Clean Control Rules . . . . .	51
3.3.8	Clean Skip Rules . . . . .	51
3.3.9	Clean Assign Rules . . . . .	52
3.3.10	Clean Construct Rules . . . . .	53

# 1 The Clean Language

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

**begin**

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

It comprises:

- C-like control flow with *break* and *return*,
- global variables,
- function calls (seen as monadic executions) with side-effects, recursion and local variables,
- parameters are modeled via functional abstractions (functions are monads); a passing of parameters to local variables might be added later,
- direct recursive function calls,
- cartouche syntax for  $\lambda$ -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  $\sigma_0$  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  $\sigma_{n+1} = \sigma_n + a_1 :: \tau_1$  list; ...;  $a_n :: \tau_n$  list; result ::  $\tau_{result-type}$  list;
```

---

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

The **record** package creates an  $\prime \sigma$  extensible record type  $\prime \sigma$  *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*

**definition** *break* :: (*unit*, (*'σ-ext*) *control-state-ext*) *MON<sub>SE</sub>*  
**where** *break* ≡ (λ *σ*. *Some*((), *σ* (| *break-status* := *True* |)))

**definition** *unset-break-status* :: (*unit*, (*'σ-ext*) *control-state-ext*) *MON<sub>SE</sub>*  
**where** *unset-break-status* ≡ (λ *σ*. *Some*((), *σ* (| *break-status* := *False* |)))

**definition** *set-return-status* :: (*unit*, (*'σ-ext*) *control-state-ext*) *MON<sub>SE</sub>*  
**where** *set-return-status* = (λ *σ*. *Some*((), *σ* (| *return-status* := *True* |)))

**definition** *unset-return-status* :: (*unit*, (*'σ-ext*) *control-state-ext*) *MON<sub>SE</sub>*  
**where** *unset-return-status* = (λ *σ*. *Some*((), *σ* (| *return-status* := *False* |)))

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

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

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

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

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

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

**lemma** *exec-stop-vs-control-independence* [*simp*]:  
 # *upd* ⇒ *exec-stop* (*upd f σ*) = *exec-stop* *σ*  
 ⟨*proof*⟩

**lemma** *exec-stop-vs-control-independence'* [simp]:  
 $\# \text{ upd} \implies (\text{upd } f (\sigma \langle \text{return-status} := b \rangle)) = (\text{upd } f \sigma) \langle \text{return-status} := b \rangle$   
 $\langle \text{proof} \rangle$

**lemma** *exec-stop-vs-control-independence''* [simp]:  
 $\# \text{ upd} \implies (\text{upd } f (\sigma \langle \text{break-status} := b \rangle)) = (\text{upd } f \sigma) \langle \text{break-status} := b \rangle$   
 $\langle \text{proof} \rangle$

### 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  $\text{tmp}$  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 <sup>1</sup> 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* ::  $((\sigma\text{-ext}) \text{ control-state-scheme} \Rightarrow$   
 $(\sigma\text{-ext}) \text{ control-state-scheme}) \Rightarrow$   
 $(\text{unit}, (\sigma\text{-ext}) \text{ control-state-scheme}) \text{MON}_{SE}$   
**where** *assign*  $f = (\lambda \sigma. \text{if } \text{exec-stop } \sigma \text{ then } \text{Some}(\langle \rangle, \sigma) \text{ else } \text{Some}(\langle \rangle, f \sigma))$

**definition** *assign-global* ::  $((\alpha \Rightarrow \alpha') \Rightarrow \sigma\text{-ext control-state-scheme} \Rightarrow \sigma\text{-ext control-state-scheme})$   
 $\Rightarrow (\sigma\text{-ext control-state-scheme} \Rightarrow \alpha')$

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

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

**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).

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

**lemma**  $\text{tl-map-hd [simp]} : \text{tl } (\text{map-hd } f S) = \text{tl } S \langle \text{proof} \rangle$

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

**definition**  $\text{assign-local} :: (('a \text{ list} \Rightarrow 'a \text{ list})$   
 $\quad \Rightarrow ' \sigma\text{-ext control-state-scheme} \Rightarrow ' \sigma\text{-ext control-state-scheme})$   
 $\quad \Rightarrow (' \sigma\text{-ext control-state-scheme} \Rightarrow 'a)$   
 $\quad \Rightarrow (\text{unit}, ' \sigma\text{-ext control-state-scheme}) \text{MON}_{SE}$

**where**  $\text{assign-local upd rhs} = \text{assign}(\lambda\sigma. ((\text{upd } o \text{map-hd}) (\% \cdot. \text{rhs } \sigma)) \sigma)$

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**  $\text{assign-local upd rhs} = \text{assign-global } (\text{upd } o \text{map-hd}) \text{rhs}$   
 $\langle \text{proof} \rangle$

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**  $\text{return}_C :: (('a \text{ list} \Rightarrow 'a \text{ list}) \Rightarrow ' \sigma\text{-ext control-state-scheme} \Rightarrow ' \sigma\text{-ext control-state-scheme})$   
 $\quad \Rightarrow (' \sigma\text{-ext control-state-scheme} \Rightarrow 'a)$   
 $\quad \Rightarrow (\text{unit}, ' \sigma\text{-ext control-state-scheme}) \text{MON}_{SE}$

**where**  $\text{return}_C \text{upd rhs} = (\lambda\sigma. \text{if exec-stop } \sigma \text{ then } \text{Some}(\cdot, \sigma)$   
 $\quad \text{else } (\text{assign-local upd rhs } ; - \text{set-return-status}) \sigma)$

### 1.1.5 Example for a Local Variable Space

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

*function-spec*  $\text{swap } (i :: \text{nat}, j :: \text{nat})$

*local-vars*  $tmp :: int$   
*defines*  $\langle tmp := A ! i \rangle ; -$   
 $\langle A[i] := A ! j \rangle ; -$   
 $\langle A[j] := tmp \rangle$

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

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

*definition push-local-swap-state*  $:: (unit, 'a local-swap-state-scheme) MON_{SE}$   
*where* *push-local-swap-state*  $\sigma =$   
 $Some((\lambda\sigma. \sigma \langle local-swap-state.tmp := undefined \# local-swap-state.tmp \sigma,$   
 $local-swap-state.result-value := undefined \#$   
 $local-swap-state.result-value \sigma \rangle))$

*definition pop-local-swap-state*  $:: (unit, 'a local-swap-state-scheme) MON_{SE}$   
*where* *pop-local-swap-state*  $\sigma =$   
 $Some(hd(local-swap-state.result-value \sigma),$   
 $\sigma \langle local-swap-state.tmp := tl(local-swap-state.tmp \sigma) \rangle)$

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.

$\langle ML \rangle$

### 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<sub>C</sub>*  $:: (unit, ('\sigma-ext) control-state-ext) MON_{SE}$   
 $\Rightarrow (unit, ('\sigma-ext) control-state-ext) MON_{SE}$   
 $\Rightarrow ('\alpha, ('\sigma-ext) control-state-ext) MON_{SE}$   
 $\Rightarrow ('\alpha, ('\sigma-ext) control-state-ext) MON_{SE}$

**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*<sub>SE</sub>( $x$ )) — yield the return value

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

*definition*  $swap\_core :: nat \times nat \Rightarrow (unit, 'a\ local\ swap\ state\ scheme) MON_{SE}$   
*where*  $swap\_core \equiv (\lambda(i,j). ((assign\ local\ tmp\ update\ (\lambda\sigma. A\ \sigma\ !\ i))\ ;-\$   
 $(assign\ global\ A\ update\ (\lambda\sigma. list\ update\ (A\ \sigma)\ (i)\ (A\ \sigma\ !\ j)))\ ;-\$   
 $(assign\ global\ A\ update\ (\lambda\sigma. list\ update\ (A\ \sigma)\ (j)\ ((hd\ o\ tmp)\ \sigma))))))$

*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-aware*.

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

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

**definition**  $call\_0_C :: ('q, ('\sigma\ ext)\ control\ state\ ext) MON_{SE} \Rightarrow ('q, ('\sigma\ ext)\ control\ state\ ext) MON_{SE}$   
**where**  $call\_0_C\ M = (\lambda\sigma. if\ exec\ stop\ \sigma\ then\ Some(undefined, \sigma)\ else\ M\ \sigma)$

The generic version using tuples is identical with  $call\_1_C$ .

**definition**  $call\_1_C :: ('\alpha \Rightarrow ('q, ('\sigma\ ext)\ control\ state\ ext) MON_{SE}) \Rightarrow$   
 $((('\sigma\ ext)\ control\ state\ ext) \Rightarrow '\alpha) \Rightarrow$   
 $('q, ('\sigma\ ext)\ control\ state\ ext) MON_{SE}$   
**where**  $call\_1_C = call_C$

**definition**  $call\_2_C :: ('\alpha \Rightarrow '\beta \Rightarrow ('q, ('\sigma\ ext)\ control\ state\ ext) MON_{SE}) \Rightarrow$   
 $((('\sigma\ ext)\ control\ state\ ext) \Rightarrow '\alpha) \Rightarrow$   
 $((('\sigma\ ext)\ control\ state\ ext) \Rightarrow '\beta) \Rightarrow$   
 $('q, ('\sigma\ ext)\ control\ state\ ext) MON_{SE}$   
**where**  $call\_2_C\ M\ A_1\ A_2 = (\lambda\sigma. if\ exec\ stop\ \sigma\ then\ Some(undefined, \sigma)\ else\ M\ (A_1\ \sigma)\ (A_2\ \sigma)\ \sigma)$

**definition**  $call-3_C :: ('\alpha \Rightarrow '\beta \Rightarrow '\gamma \Rightarrow ('_q, ('_sigma-ext) control-state-ext)MON_{SE}) \Rightarrow$   
 $((('sigma-ext) control-state-ext) \Rightarrow '\alpha) \Rightarrow$   
 $((('sigma-ext) control-state-ext) \Rightarrow '\beta) \Rightarrow$   
 $((('sigma-ext) control-state-ext) \Rightarrow '\gamma) \Rightarrow$   
 $('_q, ('_sigma-ext) control-state-ext)MON_{SE}$   
**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 `HOLLogic` from the Isabelle/HOL libraries. They incorporate the construction of types during term construction in a bottom-up manner. Consequently, the leaves of such terms should always be typed, and anonymous loose-Bound variables avoided.

$\langle ML \rangle$

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.

$\langle ML \rangle$

### 1.4 Syntactic Sugar supporting $\lambda$ -lifting for Global and Local Variables

$\langle ML \rangle$

**syntax**  $-cartouche-string :: cartouche-position \Rightarrow string (-)$

$\langle ML \rangle$

### 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  $\lambda$ -lifting cartouches and implicit reference to parameters, pre and post states),
3. the parsing local variable section with the local-variable space generation,
4. the parsing of the body in this extended variable space,
5. and optionally the support of measures for recursion proofs.

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

**definition** *old* :: 'a  $\Rightarrow$  'a **where** *old* x = x

$\langle ML \rangle$

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

**definition** *if-C* :: [( $'\sigma$ -ext) control-state-ext  $\Rightarrow$  bool,  
 ( $'\beta$ , ( $'\sigma$ -ext) control-state-ext)  $MON_{SE}$ ,  
 ( $'\beta$ , ( $'\sigma$ -ext) control-state-ext)  $MON_{SE}$ ]  $\Rightarrow$  ( $'\beta$ , ( $'\sigma$ -ext) control-state-ext)  $MON_{SE}$   
**where** *if-C* c E F = ( $\lambda\sigma$ . if exec-stop  $\sigma$   
 then Some(undefined,  $\sigma$ ) — state unchanged, return arbitrary  
 else if c  $\sigma$  then E  $\sigma$  else F  $\sigma$ )

**syntax** (*xsymbols*)  
 -if-SECLEAN :: [ $'\sigma \Rightarrow$  bool, ( $'o$ ,  $'\sigma$ )  $MON_{SE}$ , ( $'o'$ ,  $'\sigma$ )  $MON_{SE}$ ]  $\Rightarrow$  ( $'o'$ ,  $'\sigma$ )  $MON_{SE}$   
 ((if<sub>C</sub> - then - else -fi) [5,8,8]8)

**translations**  
 (if<sub>C</sub> 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  $\sigma$  then Some((),  $\sigma$ )  
 else ((MonadSE.while-SE ( $\lambda\sigma$ .  $\neg$ exec-stop  $\sigma \wedge$  c  $\sigma$ ) B) ; -  
 unset-break-status)  $\sigma$ )

**syntax** (*xsymbols*)  
 -while-C :: [ $'\sigma \Rightarrow$  bool, (unit,  $'\sigma$ )  $MON_{SE}$ ]  $\Rightarrow$  (unit,  $'\sigma$ )  $MON_{SE}$   
 ((while<sub>C</sub> - do - od) [8,8]8)

**translations**  
 while<sub>C</sub> c do b od == CONST Clean.while-C c b

**end**





## 2 Clean Semantics : A Coding-Concept Example

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

```
theory Quicksort-concept
  imports Clean
    Hoare-MonadSE
begin
```

### 2.1 The Quicksort Example

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

---

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

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

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

---

In the following, we will present the Quicksort program 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⟩

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

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

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

⟨ML⟩

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

## 2.3 Encoding swap in Clean

### 2.3.1 swap in High-level Notation

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

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

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

**function-spec**  $swap (i::nat, j::nat)$  — TODO: the hovering on parameters produces a number of report equal to the number of `Proof_Context.add_fixes` called in `Function_Specification_Parser.checkM`

```
pre      ⟨i < length A ∧ j < length A⟩
post     ⟨λres. length A = length(old A) ∧ res = ()⟩
local-vars tmp :: int
defines  ⟨ tmp := A ! i ⟩ ;−
        ⟨ A := list-update A i (A ! j) ⟩ ;−
        ⟨ A := list-update A j tmp ⟩
```

The body — heavily using the  $\lambda$ -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:

$\langle ML \rangle$

### 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* ::  $nat \times nat \Rightarrow 'a \text{ global-state-state-scheme} \Rightarrow bool$

**where** *swap'-pre*  $\equiv \lambda(i, j) \sigma. i < length(A \sigma) \wedge j < length(A \sigma)$

**definition** *swap'-post* ::  $'a \times 'b \Rightarrow 'c \text{ global-state-state-scheme} \Rightarrow 'd \text{ global-state-state-scheme} \Rightarrow unit \Rightarrow bool$

**where** *swap'-post*  $\equiv \lambda(i, j) \sigma_{pre} \sigma \text{ res}. length(A \sigma) = length(A \sigma_{pre}) \wedge \text{res} = ()$

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*

## 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**  $(\langle pivot := A ! hi \rangle ; - \langle i := lo \rangle ; - \langle j := lo \rangle ; -$   
 $(while_C \langle j \leq hi - 1 \rangle$   
 $do (if_C \langle A ! j < pivot \rangle$   
 $then call_C swap \langle (i, j) \rangle ; -$   
 $\langle i := i + 1 \rangle$   
 $else skip_{SE}$   
 $fi) ; -$   
 $\langle j := j + 1 \rangle$   
 $od) ; -$   
 $call_C swap \langle (i, j) \rangle ; -$

```

    returnC result-value-update ⟨i⟩
  )

```

The body is a fancy syntax for :

```

⟨defines
  ((assign-local pivot-update (λσ. A σ ! hi )) ;−
  (assign-local i-update (λσ. lo )) ;−

  (assign-local j-update (λσ. lo )) ;−
  (whileC (λσ. (hd o j) σ ≤ hi − 1 )
  do (ifC (λσ. A σ ! (hd o j) σ < (hd o pivot)σ )
    then callC (swap) (λσ. ((hd o i) σ, (hd o j) σ)) ;−
      assign-local i-update (λσ. ((hd o i) σ) + 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⟩

## 2.4.2 A Simulation of partition in elementary specification constructs:

**definition** *partition'-pre* ≡ λ(lo, hi) σ. lo < length (A σ) ∧ hi < length (A σ)

**definition** *partition'-post* ≡ λ(lo, hi) σ<sub>pre</sub> σ res. length (A σ) = length (A σ<sub>pre</sub>) ∧ res = 3

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

```

local-vars-test partition' nat
  pivot :: int
  i      :: nat
  j      :: nat

```

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

```

thm push-local-partition'-state-def

```

**thm** *pop-local-partition'-state-def*

**definition** *push-local-partition-state'* :: (unit, 'a local-partition'-state-scheme) *MON<sub>SE</sub>*  
**where** *push-local-partition-state'*  $\sigma = \text{Some}()$ ,  
 $\sigma(\text{local-partition-state.pivot} := \text{undefined} \# \text{local-partition-state.pivot } \sigma,$   
 $\text{local-partition-state.i} := \text{undefined} \# \text{local-partition-state.i } \sigma,$   
 $\text{local-partition-state.j} := \text{undefined} \# \text{local-partition-state.j } \sigma,$   
 $\text{local-partition-state.result-value}$   
 $:= \text{undefined} \# \text{local-partition-state.result-value } \sigma \ )$

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

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

**thm** *partition-core-def*

**definition** *partition'* :: nat  $\times$  nat  $\Rightarrow$  (nat, 'a local-partition'-state-scheme) *MON<sub>SE</sub>*  
**where** *partition'*  $\equiv \lambda(\text{lo}, \text{hi}). \text{block}_C \text{push-local-partition-state}$   
 $(\text{partition-core } (\text{lo}, \text{hi}))$   
 $\text{pop-local-partition-state}$

## 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⟩

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

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

**definition** push-local-quicksort-state' :: (unit, 'a local-quicksort'-state-scheme) MON<sub>SE</sub>

**where** push-local-quicksort-state' σ =

Some((), σ(|local-quicksort'-state.p := undefined # local-quicksort'-state.p σ,

local-quicksort'-state.result-value := undefined # local-quicksort'-state.result-value

σ |))

**definition** pop-local-quicksort-state' :: (unit, 'a local-quicksort'-state-scheme) MON<sub>SE</sub>

**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),

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

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

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

**definition**  $\text{quicksort}'\text{-core} :: (\text{nat} \times \text{nat} \Rightarrow (\text{unit}, 'a \text{ local-quicksort}'\text{-state-scheme}) \text{MON}_{SE})$   
 $\Rightarrow (\text{nat} \times \text{nat} \Rightarrow (\text{unit}, 'a \text{ local-quicksort}'\text{-state-scheme}) \text{MON}_{SE})$

**where**  $\text{quicksort}'\text{-core} \text{ quicksort-rec} \equiv \lambda(\text{lo}, \text{hi}).$   
 $((\text{if}_C (\lambda\sigma. \text{lo} < \text{hi})$   
 then  $(p_{tmp} \leftarrow \text{call}_C \text{partition } (\lambda\sigma. (\text{lo}, \text{hi})) ;$   
 $\text{assign-local } p\text{-update } (\lambda\sigma. p_{tmp}) ;-$   
 $\text{call}_C \text{quicksort-rec } (\lambda\sigma. (\text{lo}, (\text{hd } o \text{ } p) \sigma - 1)) ;-$   
 $\text{call}_C \text{quicksort-rec } (\lambda\sigma. ((\text{hd } o \text{ } p) \sigma + 1, \text{hi}))$   
 else  $\text{skip}_{SE}$   
 $\text{fi}))$

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

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

### 2.5.3 Setup for Deductive Verification

The coupling between the pre- and the post-condition state is done by the free variable (serving as a kind of ghost-variable)  $\sigma_{pre}$ . 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 :
  {λσ. ¬exec-stop σ ∧ quicksort-pre (lo, hi)(σ) ∧ σ = σpre }
    quicksort (lo, hi)
  {λr σ. ¬exec-stop σ ∧ quicksort-post(lo, hi)(σpre)(σ)(r) }
  ⟨proof⟩

```

end

## 2.6 The Squareroot Example for Symbolic Execution

```

theory SquareRoot-concept
  imports Test-Clean
begin

```

### 2.6.1 The Conceptual Algorithm in Clean Notation

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

```

⟨
  function-spec sqrt (a::int) returns int
  pre      ⟨0 ≤ a⟩
  post     ⟨λres::int. (res + 1)2 > a ∧ a ≥ (res)2⟩
  defines  (⟨tm := 1⟩ ;−
           ⟨sqsum := 1⟩ ;−
           ⟨i := 0⟩ ;−
           (whileSE ⟨sqsum ≤ a⟩ do
             ⟨i := i+1⟩ ;−
             ⟨tm := tm + 2⟩ ;−
             ⟨sqsum := tm + sqsum⟩
           od) ;−
           returnC result-value-update ⟨i⟩
           )
  ⟩

```

### 2.6.2 Definition of the Global State

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

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

```

typ Clean.control-state

```



$\langle ML \rangle$

**global-vars** *state*

*tm* :: *int*  
*i* :: *int*  
*sqsum* :: *int*

$\langle ML \rangle$

**lemma** *tm-independent* [*simp*]: # *tm-update*  
 $\langle proof \rangle$

**lemma** *i-independent* [*simp*]: # *i-update*  
 $\langle proof \rangle$

**lemma** *sqsum-independent* [*simp*]: # *sqsum-update*  
 $\langle proof \rangle$

### 2.6.3 Setting for Symbolic Execution

Some lemmas to reason about memory

**lemma** *tm-simp* :  $tm (\sigma (tm := t)) = t$   
 $\langle proof \rangle$

**lemma** *tm-simp1* :  $tm (\sigma (sqsum := s)) = tm \sigma$   $\langle proof \rangle$

**lemma** *tm-simp2* :  $tm (\sigma (i := s)) = tm \sigma$   $\langle proof \rangle$

**lemma** *sqsum-simp* :  $sqsum (\sigma (sqsum := s)) = s$   $\langle proof \rangle$

**lemma** *sqsum-simp1* :  $sqsum (\sigma (tm := t)) = sqsum \sigma$   $\langle proof \rangle$

**lemma** *sqsum-simp2* :  $sqsum (\sigma (i := t)) = sqsum \sigma$   $\langle proof \rangle$

**lemma** *i-simp* :  $i (\sigma (i := i')) = i'$   $\langle proof \rangle$

**lemma** *i-simp1* :  $i (\sigma (tm := i')) = i \sigma$   $\langle proof \rangle$

**lemma** *i-simp2* :  $i (\sigma (sqsum := i')) = i \sigma$   $\langle proof \rangle$

**lemmas** *memory-theory* =

*tm-simp tm-simp1 tm-simp2*  
*sqsum-simp sqsum-simp1 sqsum-simp2*  
*i-simp i-simp1 i-simp2*

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

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

**assumes** # *upd*

**shows**  $\sigma \models assign-global\ upd\ rhs ; - M \implies \neg exec-stop\ \sigma \implies upd\ (\lambda-. rhs)\ \sigma \models M$

*<proof>*

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

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

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

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

## 2.6.4 A Symbolic Execution Simulation

**lemma**

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

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

**and** *annotated-program:*

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

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

*<proof>*

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

**end**

## 3 Appendix : Used Monad Libraries

```
theory MonadSE
  imports Main
begin
```

### 3.1 Definition : Standard State Exception Monads

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

#### 3.1.1 Definition : Core Types and Operators

```
type-synonym ('o, '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
⟨proof⟩
```

```
lemma bind-left-unit [simp]: (x ← result c; P x) = P c
⟨proof⟩
```

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

### 3.1.2 Definition : More Operators and their Properties

**definition** *fail-SE* ::  $('o, 'σ)MON_{SE}$

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

**notation**  $fail-SE$  (*fail<sub>SE</sub>*)

**definition** *assert-SE* ::  $('σ \Rightarrow bool) \Rightarrow (bool, 'σ)MON_{SE}$

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

**notation**  $assert-SE$  (*assert<sub>SE</sub>*)

**definition** *assume-SE* ::  $('σ \Rightarrow bool) \Rightarrow (unit, 'σ)MON_{SE}$

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

**notation**  $assume-SE$  (*assume<sub>SE</sub>*)

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

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

**definition** *bind-SE'* ::  $('α, 'σ)MON_{SE} \Rightarrow ('β, 'σ)MON_{SE} \Rightarrow ('β, 'σ)MON_{SE}$  (**infixr** ;- 60)

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

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

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

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

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

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

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

**definition** *yield<sub>C</sub>* ::  $('a \Rightarrow 'b) \Rightarrow ('b, 'a) MON_{SE}$   
**where**  $yield_C f \equiv (\lambda\sigma. Some(f \ \sigma, \sigma))$

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

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

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

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

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

**lemma**  $malt\_SE\text{-}mt\ [simp]: \sqcap_{SE}\ [] = fail_{SE}$   
 $\langle proof \rangle$

**lemma**  $malt\_SE\text{-}cons\ [simp]: \sqcap_{SE}\ (a\ \# S) = (a\ \sqcap_{SE}\ (\sqcap_{SE}\ S))$   
 $\langle proof \rangle$

### 3.1.3 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**  $if\_SE\ c\ E\ F = (\lambda\sigma. if\ c\ \sigma\ then\ E\ \sigma\ else\ F\ \sigma)$

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

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

### 3.1.4 Theory of a Monadic While

Prerequisites

**fun**  $replicator :: [('a, 'σ)MON_{SE}, nat] \Rightarrow (unit, 'σ)MON_{SE} (\mathbf{infixr}\ \overset{\sim}{\sim}\ 60)$   
**where**  $f\ \overset{\sim}{\sim}\ 0 = (result\ ())$   
 $\quad | f\ \overset{\sim}{\sim}\ (Suc\ n) = (f\ ;-\ f\ \overset{\sim}{\sim}\ n)$

**fun**  $replicator2 :: [('a, 'σ)MON_{SE}, nat, ('b, 'σ)MON_{SE}] \Rightarrow ('b, 'σ)MON_{SE} (\mathbf{infixr}\ \hat{\sim}\ 60)$

**where**  $(f \hat{=} 0) M = (M)$   
 $| (f \hat{=} (Suc\ n)) M = (f ;- ((f \hat{=} n) M))$

First Step : Establishing an embedding between partial functions and relations

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

**definition**  $Rel2Mon :: ('σ \times 'σ) set \Rightarrow (unit, 'σ)MON_{SE}$

**where**  $Rel2Mon\ S = (\lambda\ \sigma. \text{if } \exists\ \sigma'. (\sigma, \sigma') \in S \text{ then } Some((), SOME\ \sigma'. (\sigma, \sigma') \in S) \text{ else } None)$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

**lemma**  $single\text{-}valued\text{-}Mon2Rel$ :  $single\text{-}valued\ (Mon2Rel\ B)$

$\langle proof \rangle$

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

**definition**  $chain :: (nat \Rightarrow 'a\ set) \Rightarrow bool$

**where**  $chain\ S = (\forall\ i. S\ i \subseteq S(Suc\ i))$

**lemma**  $chain\text{-}total$ :  $chain\ S \implies S\ i \leq S\ j \vee S\ j \leq S\ i$

$\langle proof \rangle$

**definition**  $cont :: ('a\ set \implies 'b\ set) \implies bool$

**where**  $cont\ f = (\forall\ S. chain\ S \longrightarrow f(UN\ n. S\ n) = (UN\ n. f(S\ n)))$

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

**assumes**  $cont\ f$  **shows**  $mono\ f$

$\langle proof \rangle$

**lemma**  $chain\text{-}iterates$ : **fixes**  $f :: 'a\ set \implies 'a\ set$

**assumes**  $mono\ f$  **shows**  $chain(\lambda n. (f \hat{=} n)\ \{\})$

$\langle proof \rangle$

**theorem**  $lfp\text{-}if\text{-}cont$ :

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

$\langle proof \rangle$

**lemma**  $single\text{-}valued\text{-}UN\text{-}chain$ :

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

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

$\langle proof \rangle$

**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**  $\text{single-valued}(\text{lfp } f)$   
*<proof>*

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**  $\Gamma b \text{ cd} = (\lambda c w. \{(s, t). \text{if } b \text{ s then } (s, t) \in \text{cd } O \text{ cw else } s = t\})$

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

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

**lemma** *cont-Γ*:  $\text{cont } (\Gamma c \text{ b})$   
*<proof>*

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

**theorem** *single-valued-lfp-Mon2Rel*:  $\text{single-valued } (\text{lfp}(\Gamma c (\text{Mon2Rel } B)))$   
*<proof>*

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

**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>*

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*:  
 $(\text{while}_{SE} b \text{ do } c \text{ od}) = (\text{if}_{SE} b \text{ then } (c ;- (\text{while}_{SE} b \text{ do } c \text{ od})) \text{ else } \text{result } () \text{ fi})$   
*<proof>*

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

$\langle proof \rangle$

**lemma** *bind'-cong* :  $f \sigma = g \sigma \implies (f ; - M) \sigma = (g ; - M) \sigma$   
 $\langle proof \rangle$

**lemma** *if<sub>SE</sub>-True [simp]*:  $(if_{SE} (\lambda x. True) then c else d fi) = c$   
 $\langle proof \rangle$

**lemma** *if<sub>SE</sub>-False [simp]*:  $(if_{SE} (\lambda x. False) then c else d fi) = d$   
 $\langle proof \rangle$

**lemma** *if<sub>SE</sub>-cond-cong* :  $f \sigma = g \sigma \implies$   
 $(if_{SE} f then c else d fi) \sigma =$   
 $(if_{SE} g then c else d fi) \sigma$   
 $\langle proof \rangle$

**lemma** *while<sub>SE</sub>-skip [simp]* :  $(while_{SE} (\lambda x. False) do c od) = skip_{SE}$   
 $\langle proof \rangle$

**end**

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

### 3.1.5 Chaining Monadic Computations : Definitions of Multi-bind Operators

In order to express execution sequences inside HOL— rather than arguing over a certain pattern of terms on the meta-level — and in order to make our theory amenable to formal reasoning over execution sequences, we represent them as lists of input and generalize the bind-operator of the state-exception monad accordingly. The approach is straightforward, but comes with a price: we have to encapsulate all input and output data into one type, and restrict ourselves to a uniform step function. Assume that we have a typed interface to a module with the operations  $op_1, op_2, \dots, op_n$  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 :: 'l list  $\Rightarrow$  ('l  $\Rightarrow$  ('o,' $\sigma$ )  $MON_{SE}$ )  $\Rightarrow$  ('o list,' $\sigma$ )  $MON_{SE}$ 
where mbind [] iostep  $\sigma$  = Some([],  $\sigma$ )
      | mbind (a#S) iostep  $\sigma$  =
          (case iostep a  $\sigma$  of
             None  $\Rightarrow$  Some([],  $\sigma$ )
          | Some (out,  $\sigma'$ )  $\Rightarrow$  (case mbind S iostep  $\sigma'$  of
                                     None  $\Rightarrow$  Some([out], $\sigma'$ )
                                     | Some(outs, $\sigma''$ )  $\Rightarrow$  Some(out#outs, $\sigma''$ )))
```

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

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

```
lemma mbind-unit [simp]:
  mbind [] f = (result [])
  <proof>
```

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

```
lemma mbind-nofailure [simp]:
  mbind S f  $\sigma \neq$  None
  <proof>
```

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

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

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

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

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

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

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

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

**fun**  $mbind'' :: 'l list \Rightarrow ('l \Rightarrow ('o, 's) MON_{SE}) \Rightarrow ('o list, 's) MON_{SE}$   
**where**  $mbind'' [] iostep \sigma = Some([], \sigma) |$   
 $mbind'' (a\#S) iostep \sigma =$   
 $(case iostep a \sigma of$   
 $None \Rightarrow mbind'' S iostep \sigma$   
 $| Some (out, \sigma') \Rightarrow (case mbind'' S iostep \sigma' of$   
 $None \Rightarrow None \text{ --- does not occur}$   
 $| Some(outs, \sigma'') \Rightarrow Some(out\#outs, \sigma'')))$

**notation**  $mbind''$  ( $mbind_{FailPurge}$ )

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

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

## Definition : Miscellaneous Operators and their Properties

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

**end**

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

### 3.1.6 Definition and Properties of Valid Execution Sequences

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

1. terminates (not obvious since while),
2. results in a final *True*,
3. does not fail globally (but recall the FailSave and FailPurge variants of  $m\text{bind}_{\text{FailSave}}$ -operators, that handle local exceptions in one or another way).

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

**definition** *valid-SE* ::  $'\sigma \Rightarrow (\text{bool}, '\sigma) \text{MON}_{SE} \Rightarrow \text{bool}$  (**infix**  $\models$  15)  
**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*]:  $(\sigma \models (\text{result } P)) = (P)$   
 $\langle \text{proof} \rangle$

**lemma** *exec-unit-SE'* [*simp*]:  $(\sigma_0 \models (\lambda \sigma. \text{Some } (f \sigma, \sigma))) = (f \sigma_0)$   
 $\langle \text{proof} \rangle$

**lemma** *exec-fail-SE* [*simp*]:  $(\sigma \models \text{fail}_{SE}) = \text{False}$   
 $\langle \text{proof} \rangle$

**lemma** *exec-fail-SE'* [*simp*]:  $\neg(\sigma_0 \models (\lambda \sigma. \text{None}))$   
 $\langle \text{proof} \rangle$

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

**lemma** *exec-bind-SE-failure*:  
 $A \sigma = \text{None} \Longrightarrow \neg(\sigma \models ((s \leftarrow A ; M s)))$   
 $\langle \text{proof} \rangle$

**lemma** *exec-bind-SE-failure2*:  
 $A \sigma = \text{None} \Longrightarrow \neg(\sigma \models ((A ; - M)))$   
 $\langle \text{proof} \rangle$

**lemma** *exec-bind-SE-success*:  
 $A \sigma = \text{Some}(b, \sigma') \Longrightarrow (\sigma \models ((s \leftarrow A ; M s))) = (\sigma' \models (M b))$   
 $\langle \text{proof} \rangle$

**lemma** *exec-bind-SE-success2*:  
 $A \sigma = \text{Some}(b, \sigma') \Longrightarrow (\sigma \models ((A ; - M))) = (\sigma' \models M)$   
 $\langle \text{proof} \rangle$

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

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

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

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

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

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

Recall `mbind_unit` for the base case.

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

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

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

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

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

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

**lemma** *exec-mbindFSave-failure*:  
 $\text{ioprogram } a \sigma = \text{None} \implies$

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

**lemma** *exec-mbindFStop-failure:*

*ioprogram a σ = None ⇒*

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

⟨proof⟩

**lemma** *exec-mbindFPurge-failure:*

*ioprogram a σ = None ⇒*

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

⟨proof⟩

**lemma** *exec-mbindFSave-success :*

*ioprogram a σ = Some(b,σ') ⇒*

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

⟨proof⟩

**lemma** *exec-mbindFStop-success :*

*ioprogram a σ = Some(b,σ') ⇒*

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

⟨proof⟩

**lemma** *exec-mbindFPurge-success :*

*ioprogram a σ = Some(b,σ') ⇒*

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

⟨proof⟩

**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 ⇒ (σ ⊨ (return (P [])))*

*| Some(b,σ') ⇒ (σ' ⊨ (s ← mbind<sub>FailSave</sub> S ioprogram ; return (P (b#s))))*)

⟨proof⟩

**lemma** *mbind-eq-sexec:*

**assumes**  $*$  :  $\bigwedge b \sigma'. f a \sigma = \text{Some}(b, \sigma') \Rightarrow$

$(os \leftarrow \text{mbind}_{\text{FailStop}} S f ; P (b\#os)) = (os \leftarrow \text{mbind}_{\text{FailStop}} S f ; P' (b\#os))$

**shows**  $(a \leftarrow f a ; x \leftarrow \text{mbind}_{\text{FailStop}} S f ; P (a \# x)) \sigma =$

$(a \leftarrow f a ; x \leftarrow \text{mbind}_{\text{FailStop}} S f ; P'(a \# x)) \sigma$

⟨proof⟩

**lemma** *mbind-eq-sexec':*

**assumes**  $*$  :  $\bigwedge b \sigma'. f a \sigma = \text{Some}(b, \sigma') \implies$   
 $(P(b))\sigma' = (P'(b))\sigma'$   
**shows**  $(a \leftarrow f a; P(a)) \sigma =$   
 $(a \leftarrow f a; P'(a)) \sigma$   
 $\langle \text{proof} \rangle$

**lemma** *mbind'-concat*:

$(os \leftarrow \text{mbind}_{\text{FailStop}} (S@T) f; P os) = (os \leftarrow \text{mbind}_{\text{FailStop}} S f; os' \leftarrow \text{mbind}_{\text{FailStop}} T f;$   
 $P (os @ os'))$   
 $\langle \text{proof} \rangle$

**lemma** *assert-suffix-inv* :

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

$\langle \text{proof} \rangle$

Universal splitting and symbolic execution rule

**lemma** *exec-mbindFSave-E*:

**assumes**  $seq : (\sigma \models (s \leftarrow \text{mbind}_{\text{FailSave}} (a\#S) \text{ ioprogram}; (P s)))$   
**and**  $none: \text{ioprogram } a \sigma = \text{None} \implies (\sigma \models (P [])) \implies Q$   
**and**  $some: \bigwedge b \sigma'. \text{ioprogram } a \sigma = \text{Some}(b, \sigma') \implies (\sigma' \models (s \leftarrow \text{mbind}_{\text{FailSave}} S \text{ ioprogram}; (P$   
 $(b\#s)))) \implies Q$   
**shows**  $Q$   
 $\langle \text{proof} \rangle$

The next rule reveals the particular interest in deduction; as an elimination rule, it allows for a linear conversion of a validity judgement  $\text{mbind}_{\text{FailStop}}$  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**  $seq : (\sigma \models (s \leftarrow \text{mbind}_{\text{FailStop}} (a\#S) \text{ ioprogram}; (P s)))$   
**and**  $some: \bigwedge b \sigma'. \text{ioprogram } a \sigma = \text{Some}(b, \sigma') \implies (\sigma' \models (s \leftarrow \text{mbind}_{\text{FailStop}} S \text{ ioprogram}; (P(b\#s))))$   
 $\implies Q$   
**shows**  $Q$   
 $\langle \text{proof} \rangle$

**lemma** *exec-mbindFPurge-E*:

**assumes**  $seq : (\sigma \models (s \leftarrow \text{mbind}_{\text{FailPurge}} (a\#S) \text{ ioprogram}; (P s)))$   
**and**  $none: \text{ioprogram } a \sigma = \text{None} \implies (\sigma \models (s \leftarrow \text{mbind}_{\text{FailPurge}} S \text{ ioprogram}; (P(s)))) \implies Q$   
**and**  $some: \bigwedge b \sigma'. \text{ioprogram } a \sigma = \text{Some}(b, \sigma') \implies (\sigma' \models (s \leftarrow \text{mbind}_{\text{FailPurge}} S \text{ ioprogram}; (P$   
 $(b\#s)))) \implies Q$   
**shows**  $Q$   
 $\langle \text{proof} \rangle$

**lemma** *assert-disch1* :  $P \sigma \implies (\sigma \models (x \leftarrow \text{assert}_{SE} P; M x)) = (\sigma \models (M \text{ True}))$   
 $\langle \text{proof} \rangle$

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

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

**lemma** *assert-disch4* :  $P \sigma \implies (\sigma \models (\text{assert}_{SE} P))$   
 ⟨*proof*⟩

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

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

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

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

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

**lemma** *assume-E* :  
**assumes** \* :  $\sigma \models (- \leftarrow \text{assume}_{SE} P ; M)$   
**and** \*\* :  $\bigwedge \sigma. P \sigma \implies \sigma \models M \implies Q$   
**shows**  $Q$   
 ⟨*proof*⟩

**lemma** *assume-E'* :  
**assumes** \* :  $\sigma \models \text{assume}_{SE} P ; - M$   
**and** \*\* :  $\bigwedge \sigma. P \sigma \implies \sigma \models M \implies Q$   
**shows**  $Q$   
 ⟨*proof*⟩

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

**term** *if<sub>SE</sub>*  $P \text{ then } B_1 \text{ else } B_2 \text{ fi}$

**lemma** *if-SE-D1* :  $P \sigma \implies (\sigma \models (\text{if}_{SE} P \text{ then } B_1 \text{ else } B_2 \text{ fi})) = (\sigma \models B_1)$   
 ⟨*proof*⟩

**lemma** *if-SE-D1'* :  $P \sigma \implies (\sigma \models (\text{if}_{SE} P \text{ then } B_1 \text{ else } B_2 \text{ fi}); - M) = (\sigma \models (B_1; - M))$   
 ⟨*proof*⟩

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

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

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

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

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

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

**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**  $Q$   
 ⟨proof⟩

**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**  $Q$   
 ⟨proof⟩

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

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



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

$$\begin{aligned} & \sigma \models (\text{if}_{SE} P \text{ then } B_1 \text{ else } B_2 \text{ fi}) ; - M \\ & \implies (P \sigma \implies \sigma \models B_1 ; - M \implies Q) \\ & \implies (\neg P \sigma \implies \sigma \models B_2 ; - M \implies Q) \\ & \implies Q \\ & \langle \text{proof} \rangle \end{aligned}$$

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

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

$$\begin{aligned} & \sigma \models (\text{if}_{SE} P \text{ then } B_1 \text{ else } B_2 \text{ fi}) ; - M \\ & \implies (P \sigma \implies \sigma \models B_1 ; - M \implies Q) \\ & \implies (\text{opaque} (\sigma \models (\text{if}_{SE} P \text{ then } B_1 \text{ else } B_2 \text{ fi}) ; - M) \implies Q) \\ & \implies Q \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** [*code*]:

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

**lemma** *monadic-sequence-rule*:

$$\begin{aligned} & \bigwedge X \sigma_1. (\sigma \models (- \leftarrow \text{assume}_{SE} (\lambda\sigma'. (\sigma=\sigma') \wedge P \sigma) ; x \leftarrow M ; \text{assert}_{SE} (\lambda\sigma. (x=X) \wedge \\ & (\sigma=\sigma_1) \wedge Q x \sigma))) \\ & \quad \wedge \\ & \quad (\sigma_1 \models (- \leftarrow \text{assume}_{SE} (\lambda\sigma. (\sigma=\sigma_1) \wedge Q x \sigma) ; y \leftarrow M' ; \text{assert}_{SE} (\lambda\sigma. R x y \sigma))) \\ & \implies \\ & \quad \sigma \models (- \leftarrow \text{assume}_{SE} (\lambda\sigma'. (\sigma=\sigma') \wedge P \sigma) ; x \leftarrow M ; y \leftarrow M' ; \text{assert}_{SE} (R x y)) \\ & \langle \text{proof} \rangle \end{aligned}$$

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

$\implies$

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

$\langle \text{proof} \rangle$

**lemma** *exec-skip*:

$$(\sigma \models \text{skip}_{SE} ; - M) = (\sigma \models M)$$

*<proof>*

**lemmas**  $exec\text{-}skipD = exec\text{-}skip[THEN\ iffD1]$

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**  $mbindFSave\text{-}vs\text{-}mbindFStop :$

$(\sigma \models (os \leftarrow (mbind_{FailSave}\ \iota\ ioprogram); result(length\ \iota s = length\ os \wedge P\ \iota s\ os))) =$   
 $(\sigma \models (os \leftarrow (mbind_{FailStop}\ \iota s\ ioprogram); result(P\ \iota s\ os)))$   
*<proof>*

**lemma**  $mbind_{FailSave}\text{-}vs\text{-}mbind_{FailStop}:$

**assumes**  $A: \forall\ \iota\ \sigma. ioprogram\ \iota\ \sigma \neq None$

**shows**  $(\sigma \models (os \leftarrow (mbind_{FailSave}\ \iota s\ ioprogram); P\ os)) =$   
 $(\sigma \models (os \leftarrow (mbind_{FailStop}\ \iota s\ ioprogram); P\ os))$   
*<proof>*

### 3.1.7 Miscellaneous

**no-notation**  $unit\text{-}SE\ ((result\ -)\ 8)$

**end**

**theory**  $Clean\text{-}Symbex$

**imports**  $Clean$

**begin**

## 3.2 Clean Symbolic Execution Rules

### 3.2.1 Basic NOP - Symbolic Execution Rules.

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

**lemma**  $non\text{-}exec\text{-}assign :$

**assumes**  $\neg exec\text{-}stop\ \sigma$

**shows**  $(\sigma \models (- \leftarrow assign\ f; M)) = ((f\ \sigma) \models M)$   
*<proof>*

**lemma**  $non\text{-}exec\text{-}assign' :$

**assumes**  $\neg exec\text{-}stop\ \sigma$

**shows**  $(\sigma \models (assign\ f; - M)) = ((f\ \sigma) \models M)$   
*<proof>*

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

**lemma** *exec-assign'* :  
**assumes** *exec-stop*  $\sigma$   
**shows**  $(\sigma \models (\text{assign } f; - M)) = (\sigma \models M)$   
 $\langle \text{proof} \rangle$

### 3.2.2 Assign Execution Rules.

**lemma** *non-exec-assign-global* :  
**assumes**  $\neg \text{exec-stop } \sigma$   
**shows**  $(\sigma \models (- \leftarrow \text{assign-global upd rhs; } M)) = ((\text{upd } (\lambda-. \text{ rhs } \sigma) \sigma) \models M)$   
 $\langle \text{proof} \rangle$

**lemma** *non-exec-assign-global'* :  
**assumes**  $\neg \text{exec-stop } \sigma$   
**shows**  $(\sigma \models (\text{assign-global upd rhs; } - M)) = ((\text{upd } (\lambda-. \text{ rhs } \sigma) \sigma) \models M)$   
 $\langle \text{proof} \rangle$

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

**lemma** *exec-assign-global'* :  
**assumes** *exec-stop*  $\sigma$   
**shows**  $(\sigma \models (\text{assign-global upd rhs; } - M)) = (\sigma \models M)$   
 $\langle \text{proof} \rangle$

**lemma** *non-exec-assign-local* :  
**assumes**  $\neg \text{exec-stop } \sigma$   
**shows**  $(\sigma \models (- \leftarrow \text{assign-local upd rhs; } M)) = ((\text{upd } (\text{map-hd } (\lambda-. \text{ rhs } \sigma)) \sigma) \models M)$   
 $\langle \text{proof} \rangle$

**lemma** *non-exec-assign-local'* :  
**assumes**  $\neg \text{exec-stop } \sigma$   
**shows**  $(\sigma \models (\text{assign-local upd rhs; } - M)) = ((\text{upd } (\text{map-hd } (\lambda-. \text{ rhs } \sigma)) \sigma) \models M)$   
 $\langle \text{proof} \rangle$

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

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

**lemma** *exec-assign-local'* :  
**assumes** *exec-stop*  $\sigma$   
**shows**  $(\sigma \models (\text{assign-local upd rhs};- M)) = (\sigma \models M)$   
 $\langle \text{proof} \rangle$

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

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

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

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

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

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

### 3.2.3 Basic Call Symbolic Execution Rules.

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

**lemma** *exec-call-0'* :  
**assumes** *exec-stop*  $\sigma$   
**shows**  $(\sigma \models (\text{call-0}_C M;- M')) = (\sigma \models M')$   
 $\langle \text{proof} \rangle$

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

**lemma** *exec-call-1'* :  
**assumes** *exec-stop*  $\sigma$   
**shows**  $(\sigma \models (\text{call-1}_C M A_1;- M')) = (\sigma \models M')$   
 $\langle \text{proof} \rangle$

**lemma** *exec-call* :  
**assumes** *exec-stop*  $\sigma$   
**shows**  $(\sigma \models (x \leftarrow \text{call}_C M A_1; M' x)) = (\sigma \models M' \text{ undefined})$   
 $\langle \text{proof} \rangle$

**lemma** *exec-call'* :  
**assumes** *exec-stop*  $\sigma$   
**shows**  $(\sigma \models (\text{call}_C M A_1; - M')) = (\sigma \models M')$   
 $\langle \text{proof} \rangle$

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

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

### 3.2.4 Basic Call Symbolic Execution Rules.

**lemma** *non-exec-call-0* :  
**assumes**  $\neg \text{exec-stop } \sigma$   
**shows**  $(\sigma \models (- \leftarrow \text{call-0}_C M; M')) = (\sigma \models M; - M')$   
 $\langle \text{proof} \rangle$

**lemma** *non-exec-call-0'* :  
**assumes**  $\neg \text{exec-stop } \sigma$   
**shows**  $(\sigma \models \text{call-0}_C M; - M') = (\sigma \models M; - M')$   
 $\langle \text{proof} \rangle$

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

**lemma** *non-exec-call-1'* :  
**assumes**  $\neg \text{exec-stop } \sigma$   
**shows**  $(\sigma \models \text{call-1}_C M A_1; - M') = (\sigma \models M (A_1 \sigma); - M')$   
 $\langle \text{proof} \rangle$

**lemma** *non-exec-call* :  
**assumes**  $\neg \text{exec-stop } \sigma$   
**shows**  $(\sigma \models (x \leftarrow (\text{call}_C M A_1); M' x)) = (\sigma \models (x \leftarrow M (A_1 \sigma); M' x))$   
 $\langle \text{proof} \rangle$

**lemma** *non-exec-call'* :  
**assumes**  $\neg \text{exec-stop } \sigma$   
**shows**  $(\sigma \models \text{call}_C M A_1; - M') = (\sigma \models M (A_1 \sigma); - M')$   
 $\langle \text{proof} \rangle$

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

**lemma** *non-exec-call-2'* :  
**assumes**  $\neg \text{exec-stop } \sigma$   
**shows**  $(\sigma \models \text{call-2}_C M A_1 A_2; - M') = (\sigma \models M (A_1 \sigma) (A_2 \sigma); - M')$   
 $\langle \text{proof} \rangle$

### 3.2.5 Conditional.

**lemma** *exec-If<sub>C</sub>-If<sub>SE</sub>* :  
**assumes**  $\neg \text{exec-stop } \sigma$   
**shows**  $((\text{if}_C P \text{ then } B_1 \text{ else } B_2 \text{ fi}))\sigma = ((\text{if}_{SE} P \text{ then } B_1 \text{ else } B_2 \text{ fi})) \sigma$   
 $\langle \text{proof} \rangle$

**lemma** *valid-exec-If<sub>C</sub>* :  
**assumes**  $\neg \text{exec-stop } \sigma$   
**shows**  $(\sigma \models (\text{if}_C P \text{ then } B_1 \text{ else } B_2 \text{ fi}); - M) = (\sigma \models (\text{if}_{SE} P \text{ then } B_1 \text{ else } B_2 \text{ fi}); - M)$   
 $\langle \text{proof} \rangle$

**lemma** *exec-If<sub>C</sub>'* :  
**assumes**  $\text{exec-stop } \sigma$   
**shows**  $(\sigma \models (\text{if}_C P \text{ then } B_1 \text{ else } B_2 \text{ fi}); - M) = (\sigma \models M)$   
 $\langle \text{proof} \rangle$

**lemma** *exec-While<sub>C</sub>'* :  
**assumes**  $\text{exec-stop } \sigma$   
**shows**  $(\sigma \models (\text{while}_C P \text{ do } B_1 \text{ od}); - M) = (\sigma \models M)$   
 $\langle \text{proof} \rangle$

**lemma** *if<sub>C</sub>-cond-cong* :  $f \sigma = g \sigma \implies$   
 $(\text{if}_C f \text{ then } c \text{ else } d \text{ fi}) \sigma =$   
 $(\text{if}_C g \text{ then } c \text{ else } d \text{ fi}) \sigma$   
 $\langle \text{proof} \rangle$

### 3.2.6 Break - Rules.

**lemma** *break-assign-skip [simp]*:  $\text{break}; - \text{assign } f = \text{break}$   
 $\langle \text{proof} \rangle$

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

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

**lemma** *unset-break-idem* [simp] :  
 $(unset\_break\_status ; - unset\_break\_status ; - M) = (unset\_break\_status ; - M)$   
 ⟨proof⟩

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

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

### 3.2.7 While.

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

Various tactics for various coverage criteria

**definition** *while-k* ::  $nat \Rightarrow (('sigma\text{-ext}) \text{ control-state-ext} \Rightarrow bool)$   
 $\Rightarrow (unit, ('sigma\text{-ext}) \text{ control-state-ext}) MON_{SE}$   
 $\Rightarrow (unit, ('sigma\text{-ext}) \text{ control-state-ext}) MON_{SE}$

**where**  $while\text{-}k \equiv while\text{-}C$

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

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

**lemma** *while-k-SE* :  $while\text{-}C = while\text{-}k \ k$   
 ⟨proof⟩

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

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

$\langle ML \rangle$

**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**

### 3.3 Hoare

**definition** *hoare<sub>3</sub>* :: ( $'\sigma \Rightarrow \text{bool}$ )  $\Rightarrow$  ( $'\alpha, '\sigma$ )*MON<sub>SE</sub>*  $\Rightarrow$  ( $'\alpha \Rightarrow '\sigma \Rightarrow \text{bool}$ )  $\Rightarrow$  *bool* ( $\{\{1-\}\} / (-) / \{\{1-\}\} 50$ )

**where**  $\{\{P\}\} M \{\{Q\}\} \equiv (\forall \sigma. P \sigma \longrightarrow (\text{case } M \sigma \text{ of } \text{None} \Rightarrow \text{False} \mid \text{Some}(x, \sigma') \Rightarrow Q x \sigma'))$

**definition** *hoare<sub>3</sub>'* :: ( $'\sigma \Rightarrow \text{bool}$ )  $\Rightarrow$  ( $'\alpha, '\sigma$ )*MON<sub>SE</sub>*  $\Rightarrow$  *bool* ( $\{\{1-\}\} / (-) / \dagger 50$ )

**where**  $\{\{P\}\} M \dagger \equiv (\forall \sigma. P \sigma \longrightarrow (\text{case } M \sigma \text{ of } \text{None} \Rightarrow \text{True} \mid - \Rightarrow \text{False}))$

#### 3.3.1 Basic rules

**lemma** *skip*:  $\{\{P\}\} \text{skip}_{SE} \{\{\lambda-. P\}\}$   
 $\langle \text{proof} \rangle$

**lemma** *fail*:  $\{\{P\}\} \text{fail}_{SE} \dagger$   
 $\langle \text{proof} \rangle$

**lemma** *assert*:  $\{\{P\}\} \text{assert}_{SE} P \{\{\lambda -. \text{True}\}\}$   
 $\langle \text{proof} \rangle$



**lemma** *assert-conseq*:  $\text{Collect } P \subseteq \text{Collect } Q \implies \{P\} \text{ assert}_{SE} Q \{ \lambda \cdot \cdot. \text{True} \}$   
 ⟨proof⟩

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

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

### 3.3.2 Generalized and special sequence rules

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

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

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

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

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

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

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

### 3.3.3 Generalized and special consequence rules

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

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

**and**  $(\bigwedge \sigma. Q' \sigma \longrightarrow Q \sigma)$   
**shows**  $\{\!|P|\!\} M \{\!|\lambda x \sigma. Q \sigma|\!\}$   
 $\langle proof \rangle$

**lemma** *consequence-irpt* :  
 $Collect P \subseteq Collect P'$   
 $\implies \{\!|P'|\!\} M \dagger$   
 $\implies \{\!|P|\!\} M \dagger$   
 $\langle proof \rangle$

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

### 3.3.4 Condition rules

**lemma** *cond* :  
 $\{\!|\lambda\sigma. P \sigma \wedge cond \sigma|\!\} M \{\!|Q|\!\}$   
 $\implies \{\!|\lambda\sigma. P \sigma \wedge \neg cond \sigma|\!\} M' \{\!|Q|\!\}$   
 $\implies \{\!|P|\!\} if_{SE} cond then M else M' fi \{\!|Q|\!\}$   
 $\langle proof \rangle$

**lemma** *cond-irpt* :  
 $\{\!|\lambda\sigma. P \sigma \wedge cond \sigma|\!\} M \dagger$   
 $\implies \{\!|\lambda\sigma. P \sigma \wedge \neg cond \sigma|\!\} M' \dagger$   
 $\implies \{\!|P|\!\} if_{SE} cond then M else M' fi \dagger$   
 $\langle proof \rangle$

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

### 3.3.5 While rules

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

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

**lemma** *while-irpt* :  
**assumes**  $*$  :  $\{\!|\lambda\sigma. cond \sigma \wedge P \sigma|\!\} M \{\!|\lambda-. P|\!\} \vee \{\!|\lambda\sigma. cond \sigma \wedge P \sigma|\!\} M \dagger$   
**and** *measure*:  $\forall \sigma. cond \sigma \wedge P \sigma \longrightarrow M \sigma = None \vee f(snd(the(M \sigma))) < ((f \sigma)::nat)$

**and** *enabled*:  $\forall \sigma. P \sigma \longrightarrow \text{cond } \sigma$   
**shows**  $\{\{P\}\} \text{while}_{SE} \text{cond do } M \text{ od } \dagger$   
*<proof>*

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

**definition** *hoare<sub>1</sub>* ::  $(\sigma \Rightarrow \text{bool}) \Rightarrow ('\alpha, '\sigma) \text{MON}_{SE} \Rightarrow ('\alpha \Rightarrow '\sigma \Rightarrow \text{bool}) \Rightarrow \text{bool}$  ( $\vdash_1$  ( $\{(1-)\}/(-)/\{(1-)\}$ ) 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** *hoare<sub>2</sub>* ::  $(\sigma \Rightarrow \text{bool}) \Rightarrow ('\alpha, '\sigma) \text{MON}_{SE} \Rightarrow ('\alpha \Rightarrow '\sigma \Rightarrow \text{bool}) \Rightarrow \text{bool}$  ( $\vdash_2$  ( $\{(1-)\}/(-)/\{(1-)\}$ ) 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**

### 3.3.7 Clean Control Rules

**lemma** *break1*:  
 $\{\{\lambda \sigma. P (\sigma \mid \text{break-status} := \text{True})\}\} \text{break} \{\{\lambda r \sigma. P \sigma \wedge \text{break-status } \sigma\}\}$   
*<proof>*

**lemma** *unset-break1*:  
 $\{\{\lambda \sigma. P (\sigma \mid \text{break-status} := \text{False})\}\} \text{unset-break-status} \{\{\lambda r \sigma. P \sigma \wedge \neg \text{break-status } \sigma\}\}$   
*<proof>*

**lemma** *set-return1*:  
 $\{\{\lambda \sigma. P (\sigma \mid \text{return-status} := \text{True})\}\} \text{set-return-status} \{\{\lambda r \sigma. P \sigma \wedge \text{return-status } \sigma\}\}$   
*<proof>*

**lemma** *unset-return1*:  
 $\{\{\lambda \sigma. P (\sigma \mid \text{return-status} := \text{False})\}\} \text{unset-return-status} \{\{\lambda r \sigma. P \sigma \wedge \neg \text{return-status } \sigma\}\}$   
*<proof>*

### 3.3.8 Clean Skip Rules

**lemma** *assign-global-skip*:  
 $\{\{\lambda \sigma. \text{exec-stop } \sigma \wedge P \sigma\}\} \text{assign-global upd rhs} \{\{\lambda r \sigma. \text{exec-stop } \sigma \wedge P \sigma\}\}$

$\langle proof \rangle$

**lemma** *assign-local-skip*:

$\{\{\lambda\sigma. \text{exec-stop } \sigma \wedge P \sigma\} \text{ assign-local upd rhs } \{\{\lambda r \sigma. \text{exec-stop } \sigma \wedge P \sigma\}\}$   
 $\langle proof \rangle$

**lemma** *return-skip*:

$\{\{\lambda\sigma. \text{exec-stop } \sigma \wedge P \sigma\} \text{ return}_C \text{ upd rhs } \{\{\lambda r \sigma. \text{exec-stop } \sigma \wedge P \sigma\}\}$   
 $\langle proof \rangle$

**lemma** *assign-clean-skip*:

$\{\{\lambda\sigma. \text{exec-stop } \sigma \wedge P \sigma\} \text{ assign tr } \{\{\lambda r \sigma. \text{exec-stop } \sigma \wedge P \sigma\}\}$   
 $\langle proof \rangle$

**lemma** *if-clean-skip*:

$\{\{\lambda\sigma. \text{exec-stop } \sigma \wedge P \sigma\} \text{ if}_C \text{ C then E else F fi } \{\{\lambda r \sigma. \text{exec-stop } \sigma \wedge P \sigma\}\}$   
 $\langle proof \rangle$

**lemma** *while-clean-skip*:

$\{\{\lambda\sigma. \text{exec-stop } \sigma \wedge P \sigma\} \text{ while}_C \text{ cond do body od } \{\{\lambda r \sigma. \text{exec-stop } \sigma \wedge P \sigma\}\}$   
 $\langle proof \rangle$

**lemma** *if-opcall-skip*:

$\{\{\lambda\sigma. \text{exec-stop } \sigma \wedge P \sigma\} (\text{call}_C M A_1) \{\{\lambda r \sigma. \text{exec-stop } \sigma \wedge P \sigma\}\}$   
 $\langle proof \rangle$

**lemma** *if-funcall-skip*:

$\{\{\lambda\sigma. \text{exec-stop } \sigma \wedge P \sigma\} (p_{tmp} \leftarrow \text{call}_C \text{ fun } E ; \text{assign-local upd } (\lambda\sigma. p_{tmp})) \{\{\lambda r \sigma. \text{exec-stop } \sigma \wedge P \sigma\}\}$   
 $\langle proof \rangle$

**lemma** *if-funcall-skip'*:

$\{\{\lambda\sigma. \text{exec-stop } \sigma \wedge P \sigma\} (p_{tmp} \leftarrow \text{call}_C \text{ fun } E ; \text{assign-global upd } (\lambda\sigma. p_{tmp})) \{\{\lambda r \sigma. \text{exec-stop } \sigma \wedge P \sigma\}\}$   
 $\langle proof \rangle$

### 3.3.9 Clean Assign Rules

**lemma** *assign-global*:

**assumes**  $* : \# \text{ upd}$

**shows**  $\{\{\lambda\sigma. \neg \text{exec-stop } \sigma \wedge P (\text{upd } (\lambda-. \text{rhs } \sigma) \sigma)\}$

$\text{assign-global upd rhs}$

$\{\{\lambda r \sigma. \neg \text{exec-stop } \sigma \wedge P \sigma\}\}$

$\langle proof \rangle$

**lemma** *assign-local*:

**assumes**  $* : \# (\text{upd} \circ \text{map-hd})$

**shows**  $\{\{\lambda\sigma. \neg \text{exec-stop } \sigma \wedge P ((\text{upd} \circ \text{map-hd}) (\lambda-. \text{rhs } \sigma) \sigma)\}$

$\text{assign-local upd rhs}$

$\{\lambda r \sigma. \neg \text{exec-stop } \sigma \wedge P \sigma\}$   
 <proof>

**lemma** *return-assign*:

**assumes**  $*$  :  $\# (\text{upd} \circ \text{map-hd})$

**shows**  $\{\lambda \sigma. \neg \text{exec-stop } \sigma \wedge P ((\text{upd} \circ \text{map-hd}) (\lambda-. \text{rhs } \sigma)) (\sigma \langle \text{return-status} := \text{True} \rangle))\}$   
 $\text{return}_C \text{ upd rhs}$

$\{\lambda r \sigma. P \sigma \wedge \text{return-status } \sigma\}$   
 <proof>

### 3.3.10 Clean Construct Rules

**lemma** *cond-clean* :

$\{\lambda \sigma. \neg \text{exec-stop } \sigma \wedge P \sigma \wedge \text{cond } \sigma\} M \{Q\}$   
 $\implies \{\lambda \sigma. \neg \text{exec-stop } \sigma \wedge P \sigma \wedge \neg \text{cond } \sigma\} M' \{Q\}$   
 $\implies \{\lambda \sigma. \neg \text{exec-stop } \sigma \wedge P \sigma\} \text{if}_C \text{ cond then } M \text{ else } M' \text{ fi} \{Q\}$   
 <proof>

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

**lemma** *while-clean-no-break* :

**assumes**  $*$  :  $\{\lambda \sigma. \neg \text{break-status } \sigma \wedge \text{cond } \sigma \wedge P \sigma\} M \{\lambda-. \lambda \sigma. \neg \text{break-status } \sigma \wedge P \sigma\}$   
**and measure**:  $\forall \sigma. \neg \text{exec-stop } \sigma \wedge \text{cond } \sigma \wedge P \sigma$   
 $\longrightarrow M \sigma \neq \text{None} \wedge f(\text{snd}(\text{the}(M \sigma))) < ((f \sigma)::\text{nat})$   
**(is**  $\forall \sigma. \neg \text{cond } \sigma \wedge P \sigma \longrightarrow ?\text{decrease } \sigma$   
**shows**  $\{\lambda \sigma. \neg \text{exec-stop } \sigma \wedge P \sigma\}$   
 $\text{while}_C \text{ cond do } M \text{ od}$   
 $\{\lambda-. \sigma. (\text{return-status } \sigma \vee \neg \text{cond } \sigma) \wedge \neg \text{break-status } \sigma \wedge P \sigma\}$   
**(is**  $\{\lambda \sigma. ?\text{pre}\} \text{while}_C \text{ cond do } M \text{ od } \{\lambda-. \sigma. ?\text{post1 } \sigma \wedge ?\text{post2 } \sigma\}$   
 <proof>

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

**lemma** *while-clean'*:

**assumes** *M-inv* :  $\{\lambda \sigma. \neg \text{exec-stop } \sigma \wedge \text{cond } \sigma \wedge P \sigma\} M \{\lambda-. P\}$   
**and cond-idpc** :  $\forall x \sigma. (\text{cond } (\sigma \langle \text{break-status} := x \rangle)) = \text{cond } \sigma$   
**and inv-idpc** :  $\forall x \sigma. (P (\sigma \langle \text{break-status} := x \rangle)) = P \sigma$   
**and f-is-measure** :  $\forall \sigma. \neg \text{exec-stop } \sigma \wedge \text{cond } \sigma \wedge P \sigma \longrightarrow$   
 $M \sigma \neq \text{None} \wedge f(\text{snd}(\text{the}(M \sigma))) < ((f \sigma)::\text{nat})$   
**shows**  $\{\lambda \sigma. \neg \text{exec-stop } \sigma \wedge P \sigma\}$   
 $\text{while}_C \text{ cond do } M \text{ od}$   
 $\{\lambda-. \sigma. \neg \text{break-status } \sigma \wedge P \sigma\}$

*<proof>*

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](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](https://doi.org/10.1007/978-3-319-92994-1_6).
- [5] F. Tuong and B. Wolff. Deeply integrating C11 code support into isabelle/pide. In *Formal IDE - 5th International Workshop, F-IDE 2019, Porto, Portugal, October 7, 2019. Proceedings*, 2019. URL [https://gitlri.lri.fr/ftuong/isabelle\\_c](https://gitlri.lri.fr/ftuong/isabelle_c).