

IMP2 — Simple Program Verification in Isabelle/HOL

Peter Lammich Simon Wimmer

April 10, 2026

Abstract

IMP2 is a simple imperative language together with Isabelle tooling to create a program verification environment in Isabelle/HOL. The tools include a C-like syntax, a verification condition generator, and Isabelle commands for the specification of programs. The framework is modular, i.e., it allows easy reuse of already proved programs within larger programs.

This entry comes with a quickstart guide and a large collection of examples, spanning basic algorithms with simple proofs to more advanced algorithms and proof techniques like data refinement. Some highlights from the examples are: Bisection Square Root, Extended Euclid, Exponentiation by Squaring, Binary Search, Insertion Sort, Quicksort, Depth First Search.

The abstract syntax and semantics are very simple and well-documented. They are suitable to be used in a course, as extension to the IMP language which comes with the Isabelle distribution.

While this entry is limited to a simple imperative language, the ideas could be extended to more sophisticated languages.

Contents

| | | |
|----------|-----------------------------------|----------|
| 1 | Abstract Syntax of IMP2 | 1 |
| 1.1 | Primitives | 1 |
| 1.2 | Arithmetic Expressions | 2 |
| 1.3 | Boolean Expressions | 2 |
| 1.4 | Commands | 3 |
| 1.4.1 | Minimal Concrete Syntax | 4 |
| 1.5 | Program | 4 |
| 1.6 | Default Array Index | 4 |
| 2 | Semantics of IMP | 4 |
| 2.1 | State | 4 |
| 2.1.1 | State Combination | 5 |
| 2.2 | Arithmetic Expressions | 5 |
| 2.3 | Boolean Expressions | 6 |

| | | |
|----------|--|-----------|
| 2.4 | Big-Step Semantics | 6 |
| 2.4.1 | Proof Automation | 7 |
| 2.4.2 | Automatic Derivation | 7 |
| 2.5 | Command Equivalence | 8 |
| 2.5.1 | Basic Equivalences | 9 |
| 2.6 | Execution is Deterministic | 9 |
| 2.7 | Small-Step Semantics | 10 |
| 2.7.1 | Equivalence to Big-Step Semantics | 11 |
| 2.8 | Weakest Precondition | 14 |
| 2.8.1 | Basic Properties | 14 |
| 2.8.2 | Unfold Rules | 15 |
| 2.8.3 | Weakest precondition and Program Equivalence | 16 |
| 2.8.4 | While Loops and Weakest Precondition | 16 |
| 2.9 | Invariants for While-Loops | 17 |
| 2.9.1 | Partial Correctness | 17 |
| 2.9.2 | Total Correctness | 18 |
| 2.9.3 | Standard Forms of While Rules | 19 |
| 2.10 | Modularity of Programs | 19 |
| 2.11 | Strongest Postcondition | 20 |
| 2.12 | Hoare-Triples | 20 |
| 2.12.1 | Sets of Hoare-Triples | 21 |
| 2.12.2 | Deriving Parameter Frame Adjustment Rules | 21 |
| 2.12.3 | Proof for Recursive Specifications | 22 |
| 2.13 | Completeness of While-Rule | 24 |
| 3 | Annotated Syntax | 26 |
| 3.1 | Annotations | 26 |
| 3.2 | Hoare-Triples for Annotated Commands | 26 |
| 4 | Quickstart Guide | 28 |
| 4.1 | Introductory Examples | 28 |
| 4.1.1 | Variant and Invariant Annotations | 29 |
| 4.1.2 | Recursive Procedures | 30 |
| 4.2 | The VCG | 30 |
| 4.3 | Advanced Features | 31 |
| 4.3.1 | Custom Termination Relations | 31 |
| 4.3.2 | Partial Correctness | 32 |
| 4.3.3 | Arrays | 32 |
| 4.4 | Proving Techniques | 32 |
| 4.4.1 | Auxiliary Lemmas | 33 |
| 4.4.2 | Inlining | 33 |
| 4.4.3 | Functional Refinement | 33 |
| 4.4.4 | Data Refinement | 33 |
| 4.5 | Troubleshooting | 33 |
| 4.5.1 | Invalid Variables in Annotations | 34 |
| 4.5.2 | Wrong Annotations | 34 |
| 4.5.3 | Calls to Undefined Procedures | 34 |
| 4.6 | Missing Features | 35 |
| 4.6.1 | Elaborate Warnings and Errors | 35 |
| 4.6.2 | Static Type Checking | 35 |

| | | |
|----------|--|-----------|
| 4.6.3 | Structure Types | 35 |
| 4.6.4 | Function Calls as Expressions | 36 |
| 4.6.5 | Ghost Variables | 36 |
| 4.6.6 | Concurrency | 36 |
| 4.6.7 | Pointers and Memory | 36 |
| 5 | Introduction to IMP2-VCG, based on IMP | 36 |
| 5.1 | Fancy Syntax | 37 |
| 5.2 | Operators and Arrays | 37 |
| 5.3 | Local and Global Variables | 38 |
| 5.3.1 | Parameter Passing | 38 |
| 5.4 | Recursive procedures | 38 |
| 5.4.1 | Procedure Scope | 38 |
| 5.4.2 | Syntactic sugar for procedure call with parameters | 38 |
| 5.5 | More Readable VCs | 39 |
| 5.6 | Specification Commands | 39 |
| 6 | Examples | 41 |
| 6.1 | Common Loop Patterns | 41 |
| 6.1.1 | Count Up | 41 |
| 6.1.2 | Count down | 44 |
| 6.1.3 | Approximate from Below | 44 |
| 6.1.4 | Bisection | 45 |
| 6.2 | Debugging | 46 |
| 6.2.1 | Testing Programs | 46 |
| 6.3 | More Numeric Algorithms | 46 |
| 6.3.1 | Euclid's Algorithm (with subtraction) | 46 |
| 6.3.2 | Euclid's Algorithm (with mod) | 46 |
| 6.3.3 | Extended Euclid's Algorithm | 47 |
| 6.3.4 | Exponentiation by Squaring | 49 |
| 6.3.5 | Power-Tower of 2s | 50 |
| 6.4 | Array Algorithms | 52 |
| 6.4.1 | Summation | 52 |
| 6.4.2 | Finding Least Index of Element | 52 |
| 6.4.3 | Check for Sortedness | 52 |
| 6.4.4 | Find Equilibrium Index | 53 |
| 6.4.5 | Rotate Right | 53 |
| 6.4.6 | Binary Search, Leftmost Element | 54 |
| 6.4.7 | Naive String Search | 55 |
| 6.4.8 | Insertion Sort | 59 |
| 6.4.9 | Quicksort | 62 |
| 6.5 | Data Refinement | 65 |
| 6.5.1 | Filtering | 65 |
| 6.5.2 | Merge Two Sorted Lists | 65 |
| 6.5.3 | Remove Duplicates from Array, using Bitvector Set | 68 |
| 6.6 | Recursion | 70 |
| 6.6.1 | Recursive Fibonacci | 70 |
| 6.6.2 | Homeier's Cycling Termination | 71 |
| 6.6.3 | Ackermann | 71 |

| | | |
|-------|---|----|
| 6.6.4 | McCarthy's 91 Function | 72 |
| 6.6.5 | Odd/Even | 72 |
| 6.6.6 | Pandya and Joseph's Product Producers | 73 |
| 6.7 | Graph Algorithms | 74 |
| 6.7.1 | DFS | 74 |

1 Abstract Syntax of IMP2

```
theory Syntax
imports Main
begin
```

We define the abstract syntax of the IMP2 language, and a minimal concrete syntax for direct use in terms.

1.1 Primitives

Variable and procedure names are strings.

```
type-synonym vname = string
type-synonym pname = string
```

The variable names are partitioned into local and global variables.

```
fun is-global :: vname  $\Rightarrow$  bool where
  is-global []  $\longleftrightarrow$  True
| is-global (CHR "G"#-)  $\longleftrightarrow$  True
| is-global -  $\longleftrightarrow$  False
```

```
abbreviation is-local a  $\equiv$   $\neg$ is-global a
```

Primitive values are integers, and values are arrays modeled as functions from integers to primitive values.

Note that values and primitive values are usually part of the semantics, however, as they occur as literals in the abstract syntax, we already define them here.

```
type-synonym pval = int
type-synonym val = int  $\Rightarrow$  pval
```

1.2 Arithmetic Expressions

Arithmetic expressions consist of constants, indexed array variables, and unary and binary operations. The operations are modeled by reflecting arbitrary functions into the abstract syntax.

```
datatype aexp =
  N int
```

```

| Vidx vname aexp
| Unop int  $\Rightarrow$  int aexp
| Binop int  $\Rightarrow$  int  $\Rightarrow$  int aexp aexp

```

1.3 Boolean Expressions

Boolean expressions consist of constants, the not operation, binary connectives, and comparison operations. Binary connectives and comparison operations are modeled by reflecting arbitrary functions into the abstract syntax. The not operation is the only meaningful unary Boolean operation, so we chose to model it explicitly instead of reflecting and unary Boolean function.

```

datatype bexp =
  Bc bool
| Not bexp
| BBinop bool  $\Rightarrow$  bool  $\Rightarrow$  bool bexp bexp
| Cmpop int  $\Rightarrow$  int  $\Rightarrow$  bool aexp aexp

```

1.4 Commands

The commands can roughly be put into five categories:

Skip The no-op command

Assignment commands Commands to assign the value of an arithmetic expression, copy or clear arrays, and a command to simultaneously assign all local variables, which is only used internally to simplify the definition of a small-step semantics.

Block commands The standard sequential composition, if-then-else, and while commands, and a scope command which executes a command with a fresh set of local variables.

Procedure commands Procedure call, and a procedure scope command, which executes a command in a specified procedure environment. Similar to the scope command, which introduces new local variables, and thus limits the effect of variable manipulations to the content of the command, the procedure scope command introduces new procedures, and limits the validity of their names to the content of the command. This greatly simplifies modular definition of programs, as procedure names can be used locally.

```

datatype
com =
  SKIP — No-op

```

| | | |
|--|---|--|
| — Assignment | | |
| <i>AssignIdx</i> <i>vname aexp aexp</i> | — Assign to index in array | |
| <i>ArrayCpy</i> <i>vname vname</i> | — Copy whole array | |
| <i>ArrayClear</i> <i>vname</i> | — Clear array | |
| <i>Assign-Locals</i> <i>vname</i> \Rightarrow <i>val</i> | — Internal: Assign all local variables simultaneously | |
| — Block | | |
| <i>Seq</i> <i>com com</i> | — Sequential composition | |
| <i>If</i> <i>bexp com com</i> | — Conditional | |
| <i>While</i> <i>bexp com</i> | — While-loop | |
| <i>Scope</i> <i>com</i> | — Local variable scope | |
| — Procedure | | |
| <i>PCall</i> <i>pname</i> | — Procedure call | |
| <i>PScope</i> <i>pname</i> \rightarrow <i>com com</i> | — Procedure scope | |

1.4.1 Minimal Concrete Syntax

The commands come with a minimal concrete syntax, which is compatible to the syntax of *IMP*.

| | |
|-----------------------------------|--|
| notation <i>AssignIdx</i> | $\langle \langle - \rangle ::= - \rightarrow [1000, 0, 61] 61 \rangle$ |
| notation <i>ArrayCpy</i> | $\langle \langle - \rangle ::= - \rightarrow [1000, 1000] 61 \rangle$ |
| notation <i>ArrayClear</i> | $\langle \langle CLEAR - \rangle [1000] 61 \rangle$ |
| notation <i>Seq</i> | $\langle \langle -; / - \rangle [61, 60] 60 \rangle$ |
| notation <i>If</i> | $\langle \langle IF - / THEN - / ELSE - \rangle [0, 0, 61] 61 \rangle$ |
| notation <i>While</i> | $\langle \langle WHILE - / DO - \rangle [0, 61] 61 \rangle$ |
| notation <i>Scope</i> | $\langle \langle SCOPE - \rangle [61] 61 \rangle$ |

1.5 Program

type-synonym *program* = *pname* \rightarrow *com*

1.6 Default Array Index

We define abbreviations to make arrays look like plain integer variables: Without explicitly specifying an array index, the index 0 will be used automatically.

| |
|---|
| abbreviation $V x \equiv Vid x (N 0)$ |
| abbreviation <i>Assign</i> $\langle - ::= - \rightarrow [1000, 61] 61 \rangle$ |
| where $x ::= a \equiv (x[N 0] ::= a)$ |

end

2 Semantics of IMP

```
theory Semantics
imports Syntax HOL-Eisbach.Eisbach-Tools
begin
```

2.1 State

The state maps variable names to values

```
type-synonym state = vname  $\Rightarrow$  val
```

We introduce some syntax for the null state, and a state where only certain variables are set.

```
definition null-state ( $\langle\langle\rangle\rangle$ ) where
  null-state  $\equiv$   $\lambda x. \lambda i. 0$ 
```

syntax

```
-State :: updbinds  $\Rightarrow$  'a ( $\langle\langle-\rangle\rangle$ )
```

translations

```
-State ms == -Update  $\langle\rangle$  ms
```

```
-State (-updbinds b bs) <= -Update (-State b) bs
```

2.1.1 State Combination

The state combination operator constructs a state by taking the local variables from one state, and the globals from another state.

```
definition combine-states :: state  $\Rightarrow$  state  $\Rightarrow$  state ( $\langle\langle-|->\rangle$  [0,0]
1000)
```

```
  where  $\langle s|t \rangle n =$  (if is-local n then s n else t n)
```

We prove some basic facts.

Note that we use Isabelle's context command to locally declare the definition of *combine-states* as simp lemma, such that it is unfolded automatically.

```
context notes [simp] = combine-states-def begin
```

```
lemma combine-collapse:  $\langle s|s \rangle = s$  by auto
```

```
lemma combine-nest:
```

```
   $\langle s|\langle s'|t \rangle \rangle = \langle s|t \rangle$ 
```

```
   $\langle \langle s|t' \rangle|t \rangle = \langle s|t \rangle$ 
```

```
  by auto
```

```
lemma combine-query:
```

```
  is-local x  $\Longrightarrow$   $\langle s|t \rangle x = s x$ 
```

```
  is-global x  $\Longrightarrow$   $\langle s|t \rangle x = t x$ 
```

```
  by auto
```

lemma *combine-upd*:
is-local $x \implies \langle s|t \rangle(x:=v) = \langle s(x:=v)|t \rangle$
is-global $x \implies \langle s|t \rangle(x:=v) = \langle s|t(x:=v) \rangle$
by *auto*

lemma *combine-cases*[*cases type*]:
obtains $l\ g$ **where** $s = \langle l|g \rangle$
by (*fastforce*)

end

2.2 Arithmetic Expressions

The evaluation of arithmetic expressions is straightforward.

fun *aval* :: *aexp* \Rightarrow *state* \Rightarrow *pval* **where**
aval (*N* n) $s = n$
| *aval* (*Vid* $x\ i$) $s = s\ x\ (aval\ i\ s)$
| *aval* (*Unop* $f\ a_1$) $s = f\ (aval\ a_1\ s)$
| *aval* (*Binop* $f\ a_1\ a_2$) $s = f\ (aval\ a_1\ s)\ (aval\ a_2\ s)$

2.3 Boolean Expressions

The evaluation of Boolean expressions is straightforward.

fun *bval* :: *bexp* \Rightarrow *state* \Rightarrow *bool* **where**
bval (*Bc* v) $s = v$
| *bval* (*Not* b) $s = (\neg\ bval\ b\ s)$
| *bval* (*BBinop* $f\ b_1\ b_2$) $s = f\ (bval\ b_1\ s)\ (bval\ b_2\ s)$
| *bval* (*Cmpop* $f\ a_1\ a_2$) $s = f\ (aval\ a_1\ s)\ (aval\ a_2\ s)$

2.4 Big-Step Semantics

The big-step semantics is a relation from commands and start states to end states, such that there is a terminating execution.

If there is no such execution, no end state will be related to the command and start state. This either means that the program does not terminate, or gets stuck because it tries to call an undefined procedure.

The inference rules of the big-step semantics are pretty straightforward.

inductive *big-step* :: *program* \Rightarrow *com* \times *state* \Rightarrow *state* \Rightarrow *bool*
($\langle \cdot \rangle \cdot \Rightarrow \cdot$) [1000,55,55] 55)
where
— No-Op
Skip: $\pi:(SKIP,s) \Rightarrow s$
— Assignments

$|$ *AssignIdx*: $\pi:(x[i] ::= a, s) \Rightarrow s(x := (s\ x)(\text{aval } i\ s := \text{aval } a\ s))$
 $|$ *ArrayCpy*: $\pi:(x[] ::= y, s) \Rightarrow s(x := s\ y)$
 $|$ *ArrayClear*: $\pi:(\text{CLEAR } x[], s) \Rightarrow s(x := (\lambda_. 0))$
 $|$ *Assign-Locals*: $\pi:(\text{Assign-Locals } l, s) \Rightarrow \langle l | s \rangle$

— Block commands

$|$ *Seq*: $\llbracket \pi:(c_1, s_1) \Rightarrow s_2; \pi:(c_2, s_2) \Rightarrow s_3 \rrbracket \Longrightarrow \pi:(c_1;;c_2, s_1) \Rightarrow s_3$
 $|$ *IfTrue*: $\llbracket \text{bval } b\ s; \pi:(c_1, s) \Rightarrow t \rrbracket \Longrightarrow \pi:(\text{IF } b\ \text{THEN } c_1\ \text{ELSE } c_2, s) \Rightarrow t$
 $|$ *IfFalse*: $\llbracket \neg\text{bval } b\ s; \pi:(c_2, s) \Rightarrow t \rrbracket \Longrightarrow \pi:(\text{IF } b\ \text{THEN } c_1\ \text{ELSE } c_2, s) \Rightarrow t$
 $|$ *Scope*: $\llbracket \pi:(c, \langle \langle \rangle | s \rangle) \Rightarrow s' \rrbracket \Longrightarrow \pi:(\text{SCOPE } c, s) \Rightarrow \langle s | s' \rangle$
 $|$ *WhileFalse*: $\neg\text{bval } b\ s \Longrightarrow \pi:(\text{WHILE } b\ \text{DO } c, s) \Rightarrow s$
 $|$ *WhileTrue*: $\llbracket \text{bval } b\ s_1; \pi:(c, s_1) \Rightarrow s_2; \pi:(\text{WHILE } b\ \text{DO } c, s_2) \Rightarrow s_3 \rrbracket \Longrightarrow \pi:(\text{WHILE } b\ \text{DO } c, s_1) \Rightarrow s_3$

— Procedure commands

$|$ *PCall*: $\llbracket \pi\ p = \text{Some } c; \pi:(c, s) \Rightarrow t \rrbracket \Longrightarrow \pi:(\text{PCall } p, s) \Rightarrow t$
 $|$ *PScope*: $\llbracket \pi':(c, s) \Rightarrow t \rrbracket \Longrightarrow \pi:(\text{PScope } \pi'\ c, s) \Rightarrow t$

2.4.1 Proof Automation

We do some setup to make proofs over the big-step semantics more automatic.

declare *big-step.intros* [intro]

lemmas *big-step-induct*[induct set] = *big-step.induct*[split-format(complete)]

inductive-simps *Skip-simp*: $\pi:(\text{SKIP}, s) \Rightarrow t$

inductive-simps *AssignIdx-simp*: $\pi:(x[i] ::= a, s) \Rightarrow t$

inductive-simps *ArrayCpy-simp*: $\pi:(x[] ::= y, s) \Rightarrow t$

inductive-simps *ArrayInit-simp*: $\pi:(\text{CLEAR } x[], s) \Rightarrow t$

inductive-simps *AssignLocals-simp*: $\pi:(\text{Assign-Locals } l, s) \Rightarrow t$

inductive-simps *Seq-simp*: $\pi:(c_1;;c_2, s_1) \Rightarrow s_3$

inductive-simps *If-simp*: $\pi:(\text{IF } b\ \text{THEN } c_1\ \text{ELSE } c_2, s) \Rightarrow t$

inductive-simps *Scope-simp*: $\pi:(\text{SCOPE } c, s) \Rightarrow t$

inductive-simps *PCall-simp*: $\pi:(\text{PCall } p, s) \Rightarrow t$

inductive-simps *PScope-simp*: $\pi:(\text{PScope } \pi'\ p, s) \Rightarrow t$

lemmas *big-step-simps* =

Skip-simp *AssignIdx-simp* *ArrayCpy-simp* *ArrayInit-simp*

Seq-simp *If-simp* *Scope-simp* *PCall-simp* *PScope-simp*

inductive-cases *SkipE[elim!]*: $\pi:(\text{SKIP}, s) \Rightarrow t$

inductive-cases *AssignIdxE[elim!]*: $\pi:(x[i] ::= a, s) \Rightarrow t$

inductive-cases *ArrayCpyE[elim!]*: $\pi:(x[] ::= y, s) \Rightarrow t$

inductive-cases *ArrayInitE[elim!]*: $\pi:(\text{CLEAR } x[], s) \Rightarrow t$

inductive-cases *AssignLocalsE*[*elim!*]: $\pi:(\text{Assign-Locals } l, s) \Rightarrow t$

inductive-cases *SeqE*[*elim!*]: $\pi:(c1;;c2, s1) \Rightarrow s3$

inductive-cases *IfE*[*elim!*]: $\pi:(\text{IF } b \text{ THEN } c1 \text{ ELSE } c2, s) \Rightarrow t$

inductive-cases *ScopeE*[*elim!*]: $\pi:(\text{SCOPE } c, s) \Rightarrow t$

inductive-cases *PCallE*[*elim!*]: $\pi:(\text{PCall } p, s) \Rightarrow t$

inductive-cases *PScopeE*[*elim!*]: $\pi:(\text{PScope } \pi' p, s) \Rightarrow t$

inductive-cases *WhileE*[*elim!*]: $\pi:(\text{WHILE } b \text{ DO } c, s) \Rightarrow t$

2.4.2 Automatic Derivation

lemma *Assign'*: $s' = s(x := (s \ x)(\text{aval } i \ s := \text{aval } a \ s)) \Longrightarrow \pi:(x[i] ::= a, s) \Rightarrow s'$ **by** *auto*

lemma *ArrayCpy'*: $s' = s(x := (s \ y)) \Longrightarrow \pi:(x[] ::= y, s) \Rightarrow s'$ **by** *auto*

lemma *ArrayClear'*: $s' = s(x := (\lambda-. 0)) \Longrightarrow \pi:(\text{CLEAR } x[], s) \Rightarrow s'$ **by** *auto*

lemma *Scope'*: $s_1 = \langle \langle \rangle | s \rangle \Longrightarrow \pi:(c, s_1) \Rightarrow t \Longrightarrow t' = \langle s | t \rangle \Longrightarrow \pi:(\text{Scope } c, s) \Rightarrow t'$ **by** *auto*

named-theorems *deriv-unfolds* $\langle \text{Unfold rules before derivations} \rangle$

method *bs-simp* = *simp add: combine-nest combine-upd combine-query fun-upd-same fun-upd-other del: fun-upd-apply*

method *big-step'* =
rule Skip Seq PScope
 | (*rule Assign' ArrayCpy' ArrayClear', (bs-simp;fail)*)
 | (*rule IfTrue IfFalse WhileTrue WhileFalse PCall Scope', (bs-simp;fail)*)
 | *unfold deriv-unfolds*
 | (*bs-simp; fail*)

method *big-step* =
rule Skip
 | *rule Seq, (big-step;fail), (big-step;fail)*
 | *rule PScope, (big-step;fail)*
 | (*rule Assign' ArrayCpy' ArrayClear', (bs-simp;fail)*)
 | (*rule IfTrue IfFalse, (bs-simp;fail), (big-step;fail)*)
 | *rule WhileTrue, (bs-simp;fail), (big-step;fail), (big-step;fail)*
 | *rule WhileFalse, (bs-simp;fail)*
 | *rule PCall, (bs-simp;fail), (big-step;fail)*
 | (*rule Scope', (bs-simp;fail), (big-step;fail), (bs-simp;fail)*)
 | *unfold deriv-unfolds, big-step*

schematic-goal *Map.empty*: (
 "a" ::= N 1;;

$WHILE$ Cmpop $(\lambda x y. y < x) (V \text{"n"}) (N 0) DO ($
 $\text{"a"} ::= Binop (+) (V \text{"a"}) (V \text{"a"});$
 $\text{"n"} ::= Binop (-) (V \text{"n"}) (N 1)$
 $), <\text{"n"} := (\lambda -. 5) > \Rightarrow ?s$
by *big-step*

2.5 Command Equivalence

Two commands are equivalent if they have the same semantics.

definition

$equiv\text{-}c :: com \Rightarrow com \Rightarrow bool$ (**infix** $\langle \sim \rangle$ 50) **where**
 $c \sim c' \equiv (\forall \pi s t. \pi:(c,s) \Rightarrow t = \pi:(c',s) \Rightarrow t)$

lemma *equivI[intro?]*: \llbracket

$\bigwedge s t \pi. \pi:(c,s) \Rightarrow t \implies \pi:(c',s) \Rightarrow t;$
 $\bigwedge s t \pi. \pi:(c',s) \Rightarrow t \implies \pi:(c,s) \Rightarrow t \rrbracket$
 $\implies c \sim c'$

by (*auto simp: equiv-c-def*)

lemma *equivD[dest]*: $c \sim c' \implies \pi:(c,s) \Rightarrow t \longleftrightarrow \pi:(c',s) \Rightarrow t$

by (*auto simp: equiv-c-def*)

Command equivalence is an equivalence relation, i.e. it is reflexive, symmetric, and transitive.

lemma *equiv-refl[simp, intro!]*: $c \sim c$

by (*blast intro: equivI*)

lemma *equiv-sym[sym]*: $(c \sim c') \implies (c' \sim c)$

by (*blast intro: equivI*)

lemma *equiv-trans[trans]*: $c \sim c' \implies c' \sim c'' \implies c \sim c''$

by (*blast intro: equivI*)

2.5.1 Basic Equivalences

lemma *while-unfold*:

$(WHILE\ b\ DO\ c) \sim (IF\ b\ THEN\ c;;\ WHILE\ b\ DO\ c\ ELSE\ SKIP)$

by *rule auto*

lemma *triv-if*:

$(IF\ b\ THEN\ c\ ELSE\ c) \sim c$

by (*auto intro!: equivI*)

lemma *commute-if*:

$(IF\ b1\ THEN\ (IF\ b2\ THEN\ c11\ ELSE\ c12)\ ELSE\ c2)$

\sim

$(IF\ b2\ THEN\ (IF\ b1\ THEN\ c11\ ELSE\ c2)\ ELSE\ (IF\ b1\ THEN\ c12\ ELSE\ c2))$

by (*auto intro!: equivI*)

lemma *sim-while-cong-aux*:

$\llbracket \pi:(WHILE\ b\ DO\ c,s) \Rightarrow t; \text{bval } b = \text{bval } b'; c \sim c' \rrbracket \Longrightarrow \pi:(WHILE\ b'\ DO\ c',s) \Rightarrow t$
by(*induction WHILE b DO c s t arbitrary: b c rule: big-step-induct*)
auto

lemma *sim-while-cong*: $\text{bval } b = \text{bval } b' \Longrightarrow c \sim c' \Longrightarrow WHILE\ b\ DO\ c \sim WHILE\ b'\ DO\ c'$
using *equiv-c-def sim-while-cong-aux* **by** *auto*

2.6 Execution is Deterministic

This proof is automatic.

theorem *big-step-determ*: $\llbracket \pi:(c,s) \Rightarrow t; \pi:(c,s) \Rightarrow u \rrbracket \Longrightarrow u = t$

proof (*induction arbitrary: u rule: big-step.induct*)

case (*WhileTrue b s₁ c s₂ s₃*)

then show *?case* **by** *blast*

qed *fastforce+*

2.7 Small-Step Semantics

The small step semantics is defined by a step function on a pair of command and state. Intuitively, the command is the remaining part of the program that still has to be executed. The step function is defined to stutter if the command is *SKIP*.

Moreover, the step function is explicitly partial, returning *None* on error, i.e., on an undefined procedure call.

Most steps are straightforward. For a sequential composition, steps are performed on the first command, until it has been reduced to *SKIP*, then the sequential composition is reduced to the second command.

A while command is reduced by unfolding the loop once.

A scope command is reduced to the inner command, followed by an *Assign-Locals* command to restore the original local variables.

A procedure scope command is reduced by performing a step in the inner command, with the new procedure environment, until the inner command has been reduced to *SKIP*. Then, the whole command is reduced to *SKIP*.

fun *small-step* :: *program* \Rightarrow *com* \times *state* \rightarrow *com* \times *state* **where**
small-step π (*x*[*i*]::=*a*,*s*) = *Some* (*SKIP*, *s*(*x* := (*s* *x*)(*aval* *i* *s* := *aval* *a* *s*)))
| *small-step* π (*x*[::=*y*,*s*) = *Some* (*SKIP*, *s*(*x* := *s* *y*))
| *small-step* π (*CLEAR* *x*[],*s*) = *Some* (*SKIP*, *s*(*x* := (λ -. 0)))
| *small-step* π (*Assign-Locals* *l*,*s*) = *Some* (*SKIP*, <*l*|*s*>)
| *small-step* π (*SKIP*;;*c*,*s*) = *Some* (*c*,*s*)

$| \text{small-step } \pi (c_1;;c_2,s) = (\text{case small-step } \pi (c_1,s) \text{ of Some } (c_1',s') \Rightarrow \text{Some } (c_1';;c_2,s') \mid - \Rightarrow \text{None})$
 $| \text{small-step } \pi (\text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2,s) = \text{Some } (\text{if bval } b \text{ s then } (c_1,s) \text{ else } (c_2,s))$
 $| \text{small-step } \pi (\text{SCOPE } c, s) = \text{Some } (c;;\text{Assign-Locals } s, \langle\langle\rangle\mid s\rangle)$
 $| \text{small-step } \pi (\text{WHILE } b \text{ DO } c,s) = \text{Some } (\text{IF } b \text{ THEN } c;;\text{WHILE } b \text{ DO } c \text{ ELSE SKIP, } s)$
 $| \text{small-step } \pi (\text{PCall } p, s) = (\text{case } \pi p \text{ of Some } c \Rightarrow \text{Some } (c, s) \mid - \Rightarrow \text{None})$
 $| \text{small-step } \pi (\text{PScope } \pi' \text{ SKIP, } s) = \text{Some } (\text{SKIP},s)$
 $| \text{small-step } \pi (\text{PScope } \pi' c, s) = (\text{case small-step } \pi' (c,s) \text{ of Some } (c',s') \Rightarrow \text{Some } (\text{PScope } \pi' c', s') \mid - \Rightarrow \text{None})$
 $| \text{small-step } \pi (\text{SKIP},s) = \text{Some } (\text{SKIP},s)$

We define the reflexive transitive closure of the step function.

inductive *small-steps* :: *program* \Rightarrow *com* \times *state* \Rightarrow (*com* \times *state*)
option \Rightarrow *bool* **where**
 $[simp]: \text{small-steps } \pi cs (\text{Some } cs)$
 $| \llbracket \text{small-step } \pi cs = \text{None} \rrbracket \Longrightarrow \text{small-steps } \pi cs \text{None}$
 $| \llbracket \text{small-step } \pi cs = \text{Some } cs1; \text{small-steps } \pi cs1 cs2 \rrbracket \Longrightarrow \text{small-steps } \pi cs cs2$

lemma *small-steps-append*: $\text{small-steps } \pi cs_1 (\text{Some } cs_2) \Longrightarrow \text{small-steps } \pi cs_2 cs_3 \Longrightarrow \text{small-steps } \pi cs_1 cs_3$
apply (*induction* πcs_1 *Some* cs_2 *arbitrary*: cs_2 *rule*: *small-steps.induct*)
apply (*auto intro*: *small-steps.intros*)
done

2.7.1 Equivalence to Big-Step Semantics

We show that the small-step semantics yields a final configuration if and only if the big-step semantics terminates with the respective state.

Moreover, we show that the big-step semantics gets stuck if the small-step semantics yields an error.

lemma *small-big-append*: $\text{small-step } \pi cs_1 = \text{Some } cs_2 \Longrightarrow \pi: cs_2 \Rightarrow s_3 \Longrightarrow \pi: cs_1 \Rightarrow s_3$
apply (*induction* πcs_1 *arbitrary*: $cs_2 s_3$ *rule*: *small-step.induct*)
apply (*auto split*: *option.splits if-splits*)
done

lemma *small-big-append*: $\text{small-steps } \pi cs_1 (\text{Some } cs_2) \Longrightarrow \pi: cs_2 \Rightarrow s_3 \Longrightarrow \pi: cs_1 \Rightarrow s_3$
apply (*induction* πcs_1 *Some* cs_2 *arbitrary*: cs_2 *rule*: *small-steps.induct*)
apply (*auto intro*: *small-big-append*)
done

lemma *small-imp-big*:

assumes *small-steps* π cs_1 (*Some* (*SKIP*, s_2))
shows $\pi: cs_1 \Rightarrow s_2$
using *small-big-append*[*OF* *assms*]
by *auto*

lemma *small-steps-skip-term*[*simp*]: *small-steps* π (*SKIP*, s) $cs' \longleftrightarrow cs' = \text{Some}(\text{SKIP}, s)$
apply *rule*
subgoal
apply (*induction* π (*SKIP*, s) cs' *arbitrary: s* *rule: small-steps.induct*)
by (*auto* *intro: small-steps.intros*)
by (*auto* *intro: small-steps.intros*)

lemma *small-seq*: $\llbracket c \neq \text{SKIP}; \text{small-step } \pi (c, s) = \text{Some}(c', s') \rrbracket \implies \text{small-step } \pi (c;;cx, s) = \text{Some}(c';;cx, s')$
apply (*induction* π (c, s) *arbitrary: c s c' s'* *rule: small-step.induct*)
apply *auto*
done

lemma *small-seq*: $\llbracket \text{small-steps } \pi (c, s) (\text{Some}(c', s')) \rrbracket \implies \text{small-steps } \pi (c;;cx, s) (\text{Some}(c';;cx, s'))$
apply (*induction* π (c, s) *Some* (c', s') *arbitrary: c s c' s'* *rule: small-steps.induct*)
apply (*auto* *dest: small-seq* *intro: small-steps.intros*)
by (*metis* *option.simps(1)* *prod.simps(1)* *small-seq* *small-step.simps(31)* *small-steps.intros(3)*)

lemma *small-pscope*:
 $\llbracket c \neq \text{SKIP}; \text{small-step } \pi' (c, s) = \text{Some}(c', s') \rrbracket \implies \text{small-step } \pi (\text{PScope } \pi' c, s) = \text{Some}(\text{PScope } \pi' c', s')$
apply (*induction* π (c, s) *arbitrary: c s c' s'* *rule: small-step.induct*)
apply *auto*
done

lemma *small-pscope*:
 $\text{small-steps } \pi' (c, s) (\text{Some}(c', s')) \implies \text{small-steps } \pi (\text{PScope } \pi' c, s) (\text{Some}(\text{PScope } \pi' c', s'))$
apply (*induction* π' (c, s) (*Some* (c', s')) *arbitrary: c s* *rule: small-steps.induct*)
apply *auto*
by (*metis* (*no-types*, *opaque-lifting*) *option.inject* *prod.inject* *small-pscope* *small-steps.simps* *small-steps-append* *small-steps-skip-term*)

lemma *big-imp-small*:
assumes $\pi: cs \Rightarrow t$
shows *small-steps* π cs (*Some* (*SKIP*, t))
using *assms*
proof *induction*

```

    case (Skip  $\pi$   $s$ )
  then show ?case by (auto 0 4 intro: small-steps.intros)
next
  case (AssignIdx  $\pi$   $x$   $i$   $a$   $s$ )
  then show ?case by (auto 0 4 intro: small-steps.intros)
next
  case (ArrayCpy  $\pi$   $x$   $y$   $s$ )
  then show ?case by (auto 0 4 intro: small-steps.intros)
next
  case (ArrayClear  $\pi$   $x$   $s$ )
  then show ?case by (auto 0 4 intro: small-steps.intros)
next
  case (Seq  $\pi$   $c_1$   $s_1$   $s_2$   $c_2$   $s_3$ )
  then show ?case
    by (meson small-step.simps(5) small-steps.intros(3) small-steps-append
        smalls-seq)
next
  case (IfTrue  $b$   $s$   $\pi$   $c_1$   $t$   $c_2$ )
  then show ?case by (auto 0 4 intro: small-steps.intros)
next
  case (IfFalse  $b$   $s$   $\pi$   $c_2$   $t$   $c_1$ )
  then show ?case by (auto 0 4 intro: small-steps.intros)
next
  case (Scope  $\pi$   $c$   $s$   $s'$ )
  then show ?case
    by (meson small-step.simps(17) small-step.simps(4) small-step.simps(5)
        small-steps.intros(1) small-steps.intros(3) small-steps-append smalls-seq)
next
  case (WhileFalse  $b$   $s$   $\pi$   $c$ )
  then show ?case by (auto 0 4 intro: small-steps.intros)
next
  case (WhileTrue  $b$   $s_1$   $\pi$   $c$   $s_2$   $s_3$ )
  then show ?case
    proof -
      have  $\forall ca$   $p$ . (small-steps  $\pi$   $p$  (Some (SKIP,  $s_3$ ))  $\vee$  Some ( $ca$ ,  $s_2$ )
         $\neq$  Some (WHILE  $b$  DO  $c$ ,  $s_2$ ))  $\vee$  small-step  $\pi$   $p$   $\neq$  Some ( $c$ ;  $ca$ ,  $s_1$ )
      by (metis (no-types) WhileTrue.IH(1) WhileTrue.IH(2) small-step.simps(5)
          small-steps.intros(3) small-steps-append smalls-seq)
      then have  $\forall ca$   $cb$   $cc$ . (small-steps  $\pi$  (IF  $b$  THEN  $cc$  ELSE  $ca$ ,  $s_1$ )
        (Some (SKIP,  $s_3$ ))  $\vee$  Some ( $cb$ ,  $s_2$ )  $\neq$  Some (WHILE  $b$  DO  $c$ ,  $s_2$ ))  $\vee$ 
        Some ( $cc$ ,  $s_1$ )  $\neq$  Some ( $c$ ;  $cb$ ,  $s_1$ )
      using WhileTrue.hyps(1) by force
      then show ?thesis
        using small-step.simps(18) small-steps.intros(3) by blast
    qed
next
  case (PCall  $\pi$   $p$   $c$   $s$   $t$ )
  then show ?case by (auto 0 4 intro: small-steps.intros)

```

```

next
  case (PScope  $\pi'$  c s t  $\pi$ )
  then show ?case
    by (meson small-step.simps(20) small-steps.simps small-steps-append
smalls-pscope)

```

```

next
  case (Assign-Locals  $\pi$  l s)
  then show ?case by (auto 0 4 intro: small-steps.intros)
qed

```

The big-step semantics yields a state t , iff and only iff there is a transition of the small-step semantics to $(SKIP, t)$.

theorem *big-eq-small*: $\pi: cs \Rightarrow t \iff \text{small-steps } \pi \text{ } cs \text{ } (\text{Some } (SKIP, t))$
using *big-imp-small* *small-imp-big* **by** *blast*

lemma *small-steps-determ*:
assumes *small-steps* π *cs* *None*
shows $\neg \text{small-steps } \pi \text{ } cs \text{ } (\text{Some } (SKIP, t))$
using *assms*
apply (*induction* π *cs* *None::(com \times state)* *option arbitrary: t rule:*
small-steps.induct)
apply (*auto elim: small-steps.cases*)
done

If the small-step semantics reaches a failure state, the big-step semantics gets stuck.

corollary *small-imp-big-fail*:
assumes *small-steps* π *cs* *None*
shows $\nexists t. \pi: cs \Rightarrow t$
using *assms*
by (*auto simp: big-eq-small small-steps-determ*)

2.8 Weakest Precondition

The following definitions are made wrt. a fixed program π , which becomes the first parameter of the defined constants when the context is left.

```

context
  fixes  $\pi$  :: program
begin

```

Weakest precondition: c terminates with a state that satisfies Q , when started from s .

definition *wp* c Q $s \equiv \exists t. \pi: (c, s) \Rightarrow t \wedge Q t$
— Note that this definition exploits that the semantics is deterministic! In general, we must ensure absence of infinite executions

Weakest liberal precondition: If c terminates when started from s , the new state satisfies Q .

definition $wlp\ c\ Q\ s \equiv \forall t. \pi:(c,s) \Rightarrow t \longrightarrow Q\ t$

2.8.1 Basic Properties

context

notes $[abs-def, simp] = wp-def\ wlp-def$

begin

lemma $wp\text{-imp}\text{-wlp}$: $wp\ c\ Q\ s \Longrightarrow wlp\ c\ Q\ s$

using $big\text{-step}\text{-determ}$ **by** $force$

lemma $wlp\text{-and}\text{-term}\text{-imp}\text{-wp}$: $wlp\ c\ Q\ s \wedge \pi:(c,s) \Rightarrow t \Longrightarrow wp\ c\ Q\ s$ **by** $auto$

lemma $wp\text{-equiv}$: $c \sim c' \Longrightarrow wp\ c = wp\ c'$ **by** $auto$

lemma $wp\text{-conseq}$: $wp\ c\ P\ s \Longrightarrow \llbracket \bigwedge s. P\ s \Longrightarrow Q\ s \rrbracket \Longrightarrow wp\ c\ Q\ s$ **by** $auto$

lemma $wlp\text{-equiv}$: $c \sim c' \Longrightarrow wlp\ c = wlp\ c'$ **by** $auto$

lemma $wlp\text{-conseq}$: $wlp\ c\ P\ s \Longrightarrow \llbracket \bigwedge s. P\ s \Longrightarrow Q\ s \rrbracket \Longrightarrow wlp\ c\ Q\ s$ **by** $auto$

2.8.2 Unfold Rules

lemma $wp\text{-skip}\text{-eq}$: $wp\ SKIP\ Q\ s = Q\ s$ **by** $auto$

lemma $wp\text{-assign}\text{-idx}\text{-eq}$: $wp\ (x[i]::=a)\ Q\ s = Q\ (s(x:=s\ x)\ (aval\ i\ s\ :=\ aval\ a\ s)))$ **by** $auto$

lemma $wp\text{-arraycpy}\text{-eq}$: $wp\ (x[]::=a)\ Q\ s = Q\ (s(x:=s\ a))$ **by** $auto$

lemma $wp\text{-arrayinit}\text{-eq}$: $wp\ (CLEAR\ x[])\ Q\ s = Q\ (s(x:=\lambda\cdot.\ 0))$ **by** $auto$

lemma $wp\text{-assign}\text{-locals}\text{-eq}$: $wp\ (Assign\ Locals\ l)\ Q\ s = Q\ \langle l|s \rangle$ **by** $auto$

lemma $wp\text{-seq}\text{-eq}$: $wp\ (c_1;c_2)\ Q\ s = wp\ c_1\ (wp\ c_2\ Q)\ s$ **by** $auto$

lemma $wp\text{-if}\text{-eq}$: $wp\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ Q\ s$

$= (if\ bval\ b\ s\ then\ wp\ c_1\ Q\ s\ else\ wp\ c_2\ Q\ s)$ **by** $auto$

lemma $wp\text{-scope}\text{-eq}$: $wp\ (SCOPE\ c)\ Q\ s = wp\ c\ (\lambda s'. Q\ \langle s|s' \rangle)$ **by** $auto$

lemma $wp\text{-pcall}\text{-eq}$: $\pi\ p = Some\ c \Longrightarrow wp\ (PCall\ p)\ Q\ s = wp\ c\ Q\ s$ **by** $auto$

lemmas $wp\text{-eq} = wp\text{-skip}\text{-eq}\ wp\text{-assign}\text{-idx}\text{-eq}\ wp\text{-arraycpy}\text{-eq}\ wp\text{-arrayinit}\text{-eq}$

$wp\text{-assign}\text{-locals}\text{-eq}\ wp\text{-seq}\text{-eq}\ wp\text{-scope}\text{-eq}$

lemmas $wp\text{-eq}' = wp\text{-eq}\ wp\text{-if}\text{-eq}$

lemma $wlp\text{-skip}\text{-eq}$: $wlp\ SKIP\ Q\ s = Q\ s$ **by** $auto$

lemma *wlp-assign-idx-eq*: $wlp (x[i] ::= a) Q s = Q (s(x := (s x) (aval i s := aval a s)))$ **by** *auto*

lemma *wlp-arraycpy-eq*: $wlp (x[] ::= a) Q s = Q (s(x := s a))$ **by** *auto*

lemma *wlp-arrayinit-eq*: $wlp (CLEAR x[]) Q s = Q (s(x := (\lambda-. 0)))$ **by** *auto*

lemma *wlp-assign-locals-eq*: $wlp (Assign-Locals l) Q s = Q \langle l | s \rangle$ **by** *auto*

lemma *wlp-seq-eq*: $wlp (c_1 ;; c_2) Q s = wlp c_1 (wlp c_2 Q) s$ **by** *auto*

lemma *wlp-if-eq*: $wlp (IF b THEN c_1 ELSE c_2) Q s$

$= (if bval b s then wlp c_1 Q s else wlp c_2 Q s)$ **by** *auto*

lemma *wlp-scope-eq*: $wlp (SCOPE c) Q s = wlp c (\lambda s'. Q \langle s | s' \rangle)$ $\langle \langle \rangle | s \rangle$ **by** *auto*

lemma *wlp-pcall-eq*: $\pi p = Some c \implies wlp (PCall p) Q s = wlp c Q s$ **by** *auto*

lemmas *wlp-eq = wlp-skip-eq wlp-assign-idx-eq wlp-arraycpy-eq wlp-arrayinit-eq*

wlp-assign-locals-eq wlp-seq-eq wlp-scope-eq

lemmas *wlp-eq' = wlp-eq wlp-if-eq*

end

lemma *wlp-while-unfold*: $wlp (WHILE b DO c) Q s = (if bval b s then wlp c (wlp (WHILE b DO c) Q) s else Q s)$

apply (*subst wlp-equiv[OF while-unfold]*)

apply (*simp add: wlp-eq'*)

done

lemma *wp-while-unfold*: $wp (WHILE b DO c) Q s = (if bval b s then wp c (wp (WHILE b DO c) Q) s else Q s)$

apply (*subst wp-equiv[OF while-unfold]*)

apply (*simp add: wp-eq'*)

done

end — Context fixing program

Unfold rules for procedure scope

lemma *wp-pscope-eq*: $wp \pi (PScope \pi' c) Q s = wp \pi' (c) Q s$

unfolding *wp-def* **by** *auto*

lemma *wlp-pscope-eq*: $wlp \pi (PScope \pi' c) Q s = wlp \pi' (c) Q s$

unfolding *wlp-def* **by** *auto*

2.8.3 Weakest precondition and Program Equivalence

The following three statements are equivalent:

1. The commands c and c' are equivalent
2. The weakest preconditions are equivalent, for all procedure environments
3. The weakest liberal preconditions are equivalent, for all procedure environments

lemma *wp-equiv-iff*: $(\forall \pi. wp \ \pi \ c = wp \ \pi \ c') \longleftrightarrow c \sim c'$
unfolding *equiv-c-def*
using *big-step-determ unfolding wp-def*
by (*auto; metis*)

lemma *wlp-equiv-iff*: $(\forall \pi. wlp \ \pi \ c = wlp \ \pi \ c') \longleftrightarrow c \sim c'$
unfolding *equiv-c-def wlp-def*
by (*auto; metis (no-types, opaque-lifting)*)

2.8.4 While Loops and Weakest Precondition

Exchanging the loop condition by an equivalent one, and the loop body by one with the same weakest precondition, does not change the weakest precondition of the loop.

lemma *sim-while-wp-aux*:
assumes $bval \ b = bval \ b'$
assumes $wp \ \pi \ c = wp \ \pi \ c'$
assumes $\pi: (WHILE \ b \ DO \ c, \ s) \Rightarrow t$
shows $\pi: (WHILE \ b' \ DO \ c', \ s) \Rightarrow t$
using *assms(3,2)*
apply (*induction \ \pi \ WHILE \ b \ DO \ c \ s \ t*)
apply (*auto simp: assms(1)*)
by (*metis WhileTrue big-step-determ wp-def*)

lemma *sim-while-wp*: $bval \ b = bval \ b' \Longrightarrow wp \ \pi \ c = wp \ \pi \ c' \Longrightarrow wp \ \pi \ (WHILE \ b \ DO \ c) = wp \ \pi \ (WHILE \ b' \ DO \ c')$
apply (*intro ext*)
apply (*auto 0 3 simp: wp-def intro: sim-while-wp-aux*)
done

The same lemma for weakest liberal preconditions.

lemma *sim-while-wlp-aux*:
assumes $bval \ b = bval \ b'$
assumes $wlp \ \pi \ c = wlp \ \pi \ c'$
assumes $\pi: (WHILE \ b \ DO \ c, \ s) \Rightarrow t$
shows $\pi: (WHILE \ b' \ DO \ c', \ s) \Rightarrow t$
using *assms(3,2)*
apply (*induction \ \pi \ WHILE \ b \ DO \ c \ s \ t*)
apply (*auto simp: assms(1,2)*)
by (*metis WhileTrue wlp-def*)

lemma *sim-while-wlp*: $bval\ b = bval\ b' \implies wlp\ \pi\ c = wlp\ \pi\ c' \implies wlp\ \pi\ (WHILE\ b\ DO\ c) = wlp\ \pi\ (WHILE\ b'\ DO\ c')$
apply (*intro ext*)
apply (*auto 0 3 simp: wlp-def intro: sim-while-wlp-aux*)
done

2.9 Invariants for While-Loops

We prove the standard invariant rules for while loops. We first prove them in a slightly non-standard form, summarizing the loop step and loop exit assumptions. Then, we derive the standard form with separate assumptions for step and loop exit.

2.9.1 Partial Correctness

lemma *wlp-whileI'*:
assumes *INIT*: $I\ s_0$
assumes *STEP*: $\bigwedge s. I\ s \implies (if\ bval\ b\ s\ then\ wlp\ \pi\ c\ I\ s\ else\ Q\ s)$
shows $wlp\ \pi\ (WHILE\ b\ DO\ c)\ Q\ s_0$
unfolding *wlp-def*
proof *clarify*
fix t
assume $\pi: (WHILE\ b\ DO\ c, s_0) \Rightarrow t$
thus $Q\ t$ **using** *INIT STEP*
proof (*induction* $\pi\ WHILE\ b\ DO\ c\ s_0\ t$ *rule: big-step-induct*)
case (*WhileFalse* s) **with** *STEP* **show** $Q\ s$ **by** *auto*
next
case (*WhileTrue* $s_1\ \pi\ s_2\ s_3$)
note $STEP' = WhileTrue.prem\ s(2)$
from $STEP'[OF\ \langle I\ s_1 \rangle]\ \langle bval\ b\ s_1 \rangle$ **have** $wlp\ \pi\ c\ I\ s_1$ **by** *simp*
with $\langle \pi: (c, s_1) \Rightarrow s_2 \rangle$ **have** $I\ s_2$ **unfolding** *wlp-def* **by** *blast*
moreover **have** $\langle I\ s_2 \implies Q\ s_3 \rangle$ **using** $STEP'\ WhileTrue.hyps(5)$
by *blast*
ultimately **show** $Q\ s_3$ **by** *blast*
qed
qed

lemma
assumes *INIT*: $I\ s_0$
assumes *STEP*: $\bigwedge s. I\ s \implies (if\ bval\ b\ s\ then\ wlp\ \pi\ c\ I\ s\ else\ Q\ s)$
shows $wlp\ \pi\ (WHILE\ b\ DO\ c)\ Q\ s_0$
using *STEP*
unfolding *wlp-def*
apply *clarify* **subgoal** **premises** *prems* **for** t
using $prems(2,1)\ INIT$
by (*induction* $\pi\ WHILE\ b\ DO\ c\ s_0\ t$ *rule: big-step-induct; meson*)
done

2.9.2 Total Correctness

For total correctness, each step must decrease the state wrt. a well-founded relation.

```

lemma wp-whileI':
  assumes WF: wf R
  assumes INIT: I s0
  assumes STEP:  $\bigwedge s. I s \implies (\text{if } \text{bval } b \text{ } s \text{ then } \text{wp } \pi \text{ } c \text{ } (\lambda s'. I s' \wedge (s',s) \in R) \text{ } s \text{ else } Q \text{ } s)$ 
  shows  $\text{wp } \pi \text{ } (\text{WHILE } b \text{ DO } c) \text{ } Q \text{ } s_0$ 
  using WF INIT
proof (induction rule: wf-induct-rule[where a=s0])
  case (less s)
  show  $\text{wp } \pi \text{ } (\text{WHILE } b \text{ DO } c) \text{ } Q \text{ } s$ 
  proof (rule wp-while-unfold[THEN iffD2])
    show if bval b s then wp π c (wp π (WHILE b DO c) Q) s else
Q s
    proof (split if-split; intro allI impI conjI)
      assume [simp]: bval b s

      from STEP  $\langle I s \rangle$  have  $\text{wp } \pi \text{ } c \text{ } (\lambda s'. I s' \wedge (s',s) \in R) \text{ } s$  by simp
      thus  $\text{wp } \pi \text{ } c \text{ } (\text{wp } \pi \text{ } (\text{WHILE } b \text{ DO } c) \text{ } Q) \text{ } s$  proof (rule wp-conseq)
        fix s' assume  $I s' \wedge (s',s) \in R$ 
        with less.IH show  $\text{wp } \pi \text{ } (\text{WHILE } b \text{ DO } c) \text{ } Q \text{ } s'$  by blast
        qed
      next
      assume [simp]:  $\neg \text{bval } b \text{ } s$ 
      from STEP  $\langle I s \rangle$  show  $Q \text{ } s$  by simp
      qed
    qed
  qed

```

```

lemma
  assumes WF: wf R
  assumes INIT: I s0
  assumes STEP:  $\bigwedge s. I s \implies (\text{if } \text{bval } b \text{ } s \text{ then } \text{wp } \pi \text{ } c \text{ } (\lambda s'. I s' \wedge (s',s) \in R) \text{ } s \text{ else } Q \text{ } s)$ 
  shows  $\text{wp } \pi \text{ } (\text{WHILE } b \text{ DO } c) \text{ } Q \text{ } s_0$ 
  using WF INIT
  apply (induction rule: wf-induct-rule[where a=s0])
  apply (subst wp-while-unfold)
  by (smt STEP wp-conseq)

```

2.9.3 Standard Forms of While Rules

```

lemma wlp-whileI:
  assumes INIT: I s0
  assumes STEP:  $\bigwedge s. \llbracket I \text{ } s; \text{ bval } b \text{ } s \rrbracket \implies \text{wlp } \pi \text{ } c \text{ } I \text{ } s$ 

```

assumes *FINAL*: $\bigwedge s. \llbracket I \ s; \neg bval \ b \ s \rrbracket \implies Q \ s$
shows $wlp \ \pi \ (WHILE \ b \ DO \ c) \ Q \ s_0$
using *assms wlp-whileI'* **by** *auto*

lemma *wp-whileI*:
assumes *WF*: $wf \ R$
assumes *INIT*: $I \ s_0$
assumes *STEP*: $\bigwedge s. \llbracket I \ s; bval \ b \ s \rrbracket \implies wp \ \pi \ c \ (\lambda s'. I \ s' \wedge (s', s) \in R) \ s$
assumes *FINAL*: $\bigwedge s. \llbracket I \ s; \neg bval \ b \ s \rrbracket \implies Q \ s$
shows $wp \ \pi \ (WHILE \ b \ DO \ c) \ Q \ s_0$
using *assms wp-whileI'* **by** *auto*

2.10 Modularity of Programs

Adding more procedures does not change the semantics of the existing ones.

lemma *map-leD*: $m \subseteq_m m' \implies m \ x = Some \ v \implies m' \ x = Some \ v$
by (*metis domI map-le-def*)

lemma *big-step-mono-prog*:
assumes $\pi \subseteq_m \pi'$
assumes $\pi : (c, s) \Rightarrow t$
shows $\pi' : (c, s) \Rightarrow t$
using *assms(2,1)*
apply (*induction* $\pi \ c \ s \ t$ *rule: big-step-induct*)
by (*auto dest: map-leD*)

Wrapping a set of recursive procedures into a procedure scope

lemma *localize-recursion*:
 $\pi' : (P\ Scope \ \pi \ c, \ s) \Rightarrow t \longleftrightarrow \pi : (c, s) \Rightarrow t$
by *auto*

2.11 Strongest Postcondition

context *fixes* $\pi :: program$ **begin**
definition $sp \ P \ c \ t \equiv \exists s. P \ s \wedge \pi : (c, s) \Rightarrow t$

context *notes* $[simp] = sp-def[abs-def]$ **begin**

Intuition: There exists an old value vx for the assigned variable

lemma *sp-arraycpy-eq*: $sp \ P \ (x[] := y) \ t \longleftrightarrow (\exists vx. let \ s = t(x := vx) \ in \ t \ x = s \ y \wedge P \ s)$
apply (*auto simp: big-step-simps*)
apply (*intro exI conjI, assumption, auto*) \square
apply (*intro exI conjI, assumption, auto*) \square
done

Version with renaming of assigned variable

lemma *sp-arraycpy-eq'*: $sp\ P\ (x[]::=y)\ t \longleftrightarrow t\ x = t\ y \wedge (\exists vx. P\ (t(x:=vx, y:=t\ x)))$
apply (*auto simp: big-step-simps*)
apply (*metis fun-upd-triv*)
apply (*intro exI conjI, assumption*)
apply *auto*
done

lemma *sp-skip-eq*: $sp\ P\ SKIP\ t \longleftrightarrow P\ t$ **by** *auto*

lemma *sp-seq-eq*: $sp\ P\ (c_1;;c_2)\ t \longleftrightarrow sp\ (sp\ P\ c_1)\ c_2\ t$ **by** *auto*

end
end

2.12 Hoare-Triples

A Hoare-triple summarizes the precondition, command, and post-condition.

definition *HT*

where $HT\ \pi\ P\ c\ Q \equiv (\forall s_0. P\ s_0 \longrightarrow wp\ \pi\ c\ (Q\ s_0)\ s_0)$

definition *HT-partial*

where $HT\text{-}partial\ \pi\ P\ c\ Q \equiv (\forall s_0. P\ s_0 \longrightarrow wlp\ \pi\ c\ (Q\ s_0)\ s_0)$

Consequence rule—strengthen the precondition, weaken the post-condition.

lemma *HT-conseq*:

assumes $HT\ \pi\ P\ c\ Q$

assumes $\bigwedge s. P'\ s \Longrightarrow P\ s$

assumes $\bigwedge_{s_0} s. \llbracket P\ s_0; P'\ s_0; Q\ s_0\ s \rrbracket \Longrightarrow Q'\ s_0\ s$

shows $HT\ \pi\ P'\ c\ Q'$

using *assms unfolding HT-def by (blast intro: wp-conseq)*

lemma *HT-partial-conseq*:

assumes $HT\text{-}partial\ \pi\ P\ c\ Q$

assumes $\bigwedge s. P'\ s \Longrightarrow P\ s$

assumes $\bigwedge_{s_0} s. \llbracket P\ s_0; P'\ s_0; Q\ s_0\ s \rrbracket \Longrightarrow Q'\ s_0\ s$

shows $HT\text{-}partial\ \pi\ P'\ c\ Q'$

using *assms unfolding HT-partial-def by (blast intro: wlp-conseq)*

Simple rule for presentation in lecture: Use a Hoare-triple during VCG.

lemma *wp-modularity-rule*:

$\llbracket HT\ \pi\ P\ c\ Q; P\ s; (\bigwedge s'. Q\ s\ s' \Longrightarrow Q'\ s') \rrbracket \Longrightarrow wp\ \pi\ c\ Q'\ s$

unfolding *HT-def*

by (*blast intro: wp-conseq*)

2.12.1 Sets of Hoare-Triples

type-synonym $htset = ((state \Rightarrow bool) \times com \times (state \Rightarrow state \Rightarrow bool)) \text{ set}$

definition $HTset \pi \Theta \equiv \forall (P,c,Q) \in \Theta. HT \pi P c Q$

definition $HTset-r r \pi \Theta \equiv \forall (P,c,Q) \in \Theta. HT \pi (\lambda s. r c s \wedge P s) c Q$

2.12.2 Deriving Parameter Frame Adjustment Rules

The following rules can be used to derive Hoare-triples when adding prologue and epilogue code, and wrapping the command into a scope.

This will be used to implement the local variables and parameter passing protocol of procedures.

Intuition: New precondition is weakest one we need to ensure P after prologue.

lemma *adjust-prologue:*

assumes $HT \pi P \text{ body } Q$
shows $HT \pi (wp \pi \text{ prologue } P) (\text{prologue};;\text{body}) (\lambda s_0 s. wp \pi \text{ prologue } (Q s_0 s) s_0)$
using *assms*
unfolding *HT-def*
apply (*auto simp: wp-eq*)
using *wp-def by fastforce*

Intuition: New postcondition is strongest one we can get from Q after epilogue.

We have to be careful with non-terminating epilogue, though!

lemma *adjust-epilogue:*

assumes $HT \pi P \text{ body } Q$
assumes *TERMINATES*: $\forall s. \exists t. \pi: (\text{epilogue}, s) \Rightarrow t$
shows $HT \pi P (\text{body};;\text{epilogue}) (\lambda s_0. sp \pi (Q s_0) \text{ epilogue})$
using *assms*
unfolding *HT-def*
apply (*simp add: wp-eq*)
apply (*force simp: sp-def wp-def*)
done

Intuition: Scope can be seen as assignment of locals before and after inner command. Thus, this rule is a combined forward and backward assignment rule, for the epilogue $locals:=\langle \rangle$ and the prologue $locals:=old\text{-}locals$.

lemma *adjust-scope:*

assumes $HT \ \pi \ P \ \text{body} \ Q$
shows $HT \ \pi \ (\lambda s. P \ \langle\langle\rangle|s\rangle) \ (\text{SCOPE} \ \text{body}) \ (\lambda s_0 \ s. \exists l. Q \ (\langle\langle\rangle|s_0\rangle) \ (\langle l|s\rangle))$
using *assms unfolding HT-def*
apply (*auto simp: wp-eq combine-nest*)
apply (*auto simp: wp-def*)
by (*metis combine-collapse*)

2.12.3 Proof for Recursive Specifications

Prove correct any set of Hoare-triples, e.g., mutually recursive ones.

lemma *HTsetI*:

assumes *wf R*
assumes $RL: \bigwedge P \ c \ Q \ s_0. \llbracket HTset-r \ (\lambda c' \ s'. ((c',s'),(c,s_0)) \in R) \ \pi \ \Theta; \ (P,c,Q) \in \Theta; \ P \ s_0 \rrbracket \implies wp \ \pi \ c \ (Q \ s_0) \ s_0$
shows $HTset \ \pi \ \Theta$
unfolding *HTset-def HT-def*
proof *clarsimp*
fix $P \ c \ Q \ s_0$
assume $(P,c,Q) \in \Theta \ P \ s_0$
with $\langle wf \ R \rangle$ **show** $wp \ \pi \ c \ (Q \ s_0) \ s_0$
apply (*induction (c,s₀) arbitrary: c s₀ P Q*)
using *RL unfolding HTset-r-def HT-def*
by *blast*

qed

lemma *HT-simple-recursiveI*:

assumes *wf R*
assumes $\bigwedge s. \llbracket HT \ \pi \ (\lambda s'. (f \ s', f \ s) \in R \wedge P \ s') \ c \ Q; \ P \ s \rrbracket \implies wp \ \pi \ c \ (Q \ s) \ s$
shows $HT \ \pi \ P \ c \ Q$
using *HTsetI* [**where** $R = \text{inv-image } R \ (f \ o \ \text{snd})$ **and** $\pi = \pi$ **and** $\Theta = \{(P,c,Q)\}$] *assms*
by (*auto simp: HTset-r-def HTset-def*)

lemma *HT-simple-recursive-procI*:

assumes *wf R*
assumes $\bigwedge s. \llbracket HT \ \pi \ (\lambda s'. (f \ s', f \ s) \in R \wedge P \ s') \ (PCall \ p) \ Q; \ P \ s \rrbracket \implies wp \ \pi \ (PCall \ p) \ (Q \ s) \ s$
shows $HT \ \pi \ P \ (PCall \ p) \ Q$
using *HTsetI* [**where** $R = \text{inv-image } R \ (f \ o \ \text{snd})$ **and** $\pi = \pi$ **and** $\Theta = \{(P,PCall \ p,Q)\}$] *assms*
by (*auto simp: HTset-r-def HTset-def*)

```

lemma
  assumes wf R
  assumes  $\bigwedge s P p Q. \llbracket$ 
     $\bigwedge P' p' Q'. (P', p', Q') \in \Theta$ 
     $\implies HT \pi (\lambda s'. ((p', s'), (p, s)) \in R \wedge P' s') (PCall p') Q'$ ;
     $(P, p, Q) \in \Theta; P s$ 
   $\rrbracket \implies wp \pi (PCall p) (Q s) s$ 
  shows  $\forall (P, p, Q) \in \Theta. HT \pi P (PCall p) Q$ 
proof -

  have  $HTset \pi \{(P, PCall p, Q) \mid P p Q. (P, p, Q) \in \Theta\}$ 
  apply (rule HTsetI[where  $R = inv\text{-image } R (\lambda x. case\ x\ of\ (PCall\ p, s) \Rightarrow (p, s))$ ])
  subgoal using  $\langle wf\ R \rangle$  by simp
  subgoal for  $P c Q s$ 
  apply clarsimp
  apply (rule assms(2)[where  $P = P$ ])
  apply simp-all
  unfolding HTset-r-def
  proof goal-cases
    case (1 p P' p' Q')

    from 1(1)[rule-format, of  $(P', PCall\ p', Q')$ , simplified] 1(2-)
    show ?case by auto
  qed
done

thus ?thesis by (auto simp: HTset-def)
qed

```

2.13 Completeness of While-Rule

Idea: Use wlp as invariant

lemma $wlp\text{-while}I'\text{-complete}$:

```

assumes  $wlp \pi (WHILE\ b\ DO\ c) Q s_0$ 
obtains  $I$  where
   $I s_0$ 
   $\bigwedge s. I s \implies if\ bval\ b\ s\ then\ wlp\ \pi\ c\ I\ s\ else\ Q\ s$ 
proof
  let ?I =  $wlp \pi (WHILE\ b\ DO\ c) Q$ 
  {
    show ?I  $s_0$  by fact
  }
next
  fix  $s$ 
  assume ?I  $s$ 
  then show  $if\ bval\ b\ s\ then\ wlp\ \pi\ c\ ?I\ s\ else\ Q\ s$ 
  apply (subst (asm)  $wlp\text{-while-unfold}$ )
  .

```

}
qed

Idea: Remaining loop iterations as variant

inductive *count-it* **for** π *b c* **where**

$\neg \text{bval } b \ s \implies \text{count-it } \pi \ b \ c \ s \ 0$

| $\llbracket \text{bval } b \ s; \pi: (c,s) \Rightarrow s'; \text{count-it } \pi \ b \ c \ s' \ n \rrbracket \implies \text{count-it } \pi \ b \ c \ s$
(*Suc n*)

lemma *count-it-determ*:

$\text{count-it } \pi \ b \ c \ s \ n \implies \text{count-it } \pi \ b \ c \ s \ n' \implies n' = n$

apply (*induction arbitrary: n' rule: count-it.induct*)

subgoal using *count-it.cases* **by** *blast*

subgoal by (*metis big-step-determ count-it.cases*)

done

lemma *count-it-ex*:

assumes $\pi: (\text{WHILE } b \ \text{DO } c, s) \Rightarrow t$

shows $\exists n. \text{count-it } \pi \ b \ c \ s \ n$

using *assms*

apply (*induction* π *WHILE b DO c s t arbitrary: b c*)

apply (*auto intro: count-it.intros*)

done

definition *variant* $\pi \ b \ c \ s \equiv \text{THE } n. \text{count-it } \pi \ b \ c \ s \ n$

lemma *variant-decreases*:

assumes *STEPB*: $\text{bval } b \ s$

assumes *STEPC*: $\pi: (c,s) \Rightarrow s'$

assumes *TERM*: $\pi: (\text{WHILE } b \ \text{DO } c, s') \Rightarrow t$

shows $\text{variant } \pi \ b \ c \ s' < \text{variant } \pi \ b \ c \ s$

proof –

from *count-it-ex*[*OF TERM*] **obtain** n' **where** *CI'*: $\text{count-it } \pi \ b \ c \ s' \ n' ..$

moreover from *count-it.intros*(2)[*OF STEPB STEPC this*] **have** $\text{count-it } \pi \ b \ c \ s \ (\text{Suc } n')$.

ultimately have $\text{variant } \pi \ b \ c \ s' = n' \ \text{variant } \pi \ b \ c \ s = \text{Suc } n'$

unfolding *variant-def* **using** *count-it-determ* **by** *blast+*

thus *?thesis* **by** *simp*

qed

lemma *wp-whileI'-complete*:

fixes $\pi \ b \ c$

defines $R \equiv \text{measure } (\text{variant } \pi \ b \ c)$

assumes *wp* π (*WHILE b DO c*) $Q \ s_0$

obtains I **where**

$wf \ R$

$I \ s_0$

$\bigwedge s. I \ s \implies \text{if } \text{bval } b \ s \ \text{then } wp \ \pi \ c \ (\lambda s'. I \ s' \wedge (s', s) \in R) \ s \ \text{else } Q \ s$

```

proof
  show  $\langle wf\ R \rangle$  unfolding R-def by auto
  let  $?I = wp\ \pi\ (WHILE\ b\ DO\ c)\ Q$ 
  {
    show  $?I\ s_0$  by fact
  }
  next
  fix s
  assume  $?I\ s$ 
  then show if bval b s then wp  $\pi\ c\ (\lambda s'. ?I\ s' \wedge (s',s) \in R)\ s$  else Q
  apply (subst (asm) wp-while-unfold)
  apply clarsimp
  by (auto simp: wp-def R-def intro: variant-decreases)
}
qed

```

end

3 Annotated Syntax

```

theory Annotated-Syntax
imports Semantics
begin

```

Unfold theorems to strip annotations from program, before it is defined as constant

```

  named-theorems vcg-annotation-defs  $\langle$ Definitions of Annotations $\rangle$ 

```

Marker that is inserted around all annotations by the specification parser.

```

  definition ANNOTATION  $\equiv \lambda x. x$ 

```

3.1 Annotations

The specification parser must interpret the annotations in the program.

```

  definition WHILE-annotI  $:: (state \Rightarrow bool) \Rightarrow bexp \Rightarrow com \Rightarrow com$ 

```

```

     $\langle (WHILE\ \{-}\ -/\ DO\ -)\rangle\ [0,\ 0,\ 61]\ 61$ 

```

```

  where [vcg-annotation-defs]: WHILE-annotI ( $I::state \Rightarrow bool$ )  $\equiv$ 
  While

```

lemmas *annotate-whileI* = *WHILE-annotI-def[symmetric]*

definition *WHILE-annotRVI* :: 'a rel \Rightarrow (state \Rightarrow 'a) \Rightarrow (state \Rightarrow bool) \Rightarrow bexp \Rightarrow com \Rightarrow com

(\langle (*WHILE* {-} {-} {-} -/ *DO* -) \rangle [0, 0, 0, 0, 61] 61)

where [*vcg-annotation-defs*]: *WHILE-annotRVI* R V I \equiv *While for* R V I

lemmas *annotate-whileRVI* = *WHILE-annotRVI-def[symmetric]*

definition *WHILE-annotVI* :: (state \Rightarrow int) \Rightarrow (state \Rightarrow bool) \Rightarrow bexp \Rightarrow com \Rightarrow com

(\langle (*WHILE* {-} {-} -/ *DO* -) \rangle [0, 0, 0, 61] 61)

where [*vcg-annotation-defs*]: *WHILE-annotVI* V I \equiv *While for* V I

lemmas *annotate-whileVI* = *WHILE-annotVI-def[symmetric]*

3.2 Hoare-Triples for Annotated Commands

The command is a function from pre-state to command, as the annotations that are contained in the command may depend on the pre-state!

type-synonym *HT'-type* = program \Rightarrow (state \Rightarrow bool) \Rightarrow (state \Rightarrow com) \Rightarrow (state \Rightarrow state \Rightarrow bool) \Rightarrow bool

definition *HT'-partial* :: *HT'-type*

where *HT'-partial* π P c Q \equiv ($\forall s_0. P s_0 \longrightarrow wlp \pi (c s_0) (Q s_0)$ s₀)

definition *HT'* :: *HT'-type*

where *HT'* π P c Q \equiv ($\forall s_0. P s_0 \longrightarrow wp \pi (c s_0) (Q s_0)$ s₀)

lemma *HT'-eq-HT*: *HT'* π P ($\lambda\cdot. c$) Q = *HT* π P c Q

unfolding *HT-def* *HT'-def* ..

lemma *HT'-partial-eq-HT*: *HT'-partial* π P ($\lambda\cdot. c$) Q = *HT-partial* π P c Q

unfolding *HT-partial-def* *HT'-partial-def* ..

lemmas *HT'-unfolds* = *HT'-eq-HT* *HT'-partial-eq-HT*

type-synonym 'a Θ elem-t = (state \Rightarrow 'a) \times ((state \Rightarrow bool) \times (state \Rightarrow com) \times (state \Rightarrow state \Rightarrow bool))

definition *HT'set* :: program \Rightarrow 'a Θ elem-t set \Rightarrow bool **where** *HT'set* π $\Theta \equiv \forall (n,(P,c,Q)) \in \Theta. HT' \pi P c Q$

definition *HT'set-r* :: - \Rightarrow program \Rightarrow 'a Θ elem-t set \Rightarrow bool **where** *HT'set-r* π $\Theta \equiv \forall (n,(P,c,Q)) \in \Theta. HT' \pi (\lambda s. r n s \wedge P s) c Q$

```

lemma HT'setI:
  assumes wf R
  assumes RL:  $\bigwedge f P c Q s_0. \llbracket HT'set-r (\lambda f' s'. ((f' s'), (f s_0))) \in R \rrbracket$ 
 $\pi \Theta; (f, (P, c, Q)) \in \Theta; P s_0 \rrbracket \implies wp \pi (c s_0) (Q s_0) s_0$ 
  shows HT'set  $\pi \Theta$ 
  unfolding HT'set-def HT'-def
proof clarsimp
  fix f_0 P c Q s_0
  assume  $(f_0, (P, c, Q)) \in \Theta P s_0$ 
  with  $\langle wf R \rangle$  show  $wp \pi (c s_0) (Q s_0) s_0$ 
  proof (induction f_0 s_0 arbitrary: f_0 c s_0 P Q)
    case less
    note  $RL' = RL[of f_0 s_0 P, OF - less.premis]$ 
    show ?case
    apply (rule RL')
    unfolding HT'set-r-def HT'-def using less.hyps by auto
  qed
qed

```

```

lemma HT'setD:
  assumes HT'set  $\pi (insert (f, (P, c, Q)) \Theta)$ 
  shows HT'  $\pi P c Q$  and HT'set  $\pi \Theta$ 
  using assms unfolding HT'set-def by auto

```

end

4 Quickstart Guide

```

theory Quickstart-Guide
imports ../IMP2
begin

```

4.1 Introductory Examples

IMP2 provides commands to define program snippets or procedures together with their specification.

```

procedure-spec div-ab (a, b) returns c
  assumes  $\langle b \neq 0 \rangle$ 
  ensures  $\langle c = a_0 \text{ div } b_0 \rangle$ 
  defines  $\langle c = a/b \rangle$ 
  by vcg-cs

```

The specification consists of the signature (name, parameters,

return variables), precondition, postcondition, and program text.

Signature The procedure name and variable names must be valid Isabelle names. The *returns* declaration is optional, by default, nothing is returned. Multiple values can be returned by *returns* (x_1, \dots, x_n) .

Precondition An Isabelle formula. Parameter names are valid variables.

Postcondition An Isabelle formula over the return variables, and parameter names suffixed with $_0$.

Program Text The procedure body, in a C-like syntax.

The **procedure-spec** command will open a proof to show that the program satisfies the specification. The default way of discharging this goal is by using IMP2's verification condition generator, followed by manual discharging of the generated VCs as necessary.

Note that the *vcg-cs* method will apply *clarsimp* to all generated VCs, which, in our case, already solves them. You can use *vcg* to get the raw VCs.

If the VCs have been discharged, **procedure-spec** adds prologue and epilogue code for parameter passing, defines a constant for the procedure, and lifts the pre- and postcondition over the constant definition.

thm *div-ab-spec* — Final theorem proved

thm *div-ab-def* — Constant definition, with parameter passing code

The final theorem has the form $HT\text{-}mods \ \pi \ vs \ P \ c \ Q$, where π is an arbitrary procedure environment, vs is a syntactic approximation of the (global) variables modified by the procedure, P, Q are the pre- and postcondition, lifted over the parameter passing code, and c is the defined constant for the procedure.

The precondition is a function $state \Rightarrow bool$. It starts with a series of variable bindings that map program variables to logical variables, followed by precondition that was specified, wrapped in a *BB-PROTECT* constant, which serves as a tag for the VCG, and is defined as the identity ($BB\text{-}PROTECT \equiv \lambda a. a$).

The final theorem is declared to the VCG, such that the specification will be used automatically for calls to this procedure.

procedure-spec *use-div-ab(a)* **returns** r **assumes** $\langle a \neq 0 \rangle$ **ensures** $\langle r = 1 \rangle$ **defines** $\langle r = div\text{-}ab(a, a) \rangle$ **by** *vcg-cs*

4.1.1 Variant and Invariant Annotations

Loops must be annotated with variants and invariants.

```

procedure-spec mult-ab(a,b) returns c assumes  $\langle True \rangle$  ensures
c=a0*b0
defines  $\langle$ 
  if ( $a < 0$ ) {  $a = -a$ ;  $b = -b$ };
   $c=0$ ;
  while ( $a > 0$ )
    @variant  $\langle a \rangle$ 
    @invariant  $\langle 0 \leq a \wedge a \leq |a_0| \wedge c = (|a_0| - a) * b_0 * \text{sgn } a_0 \rangle$ 
    {
       $c=c+b$ ;
       $a=a-1$ 
    }
   $\rangle$ 
apply vcg-cs
apply (auto simp: algebra-simps)
done

```

The variant and invariant can use the program variables. Variables suffixed with $_0$ refer to the values of parameters at the start of the program.

The variant must be an expression of type *int*, which decreases with every loop iteration and is always ≥ 0 .

Pitfall: If the variant has a more general type, e.g., $\langle a \rangle$, an explicit type annotation must be added. Otherwise, you'll get an ugly error message directly from Isabelle's type checker!

4.1.2 Recursive Procedures

IMP2 supports mutually recursive procedures. All procedures of a mutually recursive specification have to be specified and proved simultaneously.

Each procedure has to be annotated with a variant over the parameters. On a recursive call, the variant of the callee for the arguments must be smaller than the variant of the caller (for its initial arguments).

Recursive invocations inside the specification have to be tagged by the *rec* keyword.

```

recursive-spec
  odd-imp(n) returns b assumes  $n \geq 0$  ensures  $\langle b \neq 0 \iff \text{odd } n_0 \rangle$ 
variant  $\langle n \rangle$ 
  defines  $\langle$  if ( $n == 0$ )  $b=0$  else  $b=\text{rec even-imp}(n-1)$   $\rangle$ 
and

```

```

    even-imp(n) returns b assumes  $n \geq 0$  ensures  $b \neq 0 \iff \text{even}$ 
 $n_0$  variant  $\langle n \rangle$ 
    defines  $\langle \text{if } (n == 0) \text{ } b = 1 \text{ else } b = \text{rec odd-imp}(n - 1) \rangle$ 
    by vcg-cs

```

After proving the VCs, constants are defined as usual, and the correctness theorems are lifted and declared to the VCG for future use.

```

thm odd-imp-spec even-imp-spec

```

4.2 The VCG

The VCG is designed to produce human-readable VCs. It takes care of presenting the VCs with reasonable variable names, and a location information from where a VC originates.

```

procedure-spec mult-ab'(a,b) returns c assumes  $\langle \text{True} \rangle$  ensures
 $c = a_0 * b_0$ 
defines  $\langle$ 
   $\text{if } (a < 0) \{ a = -a; b = -b \};$ 
   $c = 0;$ 
   $\text{while } (a > 0)$ 
    @variant  $\langle a \rangle$ 
    @invariant  $\langle 0 \leq a \wedge a \leq |a_0| \wedge c = (|a_0| - a) * b_0 * \text{sgn } a_0 \rangle$ 
    {
       $c = c + b;$ 
       $a = a - 1$ 
    }
   $\rangle$ 
apply vcg

```

The ∇xxx tags in the premises give a hint to the origin of each VC. Moreover, the variable names are derived from the actual variable names in the program.

```

by (auto simp: algebra-simps)

```

4.3 Advanced Features

4.3.1 Custom Termination Relations

Both for loops and recursive procedures, a custom termination relation can be specified, with the *relation* annotation. The variant must be a function into the domain of this relation.

Pitfall: You have to ensure, by type annotations, that the most general type of the relation and variant fit together. Otherwise, ugly low-level errors will be the result.

```

procedure-spec mult-ab''(a,b) returns c assumes  $\langle \text{True} \rangle$  ensures
 $c = a_0 * b_0$ 

```

```

defines ⟨
  if (a<0) {a = -a; b = -b};
  c=0;
  while (a>0)
    @relation ⟨measure nat⟩
    @variant ⟨a⟩
    @invariant ⟨0≤a ∧ a≤|a0| ∧ c = (|a0| - a) * b0 * sgn a0⟩
    {
      c=c+b;
      a=a-1
    }
  ⟩
by vcg-cs (auto simp: algebra-simps)

recursive-spec relation ⟨measure nat⟩
  odd-imp'(n) returns b assumes n≥0 ensures ⟨b≠0 ⟷ odd n0⟩
variant ⟨n⟩
  defines ⟨if (n==0) b=0 else b=rec even-imp'(n-1)⟩
and
  even-imp'(n) returns b assumes n≥0 ensures ⟨b≠0 ⟷ even
n0⟩ variant ⟨n⟩
  defines ⟨if (n==0) b=1 else b=rec odd-imp'(n-1)⟩
by vcg-cs

```

4.3.2 Partial Correctness

IMP2 supports partial correctness proofs only for while-loops. Recursive procedures must always be proved totally correct¹

```

procedure-spec (partial) nonterminating() returns a assumes
True ensures ⟨a=0⟩ defines
  ⟨while (a≠0) @invariant ⟨True⟩
    a=a-1⟩
by vcg-cs

```

4.3.3 Arrays

IMP2 provides one-dimensional arrays of integers, which are indexed by integers. Arrays do not have to be declared or allocated. By default, every index maps to zero.

In the specifications, arrays are modeled as functions of type *int* ⇒ *int*.

lemma *array-sum-aux*: $l_0 \leq l \implies \{l_0..<l + 1\} = \text{insert } l \{l_0..<l\}$
for $l_0 \ l :: \text{int}$ **by** *auto*

¹Adding partial correctness for recursion is possible, however, compared to total correctness, showing that the prove rule is sound requires some effort that we have not (yet) invested.

```

procedure-spec array-sum(a, l, h) returns s assumes  $l \leq h$  ensures
 $\langle s = (\sum_{i=l_0..<h_0} a_0 \ i) \rangle$  defines
   $\langle s = 0;$ 
    while ( $l < h$ )
      @variant  $\langle h - l \rangle$ 
      @invariant  $\langle l_0 \leq l \wedge l \leq h \wedge s = (\sum_{i=l_0..<l} a \ i) \rangle$ 
      {  $s = s + a[l]; l = l + 1$  }  $\rangle$ 
  apply vcg-cs
  apply (simp add: array-sum-aux)
done

```

4.4 Proving Techniques

This section contains a small collection of techniques to tackle large proofs.

4.4.1 Auxiliary Lemmas

Prove auxiliary lemmas, and try to keep the actual proof of the specification small. As a rule of thumb: All VCs that cannot be solved by a simple *auto* invocation should go to an auxiliary lemma.

The auxiliary lemma may either re-state the whole VC, or only prove the “essence” of the VC, such that the rest of its proof becomes automatic again. See the *array-sum* program above for an example or the latter case.

Pitfall When extracting auxiliary lemmas, it is too easy to get too general types, which may render the lemmas unprovable. As an example, omitting the explicit type constraints from *array-sum-aux* will yield an unprovable statement.

4.4.2 Inlining

More complex procedure bodies can be modularized by either splitting them into multiple procedures, or using inlining and **program-spec** to explicitly prove a specification for a part of a program. Cf. the insertion sort example for the latter technique.

4.4.3 Functional Refinement

Sometimes, it makes sense to state the algorithm functionally first, and then prove that the implementation behaves like the functional program, and, separately, that the functional program is correct. Cf. the mergesort example.

4.4.4 Data Refinement

Moreover, it sometimes makes sense to abstract the concrete variables to abstract types, over which the algorithm is then specified. For example, an array a with a range $l..<h$ can be understood as a list. Or an array can be used as a bitvector set. Cf. the mergesort and dedup examples.

4.5 Troubleshooting

We list a few common problems and their solutions here

4.5.1 Invalid Variables in Annotations

Undeclared variables in annotations are highlighted, however, no warning or error is produced. Usually, the generated VCs will not be provable. The most common mistake is to forget the $_0$ suffix when referring to parameter values in (in)variants and postconditions.

Note the highlighting of unused variables in the following example

```
procedure-spec foo( $x1, x2$ ) returns  $y$  assumes  $x1 > x2 + x3$  ensures  $y = x1_0 + x2$  defines  $\langle$   
   $y = 0;$   
  while ( $x1 > 0$ )  
    @variant  $\langle y + x3 \rangle$   
    @invariant  $\langle y > x3 \rangle$   
    {  
       $x1 = x2$   
    }  
   $\rangle$   
oops
```

Even worse, if the most general type of an annotation becomes too general, as free variables have type $'a$ by default, you will see an internal type error.

Try replacing the variant or invariant with a free variable in the above example.

4.5.2 Wrong Annotations

For total correctness, you must annotate a loop variant and invariant. For partial correctness, you must annotate an invariant, but **no variant**.

When not following this rule, the VCG will get stuck in an internal state

```
procedure-spec (partial) foo () assumes True ensures True de-  
fines <  
  while (n>0) @variant <n> @invariant <True>  
  { n=n-1 }  
>  
apply vcg  
oops
```

4.5.3 Calls to Undefined Procedures

Calling an undefined procedure usually results in a type error, as the procedure name gets interpreted as an Isabelle term, e.g., either it refers to an existing constant, or is interpreted as a free variable

4.6 Missing Features

This is an (incomplete) list of missing features.

4.6.1 Elaborate Warnings and Errors

Currently, the IMP2 tools only produce minimal error and warning messages. Quite often, the user sees the raw error message as produced by Isabelle unfiltered, including all internal details of the tools.

4.6.2 Static Type Checking

We do no static type checking at all. In particular, we do not check, nor does our semantic enforce, that procedures are called with the same number of arguments as they were declared. Programs that violate this convention may even have provable properties, as argument and parameter passing is modeled as macros on top of the semantics, and the semantics has no notion of failure.

4.6.3 Structure Types

Every variable is an integer arrays. Plain integer variables are implemented as macros on top of this, by referring to index 0 .

The most urgent addition to increase usability would be record types. With them, we could model encapsulation and data refinement more explicitly, by collecting all parts of a data structure in a single (record-typed) variable.

An easy way of adding record types would follow a similar route as arrays, modeling values of variables as a recursive tree-structured datatype.

datatype $val = PRIM\ int \mid STRUCT\ fname \Rightarrow val \mid ARRAY\ int \Rightarrow val$

However, for modeling the semantics, we most likely want to introduce an explicit error state, to distinguish type errors (e.g. accessing a record field of an integer value) from nontermination.

4.6.4 Function Calls as Expressions

Currently, function calls are modeled as statements, and thus, cannot be nested into expressions. Doing so would require to simultaneously specify the semantics of commands and expressions, which makes things more complex.

As the language is intended to be simple, we have not done this.

4.6.5 Ghost Variables

Ghost variables are a valuable tool for expressing (data) refinement, and hinting the VCG towards the abstract algorithm structure.

We believe that we can add ghost variables with annotations on top of the VCG, without actually changing the program semantics.

4.6.6 Concurrency

IMP2 is a single threaded language. We have no current plans to add concurrency, as this would greatly complicate both the semantics and the VCG, which is contrary to the goal of a simple language for educational purposes.

4.6.7 Pointers and Memory

Adding pointers and memory allocation to IMP2 is theoretically possible, but, again, this would complicate the semantics and the VCG.

However, as the author has some experience in VCGs using separation logic, he might actually add pointers and memory allocation to IMP2 in the near future.

end

5 Introduction to IMP2-VCG, based on IMP

```
theory IMP2-from-IMP
imports ../IMP2
begin
```

This document briefly introduces the extensions of IMP2 over IMP.

5.1 Fancy Syntax

Standard Syntax

```
definition exp-count-up1 ≡
  "a" ::= N 1;;
  "c" ::= N 0;;
  WHILE Cmpop (<) (V "c") (V "n") DO (
    "a" ::= Binop (*) (N 2) (V "a");;
    "c" ::= Binop (+) (V "c") (N 1))
```

Fancy Syntax

```
definition exp-count-up2 ≡ imp<
  — Initialization
  a = 1;
  c = 0;
  while (c < n) { — Iterate until c has reached n
    a = 2 * a; — Double a
    c = c + 1 — Increment c
  }
>
```

```
lemma exp-count-up1 = exp-count-up2
  unfolding exp-count-up1-def exp-count-up2-def ..
```

5.2 Operators and Arrays

We reflect arbitrary Isabelle functions into the syntax:

```
value bval (Cmpop (≤) (Binop (+) (Unop uminus (V "x")) (N 42))
(N 50)) <"x" := (λ-. -5)>
```

```
thm aval.simps bval.simps
```

Every variable is an array, indexed by integers, no bounds. Syntax shortcuts to access index 0.

term $\langle \text{Vidx } "a" (i::aexp) \rangle$ — Array access at index i

lemma $V "x" = \text{Vidx } "x" (N 0) ..$ — Shortcut for access at index 0

New commands:

term $\langle \text{AssignIdx } "a" (i::aexp) (v::aexp) \rangle$ — Assign at index. Replaces assign.

term $\langle "a"[i] ::= v \rangle$ — Standard syntax

term $\langle \text{imp} \langle a[i] = v \rangle \rangle$ — Fancy syntax

lemma $\langle \text{Assign } "x" v = \text{AssignIdx } "x" (N 0) v \rangle ..$ — Shortcut for assignment to index 0

term $\langle "x" ::= v \rangle$ **term** $\langle \text{imp} \langle x = v+1 \rangle \rangle$

Note: In fancy syntax, assignment between variables is always parsed as array copy. This is no problem unless a variable is used as both, array and plain value, which should be avoided anyway.

term $\langle \text{ArrayCpy } "d" "s" \rangle$ — Copy whole array. Both operands are variable names.

term $\langle "d"[] ::= "s" \rangle$ **term** $\langle \text{imp} \langle d = s \rangle \rangle$

term $\langle \text{ArrayClear } "a" \rangle$ — Initialize array to all zeroes.

term $\langle \text{CLEAR } "a"[] \rangle$ **term** $\langle \text{imp} \langle \text{clear } a[] \rangle \rangle$

Semantics of these is straightforward

thm $\text{big-step.AssignIdx big-step.ArrayCpy big-step.ArrayClear}$

5.3 Local and Global Variables

term $\langle \text{is-global} \rangle$ **term** $\langle \text{is-local} \rangle$ — Partitions variable names

term $\langle \langle s_1 | s_2 \rangle \rangle$ — State with locals from s_1 and globals from s_2

term $\langle \text{SCOPE } c \rangle$ **term** $\langle \text{imp} \langle \text{scope } \{ \text{skip} \} \rangle \rangle$ — Execute c with fresh set of local variables

thm big-step.Scope

5.3.1 Parameter Passing

Parameters and return values by global variables: This is syntactic sugar only:

context **fixes** $f :: com$ **begin**

term $\langle \text{imp} \langle (r1, r2) = f(x1, x2, x3) \rangle \rangle$

end

5.4 Recursive procedures

```
term <PCall "name">
thm big-step.PCall
```

5.4.1 Procedure Scope

Execute command with local set of procedures

```
term <PScope  $\pi$  c>
thm big-step.PScope
```

5.4.2 Syntactic sugar for procedure call with parameters

```
term <imp<(r1,r2) = rec name(x1,x2,x3)>>
```

5.5 More Readable VCs

```
lemmas nat-distrib = nat-add-distrib nat-diff-distrib Suc-diff-le nat-mult-distrib
nat-div-distrib
```

```
lemma  $s_0$  "n" 0  $\geq$  0  $\implies$  wlp  $\pi$  exp-count-up1 ( $\lambda s. s$  "a" 0 =
 $2^{\wedge}nat (s_0$  "n" 0))  $s_0$ 
  unfolding exp-count-up1-def
  apply (subst annotate-whileI[where
     $I = \lambda s. s$  "n" 0 =  $s_0$  "n" 0  $\wedge$   $s$  "a" 0 =  $2^{\wedge}nat (s$  "c" 0)
 $\wedge$  0  $\leq s$  "c" 0  $\wedge s$  "c" 0  $\leq s_0$  "n" 0
  ])
  apply (i-vcg-preprocess; i-vcg-gen; clarsimp)
```

The postprocessor converts from states applied to string names to actual variables

```
apply i-vcg-postprocess
by (auto simp: algebra-simps nat-distrib)
```

```
lemma  $s_0$  "n" 0  $\geq$  0  $\implies$  wlp  $\pi$  exp-count-up1 ( $\lambda s. s$  "a" 0 =
 $2^{\wedge}nat (s_0$  "n" 0))  $s_0$ 
  unfolding exp-count-up1-def
  apply (subst annotate-whileI[where
     $I = \lambda s. s$  "n" 0 =  $s_0$  "n" 0  $\wedge$   $s$  "a" 0 =  $2^{\wedge}nat (s$  "c" 0)
 $\wedge$  0  $\leq s$  "c" 0  $\wedge s$  "c" 0  $\leq s_0$  "n" 0
  ])
  apply (i-vcg-preprocess; i-vcg-gen; clarsimp)
```

The postprocessor is invoked by default

```
apply vcg
oops
```

5.6 Specification Commands

IMP2 provides a set of commands to simplify specification and annotation of programs.

Old way of proving a specification:

```

lemma let n = s0 "n" 0 in n ≥ 0
  ⇒ wlp π exp-count-up1 (λs. let a = s "a" 0; n0 = s0 "n" 0 in
a = 2nat (n0)) s0
  unfolding exp-count-up1-def
  apply (subst annotate-whileI[where
    I=λs. s "n" 0 = s0 "n" 0 ∧ s "a" 0 = 2nat (s "c" 0)
    ∧ 0 ≤ s "c" 0 ∧ s "c" 0 ≤ s0 "n" 0

    ])
  apply vcg
  apply (auto simp: algebra-simps nat-distrib)
  done

```

```

lemma VAR (s x) P = (let v=s x in P v) unfolding VAR-def by
simp

```

IMP2 specification commands

```

program-spec (partial) exp-count-up
  assumes 0 ≤ n — Precondition. Use variable names of
program.
  ensures a = 2nat n0 — Postcondition. Use variable names
of programs. Suffix with ·0 to refer to initial state
  defines — Program
  <
    a = 1;
    c = 0;
    while (c < n)
      @invariant ⟨n=n0 ∧ a=2nat c ∧ 0 ≤ c ∧ c ≤ n⟩ — Invar
annotation. Variable names and suffix ·0 for variables from initial
state.
    {
      a=2*a;
      c=c+1
    }
  >
  apply vcg
  by (auto simp: algebra-simps nat-distrib)

thm exp-count-up-spec
thm exp-count-up-def

```

```

procedure-spec exp-count-up-proc(n) returns a
  assumes  $0 \leq n$ 
  ensures  $a = 2^{\widehat{\text{nat}} n_0}$ 
  defines
  <
    a = 1;
    c = 0;
    while (c < n)
      @invariant  $\langle n = n_0 \wedge a = 2^{\widehat{\text{nat}} c} \wedge 0 \leq c \wedge c \leq n \rangle$ 
      @variant  $\langle n - c \rangle$ 
      {
        a = 2 * a;
        c = c + 1
      }
  >
  apply vcg
  by (auto simp: algebra-simps nat-distrib)

```

Simple Recursion

```

recursive-spec
  exp-rec(n) returns a assumes  $0 \leq n$  ensures  $a = 2^{\widehat{\text{nat}} n_0}$  variant
  n
  defines  $\langle \text{if } (n == 0) \text{ } a = 1 \text{ else } \{ t = \text{rec } \text{exp-rec}(n-1); a = 2 * t \} \rangle$ 
  apply vcg
  apply (auto simp: algebra-simps nat-distrib)
  by (metis Suc-le-D diff-Suc-Suc dvd-1-left dvd-imp-le minus-nat.diff-0
nat-0-iff nat-int neq0-conv of-nat-0 order-class.order.antisym pos-int-cases
power-Suc zless-nat-eq-int-zless)

```

Mutual Recursion: See Examples

```

end
theory Examples
imports ../IMP2 ../lib/IMP2-Aux-Lemmas
begin

```

6 Examples

```

lemmas nat-distrib = nat-add-distrib nat-diff-distrib Suc-diff-le nat-mult-distrib
nat-div-distrib

```

6.1 Common Loop Patterns

6.1.1 Count Up

Counter *c* counts from 0 to *n*, such that loop is executed *n* times.
The result is computed in an accumulator *a*.

The invariant states that we have computed the function for the counter value c

The variant is the difference between n and c , i.e., the number of loop iterations that we still have to do

```

program-spec exp-count-up
assumes  $0 \leq n$ 
ensures  $a = 2^{\widehat{\text{nat}} n_0}$ 
defines <
   $a = 1;$ 
   $c = 0;$ 
  while ( $c < n$ )
    @variant  $\langle n - c \rangle$ 
    @invariant  $\langle 0 \leq c \wedge c \leq n \wedge a = 2^{\widehat{\text{nat}} c} \rangle$ 
    {
       $G\text{-par} = a;$  scope {  $a = G\text{-par}; a = 2 * a; G\text{-ret} = a$  };  $a = G\text{-ret};$ 
       $c = c + 1$ 
    }
  >
apply vcg
by (auto simp: algebra-simps nat-distrib)

```

```

program-spec sum-prog
assumes  $n \geq 0$  ensures  $s = \sum \{0..n_0\}$ 
defines <
   $s = 0;$ 
   $i = 0;$ 
  while ( $i < n$ )
    @variant  $\langle n_0 - i \rangle$ 
    @invariant  $\langle n_0 = n \wedge 0 \leq i \wedge i \leq n \wedge s = \sum \{0..i\} \rangle$ 
    {
       $i = i + 1;$ 
       $s = s + i$ 
    }
  >
apply vcg-cs
by (simp-all add: intvs-incdec)

```

```

program-spec sq-prog
assumes  $n \geq 0$  ensures  $a = n_0 * n_0$ 
defines <
   $a = 0;$ 
   $z = 1;$ 
   $i = 0;$ 
  while ( $i < n$ )
    @variant  $\langle n_0 - i \rangle$ 
    @invariant  $\langle n_0 = n \wedge 0 \leq i \wedge i \leq n \wedge a = i * i \wedge z = 2 * i +$ 
1>
    {

```

```

    a = a + z;
    z = z + 2;
    i = i + 1
  }
}
>
by vcg-cs (simp add: algebra-simps)

fun factorial :: int  $\Rightarrow$  int where
  factorial i = (if i  $\leq$  0 then 1 else i * factorial (i - 1))

program-spec factorial-prog
assumes  $n \geq 0$  ensures  $a = \text{factorial } n_0$ 
defines  $\langle$ 
  a = 1;
  i = 1;
  while ( $i \leq n$ )
    @variant  $\langle n_0 + 1 - i \rangle$ 
    @invariant  $\langle n_0 = n \wedge 1 \leq i \wedge i \leq n + 1 \wedge a = \text{factorial } (i -$ 
1)  $\rangle$ 
  {
    a = a * i;
    i = i + 1
  }
   $\rangle$ 
by vcg (simp add: antisym-conv)+

```

```

fun fib :: int  $\Rightarrow$  int where
  fib i = (if i  $\leq$  0 then 0 else if i = 1 then 1 else fib (i - 2) + fib (i - 1))

```

```

lemma fib-simps[simp]:
   $i \leq 0 \Longrightarrow \text{fib } i = 0$ 
   $i = 1 \Longrightarrow \text{fib } i = 1$ 
   $i > 1 \Longrightarrow \text{fib } i = \text{fib } (i - 2) + \text{fib } (i - 1)$ 
by simp+

```

```

lemmas [simp del] = fib.simps

```

With precondition

```

program-spec fib-prog
assumes  $n \geq 0$  ensures  $a = \text{fib } n$ 
defines  $\langle$ 
  a = 0; b = 1;
  i = 0;
  while ( $i < n$ )
    @variant  $\langle n_0 - i \rangle$ 
    @invariant  $\langle n = n_0 \wedge 0 \leq i \wedge i \leq n \wedge a = \text{fib } i \wedge b = \text{fib } (i +$ 
1)  $\rangle$ 

```

```

    {
      c = b;
      b = a + b;
      a = c;
      i = i + 1
    }
  }
  >
  by vcg-cs (simp add: algebra-simps)

```

Without precondition, returning 0 for negative numbers

```

program-spec fib-prog'
  assumes True ensures a = fib n0
  defines <
    a = 0; b = 1;
    i = 0;
    while (i < n)
      @variant <n0 - i>
      @invariant <n = n0 ∧ (0 ≤ i ∧ i ≤ n ∨ n0 < 0 ∧ i = 0) ∧ a =
fib i ∧ b = fib (i + 1)>
    {
      c = b;
      b = a + b;
      a = c;
      i = i + 1
    }
  }
  >
  by vcg-cs (auto simp: algebra-simps)

```

6.1.2 Count down

Essentially the same as count up, but we (ab)use the input variable as a counter.

The invariant is the same as for count-up. Only that we have to compute the actual number of loop iterations by $n_0 - n$. We locally introduce the name c for that.

```

program-spec exp-count-down
  assumes 0 ≤ n
  ensures a = 2nat n0
  defines <
    a = 1;
    while (n > 0)
      @variant <n>
      @invariant <let c = n0 - n in 0 ≤ n ∧ n ≤ n0 ∧ a = 2nat c>
    {
      a = 2 * a;
      n = n - 1
    }
  }

```

```

>
apply vcg-cs
by (auto simp: algebra-simps nat-distrib)

```

6.1.3 Approximate from Below

Used to invert a monotonic function. We count up, until we overshoot the desired result, then we subtract one.

The invariant states that the $r-1$ is not too big. When the loop terminates, $r-1$ is not too big, but r is already too big, so $r-1$ is the desired value (rounding down).

The variant measures the gap that we have to the correct result. Note that the loop will do a final iteration, when the result has been reached exactly. We account for that by adding one, such that the measure also decreases in this case.

```

program-spec sqr-approx-below
assumes  $0 \leq n$ 
ensures  $0 \leq r \wedge r^2 \leq n_0 \wedge n_0 < (r+1)^2$ 
defines <
   $r=1$ ;
  while ( $r*r \leq n$ )
    @variant < $n + 1 - r*r$ >
    @invariant < $0 \leq r \wedge (r-1)^2 \leq n_0$ >
    {  $r = r + 1$  };
   $r = r - 1$ 
>
apply vcg
by (auto simp: algebra-simps power2-eq-square)

```

6.1.4 Bisection

A more efficient way of inverting monotonic functions is by bisection, that is, one keeps track of a possible interval for the solution, and halves the interval in each step. The program will need $O(\log n)$ iterations, and is thus very efficient in practice.

Although the final algorithm looks quite simple, getting it right can be quite tricky.

The invariant is surprisingly easy, just stating that the solution is in the interval $l..<h$.

```

lemma  $\bigwedge h l n_0 :: int.$ 
   $\llbracket \text{"invar-final"}; 0 \leq n_0; \neg 1 + l < h; 0 \leq l; l < h; l * l \leq n_0;$ 
   $n_0 < h * h \rrbracket$ 
   $\implies n_0 < 1 + (l * l + l * 2)$ 
by (smt mult.commute semiring-normalization-rules(3))

```

```

program-spec sqr-bisect
assumes  $0 \leq n$  ensures  $r^2 \leq n_0 \wedge n_0 < (r+1)^2$ 
defines <
   $l=0; h=n+1;$ 
  while  $(l+1 < h)$ 
    @variant  $\langle h-l \rangle$ 
    @invariant  $\langle 0 \leq l \wedge l < h \wedge l^2 \leq n \wedge n < h^2 \rangle$ 
    {
       $m = (l + h) / 2;$ 
      if  $(m*m \leq n)$   $l=m$  else  $h=m$ 
    };
   $r=l$ 
  >
apply vcg

```

We use quick-and-dirty apply style proof to discharge the VCs

```

apply (auto simp: power2-eq-square algebra-simps add-pos-pos)
apply (smt not-sum-squares-lt-zero)
by (smt mult.commute semiring-normalization-rules(3))

```

6.2 Debugging

6.2.1 Testing Programs

Stepwise

```

schematic-goal Map.empty: (sqr-approx-below, <"n" := λ-. 4>) ⇒ ?s
unfolding sqr-approx-below-def
apply big-step'
apply big-step'
apply big-step'
apply big-step'
apply big-step'
apply big-step'
apply big-step'
apply big-step'
apply big-step'
done

```

Or all steps at once

```

schematic-goal Map.empty: (sqr-bisect, <"n" := λ-. 4900000001>) ⇒ ?s
unfolding sqr-bisect-def
by big-step

```

6.3 More Numeric Algorithms

6.3.1 Euclid's Algorithm (with subtraction)

```

thm gcd.commute gcd-diff1

```

```

program-spec euclid1
  assumes  $a > 0 \wedge b > 0$ 
  ensures  $a = \text{gcd } a_0 \ b_0$ 
  defines <
    while ( $a \neq b$ )
      @invariant < $\text{gcd } a \ b = \text{gcd } a_0 \ b_0 \wedge (a > 0 \wedge b > 0)$ >
      @variant < $a + b$ >
      {
        if ( $a < b$ )  $b = b - a$ 
        else  $a = a - b$ 
      }
    >
  apply vcg-cs
  apply (metis gcd.commute gcd-diff1)
  apply (metis gcd-diff1)
  done

```

6.3.2 Euclid's Algorithm (with mod)

thm *gcd-red-int[symmetric]*

```

program-spec euclid2
  assumes  $a > 0 \wedge b > 0$ 
  ensures  $a = \text{gcd } a_0 \ b_0$ 
  defines <
    while ( $b \neq 0$ )
      @invariant < $\text{gcd } a \ b = \text{gcd } a_0 \ b_0 \wedge b \geq 0 \wedge a > 0$ >
      @variant < $b$ >
      {
         $t = a;$ 
         $a = b;$ 
         $b = t \text{ mod } b$ 
      }
    >
  apply vcg-cs
  by (simp add: gcd-red-int[symmetric])

```

6.3.3 Extended Euclid's Algorithm

locale *extended-euclid-aux-lemmas* **begin**

lemma *aux2*:

```

  fixes  $a \ b :: \text{int}$ 
  assumes  $b = t * b_0 + s * a_0 \ q = a \text{ div } b \ \text{gcd } a \ b = \text{gcd } a_0 \ b_0$ 
  shows  $\text{gcd } b \ (a - (a_0 * (s * q) + b_0 * (t * q))) = \text{gcd } a_0 \ b_0$ 
proof -
  have  $a - (a_0 * (s * q) + b_0 * (t * q)) = a - b * q$ 
    unfolding < $b = -$ > by (simp add: algebra-simps)
  also have  $a - b * q = a \text{ mod } b$ 

```

```

    unfolding ⟨q = -⟩ by (simp add: algebra-simps)
  finally show ?thesis
    unfolding ⟨gcd a b = -⟩[symmetric] by (simp add: gcd-red-int[symmetric])
qed

```

lemma *aux3*:

```

  fixes a b :: int
  assumes b = t * b0 + s * a0 q = a div b b > 0
  shows t * (b0 * q) + s * (a0 * q) ≤ a
proof -
  have t * (b0 * q) + s * (a0 * q) = q * b
    unfolding ⟨b = -⟩ by (simp add: algebra-simps)
  then show ?thesis
    using ⟨b > 0⟩
    by (simp add: algebra-simps ⟨q = a div b⟩)
      (simp flip: minus-mod-eq-mult-div)
qed

```

end

The following is a direct translation of the pseudocode for the Extended Euclidean algorithm as described by the English version of Wikipedia (https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm):

program-spec *euclid-extended*

```

  assumes a>0 ∧ b>0
  ensures old-r = gcd a0 b0 ∧ gcd a0 b0 = a0 * old-s + b0 * old-t
  defines ⟨
    s = 0;    old-s = 1;
    t = 1;    old-t = 0;
    r = b;    old-r = a;
    while (r≠0)
      @invariant ⟨
        gcd old-r r = gcd a0 b0 ∧ r ≥ 0 ∧ old-r > 0
        ∧ a0 * old-s + b0 * old-t = old-r ∧ a0 * s + b0 * t = r
      ⟩
      @variant ⟨r⟩
    {
      quotient = old-r / r;
      temp = old-r;
      old-r = r;
      r = temp - quotient * r;
      temp = old-s;
      old-s = s;
      s = temp - quotient * s;
      temp = old-t;
      old-t = t;
      t = temp - quotient * t
    }
  ⟩

```

```

>
proof –
  interpret extended-euclid-aux-lemmas .
  show ?thesis
    apply vcg-cs
    apply (simp add: algebra-simps)
    apply (simp add: aux2 aux3 minus-div-mult-eq-mod)+
  done

```

qed

Non-Wikipedia version

```

context extended-euclid-aux-lemmas begin
  lemma aux:
    fixes a b q x y:: int
    assumes  $a = \text{old-}y * b_0 + \text{old-}x * a_0$   $b = y * b_0 + x * a_0$   $q = a$ 
    div b
    shows
       $a \bmod b + (a_0 * (x * q) + b_0 * (y * q)) = a$ 
    proof –
      have  $*$ :  $a_0 * (x * q) + b_0 * (y * q) = q * b$ 
        unfolding  $\langle b = \rightarrow \rangle$  by (simp add: algebra-simps)
      show ?thesis
        unfolding  $*$  unfolding  $\langle q = \rightarrow \rangle$  by simp
    qed
  end

```

```

program-spec euclid-extended'
  assumes  $a > 0 \wedge b > 0$ 
  ensures  $a = \text{gcd } a_0 \ b_0 \wedge \text{gcd } a_0 \ b_0 = a_0 * x + b_0 * y$ 
  defines  $\langle$ 
     $x = 0;$ 
     $y = 1;$ 
     $\text{old-}x = 1;$ 
     $\text{old-}y = 0;$ 
    while ( $b \neq 0$ )
      @invariant  $\langle$ 
         $\text{gcd } a \ b = \text{gcd } a_0 \ b_0 \wedge b \geq 0 \wedge a > 0 \wedge a = a_0 * \text{old-}x + b_0 * \text{old-}y \wedge b = a_0 * x + b_0 * y$ 
       $\rangle$ 
      @variant  $\langle b \rangle$ 
    {
       $q = a / b;$ 
       $t = a;$ 
       $a = b;$ 
       $b = t \bmod b;$ 
       $t = x;$ 
       $x = \text{old-}x - q * x;$ 
       $\text{old-}x = t;$ 
    }
   $\rangle$ 

```

```

    t = y;
    y = old-y - q * y;
    old-y = t
  };
  x = old-x;
  y = old-y
}
proof -
interpret extended-euclid-aux-lemmas .
show ?thesis
  apply vcg-cs
  apply (simp add: gcd-red-int[symmetric])
  apply (simp add: algebra-simps)
  apply (rule aux; simp add: algebra-simps)
  done
qed

```

6.3.4 Exponentiation by Squaring

```

lemma ex-by-sq-aux:
  fixes x :: int and n :: nat
  assumes n mod 2 = 1
  shows  $x * (x * x)^{\wedge} (n \text{ div } 2) = x^{\wedge} n$ 
proof -
  have  $n > 0$ 
    using assms by presburger
  have  $2 * (n \text{ div } 2) = n - 1$ 
    using assms by presburger
  then have  $(x * x)^{\wedge} (n \text{ div } 2) = x^{\wedge} (n - 1)$ 
    by (simp add: semiring-normalization-rules)
  with  $\langle 0 < n \rangle$  show ?thesis
    by simp (metis Suc-pred power.simps(2))
qed

```

A classic algorithm for computing x^n works by repeated squaring, using the following recurrence:

- $x^n = x * x^{(n-1)/2^2}$ if n is odd
- $x^n = x^{n/2^2}$ if n is even

```

program-spec ex-by-sq
  assumes  $n \geq 0$ 
  ensures  $r = x_0^{\wedge} \text{nat } n_0$ 
defines <
   $r = 1$ ;
  while ( $n \neq 0$ )
    @invariant <
       $n \geq 0 \wedge r * x^{\wedge} \text{nat } n = x_0^{\wedge} \text{nat } n_0$ 
    >
  >

```

```

    @variant ⟨n⟩
    {
      if (n mod 2 == 1) {
        r = r * x
      };
      x = x * x;
      n = n / 2
    }
  }
}
apply vcs
apply (auto simp flip: odd-iff-mod-2-eq-one
simp add: nat-add-distrib nat-mult-distrib power2-eq-square power-mult
elim!: oddE evenE)
done

```

6.3.5 Power-Tower of 2s

```

fun tower2 where
  tower2 0 = 1
| tower2 (Suc n) = 2 ^ tower2 n

```

definition *tower2'* n = int (*tower2* (nat n))

```

program-spec tower2-imp
assumes ⟨m>0⟩
ensures ⟨a = tower2' m0⟩
defines ⟨
  a=1;
  while (m>0)
    @variant ⟨m⟩
    @invariant ⟨0 ≤ m ∧ m ≤ m0 ∧ a = tower2' (m0-m)⟩
  {
    n=a;

    a = 1;
    while (n>0)
      @variant ⟨n⟩
      @invariant ⟨True⟩ — This will get ugly, there is no n0 that we
could use!
    {
      a=2*a;
      n=n-1
    };

    m=m-1
  }
  ⟩
oops

```

We prove the inner loop separately instead! (It happens to be

exactly our *exp-count-down* program.)

```

program-spec tower2-imp
  assumes  $\langle m > 0 \rangle$ 
  ensures  $\langle a = \text{tower2}' m_0 \rangle$ 
  defines  $\langle$ 
     $a = 1;$ 
    while  $(m > 0)$ 
       $\text{@variant } \langle m \rangle$ 
       $\text{@invariant } \langle 0 \leq m \wedge m \leq m_0 \wedge a = \text{tower2}' (m_0 - m) \rangle$ 
      {
         $n = a;$ 
        inline exp-count-down;
         $m = m - 1$ 
      }
     $\rangle$ 
  apply vcg-cs
  by (auto simp: algebra-simps tower2'-def nat-distrib)

```

6.4 Array Algorithms

6.4.1 Summation

```

program-spec array-sum
  assumes  $l \leq h$ 
  ensures  $r = (\sum_{i=l_0..<h_0} a_0 i)$ 
  defines  $\langle$ 
     $r = 0;$ 
    while  $(l < h)$ 
       $\text{@invariant } \langle l_0 \leq l \wedge l \leq h \wedge r = (\sum_{i=l_0..<l} a_0 i) \rangle$ 
       $\text{@variant } \langle h - l \rangle$ 
      {
         $r = r + a[l];$ 
         $l = l + 1$ 
      }
     $\rangle$ 
  apply vcg-cs
  by (auto simp: intvs-incdec)

```

6.4.2 Finding Least Index of Element

```

program-spec find-least-idx
  assumes  $\langle l \leq h \rangle$ 
  ensures  $\langle \text{if } l = h_0 \text{ then } x_0 \notin a_0 \{l_0..<h_0\} \text{ else } l \in \{l_0..<h_0\} \wedge a_0 l = x_0 \wedge x_0 \notin a_0 \{l_0..<l\} \rangle$ 
  defines  $\langle$ 
    while  $(l < h \wedge a[l] \neq x)$ 
       $\text{@variant } \langle h - l \rangle$ 
       $\text{@invariant } \langle l_0 \leq l \wedge l \leq h \wedge x \notin a \{l_0..<l\} \rangle$ 
       $l = l + 1$ 
     $\rangle$ 

```

```

apply vcg-cs
by (smt atLeastLessThan-iff imageI)

```

6.4.3 Check for Sortedness

term *ran-sorted*

```

program-spec check-sorted
assumes  $\langle l \leq h \rangle$ 
ensures  $\langle r \neq 0 \longleftrightarrow \text{ran-sorted } a_0 \ l_0 \ h_0 \rangle$ 
defines  $\langle$ 
  if ( $l == h$ )  $r = 1$ 
  else {
     $l = l + 1$ ;
    while ( $l < h \wedge a[l-1] \leq a[l]$ )
       $@\text{variant } \langle h - l \rangle$ 
       $@\text{invariant } \langle l_0 < l \wedge l \leq h \wedge \text{ran-sorted } a \ l_0 \ l \rangle$ 
       $l = l + 1$ ;
  }
  if ( $l == h$ )  $r = 1$  else  $r = 0$ 
 $\rangle$ 
apply vcg-cs
apply (auto simp: ran-sorted-def)
by (smt atLeastLessThan-iff)

```

6.4.4 Find Equilibrium Index

definition *is-equil* $a \ l \ h \ i \equiv l \leq i \wedge i < h \wedge (\sum_{j=l..<i.} a \ j) = (\sum_{j=i..<h.} a \ j)$

```

program-spec equilibrium
assumes  $\langle l \leq h \rangle$ 
ensures  $\langle \text{is-equil } a \ l \ h \ i \vee i = h \wedge (\nexists i. \text{is-equil } a \ l \ h \ i) \rangle$ 
defines  $\langle$ 
   $usum = 0$ ;  $i = l$ ;
  while ( $i < h$ )
     $@\text{variant } \langle h - i \rangle$ 
     $@\text{invariant } \langle l \leq i \wedge i \leq h \wedge usum = (\sum_{j=l..<i.} a \ j) \rangle$ 
  {
     $usum = usum + a[i]$ ;  $i = i + 1$ 
  }
   $i = l$ ;  $lsum = 0$ ;
  while ( $usum \neq lsum \wedge i < h$ )
     $@\text{variant } \langle h - i \rangle$ 
     $@\text{invariant } \langle l \leq i \wedge i \leq h$ 
       $\wedge lsum = (\sum_{j=l..<i.} a \ j)$ 
       $\wedge usum = (\sum_{j=i..<h.} a \ j)$ 
       $\wedge (\forall j < i. \neg \text{is-equil } a \ l \ h \ j)$ 
     $\rangle$ 

```

```

    {
      lsum = lsum + a[i];
      usum = usum - a[i];
      i=i+1
    }
  }
}
apply vcg-cs
  apply (auto simp: intvs-incdec is-equil-def)
  apply (metis atLeastLessThan-iff eq-iff finite-atLeastLessThan-int
sum-diff1)
  apply force
by force

```

6.4.5 Rotate Right

```

program-spec rotate-right
assumes  $0 < n$ 
ensures  $\forall i \in \{0..<n\}. a\ i = a_0\ ((i-1)\ \text{mod}\ n)$ 
defines <
   $i = 0;$ 
   $prev = a[n - 1];$ 
  while ( $i < n$ )
    @invariant <
       $0 \leq i \wedge i \leq n$ 
       $\wedge (\forall j \in \{0..<i\}. a\ j = a_0\ ((j-1)\ \text{mod}\ n))$ 
       $\wedge (\forall j \in \{i..<n\}. a\ j = a_0\ j)$ 
       $\wedge prev = a_0\ ((i-1)\ \text{mod}\ n)$ 
    >
  >
  @variant < $n - i$ >
  {
     $temp = a[i];$ 
     $a[i] = prev;$ 
     $prev = temp;$ 
     $i = i + 1$ 
  }
  >
apply vcg-cs
by (simp add: zmod-minus1)

```

6.4.6 Binary Search, Leftmost Element

We first specify the pre- and postcondition

definition *bin-search-pre* $a\ l\ h \longleftrightarrow l \leq h \wedge \text{ran-sorted}\ a\ l\ h$

definition *bin-search-post* $a\ l\ h\ x\ i \longleftrightarrow$
 $l \leq i \wedge i \leq h \wedge (\forall i \in \{l..<i\}. a\ i < x) \wedge (\forall i \in \{i..<h\}. x \leq a\ i)$

Then we prove that the program is correct

program-spec *binsearch*

```

assumes  $\langle \text{bin-search-pre } a \ l \ h \rangle$ 
ensures  $\langle \text{bin-search-post } a_0 \ l_0 \ h_0 \ x_0 \ l \rangle$ 
defines  $\langle$ 
  while  $\langle l < h \rangle$ 
    @variant  $\langle h-l \rangle$ 
    @invariant  $\langle l_0 \leq l \wedge l \leq h \wedge h \leq h_0 \wedge (\forall i \in \{l_0..<l\}. a \ i < x) \wedge$ 
 $\langle (\forall i \in \{h..<h_0\}. x \leq a \ i) \rangle$ 
    {
       $m = (l + h) / 2;$ 
      if  $\langle a[m] < x \rangle$   $l = m + 1$ 
      else  $h = m$ 
    }
   $\rangle$ 
apply vcg-cs
  apply  $(\text{auto simp: algebra-simps bin-search-pre-def bin-search-post-def})$ 

```

Driving sledgehammer to its limits ...

```

apply  $(\text{smt div-add-self1 even-succ-div-two odd-two-times-div-two-succ}$ 
 $\text{ran-sorted-alt})$ 
by  $(\text{smt div-add-self1 even-succ-div-two odd-two-times-div-two-succ}$ 
 $\text{ran-sorted-alt})$ 

```

Next, we show that our postcondition (which was easy to prove) implies the expected properties of the algorithm.

lemma

```

assumes  $\text{bin-search-pre } a \ l \ h \ \text{bin-search-post } a \ l \ h \ x \ i$ 
shows  $\text{bin-search-decide-membership: } x \in a \{l..<h\} \longleftrightarrow (i < h \wedge x =$ 
 $a \ i)$ 
and  $\text{bin-search-leftmost: } x \notin a \{l..<i\}$ 
using assms apply  $-$ 
apply  $(\text{auto simp: bin-search-post-def bin-search-pre-def})$ 
apply  $(\text{smt atLeastLessThan-iff ran-sorted-alt})$ 
done

```

6.4.7 Naive String Search

```

program-spec match-string
assumes  $l1 \leq h1$ 
ensures  $(\forall j \in \{0..<i\}. a \ (l + j) = b \ (l1 + j)) \wedge (i < h1 - l1 \longrightarrow$ 
 $a \ (l + i) \neq b \ (l1 + i))$ 
 $\wedge 0 \leq i \wedge i \leq h1 - l1$ 
defines  $\langle$ 
   $i = 0;$ 
  while  $\langle l1 + i < h1 \wedge a[l + i] == b[l1 + i] \rangle$ 
    @invariant  $\langle (\forall j \in \{0..<i\}. a \ (l + j) = b \ (l1 + j)) \wedge 0 \leq i \wedge i \leq$ 
 $h1 - l1 \rangle$ 
    @variant  $\langle (h1 - (l1 + i)) \rangle$ 
    {
       $i = i + 1$ 
    }
   $\rangle$ 

```

```

}
>
by vcg-cs auto

lemma lan-eq-iff': lan a l1 (l1 + (h - l)) = lan a' l h
   $\longleftrightarrow (\forall i. 0 \leq i \wedge i < h - l \longrightarrow a (l1 + i) = a' (l + i))$  if  $l \leq h$ 
  using that
proof (induction nat (h - l) arbitrary: h)
  case 0
  then show ?case
    by auto
next
  case (Suc x)
  then have *: x = nat (h - 1 - l) l ≤ h - 1
    by auto
  note IH = Suc.hyps(1)[OF *]
  from * have 1:
    lan a l1 (l1 + (h - l)) = lan a l1 (l1 + (h - 1 - l)) @ [a (l1
+ (h - 1 - l))]
    lan a' l h = lan a' l (h - 1) @ [a' (h - 1)]
    by (auto simp: lan-bwd-simp algebra-simps lan-butlast[symmetric])
  from IH * Suc.hyps(2) Suc.prem show ?case
    unfolding 1
    apply auto
    subgoal for i
      by (cases i = h - 1 - l auto)
    done
qed

```

```

program-spec match-string'
  assumes  $l1 \leq h1$ 
  ensures  $i = h1 - l1 \longleftrightarrow lan a l (l + (h1 - l1)) = lan b l1 h1$ 
for  $i h1 l1 l a [] b []$ 
defines  $\langle inline\ match-string \rangle$ 
  by vcg-cs (auto simp: lan-eq-iff')

```

```

program-spec substring
  assumes  $l \leq h \wedge l1 \leq h1$ 
  ensures  $match = 1 \longleftrightarrow (\exists j \in \{l_0..<h_0\}. lan a j (j + (h1 - l1)) = lan b l1 h1)$ 
=  $lan b l1 h1$ 
for  $a [] b []$ 
defines  $\langle$ 
  match = 0;
  while (l < h ∧ match == 0)
   $\langle invariant \langle l_0 \leq l \wedge l \leq h \wedge match \in \{0,1\} \wedge$ 
   $(if\ match = 1$ 
   $then\ lan\ a\ l\ (l + (h1 - l1)) = lan\ b\ l1\ h1 \wedge l < h$ 
   $else\ (\forall j \in \{l_0..<l\}. lan\ a\ j\ (j + (h1 - l1)) \neq lan\ b\ l1\ h1) \rangle,$ 
   $\langle variant \langle (h - l) * (1 - match) \rangle$ 

```

```

{
  inline match-string';
  if (i == h1 - l1) {match = 1}
  else {l = l + 1}
}
>
by vcg-cs auto

```

```

program-spec substring'
  assumes  $l \leq h \wedge l1 \leq h1$ 
  ensures  $match = 1 \iff (\exists j \in \{l_0..h_0-(h1 - l1)\}. \text{ran } a \ j \ (j + (h1 - l1)) = \text{ran } b \ l1 \ h1)$ 
  for  $a[] \ b[]$ 
  defines <
    match = 0;
    if (l + (h1 - l1) ≤ h) {
      h = h - (h1 - l1) + 1;
      inline substring
    }
  >
  by vcg-cs auto

```

```

program-spec substring''
  assumes  $l \leq h \wedge l1 \leq h1$ 
  ensures  $match = 1 \iff (\exists j \in \{l_0..<h_0-(h1 - l1)\}. \text{ran } a \ j \ (j + (h1 - l1)) = \text{ran } b \ l1 \ h1)$ 
  for  $a[] \ b[]$ 
  defines <
    match = 0;
    if (l + (h1 - l1) ≤ h) {
      while (l + (h1 - l1) < h ∧ match == 0)
        @invariant<math>l_0 \leq l \wedge l \leq h - (h1 - l1) \wedge match \in \{0,1\} \wedge
          (if match = 1
            then  $\text{ran } a \ l \ (l + (h1 - l1)) = \text{ran } b \ l1 \ h1 \wedge l < h - (h1 - l1)$ 
            else  $(\forall j \in \{l_0..<l\}. \text{ran } a \ j \ (j + (h1 - l1)) \neq \text{ran } b \ l1 \ h1)$ )
          @variant<math>(h - l) * (1 - match)>
        {
          inline match-string';
          if (i == h1 - l1) {match = 1}
          else {l = l + 1}
        }
    }
  >
  by vcg-cs auto

```

lemma *ran-split*:
 $\text{ran } a \ l \ h = \text{ran } a \ l \ p \ @ \ \text{ran } a \ p \ h$ **if** $l \leq p \leq h$
using that by (*induction p; simp; simp add: ran.simps*)

```

lemma ran-eq-append-iff:
   $ran\ a\ l\ h = as\ @\ bs \longleftrightarrow (\exists\ i.\ l \leq i \wedge i \leq h \wedge as = ran\ a\ l\ i \wedge bs = ran\ a\ i\ h)$  if  $l \leq h$ 
apply safe
subgoal
  using that
proof (induction as arbitrary: l)
  case Nil
  then show ?case
    by auto
next
  case (Cons x as)
  from this(2-) have  $ran\ a\ (l + 1)\ h = as\ @\ bs\ a\ l = x\ l + 1 \leq h$ 
    apply -
    subgoal
      by simp
    subgoal
      by (smt append-Cons list.inject ran.simps ran-append1)
    subgoal
      using add1-zle-eq by fastforce
    done
  from Cons.IH[OF this(1,3)] obtain i
    where  $IH: l + 1 \leq i \wedge i \leq h \wedge as = ran\ a\ (l + 1)\ i \wedge bs = ran\ a\ i\ h$ 
    by auto
  with  $\langle a\ l = x \rangle$  show ?case
    apply (intro exI[where x = i])
    apply auto
    by (smt IH(3) ran-prepend1)
qed
apply (subst ran-split; simp)
done

```

```

lemma ran-split':
   $(\exists j \in \{l..h - (h1 - l1)\}.\ ran\ a\ j\ (j + (h1 - l1)) = ran\ b\ l1\ h1)$ 
   $= (\exists as\ bs.\ ran\ a\ l\ h = as\ @\ ran\ b\ l1\ h1\ @\ bs)$  if  $l \leq h \wedge l1 \leq h1$ 
proof safe
  fix j
  assume  $j: j \in \{l..h - (h1 - l1)\}$  and match:  $ran\ a\ j\ (j + (h1 - l1)) = ran\ b\ l1\ h1$ 
  with  $\langle l1 \leq h1 \rangle$  have  $ran\ a\ l\ h = ran\ a\ l\ j\ @\ ran\ a\ j\ (j + (h1 - l1))\ @\ ran\ a\ (j + (h1 - l1))\ h$ 
    apply (subst ran-split[where p = j], simp, simp)
    apply (subst (2) ran-split[where p = j + (h1 - l1)]; simp)
    done
  then show  $\exists as\ bs.\ ran\ a\ l\ h = as\ @\ ran\ b\ l1\ h1\ @\ bs$ 
    by (auto simp: match)
next
  fix as bs

```

```

assume  $lan\ a\ l\ h = as @ lan\ b\ l1\ h1 @ bs$ 
with that  $lan\text{-}eq\text{-}append\text{-}iff[of\ l\ h\ a\ as\ lan\ b\ l1\ h1 @ bs]$  obtain  $i$ 
where
   $as = lan\ a\ l\ i\ lan\ a\ i\ h = lan\ b\ l1\ h1 @ bs\ l \leq i\ i \leq h$ 
  by auto
with  $lan\text{-}eq\text{-}append\text{-}iff[of\ i\ h\ a\ lan\ b\ l1\ h1\ bs]$  obtain  $j$  where  $j$ :
   $lan\ b\ l1\ h1 = lan\ a\ i\ j\ bs = lan\ a\ j\ h\ i \leq j\ j \leq h$ 
  by auto
moreover have  $j = i + (h1 - l1)$ 
proof -
  have  $length\ (lan\ b\ l1\ h1) = nat\ (h1 - l1)\ length\ (lan\ a\ i\ j) =$ 
 $nat\ (j - i)$ 
  by auto
  with  $j(1,3)$  that show ?thesis
  by auto
qed
ultimately show  $\exists j \in \{l..h - (h1 - l1)\}. lan\ a\ j\ (j + (h1 - l1)) =$ 
 $lan\ b\ l1\ h1$ 
  using  $\langle l \leq i \rangle$  by auto
qed

```

```

program-spec substring-final
assumes  $l \leq h \wedge 0 \leq len$ 
ensures  $match = 1 \iff (\exists as\ bs. lan\ a\ l_0\ h_0 = as @ lan\ b\ 0\ len @ bs)$ 
for  $l\ h\ len\ match\ a[]\ b[]$ 
defines  $\langle l1 = 0; h1 = len; inline\ substring' \rangle$ 
supply  $[simp] = lan\text{-}split'[symmetric]$ 
apply vcg-cs
done

```

6.4.8 Insertion Sort

We first prove the inner loop. The specification here specifies what the algorithm does as closely as possible, such that it becomes easier to prove. In this case, sortedness is not a precondition for the inner loop to move the key element backwards over all greater elements.

```

definition insort-insert-post  $l\ j\ a_0\ a\ i$ 
 $\iff (let\ key = a_0\ j\ in$ 
   $i \in \{l-1..<j\} \quad \text{--- } i \text{ is in range}$ 
   $\text{--- Content of new array}$ 
   $\wedge (\forall k \in \{l..i\}. a\ k = a_0\ k)$ 
   $\wedge a\ (i+1) = key$ 
   $\wedge (\forall k \in \{i+2..j\}. a\ k = a_0\ (k-1))$ 
   $\wedge a = a_0\ on\ \text{---}\{l..j\}$ 
   $\text{--- Placement of } key$ 
   $\wedge (i \geq l \implies a\ i \leq key) \text{ --- Element at } i \text{ smaller than } key, \text{ if it exists}$ 

```

$\wedge (\forall k \in \{i+2..j\}. a\ k > key) \text{ — Elements } \geq i+2 \text{ greater than } key$
 $)$

for $l\ j\ i :: int$ **and** $a_0\ a :: int \Rightarrow int$

program-spec *insort-insert*
assumes $l < j$
ensures *insort-insert-post* $l\ j\ a_0\ a\ i$
defines \langle
 $key = a[j];$
 $i = j - 1;$
while $(i \geq l \wedge a[i] > key)$
 $\text{@variant } \langle i - l + 1 \rangle$
 $\text{@invariant } \langle l - 1 \leq i \wedge i < j$
 $\wedge (\forall k \in \{l..i\}. a\ k = a_0\ k)$
 $\wedge (\forall k \in \{i+2..j\}. a\ k > key \wedge a\ k = a_0\ (k - 1))$
 $\wedge a = a_0 \text{ on } -\{l..j\}$
 \rangle
 $\{$
 $a[i+1] = a[i];$
 $i = i - 1$
 $\};$
 $a[i+1] = key$
 \rangle
unfolding *insort-insert-post-def* *Let-def*
apply *vcg*
 $\text{apply } auto$
by (*smt atLeastAtMost-iff*)

Next, we show that our specification that was easy to prove implies the specification that the outer loop expects:

Invoking *insort-insert* will sort in the element

lemma *insort-insert-sorted*:
assumes $l < j$
assumes *insort-insert-post* $l\ j\ a'\ i'$
assumes *ran-sorted* $a\ l\ j$
shows *ran-sorted* $a'\ l\ (j + 1)$
using *assms* **unfolding** *insort-insert-post-def* *Let-def*
apply *auto*
subgoal
 $\text{by } (smt\ atLeastAtMost-iff\ ran-sorted-alt)$
subgoal
unfolding *ran-sorted-alt* *Ball-def*
apply *auto*
by *smt*
done

Invoking *insort-insert* will only mutate the elements

lemma *insort-insert-ran1*:
assumes $l < j$
assumes *insort-insert-post* $l\ j\ a\ a'\ i$
shows $mset\text{-}ran\ a'\ \{l..j\} = mset\text{-}ran\ a\ \{l..j\}$
proof –
from *assms* **have** *EQS*:
 $a' = a$ on $\{l..i\}$
 $a' (i+1) = a\ j$
 $a' = (a\ o\ (+)(-1))$ on $\{i+2..j\}$
unfolding *insort-insert-post-def eq-on-def Let-def* **by** *auto*

from *assms* **have** $l \leq i+1\ i+1 \leq j$ **unfolding** *insort-insert-post-def Let-def* **by** *auto*

have *ranshift*: $mset\text{-}ran\ (a\ o\ (+)(-1))\ \{i+2..j\} = mset\text{-}ran\ a\ \{i+1..j-1\}$

by (*simp add: mset-ran-shift algebra-simps*)

have $mset\text{-}ran\ a'\ \{l..j\} = mset\text{-}ran\ a'\ \{l..i\} + \{\# a' (i+1) \#\} + mset\text{-}ran\ a'\ \{i+2..j\}$
using $\langle l < j \rangle\ \langle l \leq i+1 \rangle\ \langle i+1 \leq j \rangle$
apply (*simp add: mset-ran-combine*)
by (*auto intro: arg-cong[where f=mset-ran a']*)
also **have** $\dots = mset\text{-}ran\ a\ \{l..i\} + \{\# a\ j \#\} + mset\text{-}ran\ (a\ o\ (+)(-1))\ \{i+2..j\}$
using *EQS(1,3)[THEN mset-ran-cong] EQS(2)* **by** *simp*
also **have** $\dots = mset\text{-}ran\ a\ \{l..i\} + mset\text{-}ran\ a\ \{i+1..j-1\} + \{\# a\ j \#\}$
by (*simp add: mset-ran-shift algebra-simps*)
also **have** $\dots = mset\text{-}ran\ a\ \{l..j\}$
using $\langle l < j \rangle\ \langle l \leq i+1 \rangle\ \langle i+1 \leq j \rangle$
apply (*simp add: mset-ran-combine*)
by (*auto intro: arg-cong[where f=mset-ran a]*)
finally **show** *?thesis* .

qed

The property $\llbracket ?l < ?j; \text{insort-insert-post } ?l\ ?j\ ?a\ ?a'\ ?i \rrbracket \implies mset\text{-}ran\ ?a'\ \{?l..?j\} = mset\text{-}ran\ ?a\ \{?l..?j\}$ extends to the whole array to be sorted

lemma *insort-insert-ran2*:
assumes $l < j < h$
assumes *insort-insert-post* $l\ j\ a\ a'\ i$
shows $mset\text{-}ran\ a'\ \{l..<h\} = mset\text{-}ran\ a\ \{l..<h\}$ (**is** *?thesis1*)
and $a' = a$ on $-\{l..<h\}$ (**is** *?thesis2*)
proof –
from *insort-insert-ran1* *assms* **have** $mset\text{-}ran\ a'\ \{l..j\} = mset\text{-}ran\ a\ \{l..j\}$ **by** *blast*
also **from** $\langle \text{insort-insert-post } l\ j\ a\ a'\ i \rangle$ **have** $a' = a$ on $\{j < .. < h\}$

```

unfolding insort-insert-post-def Let-def by (auto simp: eq-on-def)
hence mset-ran a' {j<..h} = mset-ran a {j<..h} by (rule mset-ran-cong)
finally (mset-ran-combine-eqs) show ?thesis1
  by (simp add: assms ivl-disj-int-two(4) ivl-disj-un-two(4) le-less)

from assms show ?thesis2
  unfolding insort-insert-post-def Let-def eq-on-def
  by auto

```

qed

Finally, we specify and prove correct the outer loop

```

program-spec insort
  assumes l<h
  ensures ran-sorted a l h  $\wedge$  mset-ran a {l..h} = mset-ran a0 {l..h}
 $\wedge$  a=a0 on  $\neg$ {l..h}
  for a[]
  defines  $\langle$ 
    j = l + 1;
    while (j<h)
     $\langle$ 
      @variant  $\langle$ h-j $\rangle$ 
      @invariant  $\langle$ 
        l<j  $\wedge$  j $\leq$ h — j in range
         $\wedge$  ran-sorted a l j — Array is sorted up to j
         $\wedge$  mset-ran a {l..h} = mset-ran a0 {l..h} — Elements in
        range only permuted
         $\wedge$  a=a0 on  $\neg$ {l..h}
       $\rangle$ 
     $\rangle$ 
     $\{$ 
      inline insort-insert;
      j=j+1
     $\}$ 
   $\rangle$ 
  apply vcg-cs
  apply (intro conjI)
  subgoal by (rule insort-insert-sorted)
  subgoal using insort-insert-ran2(1) by auto
  subgoal apply (frule (2) insort-insert-ran2(2)) by (auto simp:
eq-on-def)
  done

```

6.4.9 Quicksort

```

procedure-spec partition-aux(a,l,h,p) returns (a,i)
  assumes l $\leq$ h
  ensures mset-ran a0 {l0..h0} = mset-ran a {l0..h0}
     $\wedge$  ( $\forall j \in \{l_0..i\}. a j < p_0$ )
     $\wedge$  ( $\forall j \in \{i..h_0\}. a j \geq p_0$ )
     $\wedge$  l0 $\leq$ i  $\wedge$  i $\leq$ h0

```

$\wedge a_0 = a \text{ on } -\{l_0..<h_0\}$

```

defines <
  i=l; j=l;
  while (j<h)
    @invariant <
      l≤i ∧ i≤j ∧ j≤h
      ∧ mset-ran a0 {l0..<h0} = mset-ran a {l0..<h0}
      ∧ (∀ k∈{l..i). a k < p)
      ∧ (∀ k∈{i..j). a k ≥ p)
      ∧ (∀ k∈{j..h). a0 k = a k)
      ∧ a0 = a on -{l0..<h0}
    >
    @variant <(h-j)>
  {
    if (a[j]<p) {temp = a[i]; a[i] = a[j]; a[j] = temp; i=i+1};
    j=j+1
  }
  >
supply lran-eq-iff[simp] lran-tail[simp del]
apply vcg-cs
subgoal by (simp add: mset-ran-swap[unfolded swap-def])
subgoal by auto
done

```

```

procedure-spec partition(a,l,h,p) returns (a,i)
assumes l<h
ensures mset-ran a0 {l0..<h0} = mset-ran a {l0..<h0}
      ∧ (∀ j∈{l0..i). a j < a i)
      ∧ (∀ j∈{i..h0). a j ≥ a i)
      ∧ l0≤i ∧ i<h0 ∧ a0 (h0-1) = a i
      ∧ a0 = a on -{l0..<h0}

```

```

defines <
  p = a[h-1];
  (a,i) = partition-aux(a,l,h-1,p);
  a[h-1] = a[i];
  a[i] = p
  >
apply vcg-cs
apply (auto simp: eq-on-def mset-ran-swap[unfolded swap-def] intvs-incdec
intro: mset-ran-combine-eq-diff)
done

```

```

lemma quicksort-sorted-aux:
assumes BOUNDS: l ≤ i < h

```

assumes *LESS*: $\forall j \in \{l..<i\}. a_1 j < a_1 i$
assumes *GEQ*: $\forall j \in \{i..<h\}. a_1 i \leq a_1 j$

assumes *R1*: $mset\text{-}ran\ a_1\ \{l..<i\} = mset\text{-}ran\ a_2\ \{l..<i\}$
assumes *E1*: $a_1 = a_2\ on\ -\ \{l..<i\}$

assumes *SL*: $ran\text{-}sorted\ a_2\ l\ i$

assumes *R2*: $mset\text{-}ran\ a_2\ \{i + 1..<h\} = mset\text{-}ran\ a_3\ \{i + 1..<h\}$
assumes *E2*: $a_2 = a_3\ on\ -\ \{i + 1..<h\}$

assumes *SH*: $ran\text{-}sorted\ a_3\ (i + 1)\ h$

shows $ran\text{-}sorted\ a_3\ l\ h$

proof –

have [*simp*]: $\{l..<i\} \subseteq -\ \{i + 1..<h\}$ **by** *auto*
have [*simp*]: $a_1\ i = a_3\ i$ **using** *E1 E2* **by** (*auto simp: eq-on-def*)

note *X1* = $mset\text{-}ran\text{-}xfer\text{-}pointwise$ [**where** $P = \langle \lambda x. x < p \rangle$ **for** p , *OF R1, simplified*]
note *X2* = $eq\text{-}on\text{-}xfer\text{-}pointwise$ [**where** $P = \langle \lambda x. x < p \rangle$ **for** p , *OF E2, of $\{l..<i\}$, simplified*]
from *LESS* **have** *LESS'*: $\forall j \in \{l..<i\}. a_3 j < a_3 i$
by (*simp add: X1 X2*)

from *GEQ* **have** *GEQ1*: $\forall j \in \{i+1..<h\}. a_1 i \leq a_1 j$ **by** *auto*
have [*simp*]: $\{i + 1..<h\} \subseteq -\ \{l..<i\}$ **by** *auto*
note *X3* = $eq\text{-}on\text{-}xfer\text{-}pointwise$ [**where** $P = \langle \lambda x. x \geq p \rangle$ **for** p , *OF E1, of $\{i+1..<h\}$, simplified*]
note *X4* = $mset\text{-}ran\text{-}xfer\text{-}pointwise$ [**where** $P = \langle \lambda x. x \geq p \rangle$ **for** p , *OF R2, simplified*]
from *GEQ1* **have** *GEQ'*: $\forall j \in \{i+1..<h\}. a_3 i \leq a_3 j$ **by** (*simp add: X3 X4*)

from *SL* $eq\text{-}on\text{-}xfer\text{-}ran\text{-}sorted$ [*OF E2, of $l\ i$*] **have** *SL'*: $ran\text{-}sorted\ a_3\ l\ i$ **by** *simp*

show *?thesis* **using** $combine\text{-}sorted\text{-}pivot$ [*OF BOUNDS SL' SH LESS' GEQ'*].

qed

lemma *quicksort-mset-aux*:

assumes *B*: $l_0 \leq i < h_0$
assumes *R1*: $mset\text{-}ran\ a\ \{l_0..<i\} = mset\text{-}ran\ aa\ \{l_0..<i\}$
assumes *E1*: $a = aa\ on\ -\ \{l_0..<i\}$
assumes *R2*: $mset\text{-}ran\ aa\ \{i + 1..<h_0\} = mset\text{-}ran\ ab\ \{i + 1..<h_0\}$

```

assumes  $E2: aa = ab \text{ on } - \{i + 1..<h_0\}$ 
shows  $mset\text{-}ran\ a\ \{l_0..<h_0\} = mset\text{-}ran\ ab\ \{l_0..<h_0\}$ 
apply (rule trans)
  apply (rule mset-ran-eq-extend[OF R1 E1])
using B apply auto [2]
apply (rule mset-ran-eq-extend[OF R2 E2])
using B apply auto [2]
done

```

```

recursive-spec quicksort( $a, l, h$ ) returns  $a$ 
  assumes True
  ensures  $ran\text{-}sorted\ a\ l_0\ h_0 \wedge mset\text{-}ran\ a_0\ \{l_0..<h_0\} = mset\text{-}ran\ a\ \{l_0..<h_0\} \wedge a_0 = a \text{ on } -\{l_0..<h_0\}$ 
  variant  $h-l$ 
  defines <
    if ( $l < h$ ) {
      ( $a, i$ ) = partition( $a, l, h, a[l]$ );
       $a = rec\ quicksort(a, l, i)$ ;
       $a = rec\ quicksort(a, i+1, h)$ 
    }
  >
  apply (vcg-cs; (intro conjI)?)
  subgoal using quicksort-sorted-aux by metis
  subgoal using quicksort-mset-aux by metis
  subgoal by (smt ComplD ComplI atLeastLessThan-iff eq-on-def)
  subgoal by (auto simp: ran-sorted-def)
  done

```

6.5 Data Refinement

6.5.1 Filtering

```

program-spec array-filter-negative
  assumes  $l \leq h$ 
  ensures  $lran\ a\ l_0\ i = filter\ (\lambda x. x \geq 0)\ (lran\ a_0\ l_0\ h_0)$ 
  defines <
     $i=l; j=l;$ 
    while ( $j < h$ )
      @invariant <
         $l \leq i \wedge i \leq j \wedge j \leq h$ 
         $\wedge lran\ a\ l\ i = filter\ (\lambda x. x \geq 0)\ (lran\ a_0\ l\ j)$ 
         $\wedge lran\ a\ j\ h = lran\ a_0\ j\ h$ 
      >
      @variant  $\langle h-j \rangle$ 
    {
      if ( $a[j] \geq 0$ ) { $a[i] = a[j]; i=i+1$ };
       $j=j+1$ 
    }
  >

```

```

supply lran-eq-iff[simp] lran-tail[simp del]
apply vcg-cs
done

```

6.5.2 Merge Two Sorted Lists

We define the merge function abstractly first, as a functional program on lists.

```

fun merge where
  merge [] ys = ys
| merge xs [] = xs
| merge (x#xs) (y#ys) = (if x<y then x#merge xs (y#ys) else y#merge
(x#xs) ys)

```

lemma *merge-add-simp*[*simp*]: *merge xs [] = xs* **by** (*cases xs*) *auto*

It's straightforward to show that this produces a sorted list with the same elements.

```

lemma merge-sorted:
  assumes sorted xs sorted ys
  shows sorted (merge xs ys) ∧ set (merge xs ys) = set xs ∪ set ys
  using assms
  apply (induction xs ys rule: merge.induct)
  apply auto
done

```

lemma *merge-mset*: *mset (merge xs ys) = mset xs + mset ys*
by (*induction xs ys rule: merge.induct*) *auto*

Next, we prove an equation that characterizes one step of the while loop, on the list level.

```

lemma merge-eq: xs≠[] ∨ ys≠[] ⇒ merge xs ys = (
  if ys=[] ∨ (xs≠[] ∧ hd xs < hd ys) then hd xs # merge (tl xs) ys
  else hd ys # merge xs (tl ys)
  )
by (cases xs; cases ys; simp)

```

We do a first proof that our merge implementation on the arrays and indexes behaves like the functional merge on the corresponding lists.

The annotations use the *lran* function to map from the implementation level to the list level. Moreover, the invariant of the implementation, $l \leq h$, is carried through explicitly.

```

program-spec merge-imp'
  assumes l1 ≤ h1 ∧ l2 ≤ h2
  ensures let ms = lran m 0 j; xs0 = lran a10 l10 h10; ys0 = lran
a20 l20 h20 in

```

```

    j ≥ 0 ∧ ms = merge xs0 ys0
defines ⟨
  j = 0;
  while (l1 ≠ h1 ∨ l2 ≠ h2)
    @variant ⟨h1 + h2 - l1 - l2⟩
    @invariant ⟨let
      xs = lran a1 l1 h1; ys = lran a2 l2 h2; ms = lran m 0 j;
      xs0 = lran a10 l10 h10; ys0 = lran a20 l20 h20
    in
      l1 ≤ h1 ∧ l2 ≤ h2 ∧ 0 ≤ j ∧
      merge xs0 ys0 = ms @ merge xs ys
    ⟩
  {
    if (l2 == h2 ∨ (l1 ≠ h1 ∧ a1[l1] < a2[l2])) {
      m[j] = a1[l1];
      l1 = l1 + 1
    } else {
      m[j] = a2[l2];
      l2 = l2 + 1
    };
    j = j + 1
  }
  ⟩

```

Given the *merge-eq* theorem, which captures the essence of a loop step, and the theorems $?l \leq ?h \implies \text{lran } ?a \ ?l \ (?h + 1) = \text{lran } ?a \ ?l \ ?h @ [?a \ ?h]$, $\text{lran } ?a \ (?l + 1) \ ?h = \text{tl} (\text{lran } ?a \ ?l \ ?h)$, and $?l < ?h \implies \text{hd} (\text{lran } ?a \ ?l \ ?h) = ?a \ ?l$, which convert from the operations on arrays and indexes to operations on lists, the proof is straightforward

```

apply vcg-cs
subgoal apply (subst merge-eq) by auto
subgoal by linarith
subgoal apply (subst merge-eq) by auto
done

```

In a next step, we refine our proof to combine it with the abstract properties we have proved about merge. The program does not change (we simply inline the original one here).

```

procedure-spec merge-imp (a1, l1, h1, a2, l2, h2) returns (m, j)
  assumes l1 ≤ h1 ∧ l2 ≤ h2 ∧ sorted (lran a1 l1 h1) ∧ sorted (lran a2
l2 h2)
  ensures let ms = lran m 0 j in
    j ≥ 0
    ∧ sorted ms
    ∧ mset ms = mset (lran a10 l10 h10) + mset (lran a20 l20 h20)
  for l1 h1 l2 h2 a1[] a2[] m[] j
defines ⟨inline merge-imp'⟩

```

```

apply vcg-cs
apply (auto simp: Let-def merge-mset dest: merge-sorted)
done

thm merge-imp-spec
thm merge-imp-def

lemma [named-ss vcg-bb]:
   $UNIV \cup a = UNIV$ 
   $a \cup UNIV = UNIV$ 
by auto

lemma merge-msets-aux:  $\llbracket l \leq m; m \leq h \rrbracket \implies mset (lran\ a\ l\ m) + mset$ 
 $(lran\ a\ m\ h) = mset (lran\ a\ l\ h)$ 
by (auto simp: mset-lran mset-ran-combine ivl-disj-un-two)

recursive-spec mergesort (a,l,h) returns (b,j)
  assumes  $l \leq h$ 
  ensures  $\langle 0 \leq j \wedge sorted (lran\ b\ 0\ j) \wedge mset (lran\ b\ 0\ j) = mset (lran$ 
 $a_0\ l_0\ h_0) \rangle$ 
  variant  $\langle h-l \rangle$ 
  for  $a[]\ b[]$ 
  defines  $\langle$ 
     $if\ (l==h)\ j=0$ 
     $else\ if\ (l+1==h)\ \{$ 
       $b[0] = a[l];$ 
       $j=1$ 
     $\}$   $else\ \{$ 
       $m = (h+l) / 2;$ 
       $(a1,h1) = rec\ mergesort\ (a,l,m);$ 
       $(a2,h2) = rec\ mergesort\ (a,m,h);$ 
       $(b,j) = merge-imp\ (a1,0,h1,a2,0,h2)$ 
     $\}$ 
   $\rangle$ 
  apply vcg
    apply auto []
    apply (auto simp: lran.simps) []
    apply auto []
    apply auto []
    apply auto []
  apply (auto simp: Let-def merge-msets-aux) []
done
print-theorems

```

6.5.3 Remove Duplicates from Array, using Bitvector Set

We use an array to represent a set of integers.

If we only insert elements in range $\{0..<n\}$, this representation is called bit-vector (storing a single bit per index is enough).

definition *set-of* :: (int \Rightarrow int) \Rightarrow int set **where** *set-of a* \equiv {i. a i \neq 0}

context notes [*simp*] = *set-of-def* **begin**

lemma *set-of-empty*[*simp*]: *set-of* (λ -. 0) = {} **by** *auto*

lemma *set-of-insert*[*simp*]: $x \neq 0 \implies \text{set-of } (a(i:=x)) = \text{insert } i$
(*set-of a*) **by** *auto*

lemma *set-of-remove*[*simp*]: *set-of* ($a(i:=0)$) = *set-of a* - {i} **by**
auto

lemma *set-of-mem*[*simp*]: $i \in \text{set-of } a \iff a \ i \neq 0$ **by** *auto*

end

program-spec *dedup*

assumes $\langle l \leq h \rangle$

ensures $\langle \text{set } (\text{lan } a \ l \ i) = \text{set } (\text{lan } a_0 \ l \ h) \wedge \text{distinct } (\text{lan } a \ l \ i) \rangle$

defines \langle

i=l; j=l;

clear b[];

while (j<h)

$\text{@variant } \langle h-j \rangle$

$\text{@invariant } \langle l \leq i \wedge i \leq j \wedge j \leq h$

$\wedge \text{set } (\text{lan } a \ l \ i) = \text{set } (\text{lan } a_0 \ l \ j)$

$\wedge \text{distinct } (\text{lan } a \ l \ i)$

$\wedge \text{lan } a \ j \ h = \text{lan } a_0 \ j \ h$

$\wedge \text{set-of } b = \text{set } (\text{lan } a \ l \ i)$

\rangle

{

if ($b[a[j]] == 0$) {

a[i] = a[j]; i=i+1; b[a[j]] = 1

};

j=j+1

}

\rangle

apply *vcg-cs*

apply (*auto simp: lan-eq-iff lan-upd-inside intro: arg-cong*[**where**
f=tl]) []

apply (*auto simp: lan-eq-iff*) []

done

procedure-spec *bv-init* () **returns** *b*

assumes *True* **ensures** $\langle \text{set-of } b = \{\} \rangle$

```

defines  $\langle \text{clear } b[] \rangle$ 
by vcg-cs

procedure-spec bv-insert (x, b) returns b
  assumes True ensures  $\langle \text{set-of } b = \text{insert } x_0 \text{ (set-of } b_0) \rangle$ 
  defines  $\langle b[x] = 1 \rangle$ 
  by vcg-cs

procedure-spec bv-remove (x, b) returns b
  assumes True ensures  $\langle \text{set-of } b = \text{set-of } b_0 - \{x_0\} \rangle$ 
  defines  $\langle b[x] = 0 \rangle$ 
  by vcg-cs

procedure-spec bv-elim (x, b) returns r
  assumes True ensures  $\langle r \neq 0 \iff x_0 \in \text{set-of } b_0 \rangle$ 
  defines  $\langle r = b[x] \rangle$ 
  by vcg-cs

procedure-spec dedup' (a, l, h) returns (a, l, i)
  assumes  $\langle l \leq h \rangle$  ensures  $\langle \text{set } (\text{ran } a \text{ } l \text{ } i) = \text{set } (\text{ran } a_0 \text{ } l_0 \text{ } h_0) \wedge$ 
  distinct (ran a l i)  $\rangle$ 
  for b[]
  defines  $\langle$ 
    b = bv-init();

    i=l; j=l;

    while (j<h)
      @variant  $\langle h-j \rangle$ 
      @invariant  $\langle l \leq i \wedge i \leq j \wedge j \leq h$ 
         $\wedge \text{set } (\text{ran } a \text{ } l \text{ } i) = \text{set } (\text{ran } a_0 \text{ } l \text{ } j)$ 
         $\wedge \text{distinct } (\text{ran } a \text{ } l \text{ } i)$ 
         $\wedge \text{ran } a \text{ } j \text{ } h = \text{ran } a_0 \text{ } j \text{ } h$ 
         $\wedge \text{set-of } b = \text{set } (\text{ran } a \text{ } l \text{ } i)$ 
       $\rangle$ 
      {
        mem = bv-elim (a[j], b);
        if (mem == 0) {
          a[i] = a[j]; i=i+1; b = bv-insert(a[j], b)
        };
        j=j+1
      }
     $\rangle$ 
  apply vcg-cs
  apply (auto simp: ran-eq-iff ran-upd-inside intro: arg-cong[where
f=tl])
  done

```

6.6 Recursion

6.6.1 Recursive Fibonacci

```
recursive-spec fib-imp (i) returns r assumes True ensures  $\langle r = \text{fib } i_0 \rangle$  variant  $\langle i \rangle$   
defines  $\langle$   
  if ( $i \leq 0$ )  $r = 0$   
  else if ( $i = 1$ )  $r = 1$   
  else {  
     $r1 = \text{rec fib-imp } (i-2);$   
     $r2 = \text{rec fib-imp } (i-1);$   
     $r = r1 + r2$   
  }  
 $\rangle$   
by vcg-cs
```

6.6.2 Homeier's Cycling Termination

A contrived example from Homeier's thesis. Only the termination proof is done.

```
recursive-spec  
pedal (n, m) returns () assumes  $\langle n \geq 0 \wedge m \geq 0 \rangle$  ensures True variant  $\langle n+m \rangle$   
defines  $\langle$   
  if ( $n \neq 0 \wedge m \neq 0$ ) {  
     $G = G + m;$   
    if ( $n < m$ ) rec coast ( $n-1, m-1$ ) else rec pedal( $n-1, m$ )  
  }  
 $\rangle$   
and  
coast (n, m) returns () assumes  $\langle n \geq 0 \wedge m \geq 0 \rangle$  ensures True variant  $\langle n+m+1 \rangle$   
defines  $\langle$   
   $G = G + n;$   
  if ( $n < m$ ) rec coast ( $n, m-1$ ) else rec pedal ( $n, m$ )  
 $\rangle$   
by vcg-cs
```

6.6.3 Ackermann

```
fun ack :: nat  $\Rightarrow$  nat  $\Rightarrow$  nat where  
  ack 0 n =  $n+1$   
| ack m 0 = ack ( $m-1$ ) 1  
| ack m n = ack ( $m-1$ ) (ack m ( $n-1$ ))
```

```
lemma ack-add-simps[simp]:  
   $m \neq 0 \implies \text{ack } m \ 0 = \text{ack } (m-1) \ 1$   
   $\llbracket m \neq 0; n \neq 0 \rrbracket \implies \text{ack } m \ n = \text{ack } (m-1) (\text{ack } m \ (n-1))$   
subgoal by (cases m) auto
```

```

subgoal by (cases (m,n) rule: ack.cases) (auto)
done

```

```

recursive-spec relation less-than <*lex*> less-than
ack-imp (m,n) returns r
  assumes  $m \geq 0 \wedge n \geq 0$  ensures  $r = \text{int } (\text{ack } (\text{nat } m_0) (\text{nat } n_0))$ 
  variant (nat m, nat n)
  defines <
    if (m==0) r = n+1
    else if (n==0) r = rec ack-imp (m-1,1)
    else {
      t = rec ack-imp (m,n-1);
      r = rec ack-imp (m-1,t)
    }
  >
supply nat-distrib[simp]
by vcg-cs

```

6.6.4 McCarthy's 91 Function

A standard benchmark for verification of recursive functions. We use Homeier's version with a global variable.

```

recursive-spec p91(y) assumes True ensures if  $100 < y_0$  then  $G = y_0 - 10$  else  $G = 91$  variant  $101 - y$ 
for G
  defines <
    if (100 < y)  $G = y - 10$ 
    else {
      rec p91 (y+11);
      rec p91 (G)
    }
  >
  apply vcg-cs
  apply (auto split: if-splits)
done

```

6.6.5 Odd/Even

```

recursive-spec
odd-imp (a) returns b
  assumes True
  ensures  $b \neq 0 \iff \text{odd } a_0$ 
  variant |a|
  defines <
    if (a==0) b=0
    else if (a<0) b = rec even-imp (a+1)
    else b = rec even-imp (a-1)
  >
and

```

```

even-imp (a) returns b
  assumes True
  ensures  $b \neq 0 \longleftrightarrow \text{even } a_0$ 
  variant |a|
  defines <
    if (a==0) b=1
    else if (a<0) b = rec odd-imp (a+1)
    else b = rec odd-imp (a-1)
  >
  apply vcg
    apply auto
done

```

thm *even-imp-spec*

6.6.6 Pandya and Joseph's Product Producers

Again, taking the version from Homeier's thesis, but with a modification to also terminate for negative y .

```

recursive-spec relation <measure nat <*lex*> less-than>
  product () assumes True ensures <math>GZ = GZ_0 + GX_0 * GY_0</math> variant (|GY|, 1::nat)
  for GX GY GZ
  defines
  <
    e = even-imp (GY);
    if (e≠0) rec evenproduct() else rec oddproduct()
  >
  and
  oddproduct() assumes <math>\text{odd } GY</math> ensures <math>GZ = GZ_0 + GX_0 * GY_0</math>
  variant (|GY|, 0::nat)
  for GX GY GZ
  defines
  <
    if (GY<0) {
      GY = GY + 1;
      GZ = GZ - GX
    } else {
      GY = GY - 1;
      GZ = GZ + GX
    };
    rec evenproduct()
  >
  and
  evenproduct() assumes <math>\text{even } GY</math> ensures <math>GZ = GZ_0 + GX_0 * GY_0</math>
  variant (|GY|, 0::nat)
  for GX GY GZ
  defines
  <

```

```

    if (GY≠0) {
      GX = 2*GX;
      GY = GY / 2;
      rec product()
    }
  }
}
apply vcg-cs
  apply (auto simp: algebra-simps)
  apply presburger+
done

```

6.7 Graph Algorithms

6.7.1 DFS

A graph is stored as an array of integers. Each node is an index into this array, pointing to a size-prefixed list of successors.

Example for node i , which has successors $s1 \dots sn$:

```

  Indexes: ... | i | i+1 | ... | i+n | ...
  Data:    ... | n | s1 | ... | sn | ...

```

definition *succs* where

succs $a \ i \equiv a \ \{i+1..<a \ i\}$ for $a :: int \Rightarrow int$

definition *Edges* where

Edges $a \equiv \{(i, j). j \in \text{succs } a \ i\}$

procedure-spec *push'* ($x, \text{stack}, \text{ptr}$) **returns** (stack, ptr)

assumes $\text{ptr} \geq 0$ **ensures** $\langle \text{ran } \text{stack} \ 0 \ \text{ptr} = \text{ran } \text{stack}_0 \ 0 \ \text{ptr}_0 \ @$
 $[x_0] \wedge \text{ptr} = \text{ptr}_0 + 1 \rangle$

defines $\langle \text{stack}[\text{ptr}] = x; \text{ptr} = \text{ptr} + 1 \rangle$

by *vcg-cs*

procedure-spec *push* ($x, \text{stack}, \text{ptr}$) **returns** (stack, ptr)

assumes $\text{ptr} \geq 0$ **ensures** $\langle \text{stack} \ \{0..<\text{ptr}\} = \{x_0\} \cup \text{stack}_0 \ \{$
 $\{0..<\text{ptr}_0\} \wedge \text{ptr} = \text{ptr}_0 + 1 \rangle$

for $\text{stack}[]$

defines $\langle \text{stack}[\text{ptr}] = x; \text{ptr} = \text{ptr} + 1 \rangle$

by *vcg-cs* (*auto simp: fun-upd-image*)

program-spec *get-succs*

assumes $j \leq \text{stop} \wedge \text{stop} = a \ (j - 1) \wedge 0 \leq i$

ensures

$\text{stack} \ \{0..<i\} = \{x. (j_0 - 1, x) \in \text{Edges } a \wedge x \notin \text{set-of visited}\}$
 $\cup \text{stack}_0 \ \{0..<i_0\}$

$\wedge i \geq i_0$

for $i \ j \ \text{stop} \ \text{stack}[] \ a[] \ \text{visited}[]$

defines

```

<
  while (j < stop)
    @invariant <stack ‘ {0..<i} = {x. x ∈ a ‘ {j0..<j} ∧ x ∉ set-of
visited} ∪ stack0 ‘ {0..<i0}
      ∧ j ≤ stop ∧ i0 ≤ i ∧ j0 ≤ j
    >
    @variant <(stop - j)>
    {
      succ = a[j];
      is-elem = bv-elem(succ,visited);
      if (is-elem == 0) {
        (stack, i) = push (succ, stack, i)
      };
      j = j + 1
    }
  >
  by vcg-cs (auto simp: intvs-incr-h Edges-def succs-def)

```

```

procedure-spec pop (stack, ptr) returns (x, ptr)
assumes ptr ≥ 1 ensures <stack0 ‘ {0..<ptr0} = stack0 ‘ {0..<ptr}
∪ {x} ∧ ptr0 = ptr + 1>
for stack[]
defines <ptr = ptr - 1; x = stack[ptr]>
by vcg-cs (simp add: intvs-upper-decr)

```

```

procedure-spec stack-init () returns i
assumes True ensures <i = 0>
defines <i = 0>
by vcg-cs

```

```

lemma Edges-empty:
  Edges a “ {i} = {} if i + 1 ≥ a i
using that unfolding Edges-def succs-def by auto

```

This is one of the main insights of the algorithm: if a set of visited states is closed w.r.t. to the edge relation, then it is guaranteed to contain all the states that are reachable from any state within the set.

```

lemma reachability-invariant:
assumes reachable: (s, x) ∈ (Edges a)*
and closed: ∀ v ∈ visited. Edges a “ {v} ⊆ visited
and start: s ∈ visited
shows x ∈ visited
using reachable start closed by induction auto

```

```

program-spec (partial) dfs
assumes 0 ≤ x ∧ 0 ≤ s
ensures b = 1 ↔ x ∈ (Edges a)* “ {s} defines <
b = 0;

```

```

clear stack[];
i = stack-init();
(stack, i) = push (s, stack, i);
clear visited[];
while (b == 0 ∧ i ≠ 0)
  @invariant (0 ≤ i ∧ (s ∈ stack ‘ {0..<i} ∨ s ∈ set-of visited) ∧
(b = 0 ∨ b = 1) ∧ (
  if b = 0 then
    stack ‘ {0..<i} ⊆ (Edges a)* ‘ {s}
    ∧ (∀ v ∈ set-of visited. (Edges a) ‘ {v} ⊆ set-of visited ∪ stack ‘
{0..<i})
    ∧ (x ∉ set-of visited)
  else x ∈ (Edges a)* ‘ {s})
  )
  {
    (next, i) = pop(stack, i); — Take the top most element from the
stack.
    visited = bv-insert(next, visited); — Mark it as visited,
    if (next == x) {
      b = 1 — If it is the target, we are done.
    } else {
      — Else, put its successors on the stack if they are not yet visited.
      stop = a[next];
      j = next + 1;
      if (j ≤ stop) {
        inline get-succs
      }
    }
  }
}
}
}
>
apply vcg-cs
subgoal by (auto simp: set-of-def)
subgoal using intvs-lower-incr by (auto simp: Edges-empty)
subgoal by auto (fastforce simp: set-of-def dest!: reachability-invariant)
done

```

Assuming that the input graph is finite, we can also prove that the algorithm terminates. We will thus use an *Isabelle context* to fix a certain finite graph and a start state:

```

context
  fixes start :: int and edges
  assumes finite-graph[intro!]: finite ((Edges edges)* ‘ {start})
begin

```

lemma *sub-insert-same-iff*: $s \subset \text{insert } x \ s \iff x \notin s$ **by** *auto*

```

program-spec dfs1
  assumes  $0 \leq x \wedge 0 \leq s \wedge \text{start} = s \wedge \text{edges} = a$ 
  ensures  $b = 1 \iff x \in (\text{Edges } a)^* \ ‘ \{s\}$ 

```

```

for visited[]
defines
<
  b = 0;
  — i will point to the next free space in the stack (i.e. it is the size
of the stack)
  i = 1;
  — Initially, we put s on the stack.
  stack[0] = s;
  visited = bv-init();
  while (b == 0 ∧ i ≠ 0)
    @invariant <
      0 ≤ i ∧ (s ∈ stack ‘ {0..i} ∨ s ∈ set-of visited) ∧ (b = 0 ∨ b =
1) ∧
      set-of visited ⊆ (Edges edges)* “ {start} ∧ (
      if b = 0 then
        stack ‘ {0..i} ⊆ (Edges a)* “ {s}
        ∧ (∀ v ∈ set-of visited. (Edges a) “ {v} ⊆ set-of visited ∪ stack ‘
{0..i})
        ∧ (x ∉ set-of visited)
        else x ∈ (Edges a)* “ {s}
      )
    >
    @relation <finite-psupset ((Edges edges)* “ {start}) <*lex*> less-than>
    @variant <( set-of visited, nat i )>
    {
      — Take the top most element from the stack.
      (next, i) = pop(stack, i);
      if (next == x) {
        — If it is the target, we are done.
        visited = bv-insert(next, visited);
        b = 1
      } else {
        is-elem = bv-elem(next, visited);
        if (is-elem == 0) {
          visited = bv-insert(next, visited);
          — Else, put its successors on the stack if they are not yet visited.
          stop = a[next];
          j = next + 1;
          if (j ≤ stop) {
            inline get-succs
          }
        }
      }
    }
  }
>
apply vcg-cs
subgoal by auto
subgoal by (auto simp add: image-constant-conv)
subgoal by (clarsimp simp: finite-psupset-def sub-insert-same-iff)

```

```
subgoal by (auto simp: set-of-def)
subgoal by (clarsimp simp: finite-psupset-def sub-insert-same-iff)
subgoal by (auto simp: Edges-empty)
subgoal by (clarsimp simp: finite-psupset-def sub-insert-same-iff)
subgoal by (auto simp: set-of-def)
subgoal by auto (fastforce simp: set-of-def dest!: reachability-invariant)
done
```

end

end