

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	9
2.7.1	Equivalence to Big-Step Semantics	11
2.8	Weakest Precondition	12
2.8.1	Basic Properties	12
2.8.2	Unfold Rules	13
2.8.3	Weakest precondition and Program Equivalence	14
2.8.4	While Loops and Weakest Precondition	14
2.9	Invariants for While-Loops	15
2.9.1	Partial Correctness	15
2.9.2	Total Correctness	15
2.9.3	Standard Forms of While Rules	16
2.10	Modularity of Programs	16
2.11	Strongest Postcondition	17
2.12	Hoare-Triples	17
2.12.1	Sets of Hoare-Triples	18
2.12.2	Deriving Parameter Frame Adjustment Rules	18
2.12.3	Proof for Recursive Specifications	19
2.13	Completeness of While-Rule	20
3	Annotated Syntax	21
3.1	Annotations	21
3.2	Hoare-Triples for Annotated Commands	21
4	Quickstart Guide	22
4.1	Introductory Examples	23
4.1.1	Variant and Invariant Annotations	24
4.1.2	Recursive Procedures	24
4.2	The VCG	25
4.3	Advanced Features	25
4.3.1	Custom Termination Relations	25
4.3.2	Partial Correctness	26
4.3.3	Arrays	26
4.4	Proving Techniques	27
4.4.1	Auxiliary Lemmas	27
4.4.2	Inlining	27
4.4.3	Functional Refinement	28
4.4.4	Data Refinement	28
4.5	Troubleshooting	28
4.5.1	Invalid Variables in Annotations	28
4.5.2	Wrong Annotations	29
4.5.3	Calls to Undefined Procedures	29
4.6	Missing Features	29
4.6.1	Elaborate Warnings and Errors	29
4.6.2	Static Type Checking	29

4.6.3	Structure Types	30
4.6.4	Function Calls as Expressions	30
4.6.5	Ghost Variables	30
4.6.6	Concurrency	30
4.6.7	Pointers and Memory	31
5	Introduction to IMP2-VCG, based on IMP	31
5.1	Fancy Syntax	31
5.2	Operators and Arrays	32
5.3	Local and Global Variables	32
5.3.1	Parameter Passing	32
5.4	Recursive procedures	33
5.4.1	Procedure Scope	33
5.4.2	Syntactic sugar for procedure call with parameters	33
5.5	More Readable VCs	33
5.6	Specification Commands	33
6	Examples	35
6.1	Common Loop Patterns	35
6.1.1	Count Up	35
6.1.2	Count down	37
6.1.3	Approximate from Below	38
6.1.4	Bisection	38
6.2	Debugging	39
6.2.1	Testing Programs	39
6.3	More Numeric Algorithms	39
6.3.1	Euclid's Algorithm (with subtraction)	39
6.3.2	Euclid's Algorithm (with mod)	40
6.3.3	Extended Euclid's Algorithm	40
6.3.4	Exponentiation by Squaring	42
6.3.5	Power-Tower of 2s	43
6.4	Array Algorithms	44
6.4.1	Summation	44
6.4.2	Finding Least Index of Element	44
6.4.3	Check for Sortedness	44
6.4.4	Find Equilibrium Index	45
6.4.5	Rotate Right	46
6.4.6	Binary Search, Leftmost Element	46
6.4.7	Naive String Search	47
6.4.8	Insertion Sort	49
6.4.9	Quicksort	51
6.5	Data Refinement	53
6.5.1	Filtering	53
6.5.2	Merge Two Sorted Lists	53
6.5.3	Remove Duplicates from Array, using Bitvector Set	56
6.6	Recursion	58
6.6.1	Recursive Fibonacci	58
6.6.2	Homeier's Cycling Termination	58
6.6.3	Ackermann	58

6.6.4	McCarthy's 91 Function	59
6.6.5	Odd/Even	59
6.6.6	Pandya and Joseph's Product Producers	60
6.7	Graph Algorithms	61
6.7.1	DFS	61

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 ⇒ int aexp
| Binop int ⇒ int ⇒ 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 ⇒ bool ⇒ bool bexp bexp
| Cmpop int ⇒ int ⇒ 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
 | *AssignIdx vname aexp aexp* — Assign to index in array
 | *ArrayCpy vname vname* — Copy whole array
 | *ArrayClear vname* — Clear array
 | *Assign-Locals vname ⇒ val* — Internal: Assign all local variables simultaneously

— Block
 | *Seq com com* — Sequential composition
 | *If bexp com com* — Conditional
 | *While bexp com* — While-loop
 | *Scope com* — Local variable scope

— Procedure
 | *PCall pname* — Procedure call
 | *PScope pname ↪ com com* — 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 *AssignIdx* ($\langle \text{-}[-] ::= \rightarrow [1000, 0, 61] 61 \rangle$)

notation *ArrayCpy* ($\langle \text{-}[] ::= \rightarrow [1000, 1000] 61 \rangle$)

notation *ArrayClear* ($\langle \text{CLEAR } \text{-}[] \rangle [1000] 61$)

notation *Seq* ($\langle \text{-};/ \rightarrow [61, 60] 60 \rangle$)

notation *If* ($\langle \text{(IF } \text{-/ THEN } \text{-/ ELSE } \text{-)} \rangle [0, 0, 61] 61$)

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

notation *Scope* ($\langle \text{SCOPE } \rightarrow [61] 61 \rangle$)

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 $V x \equiv \text{Vidx } x (N 0)$

abbreviation *Assign* ($\langle \text{-} ::= \rightarrow [1000, 61] 61 \rangle$)

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  $\equiv$  -Update  $\langle\rangle$  ms
```

```
-State (-updbinds b bs)  $\leq$  -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 ( $\langle\langle-|- \rangle\rangle$  [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:  $\langle s|s \rangle = s$   $\langle$ proof $\rangle$ 
```

```
lemma combine-nest:
```

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

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

```
 $\langle$ proof $\rangle$ 
```

```
lemma combine-query:
```

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

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

```
 $\langle$ proof $\rangle$ 
```

lemma *combine-upd*:
is-local $x \implies \langle s|t \rangle(x:=v) = \langle s(x:=v)|t \rangle$
is-global $x \implies \langle s|t \rangle(x:=v) = \langle s|t(x:=v) \rangle$
 $\langle \text{proof} \rangle$

lemma *combine-cases*[*cases type*]:
obtains $l\ g$ **where** $s = \langle l|g \rangle$
 $\langle \text{proof} \rangle$

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* $x\ i$) $s = s\ x\ (\text{aval}\ i\ s)$
| *aval* (*Unop* $f\ a_1$) $s = f\ (\text{aval}\ a_1\ s)$
| *aval* (*Binop* $f\ a_1\ a_2$) $s = f\ (\text{aval}\ a_1\ s)\ (\text{aval}\ a_2\ s)$

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) $s = (\neg\ \text{bval}\ b\ s)$
| *bval* (*BBinop* $f\ b_1\ b_2$) $s = f\ (\text{bval}\ b_1\ s)\ (\text{bval}\ b_2\ s)$
| *bval* (*Cmpop* $f\ a_1\ a_2$) $s = f\ (\text{aval}\ a_1\ s)\ (\text{aval}\ a_2\ s)$

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*
 $(\langle \cdot : - \Rightarrow \cdot \rangle [1000, 55, 55] 55)$
where
— *No-Op*
Skip: $\pi : (\text{SKIP}, s) \Rightarrow s$
— *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'*: $s' = s(x := (s \ x)(\text{aval } i \ s := \text{aval } a \ s)) \Longrightarrow \pi:(x[i] ::= a, s) \Rightarrow s' \langle \text{proof} \rangle$

lemma *ArrayCpy'*: $s' = s(x := (s \ y)) \Longrightarrow \pi:(x[] ::= y, s) \Rightarrow s' \langle \text{proof} \rangle$

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

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

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;;

$WHILE\ Cmpop\ (\lambda x\ y.\ y < x)\ (V\ 'n')\ (N\ 0)\ DO\ (
'a'' ::= Binop\ (+)\ (V\ 'a'')\ (V\ 'a'');;
'n'' ::= Binop\ (-)\ (V\ 'n'')\ (N\ 1)
), <'n'' := (\lambda -. 5)> \Rightarrow ?s
\langle proof \rangle$

2.5 Command Equivalence

Two commands are equivalent if they have the same semantics.

definition

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

lemma *equivI[intro?]*: \llbracket

$\bigwedge s\ t\ \pi.\ \pi:(c,s) \Rightarrow t \Longrightarrow \pi:(c',s) \Rightarrow t;$
 $\bigwedge s\ t\ \pi.\ \pi:(c',s) \Rightarrow t \Longrightarrow \pi:(c,s) \Rightarrow t\rrbracket$
 $\Longrightarrow c \sim c'$
 $\langle proof \rangle$

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

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

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

lemma *equiv-sym[sym]*: $(c \sim c') \Longrightarrow (c' \sim c)$
 $\langle proof \rangle$

lemma *equiv-trans[trans]*: $c \sim c' \Longrightarrow c' \sim c'' \Longrightarrow c \sim c''$
 $\langle proof \rangle$

2.5.1 Basic Equivalences

lemma *while-unfold*:

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

lemma *triv-if*:

$(IF\ b\ THEN\ c\ ELSE\ c) \sim c$
 $\langle proof \rangle$

lemma *commute-if*:

$(IF\ b1\ THEN\ (IF\ b2\ THEN\ c11\ ELSE\ c12)\ ELSE\ c2)$
 \sim
 $(IF\ b2\ THEN\ (IF\ b1\ THEN\ c11\ ELSE\ c2)\ ELSE\ (IF\ b1\ THEN\ c12\ ELSE\ c2))$
 $\langle proof \rangle$

lemma *sim-while-cong-aux*:

$\llbracket \pi:(WHILE\ b\ DO\ c,s) \Rightarrow t; \text{bval } b = \text{bval } b'; c \sim c' \rrbracket \Longrightarrow \pi:(WHILE\ b'\ DO\ c',s) \Rightarrow t$
 ⟨proof⟩

lemma *sim-while-cong*: $\text{bval } b = \text{bval } b' \Longrightarrow c \sim c' \Longrightarrow WHILE\ b\ DO\ c \sim WHILE\ b'\ DO\ c'$
 ⟨proof⟩

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⟩

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*, $s(x := (s\ x)(\text{aval } i\ s := \text{aval } a\ s))$)
 | *small-step* π ($x[] ::= y, s$) = *Some* (*SKIP*, $s(x := s\ y)$)
 | *small-step* π (*CLEAR* $x[], s$) = *Some* (*SKIP*, $s(x := (\lambda-. 0))$)
 | *small-step* π (*Assign-Locals* l, s) = *Some* (*SKIP*, $\langle l | s \rangle$)
 | *small-step* π (*SKIP*;; c, s) = *Some* (c, s)
 | *small-step* π ($c_1;; c_2, s$) = (*case* *small-step* π (c_1, s) *of* *Some* (c_1', s')
 \Rightarrow *Some* ($c_1';; c_2, s'$) | - \Rightarrow *None*)
 | *small-step* π (*IF* b *THEN* c_1 *ELSE* c_2, s) = *Some* (*if* *bval* $b\ s$ *then*
 (c_1, s) *else* (c_2, s))
 | *small-step* π (*SCOPE* c, s) = *Some* ($c;; \text{Assign-Locals } s, \langle \langle \rangle | s \rangle$)

$| \text{small-step } \pi (\text{WHILE } b \text{ DO } c, s) = \text{Some } (\text{IF } b \text{ THEN } c;; \text{WHILE } b \text{ DO } c \text{ ELSE SKIP}, s)$
 $| \text{small-step } \pi (\text{PCall } p, s) = (\text{case } \pi p \text{ of Some } c \Rightarrow \text{Some } (c, s) \mid - \Rightarrow \text{None})$
 $| \text{small-step } \pi (\text{PScope } \pi' \text{ SKIP}, s) = \text{Some } (\text{SKIP}, s)$
 $| \text{small-step } \pi (\text{PScope } \pi' c, s) = (\text{case small-step } \pi' (c, s) \text{ of Some } (c', s') \Rightarrow \text{Some } (\text{PScope } \pi' c', s') \mid - \Rightarrow \text{None})$
 $| \text{small-step } \pi (\text{SKIP}, s) = \text{Some } (\text{SKIP}, s)$

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**
 $[simp]: \text{small-steps } \pi \text{ cs } (\text{Some } cs) \Rightarrow \text{small-steps } \pi \text{ cs } (\text{Some } cs)$
 $| \llbracket \text{small-step } \pi \text{ cs} = \text{None} \rrbracket \Longrightarrow \text{small-steps } \pi \text{ cs } \text{None}$
 $| \llbracket \text{small-step } \pi \text{ cs} = \text{Some } cs_1; \text{small-steps } \pi \text{ cs}_1 \text{ cs}_2 \rrbracket \Longrightarrow \text{small-steps } \pi \text{ cs } cs_2$

lemma *small-steps-append*: $\text{small-steps } \pi \text{ cs}_1 (\text{Some } cs_2) \Longrightarrow \text{small-steps } \pi \text{ cs}_2 \text{ cs}_3 \Longrightarrow \text{small-steps } \pi \text{ cs}_1 \text{ cs}_3$
 $\langle \text{proof} \rangle$

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*: $\text{small-step } \pi \text{ cs}_1 = \text{Some } cs_2 \Longrightarrow \pi: cs_2 \Rightarrow s_3 \Longrightarrow \pi: cs_1 \Rightarrow s_3$
 $\langle \text{proof} \rangle$

lemma *small-big-append*: $\text{small-steps } \pi \text{ cs}_1 (\text{Some } cs_2) \Longrightarrow \pi: cs_2 \Rightarrow s_3 \Longrightarrow \pi: cs_1 \Rightarrow s_3$
 $\langle \text{proof} \rangle$

lemma *small-imp-big*:
assumes $\text{small-steps } \pi \text{ cs}_1 (\text{Some } (\text{SKIP}, s_2))$
shows $\pi: cs_1 \Rightarrow s_2$
 $\langle \text{proof} \rangle$

lemma *small-steps-skip-term* $[simp]$: $\text{small-steps } \pi (\text{SKIP}, s) \text{ cs}' \longleftrightarrow \text{cs}' = \text{Some } (\text{SKIP}, s)$
 $\langle \text{proof} \rangle$

lemma *small-seq*: $\llbracket c \neq \text{SKIP}; \text{small-step } \pi (c, s) = \text{Some } (c', s') \rrbracket \Longrightarrow \text{small-step } \pi (c;; cx, s) = \text{Some } (c';; cx, s')$
 $\langle \text{proof} \rangle$

lemma *small-Seq*: $\llbracket \text{small-steps } \pi (c,s) (\text{Some } (c',s')) \rrbracket \implies \text{small-steps } \pi (c;;cx,s) (\text{Some } (c';;cx,s'))$
 $\langle \text{proof} \rangle$

lemma *small-pscope*:
 $\llbracket c \neq \text{SKIP}; \text{small-step } \pi' (c,s) = \text{Some } (c',s') \rrbracket \implies \text{small-step } \pi (P\text{Scope } \pi' c,s) = \text{Some } (P\text{Scope } \pi' c',s')$
 $\langle \text{proof} \rangle$

lemma *small-pscope*:
 $\text{small-steps } \pi' (c, s) (\text{Some } (c', s')) \implies \text{small-steps } \pi (P\text{Scope } \pi' c, s) (\text{Some } (P\text{Scope } \pi' c', s'))$
 $\langle \text{proof} \rangle$

lemma *big-imp-small*:
assumes $\pi: cs \Rightarrow t$
shows $\text{small-steps } \pi cs (\text{Some } (\text{SKIP}, t))$
 $\langle \text{proof} \rangle$

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*: $\pi: cs \Rightarrow t \iff \text{small-steps } \pi cs (\text{Some } (\text{SKIP}, t))$
 $\langle \text{proof} \rangle$

lemma *small-steps-determ*:
assumes $\text{small-steps } \pi cs \text{ None}$
shows $\neg \text{small-steps } \pi cs (\text{Some } (\text{SKIP}, t))$
 $\langle \text{proof} \rangle$

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

corollary *small-imp-big-fail*:
assumes $\text{small-steps } \pi cs \text{ None}$
shows $\nexists t. \pi: cs \Rightarrow t$
 $\langle \text{proof} \rangle$

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

<proof>

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

lemma $wp\text{-equiv}$: $c \sim c' \Longrightarrow wp\ c = wp\ c'$ *<proof>*

lemma $wp\text{-conseq}$: $wp\ c\ P\ s \Longrightarrow \llbracket \wedge s. P\ s \Longrightarrow Q\ s \rrbracket \Longrightarrow wp\ c\ Q\ s$ *<proof>*

lemma $wlp\text{-equiv}$: $c \sim c' \Longrightarrow wlp\ c = wlp\ c'$ *<proof>*

lemma $wlp\text{-conseq}$: $wlp\ c\ P\ s \Longrightarrow \llbracket \wedge s. P\ s \Longrightarrow Q\ s \rrbracket \Longrightarrow wlp\ c\ Q\ s$ *<proof>*

2.8.2 Unfold Rules

lemma $wp\text{-skip}\text{-eq}$: $wp\ SKIP\ Q\ s = Q\ s$ *<proof>*

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

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

lemma $wp\text{-arrayinit}\text{-eq}$: $wp\ (CLEAR\ x[])\ Q\ s = Q\ (s(x:=s\ (\lambda_.\ 0)))$ *<proof>*

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

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

lemma $wp\text{-if}\text{-eq}$: $wp\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ Q\ s = (if\ bval\ b\ s\ then\ wp\ c_1\ Q\ s\ else\ wp\ c_2\ Q\ s)$ *<proof>*

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

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

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

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

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

lemma $wlp\text{-skip}\text{-eq}$: $wlp \text{ } SKIP \text{ } Q \text{ } s = Q \text{ } s$ $\langle proof \rangle$

lemma $wlp\text{-assign}\text{-idx}\text{-eq}$: $wlp \text{ } (x[i] := a) \text{ } Q \text{ } s = Q \text{ } (s(x := (s \text{ } x)(aval \text{ } i \text{ } s := aval \text{ } a \text{ } s)))$ $\langle proof \rangle$

lemma $wlp\text{-arraycpy}\text{-eq}$: $wlp \text{ } (x[] := a) \text{ } Q \text{ } s = Q \text{ } (s(x := s \text{ } a))$ $\langle proof \rangle$

lemma $wlp\text{-arrayinit}\text{-eq}$: $wlp \text{ } (CLEAR \text{ } x[]) \text{ } Q \text{ } s = Q \text{ } (s(x := (\lambda \text{ } \cdot \text{ } 0)))$ $\langle proof \rangle$

lemma $wlp\text{-assign}\text{-locals}\text{-eq}$: $wlp \text{ } (Assign\text{-Locals} \text{ } l) \text{ } Q \text{ } s = Q \text{ } \langle l | s \rangle$ $\langle proof \rangle$

lemma $wlp\text{-seq}\text{-eq}$: $wlp \text{ } (c_1 ; c_2) \text{ } Q \text{ } s = wlp \text{ } c_1 \text{ } (wlp \text{ } c_2 \text{ } Q) \text{ } s$ $\langle proof \rangle$

lemma $wlp\text{-if}\text{-eq}$: $wlp \text{ } (IF \text{ } b \text{ } THEN \text{ } c_1 \text{ } ELSE \text{ } c_2) \text{ } Q \text{ } s$
 $= (if \text{ } bval \text{ } b \text{ } s \text{ } then \text{ } wlp \text{ } c_1 \text{ } Q \text{ } s \text{ } else \text{ } wlp \text{ } c_2 \text{ } Q \text{ } s)$ $\langle proof \rangle$

lemma $wlp\text{-scope}\text{-eq}$: $wlp \text{ } (SCOPE \text{ } c) \text{ } Q \text{ } s = wlp \text{ } c \text{ } (\lambda s'. Q \text{ } \langle s | s' \rangle)$
 $\langle \langle \rangle | s \rangle$ $\langle proof \rangle$

lemma $wlp\text{-pcall}\text{-eq}$: $\pi \text{ } p = Some \text{ } c \implies wlp \text{ } (PCall \text{ } p) \text{ } Q \text{ } s = wlp \text{ } c \text{ } Q \text{ } s$ $\langle proof \rangle$

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

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

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

end

lemma $wlp\text{-while}\text{-unfold}$: $wlp \text{ } (WHILE \text{ } b \text{ } DO \text{ } c) \text{ } Q \text{ } s = (if \text{ } bval \text{ } b \text{ } s \text{ } then \text{ } wlp \text{ } c \text{ } (wlp \text{ } (WHILE \text{ } b \text{ } DO \text{ } c) \text{ } Q) \text{ } s \text{ } else \text{ } Q \text{ } s)$
 $\langle proof \rangle$

lemma $wp\text{-while}\text{-unfold}$: $wp \text{ } (WHILE \text{ } b \text{ } DO \text{ } c) \text{ } Q \text{ } s = (if \text{ } bval \text{ } b \text{ } s \text{ } then \text{ } wp \text{ } c \text{ } (wp \text{ } (WHILE \text{ } b \text{ } DO \text{ } c) \text{ } Q) \text{ } s \text{ } else \text{ } Q \text{ } s)$
 $\langle proof \rangle$

end — Context fixing program

Unfold rules for procedure scope

lemma $wp\text{-pscope}\text{-eq}$: $wp \text{ } \pi \text{ } (PScope \text{ } \pi' \text{ } c) \text{ } Q \text{ } s = wp \text{ } \pi' \text{ } (c) \text{ } Q \text{ } s$
 $\langle proof \rangle$

lemma $wlp\text{-pscope}\text{-eq}$: $wlp \text{ } \pi \text{ } (PScope \text{ } \pi' \text{ } c) \text{ } Q \text{ } s = wlp \text{ } \pi' \text{ } (c) \text{ } Q \text{ } s$
 $\langle proof \rangle$

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*: $(\forall \pi. wp \ \pi \ c = wp \ \pi \ c') \longleftrightarrow c \sim c'$
 $\langle proof \rangle$

lemma *wlp-equiv-iff*: $(\forall \pi. wlp \ \pi \ c = wlp \ \pi \ c') \longleftrightarrow c \sim c'$
 $\langle proof \rangle$

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 $\pi: (WHILE \ b' \ DO \ c', \ s) \Rightarrow t$
 $\langle proof \rangle$

lemma *sim-while-wp*: $bval \ b = bval \ b' \Longrightarrow wp \ \pi \ c = wp \ \pi \ c' \Longrightarrow wp \ \pi \ (WHILE \ b \ DO \ c) = wp \ \pi \ (WHILE \ b' \ DO \ c')$
 $\langle proof \rangle$

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 $\pi: (WHILE \ b' \ DO \ c', \ s) \Rightarrow t$
 $\langle proof \rangle$

lemma *sim-while-wlp*: $bval \ b = bval \ b' \Longrightarrow wlp \ \pi \ c = wlp \ \pi \ c' \Longrightarrow wlp \ \pi \ (WHILE \ b \ DO \ c) = wlp \ \pi \ (WHILE \ b' \ DO \ c')$
 $\langle proof \rangle$

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*: $\bigwedge s. I s \implies (\text{if } \text{bval } b \text{ } s \text{ then } \text{wlp } \pi \text{ } c \text{ } I s \text{ else } Q s)$
shows $\text{wlp } \pi \text{ } (WHILE \text{ } b \text{ } DO \text{ } c) \text{ } Q s_0$
<proof>

lemma
assumes *INIT*: $I s_0$
assumes *STEP*: $\bigwedge s. I s \implies (\text{if } \text{bval } b \text{ } s \text{ then } \text{wlp } \pi \text{ } c \text{ } I s \text{ else } Q s)$
shows $\text{wlp } \pi \text{ } (WHILE \text{ } b \text{ } DO \text{ } c) \text{ } Q s_0$
<proof>

2.9.2 Total Correctness

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

lemma *wp-whileI'*:
assumes *WF*: $\text{wf } R$
assumes *INIT*: $I s_0$
assumes *STEP*: $\bigwedge s. I s \implies (\text{if } \text{bval } b \text{ } s \text{ then } \text{wp } \pi \text{ } c \text{ } (\lambda s'. I s' \wedge (s', s) \in R) \text{ } s \text{ else } Q s)$
shows $\text{wp } \pi \text{ } (WHILE \text{ } b \text{ } DO \text{ } c) \text{ } Q s_0$
<proof>

lemma
assumes *WF*: $\text{wf } R$
assumes *INIT*: $I s_0$
assumes *STEP*: $\bigwedge s. I s \implies (\text{if } \text{bval } b \text{ } s \text{ then } \text{wp } \pi \text{ } c \text{ } (\lambda s'. I s' \wedge (s', s) \in R) \text{ } s \text{ else } Q s)$
shows $\text{wp } \pi \text{ } (WHILE \text{ } b \text{ } DO \text{ } c) \text{ } Q s_0$
<proof>

2.9.3 Standard Forms of While Rules

lemma *wlp-whileI*:
assumes *INIT*: $I s_0$
assumes *STEP*: $\bigwedge s. \llbracket I s; \text{bval } b \text{ } s \rrbracket \implies \text{wlp } \pi \text{ } c \text{ } I s$
assumes *FINAL*: $\bigwedge s. \llbracket I s; \neg \text{bval } b \text{ } s \rrbracket \implies Q s$
shows $\text{wlp } \pi \text{ } (WHILE \text{ } b \text{ } DO \text{ } c) \text{ } Q s_0$
<proof>

lemma *wp-whileI*:
assumes *WF*: $\text{wf } R$
assumes *INIT*: $I s_0$

assumes *STEP*: $\bigwedge s. \llbracket I \ s; \text{bval } b \ s \rrbracket \implies wp \ \pi \ c \ (\lambda s'. I \ s' \wedge (s',s) \in R) \ s$
assumes *FINAL*: $\bigwedge s. \llbracket I \ s; \neg \text{bval } b \ s \rrbracket \implies Q \ s$
shows $wp \ \pi \ (\text{WHILE } b \ \text{DO } c) \ Q \ s_0$
 $\langle \text{proof} \rangle$

2.10 Modularity of Programs

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

lemma *map-leD*: $m \subseteq_m m' \implies m \ x = \text{Some } v \implies m' \ x = \text{Some } v$
 $\langle \text{proof} \rangle$

lemma *big-step-mono-prog*:

assumes $\pi \subseteq_m \pi'$
assumes $\pi : (c, s) \Rightarrow t$
shows $\pi' : (c, s) \Rightarrow t$
 $\langle \text{proof} \rangle$

Wrapping a set of recursive procedures into a procedure scope

lemma *localize-recursion*:

$\pi' : (\text{PScope } \pi \ c, \ s) \Rightarrow t \longleftrightarrow \pi : (c, s) \Rightarrow t$
 $\langle \text{proof} \rangle$

2.11 Strongest Postcondition

context fixes $\pi :: \text{program begin}$

definition $sp \ P \ c \ t \equiv \exists s. P \ s \wedge \pi : (c, s) \Rightarrow t$

context notes $[simp] = sp\text{-def}[abs\text{-def}] \text{ begin}$

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

lemma *sp-arraycpy-eq*: $sp \ P \ (x[] := y) \ t \longleftrightarrow (\exists vx. \text{let } s = t(x := vx) \text{ in } t \ x = s \ y \wedge P \ s)$
 $\langle \text{proof} \rangle$

Version with renaming of assigned variable

lemma *sp-arraycpy-eq'*: $sp \ P \ (x[] := y) \ t \longleftrightarrow t \ x = t \ y \wedge (\exists vx. P \ (t(x := vx, y := t \ x)))$
 $\langle \text{proof} \rangle$

lemma *sp-skip-eq*: $sp \ P \ \text{SKIP} \ t \longleftrightarrow P \ t \ \langle \text{proof} \rangle$

lemma *sp-seq-eq*: $sp \ P \ (c_1;;c_2) \ t \longleftrightarrow sp \ (sp \ P \ c_1) \ c_2 \ t \ \langle \text{proof} \rangle$

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 $\bigwedge s. P' s \implies P s$

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

shows $HT \pi P' c Q'$

<proof>

lemma *HT-partial-conseq*:

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

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

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

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

<proof>

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

lemma *wp-modularity-rule*:

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

<proof>

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

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 $HT \pi (wp \pi \text{ prologue } P) (\text{prologue};;\text{body}) (\lambda s_0 s. wp \pi \text{ prologue } (\lambda s_0. Q s_0 s) s_0)$
 $\langle \text{proof} \rangle$

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 $HT \pi P (\text{body};;\text{epilogue}) (\lambda s_0. sp \pi (Q s_0) \text{ epilogue})$
 $\langle \text{proof} \rangle$

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 $\text{locals}:=\langle \rangle$ and the prologue $\text{locals}:=\text{old-locals}$.

lemma *adjust-scope*:

assumes $HT \pi P \text{ body } Q$
shows $HT \pi (\lambda s. P \langle \langle \rangle | s \rangle) (SCOPE \text{ body}) (\lambda s_0 s. \exists l. Q (\langle \langle \rangle | s_0 \rangle) (\langle l | s \rangle))$
 $\langle \text{proof} \rangle$

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 $HTset \pi \Theta$
 $\langle \text{proof} \rangle$

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 $HT \pi P c Q$

$\langle proof \rangle$

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 $HT \ \pi \ P \ (PCall \ p) \ Q$

$\langle proof \rangle$

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 $\forall (P, p, Q) \in \Theta. HT \ \pi \ P \ (PCall \ p) \ Q$

$\langle proof \rangle$

2.13 Completeness of While-Rule

Idea: Use wlp as invariant

lemma *wlp-whileI'-complete:*

assumes $wlp \ \pi \ (WHILE \ b \ DO \ c) \ Q \ s_0$

obtains I **where**

$I \ s_0$

$\bigwedge s. I \ s \implies \text{if } bval \ b \ s \text{ then } wlp \ \pi \ c \ I \ s \text{ else } Q \ s$

$\langle proof \rangle$

Idea: Remaining loop iterations as variant

inductive *count-it* **for** $\pi \ b \ c$ **where**

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

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

lemma *count-it-determ:*

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

$\langle proof \rangle$

lemma *count-it-ex:*

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

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

$\langle proof \rangle$

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

lemma *variant-decreases*:
assumes *STEPB*: $bval\ b\ s$
assumes *STEPB*: $\pi: (c,s) \Rightarrow s'$
assumes *TERM*: $\pi: (WHILE\ b\ DO\ c,s') \Rightarrow t$
shows $variant\ \pi\ b\ c\ s' < variant\ \pi\ b\ c\ s$
 $\langle proof \rangle$

lemma *wp-whileI'-complete*:
fixes $\pi\ b\ c$
defines $R \equiv measure\ (variant\ \pi\ b\ c)$
assumes $wp\ \pi\ (WHILE\ b\ DO\ c)\ Q\ s_0$
obtains I **where**
 $wf\ R$
 $I\ s_0$
 $\bigwedge s. I\ s \implies if\ bval\ b\ s\ then\ wp\ \pi\ c\ (\lambda s'. I\ s' \wedge (s',s) \in R)\ s\ else\ Q\ s$
 $\langle proof \rangle$

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 ($\forall s_0. P s_0 \longrightarrow wlp \pi (c s_0) (Q s_0) s_0$)

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

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

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

\langle *proof* \rangle

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

\langle *proof* \rangle

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  $\pi \Theta$ 
  <proof>

```

```

lemma HT'setD:
  assumes HT'set  $\pi (insert (f,(P,c,Q)) \Theta)$ 
  shows HT'  $\pi P c Q$  and HT'set  $\pi \Theta$ 
  <proof>

```

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 <b≠0>
  ensures <c=a0 div b0
  defines <c = a/b>
  <proof>

```

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 $HT-mods\ \pi\ vs\ P\ c\ Q$, 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 ($BB-PROTECT \equiv \lambda a. a$).

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** $\langle r = div-ab(a, a) \rangle$ $\langle proof \rangle$

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** $c = a_0 * b_0$
defines \langle
 if $(a < 0)$ $\{ a = -a; b = -b \};$
 $c = 0;$
 while $(a > 0)$
 @variant $\langle a \rangle$
 @invariant $\langle 0 \leq a \wedge a \leq |a_0| \wedge c = (|a_0| - a) * b_0 * sgn\ a_0 \rangle$
 $\{$

```

    c=c+b;
    a=a-1
  }
}
⟨proof⟩

```

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., *'a*, 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  $\langle \text{if } (n == 0) \text{ } b = 0 \text{ else } b = \text{rec even-imp}(n-1) \rangle$ 
and
  even-imp(n) returns b assumes  $n \geq 0$  ensures  $\langle b \neq 0 \iff \text{even } n_0 \rangle$ 
variant  $\langle n \rangle$ 
  defines  $\langle \text{if } (n == 0) \text{ } b = 1 \text{ else } b = \text{rec odd-imp}(n-1) \rangle$ 
  ⟨proof⟩

```

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  $\langle True \rangle$  ensures
c = a0 * b0
defines  $\langle$ 
  if ( $a < 0$ ) {  $a = -a$ ;  $b = -b$ };
   $c = 0$ ;
  while ( $a > 0$ )
     $@variant \langle a \rangle$ 
     $@invariant \langle 0 \leq a \wedge a \leq |a_0| \wedge c = (|a_0| - a) * b_0 * sgn a_0 \rangle$ 
    {
       $c = c + b$ ;
       $a = a - 1$ 
    }
   $\rangle$ 
 $\langle proof \rangle$ 

```

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  $\langle True \rangle$  ensures
c = a0 * b0
defines  $\langle$ 
  if ( $a < 0$ ) {  $a = -a$ ;  $b = -b$ };
   $c = 0$ ;
  while ( $a > 0$ )
     $@relation \langle measure \ nat \rangle$ 
     $@variant \langle a \rangle$ 
     $@invariant \langle 0 \leq a \wedge a \leq |a_0| \wedge c = (|a_0| - a) * b_0 * sgn a_0 \rangle$ 
    {
       $c = c + b$ ;
       $a = a - 1$ 
    }
   $\rangle$ 
 $\langle proof \rangle$ 

```

```

recursive-spec relation  $\langle measure \ nat \rangle$ 
  odd-imp'(n) returns b assumes  $n \geq 0$  ensures  $\langle b \neq 0 \iff odd \ n \rangle$ 
variant  $\langle n \rangle$ 
  defines  $\langle if \ (n == 0) \ b = 0 \ else \ b = rec \ even-imp'(n-1) \rangle$ 
and

```

```

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

```

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  $\langle a = 0 \rangle$  defines
   $\langle \text{while } (a \neq 0) \text{ } @\text{invariant } \langle \text{True} \rangle$ 
   $\text{ } a = a - 1 \rangle$ 
   $\langle \text{proof} \rangle$ 

```

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* \Rightarrow *int*.

```

  lemma array-sum-aux:  $l_0 \leq l \implies \{l_0..<l + 1\} = \text{insert } l \{l_0..<l\}$ 
  for  $l_0 \ l :: \text{int}$   $\langle \text{proof} \rangle$ 

```

```

  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  $\langle l_0 \leq l \wedge l \leq h \wedge s = (\sum i=l_0..<l. a \ i) \rangle$ 
   $\{ s = s + a[l]; \ l = l + 1 \}$ 
   $\langle \text{proof} \rangle$ 

```

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

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.

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 = x10+x2 defines ⟨
  y=0;
  while (x1>0)
    @variant ⟨y + x3⟩
    @invariant ⟨y>x3⟩
    {
      x1=x2
    }
  ⟩
⟨proof⟩
```

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 defines ⟨
  while (n>0) @variant ⟨n⟩ @invariant ⟨True⟩
  { n=n-1 }
  ⟩
⟨proof⟩
```

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 | STRUCT fname  $\Rightarrow$  val | 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* \equiv
 $\text{"a" ::= N 1};$
 $\text{"c" ::= N 0};$
 $\text{WHILE Cmpop } (<) (V \text{"c"}) (V \text{"n"}) \text{ DO } ($
 $\text{"a" ::= Binop } (*) (N 2) (V \text{"a"});;$
 $\text{"c" ::= Binop } (+) (V \text{"c"}) (N 1))$

Fancy Syntax

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

lemma *exp-count-up1 = exp-count-up2*
 $\langle \text{proof} \rangle$

5.2 Operators and Arrays

We reflect arbitrary Isabelle functions into the syntax:

value *bval* $(\text{Cmpop } (\leq) (\text{Binop } (+) (\text{Unop } \text{uminus } (V \text{"x"})) (N 42)) (N 50)) <\text{"x"} := (\lambda -. -5)>$

thm *aval.simps bval.simps*

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

term $\langle \text{Vidx } \text{"a"} (i::aexp) \rangle$ — Array access at index i
lemma $V \text{"x"} = \text{Vidx } \text{"x"} (N 0) \langle \text{proof} \rangle$

New commands:

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

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

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

lemma $\langle \text{Assign } \text{"x"} v = \text{AssignIdx } \text{"x"} (N 0) v \rangle \langle \text{proof} \rangle$

term $\langle \text{"x"} ::= v \rangle$ **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 *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* $f :: \text{com}$ **begin**

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

end

5.4 Recursive procedures

term $\langle \text{PCall } "name" \rangle$

thm *big-step.PCall*

5.4.1 Procedure Scope

Execute command with local set of procedures

term $\langle \text{PScope } \pi \ c \rangle$

thm *big-step.PScope*

5.4.2 Syntactic sugar for procedure call with parameters

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

5.5 More Readable VCs

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

lemma $s_0 \text{ ''}n'' 0 \geq 0 \implies \text{wlp } \pi \text{ exp-count-up1 } (\lambda s. s \text{ ''}a'' 0 = 2^{\widehat{\text{nat}}(s_0 \text{ ''}n'' 0)}) s_0$
 ⟨proof⟩

lemma $s_0 \text{ ''}n'' 0 \geq 0 \implies \text{wlp } \pi \text{ exp-count-up1 } (\lambda s. s \text{ ''}a'' 0 = 2^{\widehat{\text{nat}}(s_0 \text{ ''}n'' 0)}) s_0$
 ⟨proof⟩

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* $n = s_0 \text{ ''}n'' 0$ *in* $n \geq 0$
 $\implies \text{wlp } \pi \text{ exp-count-up1 } (\lambda s. \text{let } a = s \text{ ''}a'' 0; n_0 = s_0 \text{ ''}n'' 0 \text{ in } a = 2^{\widehat{\text{nat}}(n_0)}) s_0$
 ⟨proof⟩

lemma $\text{VAR } (s \ x) \ P = (\text{let } v = s \ x \ \text{in } P \ v)$ ⟨proof⟩

IMP2 specification commands

program-spec (*partial*) *exp-count-up*
assumes $0 \leq n$ — Precondition. Use variable names of program.
ensures $a = 2^{\widehat{\text{nat}} n_0}$ — Postcondition. Use variable names of programs. Suffix with \cdot_0 to refer to initial state
defines — Program
 <
 $a = 1;$
 $c = 0;$
while $(c < n)$
 @invariant $\langle n = n_0 \wedge a = 2^{\widehat{\text{nat}} c} \wedge 0 \leq c \wedge c \leq n \rangle$ — Invar annotation. Variable names and suffix \cdot_0 for variables from initial state.
 {
 $a = 2 * a;$
 $c = c + 1$
 }
 >
 ⟨proof⟩

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  $\langle n - c \rangle$ 
    {
       $a = 2 * a$ ;
       $c = c + 1$ 
    }
  >
  <proof>

```

Simple Recursion

```

recursive-spec
  exp-rec( $n$ ) returns  $a$  assumes  $0 \leq n$  ensures  $a = 2^{\widehat{\text{nat}} n_0}$  variant
   $n$ 
  defines <if ( $n == 0$ )  $a = 1$  else { $t = \text{rec } \text{exp-rec}(n-1)$ ;  $a = 2 * t$ }>
  <proof>

```

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 {  $a = G\text{-par}; a = 2 * a; G\text{-ret} = a$  };  $a = G\text{-ret};$ 
       $c = c + 1$ 
    }
  >
  <proof>

```

```

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  $\langle n_0 = n \wedge 0 \leq i \wedge i \leq n \wedge s = \sum \{0..i\} \rangle$ 
    {
       $i = i + 1;$ 
       $s = s + i$ 
    }
  >
  <proof>

```

```

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>
    {
       $a = a + z;$ 
       $z = z + 2;$ 
       $i = i + 1$ 
    }
  >
  <proof>

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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  $(i < n)$ 
       $\text{@variant } \langle n_0 - i \rangle$ 
       $\text{@invariant } \langle n = n_0 \wedge 0 \leq i \wedge i \leq n \wedge a = \text{fib } i \wedge b = \text{fib } (i +$ 
1)  $\rangle$ 
      {
         $c = b;$ 
         $b = a + b;$ 
         $a = c;$ 
         $i = i + 1$ 
      }
     $\rangle$ 
     $\langle \text{proof} \rangle$ 

```

Without precondition, returning 0 for negative numbers

```

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

```

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  $0 \leq n$ 
  ensures  $a = 2^{\widehat{\text{nat}} } n_0$ 
  defines <
     $a = 1;$ 
    while ( $n > 0$ )
      @variant  $\langle n \rangle$ 
      @invariant  $\langle \text{let } c = n_0 - n \text{ in } 0 \leq n \wedge n \leq n_0 \wedge a = 2^{\widehat{\text{nat}} } c \rangle$ 
      {
         $a = 2 * a;$ 
         $n = n - 1$ 
      }
    >
  <proof>

```

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  $\langle$ 
     $r = 1;$ 
    while  $(r * r \leq n)$ 
       $\text{@variant } \langle n + 1 - r * r \rangle$ 
       $\text{@invariant } \langle 0 \leq r \wedge (r-1)^2 \leq n_0 \rangle$ 
       $\{ r = r + 1 \};$ 
       $r = r - 1$ 
     $\rangle$ 
   $\langle \text{proof} \rangle$ 

```

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  $\bigwedge h \ l \ n_0 :: \text{int.}$ 
   $\llbracket \text{"invar-final"}; 0 \leq n_0; \neg 1 + l < h; 0 \leq l; l < h; l * l \leq n_0;$ 
   $n_0 < h * h \rrbracket$ 
   $\implies n_0 < 1 + (l * l + l * 2)$ 
   $\langle \text{proof} \rangle$ 

```

```

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

```

<proof>

6.2 Debugging

6.2.1 Testing Programs

Stepwise

schematic-goal *Map.empty*: (*sqr-approx-below*, *<"n" := λ-. 4>*) \Rightarrow ?s
<proof>

Or all steps at once

schematic-goal *Map.empty*: (*sqr-bisect*, *<"n" := λ-. 4900000001>*) \Rightarrow ?s
<proof>

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 \langle
 while ($a \neq b$)
 @invariant $\langle \text{gcd } a \ b = \text{gcd } a_0 \ b_0 \wedge (a > 0 \wedge b > 0) \rangle$
 @variant $\langle a + b \rangle$
 {
 if ($a < b$) $b = b - a$
 else $a = a - b$
 }
 \rangle
<proof>

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 \langle
 while ($b \neq 0$)
 @invariant $\langle \text{gcd } a \ b = \text{gcd } a_0 \ b_0 \wedge b \geq 0 \wedge a > 0 \rangle$
 @variant $\langle b \rangle$
 {
 $t = a;$
 $a = b;$
 }
 \rangle

```

    b = t mod b
  }
}
⟨proof⟩

```

6.3.3 Extended Euclid's Algorithm

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

lemma *aux2*:

```

  fixes a b :: int
  assumes b = t * b0 + s * a0 q = a div b gcd a b = gcd a0 b0
  shows gcd b (a - (a0 * (s * q) + b0 * (t * q))) = gcd a0 b0
⟨proof⟩

```

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⟩

```

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⟩

```

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⟩

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 <

$x = 0$;

$y = 1$;

$\text{old-}x = 1$;

$\text{old-}y = 0$;

while ($b \neq 0$)

 @invariant <

$\text{gcd } a \ b = \text{gcd } a_0 \ b_0 \wedge b \geq 0 \wedge a > 0 \wedge a = a_0 * \text{old-}x + b_0 * \text{old-}y \wedge b = a_0 * x + b_0 * y$

 >

 @variant < b >

{

$q = a / b$;

$t = a$;

$a = b$;

$b = t \bmod b$;

$t = x$;

$x = \text{old-}x - q * x$;

$\text{old-}x = t$;

$t = y$;

$y = \text{old-}y - q * y$;

$\text{old-}y = t$

};

$x = \text{old-}x$;

$y = \text{old-}y$

>

⟨proof⟩

6.3.4 Exponentiation by Squaring

lemma *ex-by-sq-aux*:
fixes $x :: \text{int}$ **and** $n :: \text{nat}$
assumes $n \bmod 2 = 1$
shows $x * (x * x)^{\wedge (n \text{ div } 2)} = x^{\wedge n}$
 $\langle \text{proof} \rangle$

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

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

program-spec *ex-by-sq*
assumes $n \geq 0$
ensures $r = x_0^{\wedge \text{nat } n}$
defines \langle
 $r = 1;$
 $\text{while } (n \neq 0)$
 @invariant \langle
 $n \geq 0 \wedge r * x^{\wedge \text{nat } n} = x_0^{\wedge \text{nat } n}$
 \rangle
 @variant $\langle n \rangle$
 $\{$
 if $(n \bmod 2 == 1)$ $\{$
 $r = r * x$
 $\};$
 $x = x * x;$
 $n = n / 2$
 $\}$
 \rangle
 $\langle \text{proof} \rangle$

6.3.5 Power-Tower of 2s

fun *tower2* **where**
 $\text{tower2 } 0 = 1$
 $|\ \text{tower2 } (\text{Suc } n) = 2^{\wedge \text{tower2 } n}$

definition $\text{tower2}' n = \text{int } (\text{tower2 } (\text{nat } n))$

program-spec *tower2-imp*
assumes $\langle m > 0 \rangle$
ensures $\langle a = \text{tower2}' m_0 \rangle$
defines \langle
 $a = 1;$
 $\text{while } (m > 0)$
 @variant $\langle m \rangle$
 @invariant $\langle 0 \leq m \wedge m \leq m_0 \wedge a = \text{tower2}' (m_0 - m) \rangle$
 \rangle

```

{
  n=a;

  a = 1;
  while (n>0)
    @variant ⟨n⟩
    @invariant ⟨True⟩ — This will get ugly, there is no  $n_0$  that we
could use!
    {
      a=2*a;
      n=n-1
    };

  m=m-1
}
⟩
⟨proof⟩

```

We prove the inner loop separately instead! (It happens to be exactly our *exp-count-down* program.)

```

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;
      inline exp-count-down;
      m=m-1
    }
  ⟩
⟨proof⟩

```

6.4 Array Algorithms

6.4.1 Summation

```

program-spec array-sum
assumes l≤h
ensures r = (∑ i=l0..<h0. a0 i)
defines ⟨
  r = 0;
  while (l<h)
    @invariant ⟨l0≤l ∧ l≤h ∧ r = (∑ i=l0..<l. a0 i)⟩
    @variant ⟨h-l⟩
    {
      r = r + a[l];
    }
  ⟩

```

```

    l=l+1
  }
}
⟨proof⟩

```

6.4.2 Finding Least Index of Element

```

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

```

6.4.3 Check for Sortedness

term *ran-sorted*

```

program-spec check-sorted
  assumes ⟨ $l \leq h$ ⟩
  ensures ⟨ $r \neq 0 \iff \text{ran-sorted } a_0 \ l_0 \ h_0$ ⟩
  defines ⟨
    if ( $l==h$ )  $r=1$ 
    else {
       $l=l+1$ ;
      while ( $l < h \wedge a[l-1] \leq a[l]$ )
        @variant ⟨ $h-l$ ⟩
        @invariant ⟨ $l_0 < l \wedge l \leq h \wedge \text{ran-sorted } a \ l_0 \ l$ ⟩
         $l=l+1$ ;
    }
    if ( $l==h$ )  $r=1$  else  $r=0$ 
  }
  ⟨proof⟩

```

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 ⟨ $l \leq h$ ⟩
  ensures ⟨ $\text{is-equil } a \ l \ h \ i \vee i=h \wedge (\nexists i. \text{is-equil } a \ l \ h \ i)$ ⟩
  defines ⟨
     $usum=0$ ;  $i=l$ ;

```

```

while (i < h)
  @variant <h-i>
  @invariant <l ≤ i ∧ i ≤ h ∧ usum = (∑ j=l..<i. a j)>
  {
    usum = usum + a[i]; i=i+1
  };
i=l; lsum=0;
while (usum ≠ lsum ∧ i < h)
  @variant <h-i>
  @invariant <l ≤ i ∧ i ≤ h
    ∧ lsum=(∑ j=l..<i. a j)
    ∧ usum=(∑ j=i..<h. a j)
    ∧ (∀ j < i. ¬is-equil a l h j)
  >
  {
    lsum = lsum + a[i];
    usum = usum - a[i];
    i=i+1
  }
}
<proof>

```

6.4.5 Rotate Right

program-spec *rotate-right*

assumes $0 < n$

ensures $\forall i \in \{0..<n\}. a\ i = a_0\ ((i-1) \bmod n)$

defines <

$i = 0;$

$prev = a[n - 1];$

while ($i < n$)

@invariant <

$0 \leq i \wedge i \leq n$

$\wedge (\forall j \in \{0..<i\}. a\ j = a_0\ ((j-1) \bmod n))$

$\wedge (\forall j \in \{i..<n\}. a\ j = a_0\ j)$

$\wedge prev = a_0\ ((i-1) \bmod n)$

>

@variant < $n - i$ >

{

$temp = a[i];$

$a[i] = prev;$

$prev = temp;$

$i = i + 1$

}

>

<proof>

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 $(l < h)$
 @variant $\langle h-l \rangle$
 @invariant $\langle l_0 \leq l \wedge l \leq h \wedge h \leq h_0 \wedge (\forall i \in \{l_0..<l\}. a\ i < x) \wedge$
 $(\forall i \in \{h..<h_0\}. x \leq a\ i) \rangle$
 {
 $m = (l + h) / 2;$
 if $(a[m] < x)$ $l = m + 1$
 else $h = m$
 }
 \rangle
 $\langle \text{proof} \rangle$

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

lemma

assumes *bin-search-pre* $a\ l\ h$ *bin-search-post* $a\ l\ h\ x\ i$
shows *bin-search-decide-membership*: $x \in a\ \{l..<h\} \longleftrightarrow (i < h \wedge x = a\ i)$
and *bin-search-leftmost*: $x \notin a\ \{l..<i\}$
 $\langle \text{proof} \rangle$

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 $(l1 + i < h1 \wedge a[l + i] == b[l1 + i])$
 @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

```

}
>
<proof>

lemma ran-eq-iff':  $\text{ran } a \ l1 \ (l1 + (h - l)) = \text{ran } a' \ l \ h$ 
 $\longleftrightarrow (\forall i. 0 \leq i \wedge i < h - l \longrightarrow a \ (l1 + i) = a' \ (l + i))$  if  $l \leq h$ 
<proof>

program-spec match-string'
  assumes  $l1 \leq h1$ 
ensures  $i = h1 - l1 \longleftrightarrow \text{ran } a \ l \ (l + (h1 - l1)) = \text{ran } b \ l1 \ h1$ 
for  $i \ h1 \ l1 \ l \ a[] \ b[]$ 
defines <inline match-string>
  <proof>

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

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

```

```

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

```

lemma *ran-split*:
 $\text{ran } a \ l \ h = \text{ran } a \ l \ p \ @ \ \text{ran } a \ p \ h$ **if** $l \leq p \ p \leq h$
 \langle *proof* \rangle

lemma *ran-eq-append-iff*:
 $\text{ran } a \ l \ h = as \ @ \ bs \longleftrightarrow (\exists i. l \leq i \wedge i \leq h \wedge as = \text{ran } a \ l \ i \wedge bs$
 $= \text{ran } a \ i \ h)$ **if** $l \leq h$
 \langle *proof* \rangle

lemma *ran-split'*:
 $(\exists j \in \{l..h - (h1 - l1)\}. \text{ran } a \ j \ (j + (h1 - l1)) = \text{ran } b \ l1 \ h1)$
 $= (\exists as \ bs. \text{ran } a \ l \ h = as \ @ \ \text{ran } b \ l1 \ h1 \ @ \ bs)$ **if** $l \leq h \ l1 \leq h1$
 \langle *proof* \rangle

```

program-spec substring-final
  assumes  $l \leq h \wedge 0 \leq len$ 
  ensures  $match = 1 \longleftrightarrow (\exists as \ bs. \text{ran } a \ l_0 \ h_0 = as \ @ \ \text{ran } b \ 0 \ len$ 
   $@ \ bs)$ 
  for  $l \ h \ len \ match \ a[] \ b[]$ 
  defines  $\langle l1 = 0; h1 = len; \text{inline } \text{substring}' \rangle$ 
   $\langle$ proof $\rangle$ 

```

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* $l\ j\ a_0\ a\ i$
 \longleftrightarrow (let $key = a_0\ j$ in
 $i \in \{l-1..<j\}$ — i is in range
— 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 $-\{l..j\}$
— Placement of key
 $\wedge (i \geq l \longrightarrow a\ i \leq key)$ — Element at i smaller than key , if it exists
 $\wedge (\forall k \in \{i+2..j\}. a\ k > key)$ — Elements $\geq i+2$ 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 $(i \geq l \wedge a[i] > key)$
 \langle *variant* $\langle i-l+1 \rangle$
 \langle *invariant* $\langle l-1 \leq i \wedge i < j$
 $\wedge (\forall k \in \{l..i\}. a\ k = a_0\ k)$
 $\wedge (\forall k \in \{i+2..j\}. a\ k > key \wedge a\ k = a_0\ (k-1))$
 $\wedge a = a_0$ on $-\{l..j\}$
 \rangle
 $\{$
 $a[i+1] = a[i];$
 $i = i-1$
 $\}$;
 $a[i+1] = key$
 \rangle
 \langle *proof* \rangle

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 insert-insert-post  $l\ j\ a\ a'\ i'$ 
assumes ran-sorted  $a\ l\ j$ 
shows ran-sorted  $a'\ l\ (j + 1)$ 
<proof>

```

Invoking *insert-insert* will only mutate the elements

```

lemma insert-insert-ran1:
  assumes  $l < j$ 
  assumes insert-insert-post  $l\ j\ a\ a'\ i$ 
  shows  $mset\text{-}ran\ a'\ \{l..j\} = mset\text{-}ran\ a\ \{l..j\}$ 
<proof>

```

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

```

lemma insert-insert-ran2:
  assumes  $l < j < h$ 
  assumes insert-insert-post  $l\ j\ a\ a'\ i$ 
  shows  $mset\text{-}ran\ a'\ \{l..<h\} = mset\text{-}ran\ a\ \{l..<h\}$  (is ?thesis1)
  and  $a' = a\ \text{on}\ -\{l..<h\}$  (is ?thesis2)
<proof>

```

Finally, we specify and prove correct the outer loop

```

program-spec insert
  assumes  $l < h$ 
  ensures ran-sorted  $a\ l\ h \wedge mset\text{-}ran\ a\ \{l..<h\} = mset\text{-}ran\ a_0\ \{l..<h\}$ 
 $\wedge a = a_0\ \text{on}\ -\{l..<h\}$ 
  for  $a[]$ 
  defines <
     $j = l + 1;$ 
    while  $(j < h)$ 
    @variant  $\langle h - j \rangle$ 
    @invariant <
       $l < j \wedge j \leq h$  —  $j$  in range
       $\wedge \textit{ran-sorted}\ a\ l\ j$  — Array is sorted up to  $j$ 
       $\wedge mset\text{-}ran\ a\ \{l..<h\} = mset\text{-}ran\ a_0\ \{l..<h\}$  — Elements in
range only permuted
       $\wedge a = a_0\ \text{on}\ -\{l..<h\}$ 
    >
  >
  {
    inline insert-insert;
     $j = j + 1$ 
  }
>
<proof>

```

6.4.9 Quicksort

procedure-spec *partition-aux*(a, l, h, p) **returns** (a, i)

assumes $l \leq h$
ensures $mset\text{-}ran\ a_0\ \{l_0..<h_0\} = mset\text{-}ran\ a\ \{l_0..<h_0\}$
 $\wedge (\forall j \in \{l_0..<i\}. a\ j < p_0)$
 $\wedge (\forall j \in \{i..<h_0\}. a\ j \geq p_0)$
 $\wedge l_0 \leq i \wedge i \leq h_0$
 $\wedge a_0 = a\ on\ -\{l_0..<h_0\}$

defines \langle
 $i=l; j=l;$
while ($j < h$)
 @invariant \langle
 $l \leq i \wedge i \leq j \wedge j \leq h$
 $\wedge mset\text{-}ran\ a_0\ \{l_0..<h_0\} = mset\text{-}ran\ a\ \{l_0..<h_0\}$
 $\wedge (\forall k \in \{l_0..<i\}. a\ k < p)$
 $\wedge (\forall k \in \{i..<j\}. a\ k \geq p)$
 $\wedge (\forall k \in \{j..<h\}. a_0\ k = a\ k)$
 $\wedge a_0 = a\ on\ -\{l_0..<h_0\}$
 \rangle
 @variant $\langle (h-j) \rangle$
 {
 if ($a[j] < p$) { $temp = a[i]; a[i] = a[j]; a[j] = temp; i=i+1$ };
 $j=j+1$
 }
 \rangle
 $\langle proof \rangle$

procedure-spec $partition(a, l, h, p)$ **returns** (a, i)
assumes $l < h$
ensures $mset\text{-}ran\ a_0\ \{l_0..<h_0\} = mset\text{-}ran\ a\ \{l_0..<h_0\}$
 $\wedge (\forall j \in \{l_0..<i\}. a\ j < a\ i)$
 $\wedge (\forall j \in \{i..<h_0\}. a\ j \geq a\ i)$
 $\wedge l_0 \leq i \wedge i < h_0 \wedge a_0\ (h_0-1) = a\ i$
 $\wedge a_0 = a\ on\ -\{l_0..<h_0\}$

defines \langle
 $p = a[h-1];$
 $(a, i) = partition\text{-}aux(a, l, h-1, p);$
 $a[h-1] = a[i];$
 $a[i] = p$
 \rangle
 $\langle proof \rangle$

lemma *quicksort-sorted-aux*:

assumes *BOUNDS*: $l \leq i < h$

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 $ran\text{-}sorted\ a_3\ l\ h$
 $\langle proof \rangle$

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*: $aa = ab\ on\ -\ \{i + 1..<h_0\}$
shows $mset\text{-}ran\ a\ \{l_0..<h_0\} = mset\text{-}ran\ ab\ \{l_0..<h_0\}$
 $\langle proof \rangle$

recursive-spec *quicksort*(a, l, h) **returns** a

assumes *True*
ensures $ran\text{-}sorted\ a\ l_0\ h_0 \wedge mset\text{-}ran\ a_0\ \{l_0..<h_0\} = mset\text{-}ran\ a\ \{l_0..<h_0\} \wedge a_0 = a\ on\ -\ \{l_0..<h_0\}$
variant $h - l$
defines \langle
 if ($l < h$) {
 $(a, i) = partition(a, l, h, a[l]);$
 $a = rec\ quicksort(a, l, i);$
 $a = rec\ quicksort(a, i + 1, h)$
 }
 \rangle
 $\langle proof \rangle$

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 \langle
 $i = l; j = l;$
 while ($j < h$)
 @invariant \langle

```

      l ≤ i ∧ i ≤ j ∧ j ≤ h
      ∧ lran a l i = filter (λx. x ≥ 0) (lran a0 l j)
      ∧ lran a j h = lran a0 j h
    }
  @variant ⟨h-j⟩
  {
    if (a[j] ≥ 0) { a[i] = a[j]; i=i+1 };
    j=j+1
  }
}
⟨proof⟩

```

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 ⟨proof⟩

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
  ⟨proof⟩

```

```

lemma merge-mset: mset (merge xs ys) = mset xs + mset ys
  ⟨proof⟩

```

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)
)
  ⟨proof⟩

```

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  $l1 \leq h1 \wedge l2 \leq h2$ 
  ensures let  $ms = \text{lan } m \ 0 \ j$ ;  $xs_0 = \text{lan } a1_0 \ l1_0 \ h1_0$ ;  $ys_0 = \text{lan } a2_0 \ l2_0 \ h2_0$  in
     $j \geq 0 \wedge ms = \text{merge } xs_0 \ ys_0$ 
  defines  $\langle$ 
     $j = 0$ ;
    while  $(l1 \neq h1 \vee l2 \neq h2)$ 
      @variant  $\langle h1 + h2 - l1 - l2 \rangle$ 
      @invariant  $\langle$  let
         $xs = \text{lan } a1 \ l1 \ h1$ ;  $ys = \text{lan } a2 \ l2 \ h2$ ;  $ms = \text{lan } m \ 0 \ j$ ;
         $xs_0 = \text{lan } a1_0 \ l1_0 \ h1_0$ ;  $ys_0 = \text{lan } a2_0 \ l2_0 \ h2_0$ 
      in
         $l1 < h1 \wedge l2 < h2 \wedge 0 \leq j \wedge$ 
         $\text{merge } xs_0 \ ys_0 = ms @ \text{merge } xs \ ys$ 
       $\rangle$ 
     $\rangle$ 
    {
      if  $(l2 == h2 \vee (l1 \neq h1 \wedge a1[l1] < a2[l2]))$  {
         $m[j] = a1[l1]$ ;
         $l1 = l1 + 1$ 
      } else {
         $m[j] = a2[l2]$ ;
         $l2 = l2 + 1$ 
      }
    };
     $j = j + 1$ 
  }
 $\rangle$ 

```

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

<proof>

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  $l1 \leq h1 \wedge l2 \leq h2 \wedge \text{sorted} (\text{lan } a1 \ l1 \ h1) \wedge \text{sorted} (\text{lan } a2 \ l2 \ h2)$ 
  ensures let  $ms = \text{lan } m \ 0 \ j$  in
     $j \geq 0$ 
     $\wedge \text{sorted } ms$ 
     $\wedge \text{mset } ms = \text{mset} (\text{lan } a1_0 \ l1_0 \ h1_0) + \text{mset} (\text{lan } a2_0 \ l2_0 \ h2_0)$ 
  for  $l1 \ h1 \ l2 \ h2 \ a1 [] \ a2 [] \ m [] \ j$ 
  defines <inline merge-imp'>
  <proof>

```

thm *merge-imp-spec*

thm *merge-imp-def*

lemma [*named-ss vcg-bb*]:

$UNIV \cup a = UNIV$

$a \cup UNIV = UNIV$

<proof>

lemma *merge-msets-aux*: $\llbracket l \leq m; m \leq h \rrbracket \implies mset (lran\ a\ l\ m) + mset (lran\ a\ m\ h) = mset (lran\ a\ l\ h)$

<proof>

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

<proof>

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]: $set-of (\lambda-. 0) = \{\}$ *<proof>*
lemma *set-of-insert*[simp]: $x \neq 0 \implies set-of (a(i:=x)) = insert\ i$
(set-of a) <proof>
lemma *set-of-remove*[simp]: $set-of (a(i:=0)) = set-of\ a - \{i\}$ *<proof>*
lemma *set-of-mem*[simp]: $i \in set-of\ a \iff a\ i \neq 0$ *<proof>*
end

program-spec *dedup*
assumes $\langle l \leq h \rangle$
ensures $\langle set\ (lran\ a\ l\ i) = set\ (lran\ a_0\ l\ h) \wedge distinct\ (lran\ a\ l\ i) \rangle$
defines \langle
 $i=l; j=l;$
 $clear\ b[];$
 $while\ (j < h)$
 $\quad @variant\ \langle h-j \rangle$
 $\quad @invariant\ \langle l \leq i \wedge i \leq j \wedge j \leq h$
 $\quad \wedge set\ (lran\ a\ l\ i) = set\ (lran\ a_0\ l\ j)$
 $\quad \wedge distinct\ (lran\ a\ l\ i)$
 $\quad \wedge lran\ a\ j\ h = lran\ a_0\ j\ h$
 $\quad \wedge set-of\ b = set\ (lran\ a\ l\ i)$
 $\quad \rangle$
 $\{$
 $\quad if\ (b[a[j]] == 0) \{$
 $\quad \quad a[i] = a[j]; i=i+1; b[a[j]] = 1$
 $\quad \quad \};$
 $\quad \quad j=j+1$
 $\quad \}$
 $\}$
 \rangle
<proof>

procedure-spec *bv-init* () **returns** *b*
assumes *True* **ensures** $\langle set-of\ b = \{\} \rangle$
defines $\langle clear\ b[] \rangle$
<proof>

procedure-spec *bv-insert* (*x, b*) **returns** *b*
assumes *True* **ensures** $\langle set-of\ b = insert\ x_0\ (set-of\ b_0) \rangle$
defines $\langle b[x] = 1 \rangle$
<proof>

procedure-spec *bv-remove* (*x, b*) **returns** *b*
assumes *True* **ensures** $\langle set-of\ b = set-of\ b_0 - \{x_0\} \rangle$
defines $\langle b[x] = 0 \rangle$
<proof>

procedure-spec *bv-elem* (*x, b*) **returns** *r*
assumes *True* **ensures** $\langle r \neq 0 \iff x_0 \in set-of\ b_0 \rangle$
defines $\langle r = b[x] \rangle$

$\langle proof \rangle$

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

    i=l; j=l;

    while (j<h)
      @variant  $\langle h-j \rangle$ 
      @invariant  $\langle l \leq i \wedge i \leq j \wedge j \leq h$ 
         $\wedge set (lran\ a\ l\ i) = set (lran\ a_0\ l\ j)$ 
         $\wedge distinct (lran\ a\ l\ i)$ 
         $\wedge lran\ a\ j\ h = lran\ a_0\ j\ h$ 
         $\wedge set-of\ b = set (lran\ a\ l\ i)$ 
       $\rangle$ 
      {
        mem = bv-elem (a[j],b);
        if (mem == 0) {
          a[i] = a[j]; i=i+1; b = bv-insert(a[j],b)
        };
        j=j+1
      }
     $\rangle$ 
   $\langle proof \rangle$ 
```

6.6 Recursion

6.6.1 Recursive Fibonacci

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

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 ( $n, m$ )
   $\rangle$ 
   $\langle$ proof $\rangle$ 

```

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]:
   $m \neq 0 \implies \text{ack } m \ 0 = \text{ack } (m-1) \ 1$ 
   $\llbracket m \neq 0; n \neq 0 \rrbracket \implies \text{ack } m \ n = \text{ack } (m-1) \ (\text{ack } m \ (n-1))$ 
   $\langle$ proof $\rangle$ 

```

```

recursive-spec relation less-than  $\langle *lex* \rangle$  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  $\langle$ 
    if ( $m == 0$ )  $r = n+1$ 
    else if ( $n == 0$ )  $r = \text{rec ack-imp } (m-1, 1)$ 
    else {
       $t = \text{rec ack-imp } (m, n-1);$ 
       $r = \text{rec ack-imp } (m-1, t)$ 
    }
   $\rangle$ 
   $\langle$ proof $\rangle$ 

```

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 < y0 then G = y0 - 10 else G = 91 variant 101 - y
for G
defines ⟨
  if (100 < y) G = y - 10
  else {
    rec p91 (y + 11);
    rec p91 (G)
  }
  ⟨proof⟩

```

6.6.5 Odd/Even

```

recursive-spec
odd-imp (a) returns b
assumes True
ensures b ≠ 0 ↔ odd a0
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)
  ⟨proof⟩
and
even-imp (a) returns b
assumes True
ensures b ≠ 0 ↔ even a0
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)
  ⟨proof⟩

```

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 ⟨GZ = GZ0 + GX0*GY0⟩ variant
(|GY|, 1 :: nat)

```

```

for  $GX\ GY\ GZ$ 
defines
<
   $e = \text{even-imp}(GY)$ ;
  if ( $e \neq 0$ ) rec  $\text{evenproduct}()$  else rec  $\text{oddproduct}()$ 
>
and
   $\text{oddproduct}()$  assumes  $\langle \text{odd } GY \rangle$  ensures  $\langle GZ = GZ_0 + GX_0 * GY_0 \rangle$ 
variant ( $|GY|, 0 :: \text{nat}$ )
for  $GX\ GY\ GZ$ 
defines
<
  if ( $GY < 0$ ) {
     $GZ = GZ + GX$ ;
     $GZ = GZ - GX$ 
  } else {
     $GZ = GZ - GX$ ;
     $GZ = GZ + GX$ 
  };
  rec  $\text{evenproduct}()$ 
>
and
   $\text{evenproduct}()$  assumes  $\langle \text{even } GY \rangle$  ensures  $\langle GZ = GZ_0 + GX_0 * GY_0 \rangle$ 
variant ( $|GY|, 0 :: \text{nat}$ )
for  $GX\ GY\ GZ$ 
defines
<
  if ( $GY \neq 0$ ) {
     $GZ = 2 * GX$ ;
     $GZ = GZ / 2$ ;
    rec  $\text{product}()$ 
  }
>
<proof>

```

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 $s_1 \dots s_n$:

Indexes:	...		i		$i+1$...		$i+n$...
Data:	...		n		s_1		...		s_n		...

definition *succs* **where**

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

definition *Edges* where

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

procedure-spec *push'* ($x, stack, ptr$) **returns** ($stack, ptr$)

assumes $ptr \geq 0$ **ensures** $\langle lan\ stack\ 0\ ptr = lan\ stack_0\ 0\ ptr_0\ @$
 $[x_0] \wedge ptr = ptr_0 + 1 \rangle$

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

$\langle proof \rangle$

procedure-spec *push* ($x, stack, ptr$) **returns** ($stack, ptr$)

assumes $ptr \geq 0$ **ensures** $\langle stack\ ' \{0..<ptr\} = \{x_0\} \cup stack_0\ ' \{0..<ptr_0\} \wedge ptr = ptr_0 + 1 \rangle$

for $stack[]$

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

$\langle proof \rangle$

program-spec *get-succs*

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

ensures

$stack\ ' \{0..<i\} = \{x. (j_0 - 1, x) \in Edges\ a \wedge x \notin set-of\ visited\}$
 $\cup stack_0\ ' \{0..<i_0\}$

$\wedge i \geq i_0$

for $i\ j\ stop\ stack[]\ a[]\ visited[]$

defines

\langle

while ($j < stop$)

$\ @invariant\ \langle stack\ ' \{0..<i\} = \{x. x \in a\ ' \{j_0..<j\} \wedge x \notin set-of\ visited\} \cup stack_0\ ' \{0..<i_0\}$

$\ \wedge j \leq stop \wedge i_0 \leq i \wedge j_0 \leq j$

$\ \rangle$

$\ @variant\ \langle (stop - j) \rangle$

$\ \{$

$\ succ = a[j];$

$\ is_elem = bv_elem(succ, visited);$

$\ if\ (is_elem == 0)\ \{$

$\ \ (stack, i) = push\ (succ, stack, i)$

$\ \};$

$\ j = j + 1$

$\ \}$

$\ \rangle$

$\langle proof \rangle$

procedure-spec *pop* ($stack, ptr$) **returns** (x, ptr)

assumes $ptr \geq 1$ **ensures** $\langle stack_0\ ' \{0..<ptr_0\} = stack_0\ ' \{0..<ptr\} \cup \{x\} \wedge ptr_0 = ptr + 1 \rangle$

for $stack[]$

defines $\langle ptr = ptr - 1; x = stack[ptr] \rangle$

$\langle proof \rangle$

procedure-spec *stack-init* () returns *i*
assumes *True* **ensures** $\langle i = 0 \rangle$
defines $\langle i = 0 \rangle$
 $\langle \text{proof} \rangle$

lemma *Edges-empty*:
Edges a “ $\{i\} = \{\}$ **if** $i + 1 \geq a$ *i*
 $\langle \text{proof} \rangle$

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*: $(s, x) \in (\text{Edges } a)^*$
and *closed*: $\forall v \in \text{visited}. \text{Edges } a \text{ “ } \{v\} \subseteq \text{visited}$
and *start*: $s \in \text{visited}$
shows $x \in \text{visited}$
 $\langle \text{proof} \rangle$

program-spec (*partial*) *dfs*
assumes $0 \leq x \wedge 0 \leq s$
ensures $b = 1 \iff x \in (\text{Edges } a)^* \text{ “ } \{s\}$ **defines** \langle
 $b = 0;$
 $\text{clear } \text{stack}[];$
 $i = \text{stack-init}();$
 $(\text{stack}, i) = \text{push}(s, \text{stack}, i);$
 $\text{clear } \text{visited}[];$
 $\text{while } (b == 0 \wedge i \neq 0)$
 $\quad @\text{invariant } \langle 0 \leq i \wedge (s \in \text{stack “ } \{0..<i\} \vee s \in \text{set-of visited}) \wedge$
 $(b = 0 \vee b = 1) \wedge ($
 $\quad \text{if } b = 0 \text{ then}$
 $\quad \quad \text{stack “ } \{0..<i\} \subseteq (\text{Edges } a)^* \text{ “ } \{s\}$
 $\quad \quad \wedge (\forall v \in \text{set-of visited}. (\text{Edges } a) \text{ “ } \{v\} \subseteq \text{set-of visited} \cup \text{stack “}$
 $\quad \quad \{0..<i\})$
 $\quad \quad \wedge (x \notin \text{set-of visited})$
 $\quad \quad \text{else } x \in (\text{Edges } a)^* \text{ “ } \{s\}$
 $\quad)$
 $\quad \{$
 $\quad \quad (\text{next}, i) = \text{pop}(\text{stack}, i);$ — Take the top most element from the
 $\quad \quad \text{stack.}$
 $\quad \quad \text{visited} = \text{bv-insert}(\text{next}, \text{visited});$ — Mark it as visited,
 $\quad \quad \text{if } (\text{next} == x) \{$
 $\quad \quad \quad b = 1$ — If it is the target, we are done.
 $\quad \quad \} \text{ else } \{$
 $\quad \quad \quad$ — Else, put its successors on the stack if they are not yet visited.
 $\quad \quad \quad \text{stop} = a[\text{next}];$
 $\quad \quad \quad j = \text{next} + 1;$
 $\quad \quad \quad \text{if } (j \leq \text{stop}) \{$

```

      inline get-succs
    }
  }
}
>
⟨proof⟩

```

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:  $s \subset \text{insert } x \ s \longleftrightarrow x \notin s$  ⟨proof⟩

program-spec dfs1
  assumes  $0 \leq x \wedge 0 \leq s \wedge \text{start} = s \wedge \text{edges} = a$ 
  ensures  $b = 1 \longleftrightarrow x \in (\text{Edges } a)^* \text{ “ } \{s\}$ 
  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  $\wedge$  i  $\neq$  0)
      @invariant <
         $0 \leq i \wedge (s \in \text{stack } \text{“ } \{0..<i\} \vee s \in \text{set-of visited}) \wedge (b = 0 \vee b =$ 
        1)  $\wedge$ 
         $\text{set-of visited} \subseteq (\text{Edges edges})^* \text{ “ } \{start\} \wedge ($ 
        if b = 0 then
           $\text{stack } \text{“ } \{0..<i\} \subseteq (\text{Edges } a)^* \text{ “ } \{s\}$ 
           $\wedge (\forall v \in \text{set-of visited. } (\text{Edges } a) \text{ “ } \{v\} \subseteq \text{set-of visited} \cup \text{stack } \text{“ } \{0..<i\})$ 
           $\wedge (x \notin \text{set-of visited})$ 
        else  $x \in (\text{Edges } a)^* \text{ “ } \{s\}$ 
      >
    >
    @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
      }
    }
  }
}
}
}
}
>
⟨proof⟩

end

end

```