IMP2 — Simple Program Verification in Isabelle/HOL

Peter Lammich Simon Wimmer

March 17, 2025

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

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* ⇒ *bool* **where**
  is-global [] ⟷ *True*
| *is-global* (*CHR* ''*G*''#-) ⟷ *True*
| *is-global* - ⟷ *False*

abbreviation *is-local a* ≡ ¬*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* ⇒ *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* (‹-[-] ::= -› [*1000*, *0*, *61*] *61*)
notation *ArrayCpy* (‹-[] ::= -› [*1000*, *1000*] *61*)
notation *ArrayClear* (‹*CLEAR* -[]› [*1000*] *61*)

notation *Seq* (‹-;;/ -› [*61*, *60*] *60*)
notation *If* (‹(*IF* -/ *THEN* -/ *ELSE* -)› [*0*, *0*, *61*] *61*)
notation *While* (‹(*WHILE* -/ *DO* -)› [*0*, *61*] *61*)
notation *Scope* (‹*SCOPE* -› [*61*] *61*)

1.5 Program

type-synonym *program* = *pname* ⇀ *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* ≡ *Vidx x* (*N 0*)
abbreviation *Assign* (‹- ::= -› [*1000*, *61*] *61*)
  where *x* ::= *a* ≡ (*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* ⇒ *val*

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

definition *null-state* (‹<>›) **where**
  null-state ≡ λ*x*. λ*i*. *0*
syntax
  -State :: *updbinds* => ′*a* (‹<->›)
translations
  -State ms == *-Update* <> *ms*
  -State (*-updbinds b bs*) <= *-Update* (*-State b*) *bs*

2.1.1 State Combination

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

definition *combine-states* :: *state* ⇒ *state* ⇒ *state* (‹<-|->› [*0*,*0*] *1000*)
  where <*s*|*t*> *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*: <*s*|*s*> = *s* ⟨*proof*⟩

lemma *combine-nest*:
  <*s*|<*s*′|*t*>> = <*s*|*t*>
  <<*s*|*t*′>|*t*> = <*s*|*t*>
  ⟨*proof*⟩

lemma *combine-query*:
  is-local x ⟹ <*s*|*t*> *x* = *s x*
  is-global x ⟹ <*s*|*t*> *x* = *t x*
  ⟨*proof*⟩

lemma *combine-upd*:
  $is\text{-}local\ x \Longrightarrow <s|t>(x{:=}v) = <s(x{:=}v)|t>$
  $is\text{-}global\ x \Longrightarrow <s|t>(x{:=}v) = <s|t(x{:=}v)>$
  ⟨*proof*⟩

lemma *combine-cases*[*cases type*]:
  obtains *l g* **where** $s = <l|g>$
  ⟨*proof*⟩

end

2.2 Arithmetic Expressions

The evaluation of arithmetic expressions is straightforward.

fun *aval* :: *aexp* ⇒ *state* ⇒ *pval* **where**
  aval (*N n*) *s* = *n*
| *aval* (*Vidx x i*) *s* = *s x* (*aval i s*)
| *aval* (*Unop f* a_1) *s* = *f* (*aval* a_1 *s*)
| *aval* (*Binop f* a_1 a_2) *s* = *f* (*aval* a_1 *s*) (*aval* a_2 *s*)

2.3 Boolean Expressions

The evaluation of Boolean expressions is straightforward.

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

2.4 Big-Step Semantics

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

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

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

inductive *big-step* :: *program* ⇒ *com* × *state* ⇒ *state* ⇒ *bool*
  (‹-: - ⇒ -› [*1000*,*55*,*55*] *55*)
where
  — No-Op
  Skip: π:(*SKIP*,*s*) ⇒ *s*

  — Assignments

| *AssignIdx*: π:(*x*[*i*] ::= *a*,*s*) ⇒ *s*(*x* := (*s x*)(*aval i s* := *aval a s*))
| *ArrayCpy*: π:(*x*[] ::= *y*,*s*) ⇒ *s*(*x* := *s y*)
| *ArrayClear*: π:(*CLEAR x*[],*s*) ⇒ *s*(*x* := (λ-. *0*))
| *Assign-Locals*: π:(*Assign-Locals l*,*s*) ⇒ <*l*|*s*>

— Block commands
| *Seq*: ⟦ π:(c_1,s_1) ⇒ s_2; π:(c_2,s_2) ⇒ s_3 ⟧ ⟹ π:(c_1;;c_2, s_1) ⇒ s_3
| *IfTrue*: ⟦ *bval b s*; π:(c_1,*s*) ⇒ *t* ⟧ ⟹ π:(*IF b THEN* c_1 *ELSE* c_2, *s*) ⇒ *t*
| *IfFalse*: ⟦ ¬*bval b s*; π:(c_2,*s*) ⇒ *t* ⟧ ⟹ π:(*IF b THEN* c_1 *ELSE* c_2, *s*) ⇒ *t*
| *Scope*: ⟦ π:(*c*,<<>|*s*>) ⇒ *s′* ⟧ ⟹ π:(*SCOPE c*, *s*) ⇒ <*s*|*s′*>
| *WhileFalse*: ¬*bval b s* ⟹ π:(*WHILE b DO c*,*s*) ⇒ *s*
| *WhileTrue*: ⟦ *bval b* s_1; π:(*c*,s_1) ⇒ s_2; π:(*WHILE b DO c*, s_2) ⇒ s_3 ⟧
⟹ π:(*WHILE b DO c*, s_1) ⇒ s_3

— Procedure commands
| *PCall*: ⟦ π *p* = *Some c*; π:(*c*,*s*) ⇒ *t* ⟧ ⟹ π:(*PCall p*,*s*) ⇒ *t*
| *PScope*: ⟦ π′:(*c*,*s*) ⇒ *t* ⟧ ⟹ π:(*PScope* π′ *c*, *s*) ⇒ *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*: π:(*SKIP*,*s*) ⇒ *t*
inductive-simps *AssignIdx-simp*: π:(*x*[*i*] ::= *a*,*s*) ⇒ *t*
inductive-simps *ArrayCpy-simp*: π:(*x*[] ::= *y*,*s*) ⇒ *t*
inductive-simps *ArrayInit-simp*: π:(*CLEAR x*[],*s*) ⇒ *t*
inductive-simps *AssignLocals-simp*: π:(*Assign-Locals l*,*s*) ⇒ *t*

inductive-simps *Seq-simp*: π:(*c1*;;*c2*,*s1*) ⇒ *s3*
inductive-simps *If-simp*: π:(*IF b THEN c1 ELSE c2*,*s*) ⇒ *t*
inductive-simps *Scope-simp*: π:(*SCOPE c*,*s*) ⇒ *t*
inductive-simps *PCall-simp*: π:(*PCall p*,*s*) ⇒ *t*
inductive-simps *PScope-simp*: π:(*PScope* π′ *p*,*s*) ⇒ *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*!]: π:(*SKIP*,*s*) ⇒ *t*
inductive-cases *AssignIdxE*[*elim*!]: π:(*x*[*i*] ::= *a*,*s*) ⇒ *t*
inductive-cases *ArrayCpyE*[*elim*!]: π:(*x*[] ::= *y*,*s*) ⇒ *t*
inductive-cases *ArrayInitE*[*elim*!]: π:(*CLEAR x*[],*s*) ⇒ *t*

inductive-cases *AssignLocalsE*[*elim*!]: π:(*Assign-Locals l*,*s*) ⇒ *t*

inductive-cases *SeqE*[*elim*!]: π:(*c1*;;*c2*,*s1*) ⇒ *s3*
inductive-cases *IfE*[*elim*!]: π:(*IF b THEN c1 ELSE c2*,*s*) ⇒ *t*
inductive-cases *ScopeE*[*elim*!]: π:(*SCOPE c*,*s*) ⇒ *t*
inductive-cases *PCallE*[*elim*!]: π:(*PCall p*,*s*) ⇒ *t*
inductive-cases *PScopeE*[*elim*!]: π:(*PScope* π′ *p*,*s*) ⇒ *t*

inductive-cases *WhileE*[*elim*]: π:(*WHILE b DO c*,*s*) ⇒ *t*

2.4.2 Automatic Derivation

lemma *Assign′*: $s' = s(x := (s\ x)(aval\ i\ s := aval\ a\ s)) \Longrightarrow$ π:(*x*[*i*] ::= *a*, *s*) ⇒ *s′* ⟨*proof*⟩
lemma *ArrayCpy′*: $s' = s(x := (s\ y)) \Longrightarrow$ π:(*x*[] ::= *y*, *s*) ⇒ *s′* ⟨*proof*⟩
lemma *ArrayClear′*: $s' = s(x := (\lambda\text{-}.\ 0)) \Longrightarrow$ π:(*CLEAR x*[], *s*) ⇒ *s′* ⟨*proof*⟩

lemma *Scope′*: s_1 = <<>|*s*> ⟹ π:(*c*,s_1) ⇒ *t* ⟹ *t′* = <*s*|*t*> ⟹ π:(*Scope c*,*s*) ⇒ *t′* ⟨*proof*⟩

named-theorems *deriv-unfolds* ‹*Unfold rules before derivations*›

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* (
  $''a'' ::= Binop\ (+)\ (V\ ''a'')\ (V\ ''a'');;$
  $''n'' ::= Binop\ (-)\ (V\ ''n'')\ (N\ 1)$
$),<''n'':=(\lambda\text{-}.\ 5)>) \Rightarrow ?s$
⟨*proof*⟩

2.5 Command Equivalence

Two commands are equivalent if they have the same semantics.

definition
  equiv-c :: *com* ⇒ *com* ⇒ *bool* (**infix** ‹∼› *50*) **where**
  $c \sim c' \equiv (\forall \pi\ s\ t.\ \pi{:}(c,s) \Rightarrow t = \pi{:}(c',s) \Rightarrow t)$

lemma *equivI*[*intro?*]: ⟦
  $\bigwedge s\ t\ \pi.\ \pi{:}(c,s) \Rightarrow t \Longrightarrow \pi{:}(c',s) \Rightarrow t;$
  $\bigwedge s\ t\ \pi.\ \pi{:}(c',s) \Rightarrow t \Longrightarrow \pi{:}(c,s) \Rightarrow t$⟧
  $\Longrightarrow c \sim c'$
  ⟨*proof*⟩

lemma *equivD*[*dest*]: $c \sim c' \Longrightarrow \pi{:}(c,s) \Rightarrow t \longleftrightarrow \pi{:}(c',s) \Rightarrow t$
  ⟨*proof*⟩

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

lemma *equiv-refl*[*simp*, *intro*!]: $c \sim c$
  ⟨*proof*⟩
lemma *equiv-sym*[*sym*]: $(c \sim c') \Longrightarrow (c' \sim c)$
  ⟨*proof*⟩
lemma *equiv-trans*[*trans*]: $c \sim c' \Longrightarrow c' \sim c'' \Longrightarrow c \sim c''$
  ⟨*proof*⟩

2.5.1 Basic Equivalences

lemma *while-unfold*:
  (*WHILE b DO c*) ∼ (*IF b THEN c*;; *WHILE b DO c ELSE SKIP*)
  ⟨*proof*⟩

lemma *triv-if*:
  (*IF b THEN c ELSE c*) ∼ *c*
  ⟨*proof*⟩

lemma *commute-if*:
  (*IF b1 THEN* (*IF b2 THEN c11 ELSE c12*) *ELSE c2*)
   ∼
   (*IF b2 THEN* (*IF b1 THEN c11 ELSE c2*) *ELSE* (*IF b1 THEN c12 ELSE c2*))
  ⟨*proof*⟩

lemma *sim-while-cong-aux*:

$[\![\pi{:}(WHILE\ b\ DO\ c{,}s) \Rightarrow t;\ bval\ b = bval\ b';\ c \sim c'\,]\!] \Longrightarrow \pi{:}(WHILE\ b'\ DO\ c'{,}s) \Rightarrow t$
⟨*proof*⟩

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

2.6 Execution is Deterministic

This proof is automatic.

theorem *big-step-determ*: $[\![\ \pi{:}(c{,}s) \Rightarrow t;\ \pi{:}(c{,}s) \Rightarrow u\ ]\!] \Longrightarrow u = t$
⟨*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* ⇒ *com* × *state* ⇀ *com* × *state* **where**
small-step π (*x*[*i*]::=*a*,*s*) = *Some* (*SKIP*, *s*(*x* := (*s x*)(*aval i s* := *aval a s*)))
| *small-step* π (*x*[]::=*y*,*s*) = *Some* (*SKIP*, *s*(*x* := *s y*))
| *small-step* π (*CLEAR x*[],*s*) = *Some* (*SKIP*, *s*(*x* := (λ-. *0*)))
| *small-step* π (*Assign-Locals l*,*s*) = *Some* (*SKIP*,<*l*|*s*>)
| *small-step* π (*SKIP*;;*c*,*s*) = *Some* (*c*,*s*)
| *small-step* π (c_1;;c_2,*s*) = (*case small-step* π (c_1,*s*) *of Some* (c_1',*s'*) ⇒ *Some* (c_1';;c_2,*s'*) | - ⇒ *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*;;*Assign-Locals s*, <<>|*s*>)

| *small-step* π (*WHILE b DO c*,*s*) = *Some* (*IF b THEN c*;;*WHILE b DO c ELSE SKIP*, *s*)
| *small-step* π (*PCall p*, *s*) = (*case* π *p of Some c* $\Rightarrow$ *Some* (*c*, *s*) | - $\Rightarrow$ *None*)
| *small-step* π (*PScope* π' *SKIP*, *s*) = *Some* (*SKIP*,*s*)
| *small-step* π (*PScope* π' *c*, *s*) = (*case small-step* π' (*c*,*s*) *of Some* (c',s') $\Rightarrow$ *Some* (*PScope* π' c', s') | - $\Rightarrow$ *None*)
| *small-step* π (*SKIP*,*s*) = *Some* (*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*]: *small-steps* π *cs* (*Some cs*)
| ⟦ *small-step* π *cs* = *None* ⟧ $\Longrightarrow$ *small-steps* π *cs None*
| ⟦ *small-step* π *cs* = *Some cs1*; *small-steps* π *cs1 cs2* ⟧ $\Longrightarrow$ *small-steps* π *cs cs2*

lemma *small-steps-append*: *small-steps* π cs_1 (*Some* cs_2) $\Longrightarrow$ *small-steps* π cs_2 cs_3 $\Longrightarrow$ *small-steps* π cs_1 cs_3
⟨*proof*⟩

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*: *small-step* π cs_1 = *Some* cs_2 $\Longrightarrow$ π: cs_2 $\Rightarrow$ s_3 $\Longrightarrow$ π: cs_1 $\Rightarrow$ s_3
⟨*proof*⟩

lemma *smalls-big-append*: *small-steps* π cs_1 (*Some* cs_2) $\Longrightarrow$ π: cs_2 $\Rightarrow$ s_3 $\Longrightarrow$ π: cs_1 $\Rightarrow$ s_3
⟨*proof*⟩

lemma *small-imp-big*:
assumes *small-steps* π cs_1 (*Some* (*SKIP*,s_2))
shows π: cs_1 $\Rightarrow$ s_2
⟨*proof*⟩

lemma *small-steps-skip-term*[*simp*]: *small-steps* π (*SKIP*, *s*) cs' $\longleftrightarrow$ cs'=*Some* (*SKIP*,*s*)
⟨*proof*⟩

lemma *small-seq*: ⟦$c\neq$*SKIP*; *small-step* π (*c*,*s*) = *Some* (c',s')⟧ $\Longrightarrow$ *small-step* π (*c*;;*cx*,*s*) = *Some* (c';;*cx*,s')
⟨*proof*⟩

lemma *smalls-seq*: $[\![$*small-steps* π (c,s) $(Some\ (c',s'))]\!] \Longrightarrow$ *small-steps* π $(c;;cx,s)$ $(Some\ (c';;cx,s'))$
⟨*proof*⟩

lemma *small-pscope*:
$[\![c \neq SKIP;$ *small-step* π' $(c,s) = Some\ (c',s')]\!] \Longrightarrow$ *small-step* π $(PScope\ \pi'\ c,s) = Some\ (PScope\ \pi'\ c',s')$
⟨*proof*⟩

lemma *smalls-pscope*:
small-steps π' $(c,\ s)$ $(Some\ (c',\ s')) \Longrightarrow$ *small-steps* π $(PScope\ \pi'\ c,\ s)$ $(Some\ (PScope\ \pi'\ c',s'))$
⟨*proof*⟩

lemma *big-imp-small*:
assumes π: $cs \Rightarrow t$
shows *small-steps* π *cs* $(Some\ (SKIP,t))$
⟨*proof*⟩

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*: π: $cs \Rightarrow t \longleftrightarrow$ *small-steps* π *cs* $(Some\ (SKIP,t))$
⟨*proof*⟩

lemma *small-steps-determ*:
assumes *small-steps* π *cs None*
shows $\neg$*small-steps* π *cs* $(Some\ (SKIP,\ t))$
⟨*proof*⟩

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

corollary *small-imp-big-fail*:
assumes *small-steps* π *cs None*
shows $\nexists t.\ \pi$: $cs \Rightarrow t$
⟨*proof*⟩

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 π :: *program*
begin

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

definition $wp\ c\ Q\ s \equiv \exists t.\ \pi\colon (c,s) \Rightarrow t \wedge Q\ t$
— 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-imp-wlp*: $wp\ c\ Q\ s \Longrightarrow wlp\ c\ Q\ s$
⟨*proof*⟩

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

lemma *wp-equiv*: $c \sim c' \Longrightarrow wp\ c = wp\ c'$ ⟨*proof*⟩
lemma *wp-conseq*: $wp\ c\ P\ s \Longrightarrow [\![\bigwedge s.\ P\ s \Longrightarrow Q\ s]\!] \Longrightarrow wp\ c\ Q\ s$ ⟨*proof*⟩

lemma *wlp-equiv*: $c \sim c' \Longrightarrow wlp\ c = wlp\ c'$ ⟨*proof*⟩
lemma *wlp-conseq*: $wlp\ c\ P\ s \Longrightarrow [\![\bigwedge s.\ P\ s \Longrightarrow Q\ s]\!] \Longrightarrow wlp\ c\ Q\ s$ ⟨*proof*⟩

2.8.2 Unfold Rules

lemma *wp-skip-eq*: $wp\ SKIP\ Q\ s = Q\ s$ ⟨*proof*⟩
lemma *wp-assign-idx-eq*: $wp\ (x[i]{::=}a)\ Q\ s = Q\ (s(x{:=}(s\ x)(aval\ i\ s := aval\ a\ s)))$ ⟨*proof*⟩
lemma *wp-arraycpy-eq*: $wp\ (x[]{::=}a)\ Q\ s = Q\ (s(x{:=}s\ a))$ ⟨*proof*⟩
lemma *wp-arrayinit-eq*: $wp\ (CLEAR\ x[])\ Q\ s = Q\ (s(x{:=}(\lambda\text{-}.\ 0)))$ ⟨*proof*⟩
lemma *wp-assign-locals-eq*: $wp\ (Assign\text{-}Locals\ l)\ Q\ s = Q <l|s>$ ⟨*proof*⟩
lemma *wp-seq-eq*: $wp\ (c_1;;c_2)\ Q\ s = wp\ c_1\ (wp\ c_2\ Q)\ s$ ⟨*proof*⟩
lemma *wp-if-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-scope-eq*: $wp\ (SCOPE\ c)\ Q\ s = wp\ c\ (\lambda s'.\ Q <s|s'>)\ <<>|s>$ ⟨*proof*⟩
lemma *wp-pcall-eq*: $\pi\ p = Some\ c \Longrightarrow wp\ (PCall\ p)\ Q\ s = wp\ c\ Q\ s$ ⟨*proof*⟩

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

wp-assign-locals-eq wp-seq-eq wp-scope-eq
lemmas *wp-eq'* = *wp-eq wp-if-eq*

lemma *wlp-skip-eq*: *wlp SKIP Q s* = *Q s* ⟨*proof*⟩
lemma *wlp-assign-idx-eq*: $wlp\ (x[i]::=a)\ Q\ s = Q\ (s(x:=(s\ x)(aval\ i\ s := aval\ a\ s)))$ ⟨*proof*⟩
lemma *wlp-arraycpy-eq*: $wlp\ (x[]::=a)\ Q\ s = Q\ (s(x:=s\ a))$ ⟨*proof*⟩
lemma *wlp-arrayinit-eq*: $wlp\ (CLEAR\ x[])\ Q\ s = Q\ (s(x:=(\lambda\text{-}.\ 0)))$ ⟨*proof*⟩
lemma *wlp-assign-locals-eq*: $wlp\ (Assign\text{-}Locals\ l)\ Q\ s = Q\ <l|s>$ ⟨*proof*⟩
lemma *wlp-seq-eq*: $wlp\ (c_1;;c_2)\ Q\ s = wlp\ c_1\ (wlp\ c_2\ Q)\ s$ ⟨*proof*⟩
lemma *wlp-if-eq*: $wlp\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ Q\ s$
$= (if\ bval\ b\ s\ then\ wlp\ c_1\ Q\ s\ else\ wlp\ c_2\ Q\ s)$ ⟨*proof*⟩

lemma *wlp-scope-eq*: $wlp\ (SCOPE\ c)\ Q\ s = wlp\ c\ (\lambda s'.\ Q\ <s|s'>)\ <<>|s>$ ⟨*proof*⟩
lemma *wlp-pcall-eq*: $\pi\ p = Some\ c \Longrightarrow wlp\ (PCall\ p)\ Q\ s = wlp\ c\ Q\ s$ ⟨*proof*⟩

lemmas *wlp-eq* = *wlp-skip-eq wlp-assign-idx-eq wlp-arraycpy-eq wlp-arrayinit-eq*
wlp-assign-locals-eq wlp-seq-eq wlp-scope-eq
lemmas *wlp-eq'* = *wlp-eq wlp-if-eq*
end

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

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

end — Context fixing program

Unfold rules for procedure scope

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

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

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

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

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 π: $(WHILE\ b\ DO\ c,\ s) \Rightarrow t$
shows π: $(WHILE\ b'\ DO\ c',\ s) \Rightarrow t$
⟨*proof*⟩

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

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 π: $(WHILE\ b\ DO\ c,\ s) \Rightarrow t$
shows π: $(WHILE\ b'\ DO\ c',\ s) \Rightarrow t$
⟨*proof*⟩

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

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*: $\bigwedge s.\ I\ s \Longrightarrow (\mathit{if}\ bval\ b\ s\ \mathit{then}\ wlp\ \pi\ c\ I\ s\ \mathit{else}\ Q\ s)$
shows $wlp\ \pi\ (WHILE\ b\ DO\ c)\ Q\ s_0$
⟨*proof*⟩

lemma
assumes *INIT*: $I\ s_0$
assumes *STEP*: $\bigwedge s.\ I\ s \Longrightarrow (\mathit{if}\ bval\ b\ s\ \mathit{then}\ wlp\ \pi\ c\ I\ s\ \mathit{else}\ Q\ s)$
shows $wlp\ \pi\ (WHILE\ b\ DO\ c)\ Q\ s_0$
⟨*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*: *wf R*
assumes *INIT*: $I\ s_0$
assumes *STEP*: $\bigwedge s.\ I\ s \Longrightarrow (\mathit{if}\ bval\ b\ s\ \mathit{then}\ wp\ \pi\ c\ (\lambda s'.\ I\ s' \wedge (s',s){\in}R)\ s\ \mathit{else}\ Q\ s)$
shows $wp\ \pi\ (WHILE\ b\ DO\ c)\ Q\ s_0$
⟨*proof*⟩

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

2.9.3 Standard Forms of While Rules

lemma *wlp-whileI*:
assumes *INIT*: $I\ \mathfrak{s}_0$
assumes *STEP*: $\bigwedge \mathfrak{s}.\ [\![I\ \mathfrak{s};\ bval\ b\ \mathfrak{s}\ ]\!] \Longrightarrow wlp\ \pi\ c\ I\ \mathfrak{s}$
assumes *FINAL*: $\bigwedge \mathfrak{s}.\ [\![\ I\ \mathfrak{s};\ \neg bval\ b\ \mathfrak{s}\ ]\!] \Longrightarrow Q\ \mathfrak{s}$
shows $wlp\ \pi\ (WHILE\ b\ DO\ c)\ Q\ \mathfrak{s}_0$
⟨*proof*⟩

lemma *wp-whileI*:
assumes *WF*: *wf R*
assumes *INIT*: $I\ \mathfrak{s}_0$

assumes *STEP*: $\bigwedge\mathfrak{s}$. $[\![I\ \mathfrak{s};\ bval\ b\ \mathfrak{s}\ ]\!] \Longrightarrow wp\ \pi\ c\ (\lambda\mathfrak{s}'.\ I\ \mathfrak{s}' \wedge$ $(\mathfrak{s}',\mathfrak{s}){\in}R)\ \mathfrak{s}$
assumes *FINAL*: $\bigwedge\mathfrak{s}$. $[\![\ I\ \mathfrak{s};\ \neg bval\ b\ \mathfrak{s}\ ]\!] \Longrightarrow Q\ \mathfrak{s}$
shows $wp\ \pi\ (WHILE\ b\ DO\ c)\ Q\ \mathfrak{s}_0$
⟨*proof*⟩

2.10 Modularity of Programs

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

lemma *map-leD*: $m{\subseteq_m}m' \Longrightarrow m\ x = Some\ v \Longrightarrow m'\ x = Some\ v$
⟨*proof*⟩

lemma *big-step-mono-prog*:
assumes $\pi \subseteq_m \pi'$
assumes $\pi{:}(c,s) \Rightarrow t$
shows $\pi'{:}(c,s) \Rightarrow t$
⟨*proof*⟩

Wrapping a set of recursive procedures into a procedure scope

lemma *localize-recursion*:
π': $(PScope\ \pi\ c,\ s) \Rightarrow t \longleftrightarrow \pi{:}(c,s) \Rightarrow t$
⟨*proof*⟩

2.11 Strongest Postcondition

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

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

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

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

Version with renaming of assigned variable

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

lemma *sp-skip-eq*: $sp\ P\ SKIP\ t \longleftrightarrow P\ t$ ⟨*proof*⟩
lemma *sp-seq-eq*: $sp\ P\ (c_1;;c_2)\ t \longleftrightarrow sp\ (sp\ P\ c_1)\ c_2\ t$ ⟨*proof*⟩

end
end

2.12 Hoare-Triples

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

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

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

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

lemma *HT-conseq*:
  assumes $HT\ \pi\ P\ c\ Q$
  assumes $\bigwedge s.\ P'\ s \Longrightarrow P\ s$
  assumes $\bigwedge s_0\ s.\ [\![P\ s_0;\ P'\ s_0;\ Q\ s_0\ s\ ]\!] \Longrightarrow Q'\ s_0\ s$
  shows $HT\ \pi\ P'\ c\ Q'$
  ⟨*proof*⟩

lemma *HT-partial-conseq*:
  assumes $HT\text{-}partial\ \pi\ P\ c\ Q$
  assumes $\bigwedge s.\ P'\ s \Longrightarrow P\ s$
  assumes $\bigwedge s_0\ s.\ [\![P\ s_0;\ P'\ s_0;\ Q\ s_0\ s\ ]\!] \Longrightarrow Q'\ s_0\ s$
  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*:
  $[\![HT\ \pi\ P\ c\ Q;\ P\ s;\ (\bigwedge s'.\ Q\ s\ s' \Longrightarrow Q'\ s')]\!] \Longrightarrow wp\ \pi\ c\ Q'\ s$
  ⟨*proof*⟩

2.12.1 Sets of Hoare-Triples

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

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

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

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

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 $HT\ \pi\ P\ body\ Q$
 assumes $TERMINATES$: $\forall s.\ \exists t.\ \pi$: $(epilogue,s) \Rightarrow t$
 shows $HT\ \pi\ P\ (body;;epilogue)\ (\lambda s_0.\ sp\ \pi\ (Q\ s_0)\ epilogue)$
 $\langle proof \rangle$

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 *locals*:=<> and the prologue *locals*:=*old-locals*.

lemma *adjust-scope*:
 assumes $HT\ \pi\ P\ body\ Q$
 shows $HT\ \pi\ (\lambda s.\ P\ {<}{<}{>}|s{>})\ (SCOPE\ body)\ (\lambda s_0\ s.\ \exists l.\ Q\ ({<}{<}{>}|s_0{>})\ ({<}l|s{>}))$
 $\langle proof \rangle$

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: $\bigwedge P\ c\ Q\ s_0.\ [\![\ HTset\text{-}r\ (\lambda c'\ s'.\ ((c',s'),(c,s_0)) \in R\ )\ \pi\ \Theta;\ (P,c,Q) \in \Theta;\ P\ s_0\ ]\!] \Longrightarrow wp\ \pi\ c\ (Q\ s_0)\ s_0$
 shows $HTset\ \pi\ \Theta$
 $\langle proof \rangle$

lemma *HT-simple-recursiveI*:
 assumes $wf\ R$
 assumes $\bigwedge s.\ [\![HT\ \pi\ (\lambda s'.\ (f\ s',\ f\ s) \in R \wedge P\ s')\ c\ Q;\ P\ s\ ]\!] \Longrightarrow wp\ \pi\ c\ (Q\ s)\ s$
 shows $HT\ \pi\ P\ c\ Q$

$\langle proof \rangle$

lemma *HT-simple-recursive-procI*:
assumes *wf R*
assumes $\bigwedge s.\ [\![HT\ \pi\ (\lambda s'.\ (f\ s',\ f\ s){\in}R \wedge P\ s')\ (PCall\ p)\ Q;\ P\ s\ ]\!]$
$\Longrightarrow wp\ \pi\ (PCall\ p)\ (Q\ s)\ s$
shows $HT\ \pi\ P\ (PCall\ p)\ Q$
$\langle proof \rangle$

lemma
assumes *wf R*
assumes $\bigwedge s\ P\ p\ Q.\ [\![$
$\bigwedge P'\ p'\ Q'.\ (P',p',Q'){\in}\Theta$
$\Longrightarrow HT\ \pi\ (\lambda s'.\ ((p',s'),(p,s)){\in}R \wedge P'\ s')\ (PCall\ p')\ Q';$
$(P,p,Q){\in}\Theta;\ P\ s$
$]\!] \Longrightarrow wp\ \pi\ (PCall\ p)\ (Q\ s)\ s$
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**
$I\ s_0$
$\bigwedge s.\ I\ s \Longrightarrow if\ bval\ b\ s\ then\ wlp\ \pi\ c\ I\ s\ else\ Q\ s$
$\langle proof \rangle$

Idea: Remaining loop iterations as variant

inductive *count-it* **for** π *b c* **where**
$\neg bval\ b\ s \Longrightarrow count\text{-}it\ \pi\ b\ c\ s\ 0$
$|\ [\![bval\ b\ s;\ \pi\colon (c,s) \Rightarrow s';\ count\text{-}it\ \pi\ b\ c\ s'\ n\ ]\!] \Longrightarrow count\text{-}it\ \pi\ b\ c\ s$
$(Suc\ n\ )$

lemma *count-it-determ*:
$count\text{-}it\ \pi\ b\ c\ s\ n \Longrightarrow count\text{-}it\ \pi\ b\ c\ s\ n' \Longrightarrow n' = n$
$\langle proof \rangle$

lemma *count-it-ex*:
assumes $\pi\colon (WHILE\ b\ DO\ c,s) \Rightarrow t$
shows $\exists n.\ count\text{-}it\ \pi\ b\ c\ s\ n$
$\langle proof \rangle$

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

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

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
    $I\ s_0$
    $\bigwedge s.\ I\ s \Longrightarrow if\ bval\ b\ s\ then\ wp\ \pi\ c\ (\lambda s'.\ I\ s' \wedge (s',s) \in R)\ s\ else\ Q\ s$
⟨*proof*⟩

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* ‹*Definitions of Annotations*›

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* ⇒ *bool*) ⇒ *bexp* ⇒ *com* ⇒ *com*

(‹(*WHILE* {-} -/ *DO* -)› [*0*, *0*, *61*] *61*)
where [*vcg-annotation-defs*]: *WHILE-annotI* (*I*::*state* ⇒ *bool*) ≡ *While*

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

definition *WHILE-annotRVI* :: *'a rel* ⇒ (*state* ⇒ *'a*) ⇒ (*state* ⇒ *bool*) ⇒ *bexp* ⇒ *com* ⇒ *com*
(‹(*WHILE* {-} {-} {-} -/ *DO* -)› [*0*, *0*, *0*, *0*, *61*] *61*)
where [*vcg-annotation-defs*]: *WHILE-annotRVI R V I* ≡ *While* **for** *R V I*

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

definition *WHILE-annotVI* :: (*state* ⇒ *int*) ⇒ (*state* ⇒ *bool*) ⇒ *bexp* ⇒ *com* ⇒ *com*
(‹(*WHILE* {-} {-} -/ *DO* -)› [*0*, *0*, *0*, *61*] *61*)
where [*vcg-annotation-defs*]: *WHILE-annotVI V I* ≡ *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* ⇒ (*state* ⇒ *bool*) ⇒ (*state* ⇒ *com*) ⇒ (*state* ⇒ *state*⇒*bool*) ⇒ *bool*

definition *HT'-partial* :: *HT'-type*
where *HT'-partial* π *P c Q* ≡ (∀ s_0. *P* s_0 ⟶ *wlp* π (*c* s_0) (*Q* s_0) s_0)

definition *HT'* :: *HT'-type*
where *HT'* π *P c Q* ≡ (∀ s_0. *P* s_0 ⟶ *wp* π (*c* s_0) (*Q* s_0) s_0)

lemma *HT'-eq-HT*: *HT'* π *P* (λ-. *c*) *Q* = *HT* π *P c Q*
⟨*proof*⟩

lemma *HT'-partial-eq-HT*: *HT'-partial* π *P* (λ-. *c*) *Q* = *HT-partial* π *P c Q*
⟨*proof*⟩

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

type-synonym *'a* Θ*elem-t* = (*state*⇒*'a*) × ((*state* ⇒ *bool*) × (*state* ⇒ *com*) × (*state* ⇒ *state*⇒*bool*))

definition *HT'set* :: *program* ⇒ *'a* Θ*elem-t set* ⇒ *bool* **where** *HT'set* π Θ ≡ ∀ (*n*,(*P*,*c*,*Q*))∈Θ. *HT'* π *P c Q*

definition *HT'set-r* :: - ⇒ *program* ⇒ *'a* Θ*elem-t set* ⇒ *bool* **where** *HT'set-r r* π Θ ≡ ∀ (*n*,(*P*,*c*,*Q*))∈Θ. *HT'* π (λ*s*. *r n s* ∧ *P s*) *c Q*

lemma $HT'setI$:
  assumes $wf\ R$
  assumes RL: $\bigwedge f\ P\ c\ Q\ s_0.\ [\![\ HT'set\text{-}r\ (\lambda f'\ s'.\ ((f'\ s'),(f\ s_0)){\in}R\ )\ \pi\ \Theta;\ (f,(P,c,Q)){\in}\Theta;\ P\ s_0\ ]\!] \Longrightarrow wp\ \pi\ (c\ s_0)\ (Q\ s_0)\ s_0$
  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 ‹$b{\neq}0$›
  ensures ‹$c{=}a_0\ div\ b_0$›
  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* $(x_1,...,x_n)$.

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* π *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** ‹$a \neq 0$› **ensures** ‹$r=1$› **defines** ‹r = *div-ab*(a,a)› ⟨*proof*⟩

4.1.1 Variant and Invariant Annotations

Loops must be annotated with variants and invariants.

procedure-spec *mult-ab*(a,b) **returns** c **assumes** ‹*True*› **ensures** $c=a_0*b_0$
defines ‹
  if ($a<0$) {$a = -a$; $b = -b$};
  $c=0$;
  while ($a>0$)
    @*variant* ‹a›
    @*invariant* ‹$0 \leq a \wedge a \leq |a_0| \wedge c = (\ |a_0| - a) * b_0 * sgn\ a_0$›
  {

```
    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≥0 ensures ‹b≠0 ⟷ odd n0›
variant ‹n›
   defines ‹if (n==0) b=0 else b=rec even-imp(n−1)›
 and
   even-imp(n) returns b assumes n≥0 ensures ‹b≠0 ⟷ even
n0› variant ‹n›
   defines ‹if (n==0) b=1 else b=rec odd-imp(n−1)›
   ⟨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** ‹*True*› **ensures** $c=a_0*b_0$
defines ‹
  if $(a<0)$ $\{a = -a;\ b = -b\}$;
  $c=0$;
  while $(a>0)$
    @*variant* ‹a›
    @*invariant* ‹$0\leq a \wedge a\leq|a_0| \wedge c = (\ |a_0| - a) * b_0 * sgn\ a_0$›
  {
    $c=c+b$;
    $a=a-1$
  }
›
⟨*proof*⟩

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

recursive-spec relation ‹*measure nat*›
  odd-imp′(n) **returns** b **assumes** $n\geq 0$ **ensures** ‹$b\neq 0 \longleftrightarrow odd\ n_0$›
variant ‹n›
  defines ‹*if* $(n==0)$ $b=0$ *else* $b=rec\ even\text{-}imp'(n-1)$›
and

even-imp′(n) **returns** b **assumes** $n \geq 0$ **ensures** ‹$b \neq 0 \longleftrightarrow$ *even* n_0› **variant** ‹n›
defines ‹*if* $(n{=}{=}0)$ $b{=}1$ *else* $b{=}$*rec odd-imp′*$(n-1)$›
⟨*proof*⟩

4.3.2 Partial Correctness

IMP2 supports partial correctness proofs only for while-loops. Recursive procedures must always be proved totally correct[1]

procedure-spec (*partial*) *nonterminating*() **returns** a **assumes** *True* **ensures** ‹$a{=}0$› **defines**
‹*while* $(a \neq 0)$ @*invariant* ‹*True*›
$a{=}a-1$›
⟨*proof*⟩

4.3.3 Arrays

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

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

lemma *array-sum-aux*: $l_0 \leq l \Longrightarrow \{l_0..<l + 1\} = insert\ l\ \{l_0..<l\}$ **for** $l_0\ l$ :: *int* ⟨*proof*⟩

procedure-spec *array-sum*(a,l,h) **returns** s **assumes** $l \leq h$ **ensures** ‹$s = (\sum i{=}l_0..<h_0.\ a_0\ i)$› **defines**
‹ $s{=}0$;
while $(l<h)$
@*variant* ‹$h-l$›
@*invariant* ‹$l_0 \leq l \wedge l \leq h \wedge s = (\sum i{=}l_0..<l.\ a\ i)$›
{ $s = s+a[l]$; $l{=}l+1$ } ›
⟨*proof*⟩

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

[1] Adding 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* = *x1*$_0$+*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* ⇒ *val* | *ARRAY int* ⇒ *val*

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

4.6.4 Function Calls as Expressions

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

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

4.6.5 Ghost Variables

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

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

4.6.6 Concurrency

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

4.6.7 Pointers and Memory

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

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

end

5 Introduction to IMP2-VCG, based on IMP

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

This document briefly introduces the extensions of IMP2 over IMP.

5.1 Fancy Syntax

Standard Syntax

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

Fancy Syntax

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

lemma *exp-count-up1* = *exp-count-up2*
⟨*proof*⟩

5.2 Operators and Arrays

We reflect arbitrary Isabelle functions into the syntax:

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

thm *aval.simps bval.simps*

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

term ‹*Vidx* ''a'' (*i*::*aexp*)› — Array access at index *i*
lemma *V* ''x'' = *Vidx* ''x'' (*N 0*) ⟨*proof*⟩

New commands:

term ‹*AssignIdx* ''a'' (*i*::*aexp*) (*v*::*aexp*)› — Assign at index. Replaces assign.
term ‹''a''[*i*] ::= *v*› — Standard syntax
term ‹**imp**‹ *a*[*i*] = *v* ›› — Fancy syntax

lemma ‹*Assign* ''x'' *v* = *AssignIdx* ''x'' (*N 0*) *v*› ⟨*proof*⟩
term ‹''x'' ::= *v*› **term** ‹**imp**‹*x* = *v*+*1*››

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 ‹*ArrayCpy ''d'' ''s''*› — Copy whole array. Both operands are variable names.
term ‹*''d''*[] ::= *''s''*› **term** ‹**imp**‹*d* = *s*››

term ‹*ArrayClear ''a''*› — Initialize array to all zeroes.
term ‹*CLEAR ''a''*[]› **term** ‹**imp**‹*clear a*[]››

Semantics of these is straightforward

thm *big-step.AssignIdx big-step.ArrayCpy big-step.ArrayClear*

5.3 Local and Global Variables

term ‹*is-global*› **term** ‹*is-local*› — Partitions variable names
term ‹$<s_1|s_2>$› — State with locals from s_1 and globals from s_2

term ‹*SCOPE c*› **term** ‹**imp**‹*scope* { *skip* }›› — Execute *c* with fresh set of local variables
thm *big-step.Scope*

5.3.1 Parameter Passing

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

context fixes *f* :: *com* **begin**
term ‹**imp**‹ (*r1*,*r2*) = *f*(*x1*,*x2*,*x3*)››
end

5.4 Recursive procedures

term ‹*PCall ''name''*›
thm *big-step.PCall*

5.4.1 Procedure Scope

Execute command with local set of procedures

term ‹*PScope* π *c*›
thm *big-step.PScope*

5.4.2 Syntactic sugar for procedure call with parameters

term ‹**imp**‹(*r1*,*r2*) = *rec name*(*x1*,*x2*,*x3*)››

5.5 More Readable VCs

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

lemma s_0 *''n'' 0* $\geq$ *0* $\Longrightarrow$ *wlp* π *exp-count-up1* (λ*s. s ''a'' 0* = *2^nat* (s_0 *''n'' 0*)) s_0
⟨*proof*⟩

lemma s_0 *''n'' 0* $\geq$ *0* $\Longrightarrow$ *wlp* π *exp-count-up1* (λ*s. s ''a'' 0* = *2^nat* (s_0 *''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 *''n'' 0 in n* $\geq$ *0*
$\Longrightarrow$ *wlp* π *exp-count-up1* (λ*s. let a* = *s ''a'' 0*; n_0 = s_0 *''n'' 0 in* *a* = *2^nat* (n_0)) s_0
⟨*proof*⟩

lemma *VAR* (*s x*) *P* = (*let v=s x in P v*) ⟨*proof*⟩

IMP2 specification commands

program-spec (*partial*) *exp-count-up*
assumes *0≤n* — Precondition. Use variable names of program.
ensures *a* = *2^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 ‹n=n₀ ∧ a=2^nat c ∧ 0≤c ∧ c≤n› — Invar annotation. Variable names and suffix ·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≤n
  ensures a = 2^nat n₀
  defines
  ‹
    a = 1;
    c = 0;
    while (c<n)
      @invariant ‹n=n₀ ∧ a=2^nat c ∧ 0≤c ∧ c≤n›
      @variant ‹n−c›
    {
      a=2∗a;
      c=c+1
    }
  ›
  ⟨proof⟩
```

Simple Recursion

```
recursive-spec
  exp-rec(n) returns a assumes 0≤n ensures a=2^nat n₀ variant
n
  defines ‹if (n==0) a=1 else {t=rec 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-distribs = 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≤n
```

```
  ensures a = 2^nat n₀
  defines ‹
    a = 1;
    c = 0;
    while (c<n)
      @variant ‹n−c›
      @invariant ‹0≤c ∧ c≤n ∧ a=2^nat c›
    {
      G-par = a;   scope { a = G-par; a=2∗a; G-ret = a }; a = G-ret;
      c=c+1
    }
  ›
  ⟨proof⟩

program-spec sum-prog
  assumes n ≥ 0 ensures s = ∑{0..n₀}
  defines ‹
    s = 0;
    i = 0;
    while (i < n)
      @variant ‹n₀ − i›
      @invariant ‹n₀ = n ∧ 0 ≤ i ∧ i ≤ n ∧ s = ∑{0..i}›
    {
      i = i + 1;
      s = s + i
    }
  ›
  ⟨proof⟩

program-spec sq-prog
  assumes n ≥ 0 ensures a = n₀ ∗ n₀
  defines ‹
    a = 0;
    z = 1;
    i = 0;
    while (i < n)
      @variant ‹n₀ − i›
      @invariant ‹n₀ = n ∧ 0 ≤ i ∧ i ≤ n ∧ a = i ∗ i ∧ z = 2 ∗ i +
1›
    {
      a = a + z;
      z = z + 2;
      i = i + 1
    }
  ›
  ⟨proof⟩

fun factorial :: int ⇒ int where
  factorial i = (if i ≤ 0 then 1 else i ∗ factorial (i − 1))
```

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

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

lemma *fib-simps*[*simp*]:
  $i \leq 0 \Longrightarrow fib\ i = 0$
  $i = 1 \Longrightarrow fib\ i = 1$
  $i > 1 \Longrightarrow fib\ i = fib\ (i - 2) + fib\ (i - 1)$
  ⟨*proof*⟩

lemmas [*simp del*] = *fib.simps*

With precondition

program-spec *fib-prog*
  assumes $n \geq 0$ **ensures** $a = fib\ n$
  defines ‹
    $a = 0$; $b = 1$;
    $i = 0$;
    while $(i < n)$
      @*variant* ‹$n_0 - i$›
      @*invariant* ‹$n = n_0 \wedge 0 \leq i \wedge i \leq n \wedge a = fib\ i \wedge b = fib\ (i + 1)$›
    {
      $c = b$;
      $b = a + b$;
      $a = c$;
      $i = i + 1$
    }
  ›
  ⟨*proof*⟩

Without precondition, returning *0* for negative numbers

```
program-spec fib-prog'
  assumes True ensures a = fib n₀
  defines ‹
    a = 0; b = 1;
    i = 0;
    while (i < n)
      @variant ‹n₀ − i›
      @invariant ‹n = n₀ ∧ (0 ≤ i ∧ i ≤ n ∨ n₀ < 0 ∧ i = 0) ∧ a =
fib i ∧ b = fib (i+ 1)›
    {
      c = b;
      b = a + b;
      a = c;
      i = i + 1
    }
  ›
  ⟨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≤n
  ensures a = 2^nat n₀
  defines ‹
    a = 1;
    while (n>0)
      @variant ‹n›
      @invariant ‹let c = n₀−n in 0≤n ∧ n≤n₀ ∧ a=2^nat c›
    {
      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≤n
  ensures 0≤r ∧ r² ≤ n₀ ∧ n₀ < (r+1)²
  defines ‹
    r=1;
    while (r*r ≤ n)
      @variant ‹n + 1 − r*r›
      @invariant ‹0≤r ∧ (r−1)² ≤ n₀›
      { r = r + 1 };
    r = r − 1
  ›
  ⟨proof⟩
```

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 halfs 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 ⋀h l n₀ :: int.
      ⟦¶''invar−final''; 0 ≤ n₀; ¬ 1 + l < h; 0 ≤ l; l < h; l * l ≤ n₀;
       n₀ < h * h⟧
      ⟹ n₀ < 1 + (l * l + l * 2)
  ⟨proof⟩
```

```
program-spec sqr-bisect
  assumes 0≤n ensures r²≤n₀ ∧ n₀<(r+1)²
  defines ‹
    l=0; h=n+1;
    while (l+1 < h)
      @variant ‹h−l›
      @invariant ‹0≤l ∧ l<h ∧ l²≤n ∧ n < h²›
    {
      m = (l + h) / 2;
      if (m*m ≤ n) l=m else h=m
    };
    r=l
  ›
```

⟨*proof*⟩

6.2 Debugging

6.2.1 Testing Programs

Stepwise

schematic-goal *Map.empty*: (*sqr-approx-below*,<''*n*'':=λ-. *4*>) ⇒ *?s*
⟨*proof*⟩

Or all steps at once

schematic-goal *Map.empty*: (*sqr-bisect*,<''*n*'':=λ-. *4900000001*>) ⇒ *?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 ∧ b>0
  ensures a = gcd a₀ b₀
  defines ‹
    while (a≠b)
      @invariant ‹gcd a b = gcd a₀ b₀ ∧ (a>0 ∧ b>0)›
      @variant ‹a+b›
    {
      if (a<b) b = b−a
      else a = a−b
    }
  ›
  ⟨proof⟩
```

6.3.2 Euclid's Algorithm (with mod)

thm *gcd-red-int*[*symmetric*]

```
program-spec euclid2
  assumes a>0 ∧ b>0
  ensures a = gcd a₀ b₀
  defines ‹
    while (b≠0)
      @invariant ‹gcd a b = gcd a₀ b₀ ∧ b≥0 ∧ a>0›
      @variant ‹b›
    {
      t = a;
      a = b;
```

    $b = t \; mod \; b$
  }
›
⟨*proof*⟩

6.3.3 Extended Euclid's Algorithm

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

lemma *aux2*:
  fixes a b :: *int*
  assumes $b = t * b_0 + s * a_0$ $q = a \; div \; b$ $gcd \; a \; b = gcd \; a_0 \; b_0$
  shows $gcd \; b \; (a - (a_0 * (s * q) + b_0 * (t * q))) = gcd \; a_0 \; b_0$
⟨*proof*⟩

lemma *aux3*:
  fixes a b :: *int*
  assumes $b = t * b_0 + s * a_0$ $q = a \; div \; b$ $b > 0$
  shows $t * (b_0 * q) + s * (a_0 * q) \leq a$
⟨*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 $a>0 \wedge b>0$
  ensures $old\text{-}r = gcd \; a_0 \; b_0 \wedge gcd \; a_0 \; b_0 = a_0 * old\text{-}s + b_0 * old\text{-}t$
defines ‹
    $s = 0$; $old\text{-}s = 1$;
    $t = 1$; $old\text{-}t = 0$;
    $r = b$; $old\text{-}r = a$;
    while $(r \neq 0)$
      @*invariant* ‹
        $gcd \; old\text{-}r \; r = gcd \; a_0 \; b_0 \wedge r \geq 0 \wedge old\text{-}r>0$
      $\wedge \; a_0 * old\text{-}s + b_0 * old\text{-}t = old\text{-}r \wedge a_0 * s + b_0 * t = r$
      ›
      @*variant* ‹r›
    {
      $quotient = old\text{-}r \; / \; r$;
      $temp = old\text{-}r$;
      $old\text{-}r = r$;
      $r = temp - quotient * r$;
      $temp = old\text{-}s$;
      $old\text{-}s = s$;
      $s = temp - quotient * s$;
      $temp = old\text{-}t$;

```
      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 = old-y ∗ b0 + old-x ∗ a0 b = y ∗ b0 + x ∗ a0 q = a
div b
    shows
    a mod b + (a0 ∗ (x ∗ q) + b0 ∗ (y ∗ q)) = a
  ⟨proof⟩
end

program-spec euclid-extended′
  assumes a>0 ∧ b>0
  ensures a = gcd a0 b0 ∧ gcd a0 b0 = a0 ∗ x + b0 ∗ y
defines ‹
    x = 0;
    y = 1;
    old-x = 1;
    old-y = 0;
    while (b≠0)
      @invariant ‹
         gcd a b = gcd a0 b0 ∧ b≥0 ∧ a>0 ∧ a = a0 ∗ old-x + b0 ∗
old-y ∧ b = a0 ∗ x + b0 ∗ y
      ›
      @variant ‹b›
    {
      q = a / b;
      t = a;
      a = b;
      b = t mod b;
      t = x;
      x = old-x − q ∗ x;
      old-x = t;
      t = y;
      y = old-y − q ∗ y;
      old-y = t
    };
    x = old-x;
    y = old-y
  ›
⟨proof⟩
```

6.3.4 Exponentiation by Squaring

lemma *ex-by-sq-aux*:
  fixes *x* :: *int* **and** *n* :: *nat*
  assumes *n mod 2 = 1*
  shows $x * (x * x)$ ^ $(n\ div\ 2) = x$ ^ n
⟨*proof*⟩

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

- $x^n = x * x^{(n-1)/22}$ if n is odd
- $x^n = x^{n/22}$ if n is even

```
program-spec ex-by-sq
  assumes n≥0
  ensures r = x₀ ^nat n₀
defines ‹
    r = 1;
    while (n≠0)
      @invariant ‹
        n ≥ 0 ∧ r * x ^nat n = x₀ ^nat n₀
      ›
      @variant ‹n›
    {
      if (n mod 2 == 1) {
        r = r * x
      };
      x = x * x;
      n = n / 2
    }
  ›
  ⟨proof⟩
```

6.3.5 Power-Tower of 2s

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

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

```
program-spec tower2-imp
  assumes ‹m>0›
  ensures ‹a = tower2′ m₀›
  defines ‹
    a=1;
    while (m>0)
      @variant ‹m›
      @invariant ‹0≤m ∧ m≤m₀ ∧ a = tower2′ (m₀−m)›
```

```
  {
    n=a;

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

    m=m-1
  }
›
⟨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 ‹$l \le h$›
  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* ‹$l_0 \le l \wedge l \le h \wedge x \notin a$‘$\{l_0..<l\}$›
      $l=l+1$
  ›
  ⟨*proof*⟩

6.4.3 Check for Sortedness

term *ran-sorted*

program-spec *check-sorted*
  assumes ‹$l \le h$›
  ensures ‹$r \neq 0 \longleftrightarrow$ *ran-sorted* $a_0\ l_0\ h_0$›
  defines ‹
    if $(l==h)$ $r=1$
    else {
      $l=l+1$;
      while $(l<h \wedge a[l-1] \le a[l])$
        @*variant* ‹$h-l$›
        @*invariant* ‹$l_0<l \wedge l \le h \wedge$ *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* $a\ l\ h\ i \equiv l \le i \wedge i<h \wedge (\sum j=l..<i.\ a\ j) = (\sum j=i..<h.\ a\ j)$

program-spec *equilibrium*
  assumes ‹$l \le h$›
  ensures ‹*is-equil* $a\ l\ h\ i \vee i=h \wedge (\nexists i.$ *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 ∀ i∈{0..<n}. a i = a₀ ((i−1) mod n)
  defines ‹
    i = 0;
    prev = a[n − 1];
    while (i < n)
      @invariant ‹
        0 ≤ i ∧ i ≤ n
        ∧ (∀ j∈{0..<i}. a j = a₀ ((j−1) mod n))
        ∧ (∀ j∈{i..<n}. a j = a₀ j)
        ∧ prev = a₀ ((i−1) mod 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* $\longleftrightarrow$ $l \le h \wedge$ *ran-sorted a l h*

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

Then we prove that the program is correct

program-spec *binsearch*
assumes ‹*bin-search-pre a l h*›
ensures ‹*bin-search-post* a_0 l_0 h_0 x_0 *l*›
defines ‹
while $(l < h)$
@*variant* ‹$h-l$›
@*invariant* ‹$l_0 \le l \wedge l \le h \wedge h \le h_0 \wedge (\forall i \in \{l_0..<l\}.\ a\ i < x) \wedge (\forall i \in \{h..<h_0\}.\ x \le a\ i)$›
{
$m = (l + h) / 2$;
if $(a[m] < x)$ $l = m + 1$
else $h = m$
}
›
⟨*proof*⟩

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*: $x \in a\text{‘}\{l..<h\} \longleftrightarrow (i<h \wedge x = a\ i)$
and *bin-search-leftmost*: $x \notin a\text{‘}\{l..<i\}$
⟨*proof*⟩

6.4.7 Naive String Search

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

```
  }
›
  ⟨proof⟩

lemma lran-eq-iff': lran a l1 (l1 + (h − l)) = lran a' l h
  ⟷ (∀ i. 0 ≤ i ∧ i < h − l ⟶ a (l1 + i) = a' (l + i)) if l ≤ h
  ⟨proof⟩

program-spec match-string'
  assumes l1 ≤ h1
ensures i = h1 − l1 ⟷ lran a l (l + (h1 − l1)) = lran b l1 h1
for i h1 l1 l a[] b[]
defines ‹inline match-string›
  ⟨proof⟩

program-spec substring
  assumes l ≤ h ∧ l1 ≤ h1
  ensures match = 1 ⟷ (∃ j ∈ {l0..<h0}. lran a j (j + (h1 − l1))
= lran b l1 h1)
  for a[] b[]
  defines ‹
  match = 0;
  while (l < h ∧ match == 0)
    @invariant‹l0 ≤ l ∧ l ≤ h ∧ match ∈ {0,1} ∧
    (if match = 1
     then lran a l (l + (h1 − l1)) = lran b l1 h1 ∧ l < h
     else (∀ j ∈ {l0..<l}. lran a j (j + (h1 − l1)) ≠ lran b l1 h1))›
    @variant‹(h − l) ∗ (1 − match)›
  {
    inline match-string';
    if (i == h1 − l1) {match = 1}
    else {l = l + 1}
  }
›
  ⟨proof⟩

program-spec substring'
  assumes l ≤ h ∧ l1 ≤ h1
  ensures match = 1 ⟷ (∃ j ∈ {l0..h0−(h1 − l1)}. lran a j (j +
(h1 − l1)) = lran b l1 h1)
  for a[] b[]
  defines ‹
  match = 0;
  if (l + (h1 − l1) ≤ h) {
    h = h − (h1 − l1) + 1;
    inline substring
  }
›
  ⟨proof⟩
```

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

lemma *lran-split*:
  $lran\ a\ l\ h = lran\ a\ l\ p\ @\ lran\ a\ p\ h$ **if** $l \leq p\ p \leq h$
  ⟨*proof*⟩

lemma *lran-eq-append-iff*:
  $lran\ a\ l\ h = as\ @\ bs \longleftrightarrow (\exists\ i.\ l \leq i \wedge i \leq h \wedge as = lran\ a\ l\ i \wedge bs = lran\ a\ i\ h)$ **if** $l \leq h$
  ⟨*proof*⟩

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

program-spec *substring-final*
  assumes $l \leq h \wedge 0 \leq len$
  ensures $match = 1 \longleftrightarrow (\exists as\ bs.\ lran\ a\ l_0\ h_0 = as\ @\ lran\ b\ 0\ len\ @\ bs)$
  for *l h len match a*[] *b*[]
  defines ‹$l1 = 0$; $h1 = len$; *inline substring'*›
  ⟨*proof*⟩

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 a₀ a i
  ⟷ (let key = a₀ j in
      i∈{l−1..<j}            — i is in range
    — Content of new array
    ∧ (∀k∈{l..i}. a k = a₀ k)
    ∧ a (i+1) = key
    ∧ (∀k∈{i+2..j}. a k = a₀ (k−1))
    ∧ a = a₀ on −{l..j}
    — Placement of key
    ∧ (i≥l ⟶ a i ≤ key)  — Element at i smaller than key, if it exists
    ∧ (∀k∈{i+2..j}. a k > key) — Elements ≥i+2 greater than key
    )

  for l j i :: int and a₀ a :: int ⇒ int

program-spec insort-insert
  assumes l<j
  ensures insort-insert-post l j a₀ a i
  defines ‹
    key = a[j];
    i = j−1;
    while (i≥l ∧ a[i]>key)
      @variant ‹i−l+1›
      @invariant ‹l−1≤i ∧ i<j
        ∧ (∀k∈{l..i}. a k = a₀ k)
        ∧ (∀k∈{i+2..j}. a k > key ∧ a k = a₀ (k−1))
        ∧ a = a₀ on −{l..j}
        ›
    {
      a[i+1] = a[i];
      i=i−1
    };
    a[i+1] = key
  ›
  ⟨proof⟩
```

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

Invoking *insort-insert* will sort in the element

```
lemma insort-insert-sorted:
  assumes l<j
```

assumes *insort-insert-post l j a a′ i′*
assumes *ran-sorted a l j*
shows *ran-sorted a′ l (j + 1)*
⟨*proof*⟩

Invoking *insort-insert* will only mutate the elements

lemma *insort-insert-ran1*:
assumes *l<j*
assumes *insort-insert-post l j a a′ i*
shows *mset-ran a′ {l..j} = mset-ran a {l..j}*
⟨*proof*⟩

The property ⟦*?l < ?j*; *insort-insert-post ?l ?j ?a ?a′ ?i*⟧ ⟹ *mset-ran ?a′ {?l..?j} = mset-ran ?a {?l..?j}* extends to the whole array to be sorted

lemma *insort-insert-ran2*:
assumes *l<j j<h*
assumes *insort-insert-post l j a a′ i*
shows *mset-ran a′ {l..<h} = mset-ran a {l..<h}* (**is** *?thesis1*)
and *a′=a on −{l..<h}* (**is** *?thesis2*)
⟨*proof*⟩

Finally, we specify and prove correct the outer loop

program-spec *insort*
assumes *l<h*
ensures *ran-sorted a l h* ∧ *mset-ran a {l..<h} = mset-ran* a_0 *{l..<h}* ∧ *a=*a_0 *on −{l..<h}*
for *a*[]
defines ‹
j = l + 1;
while (j<h)
@*variant* ‹*h−j*›
@*invariant* ‹
l<j ∧ *j≤h* — *j* in range
∧ *ran-sorted a l j* — Array is sorted up to *j*
∧ *mset-ran a {l..<h} = mset-ran* a_0 *{l..<h}* — Elements in range only permuted
∧ *a=*a_0 *on −{l..<h}*
›
{
inline insort-insert;
j=j+1
}
›
⟨*proof*⟩

6.4.9 Quicksort

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

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

defines ‹
$i{=}l;\ j{=}l;$
while $(j<h)$
@*invariant* ‹
$l \le i\ \wedge\ i \le j\ \wedge\ j \le h$
$\wedge\ mset\text{-}ran\ a_0\ \{l_0..<h_0\} = mset\text{-}ran\ a\ \{l_0..<h_0\}$
$\wedge\ (\forall k \in \{l..<i\}.\ a\ k < p)$
$\wedge\ (\forall k \in \{i..<j\}.\ a\ k \ge p)$
$\wedge\ (\forall k \in \{j..<h\}.\ a_0\ k = a\ k)$
$\wedge\ a_0 = a\ on\ -\{l_0..<h_0\}$
›
@*variant* ‹$(h-j)$›
{
if $(a[j]<p)$ $\{temp = a[i];\ a[i] = a[j];\ a[j] = temp;\ i{=}i{+}1\};$
$j{=}j{+}1$
}
›
⟨*proof*⟩

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

defines ‹
$p = a[h-1];$
$(a,i) = partition\text{-}aux(a,l,h-1,p);$
$a[h-1] = a[i];$
$a[i] = p$
›
⟨*proof*⟩

lemma *quicksort-sorted-aux*:
assumes *BOUNDS*: $l \le i\ i < h$

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

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

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

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

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

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

shows $ran\text{-}sorted\ a_3\ l\ h$
⟨*proof*⟩

lemma *quicksort-mset-aux*:
assumes *B*: $l_0 \le i\ i<h_0$
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 $mset\text{-}ran\ a\ \{l_0..<h_0\} = mset\text{-}ran\ ab\ \{l_0..<h_0\}$
⟨*proof*⟩

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 ‹
```
  if (l<h) {
    (a,i) = partition(a,l,h,a[l]);
    a = rec quicksort(a,l,i);
    a = rec quicksort(a,i+1,h)
  }
›
```
⟨*proof*⟩

6.5 Data Refinement

6.5.1 Filtering

program-spec *array-filter-negative*
assumes $l \le h$
ensures $lran\ a\ l_0\ i = filter\ (\lambda x.\ x \ge 0)\ (lran\ a_0\ l_0\ h_0)$
defines ‹
```
  i=l; j=l;
  while (j<h)
    @invariant ‹
```

$l \le i \wedge i \le j \wedge j \le h$
$\wedge$ *lran a l i = filter* ($\lambda x.\ x \ge 0$) (*lran* a_0 *l j*)
$\wedge$ *lran a j h = lran* a_0 *j h*
›
@*variant* ‹$h-j$›
{
if ($a[j] \ge 0$) {$a[i] = a[j]$; $i=i+1$};
$j=j+1$
}
›
⟨*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*) $\wedge$ *set* (*merge xs ys*) = *set xs* $\cup$ *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*≠[] $\vee$ *ys*≠[] $\Longrightarrow$ *merge xs ys* = (
if ys=[] $\vee$ (*xs*≠[] $\wedge$ *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, $l \le h$, is carried through explicitly.

program-spec *merge-imp′*
  assumes $l1 \le h1 \wedge l2 \le h2$
  ensures *let* $ms = lran\ m\ 0\ j$; $xs_0 = lran\ a1_0\ l1_0\ h1_0$; $ys_0 = lran\ a2_0\ l2_0\ h2_0$ *in*
    $j \ge 0 \wedge ms = merge\ xs_0\ ys_0$
  defines ‹

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

Given the *merge-eq* theorem, which captures the essence of a loop step, and the theorems $?l \le ?h \Longrightarrow lran\ ?a\ ?l\ (?h + 1) = lran\ ?a\ ?l\ ?h\ @\ [?a\ ?h]$, $lran\ ?a\ (?l + 1)\ ?h = tl\ (lran\ ?a\ ?l\ ?h)$, and $?l < ?h \Longrightarrow hd\ (lran\ ?a\ ?l\ ?h) = ?a\ ?l$, 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 $l1 \le h1 \wedge l2 \le h2 \wedge sorted\ (lran\ a1\ l1\ h1) \wedge sorted\ (lran\ a2\ l2\ h2)$
  ensures *let* $ms = lran\ m\ 0\ j$ *in*
      $j \ge 0$
    $\wedge\ sorted\ ms$
    $\wedge\ mset\ ms = mset\ (lran\ a1_0\ l1_0\ h1_0) + mset\ (lran\ a2_0\ l2_0\ h2_0)$
  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 ∪ *a* = *UNIV*
a ∪ *UNIV* = *UNIV*
⟨*proof*⟩

lemma *merge-msets-aux*: ⟦*l*≤*m*; *m*≤*h*⟧ ⟹ *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*≤*h*
ensures ‹*0*≤*j* ∧ *sorted* (*lran b 0 j*) ∧ *mset* (*lran b 0 j*) = *mset* (*lran* a_0 l_0 h_0))›
variant ‹*h*−*l*›
for *a*[] *b*[]
defines ‹

```
    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)
    }
```

›
⟨*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* ⇒ *int*) ⇒ *int set* **where** *set-of a* ≡ {*i*. *a i* ≠ *0*}

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

lemma *set-of-empty*[*simp*]: *set-of* (λ-. *0*) = {} ⟨*proof*⟩
lemma *set-of-insert*[*simp*]: *x*≠*0* ⟹ *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*∈*set-of a* ⟷ *a i* ≠ *0* ⟨*proof*⟩
end

```
program-spec dedup
  assumes ‹l≤h›
  ensures ‹set (lran a l i) = set (lran a₀ l h) ∧ distinct (lran a l i)›
  defines ‹
    i=l; j=l;
    clear b[];
    while (j<h)
      @variant ‹h−j›
      @invariant ‹l≤i ∧ i≤j ∧ j≤h
        ∧ set (lran a l i) = set (lran a₀ l j)
        ∧ distinct (lran a l i)
        ∧ lran a j h = lran a₀ j h
        ∧ set-of b = set (lran a l i)
       ›
    {
      if (b[a[j]] == 0) {
        a[i] = a[j]; i=i+1; b[a[j]] = 1
      };
      j=j+1
    }
  ›
  ⟨proof⟩


procedure-spec bv-init () returns b
  assumes True ensures ‹set-of b = {}›
  defines ‹clear b[]›
  ⟨proof⟩

procedure-spec bv-insert (x, b) returns b
  assumes True ensures ‹set-of b = insert x₀ (set-of b₀)›
  defines ‹b[x] = 1›
  ⟨proof⟩

procedure-spec bv-remove (x, b) returns b
  assumes True ensures ‹set-of b = set-of b₀ − {x₀}›
  defines ‹b[x] = 0›
  ⟨proof⟩

procedure-spec bv-elem (x,b) returns r
  assumes True ensures ‹r≠0 ⟷ x₀∈set-of b₀›
  defines ‹r = b[x]›
```

⟨*proof*⟩

procedure-spec *dedup'* (*a*,*l*,*h*) **returns** (*a*,*l*,*i*)
assumes ‹$l \le h$› **ensures** ‹*set* (*lran a l i*) = *set* (*lran* a_0 l_0 h_0) ∧ *distinct* (*lran a l i*) ›
for *b*[]
defines ‹

```
  b = bv-init();

  i=l; j=l;

  while (j<h)
    @variant ‹h−j›
    @invariant ‹l≤i ∧ i≤j ∧ j≤h
      ∧ set (lran a l i) = set (lran a0 l j)
      ∧ distinct (lran a l i)
      ∧ lran a j h = lran a0 j h
      ∧ set-of b = set (lran a l i)
     ›
  {
    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
  }
›
```

⟨*proof*⟩

6.6 Recursion

6.6.1 Recursive Fibonacci

recursive-spec *fib-imp* (*i*) **returns** *r* **assumes** *True* **ensures** ‹$r = fib\ i_0$› **variant** ‹*i*›
defines ‹

```
  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
  }
›
```

⟨*proof*⟩

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 ‹n≥0 ∧ m≥0› ensures True variant ‹n+m›
  defines ‹
    if (n≠0 ∧ m≠0) {
      G = G + m;
      if (n<m) rec coast (n−1,m−1) else rec pedal(n−1,m)
    }
  ›
and
  coast (n,m) returns () assumes ‹n≥0 ∧ m≥0› ensures True variant ‹n+m+1›
  defines ‹
    G = G + n;
    if (n<m) rec coast (n,m−1) else rec pedal (n,m)
  ›
  ⟨proof⟩
```

6.6.3 Ackermann

```
fun ack :: nat ⇒ nat ⇒ 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≠0 ⟹ ack m 0 = ack (m−1) 1
  ⟦m≠0; n≠0⟧ ⟹ ack m n = ack (m−1) (ack m (n−1))
  ⟨proof⟩

recursive-spec relation less-than <*lex*> less-than
  ack-imp (m,n) returns r
    assumes m≥0 ∧ n≥0 ensures r=int (ack (nat m₀) (nat n₀))
    variant (nat m, nat n)
    defines ‹
      if (m==0) r = n+1
      else if (n==0) r = rec ack-imp (m−1,1)
      else {
        t = rec ack-imp (m,n−1);
        r = rec ack-imp (m−1,t)
      }
    ›
  ⟨proof⟩
```

6.6.4 McCarthy's 91 Function

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

recursive-spec *p91*(y) **assumes** *True* **ensures** *if* $100<y_0$ *then* $G = y_0-10$ *else* $G = 91$ **variant** $101-y$
  for G
  defines ‹
    if ($100<y$) $G=y-10$
    else {
      rec p91 ($y+11$);
      rec p91 (G)
    }
  ›
  ⟨*proof*⟩

6.6.5 Odd/Even

recursive-spec
  odd-imp (a) **returns** b
    assumes *True*
    ensures $b\neq 0 \longleftrightarrow$ *odd* a_0
    variant $|a|$
    defines ‹
      if ($a==0$) $b=0$
      else if ($a<0$) b = *rec even-imp* ($a+1$)
      else b = *rec even-imp* ($a-1$)
    ›
  and
  even-imp (a) **returns** b
    assumes *True*
    ensures $b\neq 0 \longleftrightarrow$ *even* a_0
    variant $|a|$
    defines ‹
      if ($a==0$) $b=1$
      else if ($a<0$) b = *rec odd-imp* ($a+1$)
      else b = *rec odd-imp* ($a-1$)
    ›
  ⟨*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** ‹$GZ = GZ_0 + GX_0*GY_0$› **variant** ($|GY|$,*1::nat*)

for *GX GY GZ*
defines
‹
e = *even-imp* (*GY*);
if (*e*≠*0*) *rec evenproduct*() *else rec oddproduct*()
›
and
oddproduct() **assumes** ‹*odd GY*› **ensures** ‹$GZ = GZ_0 + GX_0 * GY_0$›
variant (|*GY*|,*0*::*nat*)
for *GX GY GZ*
defines
‹
if (*GY*<*0*) {
GY = *GY* + *1*;
GZ = *GZ* − *GX*
} *else* {
GY = *GY* − *1*;
GZ = *GZ* + *GX*
};
rec evenproduct()
›
and
evenproduct() **assumes** ‹*even GY*› **ensures** ‹$GZ = GZ_0 + GX_0 * GY_0$›
variant (|*GY*|,*0*::*nat*)
for *GX GY GZ*
defines
‹
if (*GY*≠*0*) {
GX = *2*∗*GX*;
GY = *GY* / *2*;
rec 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 *s1*... *sn*:

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

definition *succs* **where**
succs a i ≡ *a* ‘ {*i*+*1*..<*a i*} **for** *a* :: *int* ⇒ *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** ‹$lran\ stack\ 0\ ptr = lran\ stack_0\ 0\ ptr_0$ @ $[x_0] \wedge ptr = ptr_0 + 1$›
defines ‹$stack[ptr] = x;\ ptr = ptr + 1$›
⟨*proof*⟩

procedure-spec *push* (*x*, *stack*, *ptr*) **returns** (*stack*, *ptr*)
assumes $ptr \geq 0$ **ensures** ‹$stack$ ‘ $\{0..<ptr\} = \{x_0\} \cup stack_0$ ‘ $\{0..<ptr_0\} \wedge ptr = ptr_0 + 1$›
for *stack*[]
defines ‹$stack[ptr] = x;\ ptr = ptr + 1$›
⟨*proof*⟩

program-spec *get-succs*
assumes $j \leq stop \wedge stop = a\ (j - 1) \wedge 0 \leq i$
ensures
$stack$ ‘ $\{0..<i\} = \{x.\ (j_0 - 1, x) \in Edges\ a \wedge x \notin set\text{-}of\ visited\}$ $\cup\ stack_0$ ‘ $\{0..<i_0\}$
$\wedge\ i \geq i_0$
for *i j stop stack*[] *a*[] *visited*[]
defines

```
‹
  while (j < stop)
   @invariant ‹stack ‘ {0..<i} = {x. x ∈ a ‘ {j0..<j} ∧ x ∉ set-of
visited} ∪ stack0 ‘ {0..<i0}
    ∧ j ≤ stop ∧ i0 ≤ i ∧ j0 ≤ j
  ›
  @variant ‹(stop − j)›
  {
    succ = a[j];
    is-elem = bv-elem(succ,visited);
    if (is-elem == 0) {
      (stack, i) = push (succ, stack, i)
    };
    j = j + 1
  }
›
```

⟨*proof*⟩

procedure-spec *pop* (*stack*, *ptr*) **returns** (*x*, *ptr*)
assumes $ptr \geq 1$ **ensures** ‹$stack_0$ ‘ $\{0..<ptr_0\} = stack_0$ ‘ $\{0..<ptr\} \cup \{x\} \wedge ptr_0 = ptr + 1$›
for *stack*[]
defines ‹$ptr = ptr - 1;\ x = stack[ptr]$›
⟨*proof*⟩

procedure-spec *stack-init* () **returns** *i*
  assumes *True* **ensures** ‹$i = 0$›
  defines ‹$i = 0$›
  ⟨*proof*⟩

lemma *Edges-empty*:
  $Edges\ a\ ``\ \{i\} = \{\}$ **if** $i + 1 \geq a\ i$
  ⟨*proof*⟩

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*: $(s, x) \in (Edges\ a)^*$
    and *closed*: $\forall\, v{\in}visited.\ Edges\ a\ ``\ \{v\} \subseteq visited$
    and *start*: $s \in visited$
  shows $x \in visited$
  ⟨*proof*⟩

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

```
      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 ⊂ insert x s ⟷ x∉s ⟨proof⟩

program-spec dfs1
  assumes 0 ≤ x ∧ 0 ≤ s ∧ start = s ∧ edges = a
  ensures b = 1 ⟷ x ∈ (Edges a)* `` {s}
  for visited[]
  defines
‹
  b = 0;
  — i will point to the next free space in the stack (i.e. it is the size
of the stack)
  i = 1;
  — Initially, we put s on the stack.
  stack[0] = s;
  visited = bv-init();
  while (b == 0 ∧ i ≠ 0)
    @invariant ‹
    0 ≤ i ∧ (s ∈ stack ` {0..<i} ∨ s ∈ set-of visited) ∧ (b = 0 ∨ b =
1) ∧
    set-of visited ⊆ (Edges edges)* `` {start} ∧ (
    if b = 0 then
      stack ` {0..<i} ⊆ (Edges a)* `` {s}
      ∧ (∀ v ∈ set-of visited. (Edges a) `` {v} ⊆ set-of visited ∪ stack `
{0..<i})
      ∧ (x ∉ set-of visited)
    else x ∈ (Edges a)* `` {s})
    ›
   @relation ‹finite-psupset ((Edges edges)* `` {start}) <*lex*> less-than›
    @variant ‹ (set-of visited, nat i)›
  {
    — Take the top most element from the stack.
    (next, i) = pop(stack, i);
    if (next == x) {
      — If it is the target, we are done.
```

```
      visited = bv-insert(next, visited);
      b = 1
    } else {
      is-elem = bv-elem(next, visited);
      if (is-elem == 0) {
        visited = bv-insert(next, visited);
      — Else, put its successors on the stack if they are not yet visited.
        stop = a[next];
        j = next + 1;
        if (j ≤ stop) {
          inline get-succs
        }
      }
    }
  }
›
  ⟨proof⟩
```

end

end